```go
import (
  "sync/atomic"
)

var count uint64

// Count safely gets a unique count number.
func Count() uint64 {
  atomic.AddUint64(&count, 1)
  return count
}
```

```go
import (
  "testing"
  "github.com/matryer/mypackage"
)

func TestCount(t *testing.T) {
  if mypackage.Count() != 1 {
    t.Error("expected 1")
  }
  if mypackage.Count() != 2 {
    t.Error("expected 2")
  }
  if mypackage.Count() != 3 {
```

Software Testing for Continuous Delivery

Seminar 6: Unit Testing Cont & Specification-based Testing (theory)

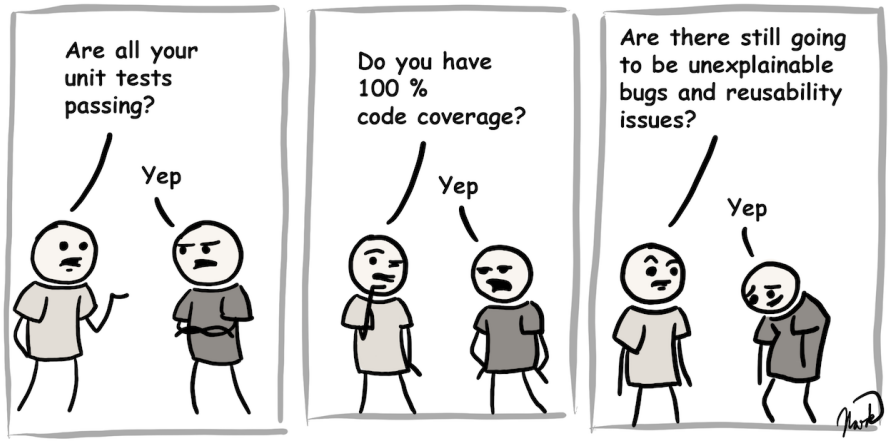
Dr. Byron J. Williams
September 11, 2019

UF Herbert Wertheim College of Engineering
Department of Computer & Information Science & Engineering
UNIVERSITY *of* FLORIDA

Content adapted from “Developer Testing” by Alexander Tarlinder 1st Edition (2016)
Images courtesy Google Image Search

FCS Research

Code Coverage



- Used to find untested parts of your code
  - oft referred to as “test” coverage
- Measures the ‘amount’ of code executed by your test suite (i.e., has an assertion against it)
- Does not indicate how good your tests are
- Can be used as *part* of an exit criteria along with reliability measures (e.g., MTTF)
- Low numbers are more telling than high coverage numbers (e.g., 99)
- What are good values for code coverage?

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

Code Coverage Metrics

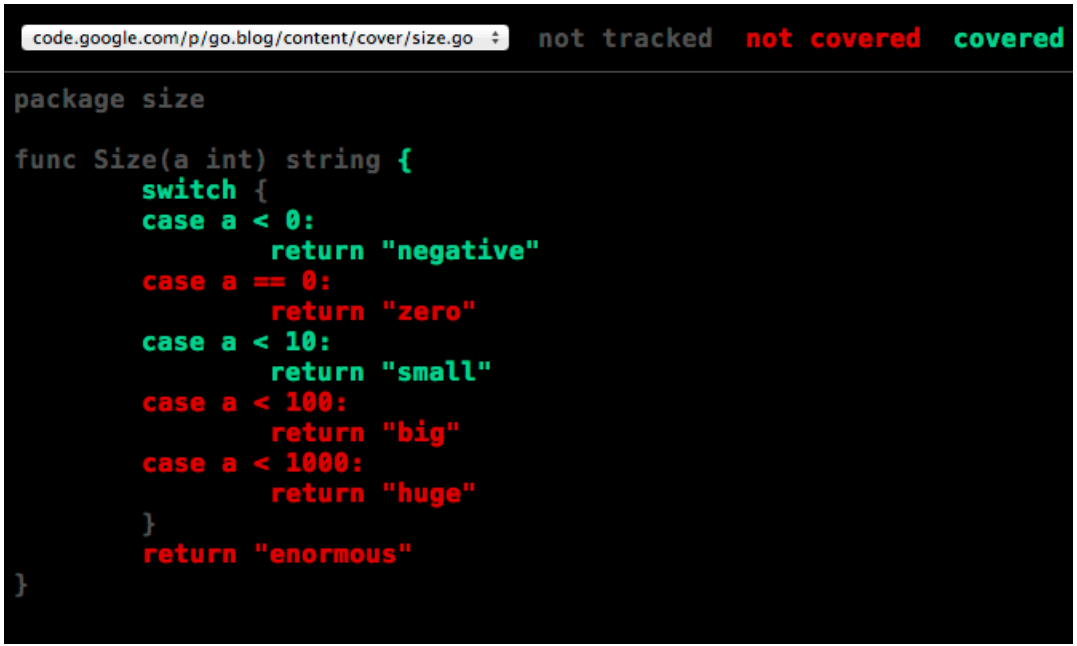
- **Function coverage:** how many of the functions defined have been called.
- **Statement coverage**: how many of the statements in the program have been executed.
- **Branches coverage:** how many of the branches of the control structures (if statements for instance) have been executed.
- **Condition coverage:** how many of the boolean sub-expressions have been tested for a true and a false value.
- **Line coverage:** how many of lines of source code have been tested.

https://www.atlassian.com/continuous-delivery/software-testing/code-coverage

Test Coverage Example (titlecase.js)

```
Dropbox/code-db/titlecase via ⬢ v12.7.0
 [I] ➜ yarn run test --coverage
yarn run v1.17.3
$ jest --coverage
 PASS  ./titlecase.test.js
  ✓ jest framework runs correctly (1ms)
  titlecase returns the appropriate Title Cased movie title for the movie entered
    ✓ should not capitalize articles found past the first word (2ms)
    ✓ should capitalize the first letter after a hyphen
    ✓ should accept a number as a title
    ✓ should properly captialize mixed case strings (1ms)
    ✓ should capitalize the first letter of each word in a multi-word title
    ✓ should capitalize a single word movie title
    ✓ should capitalize a single letter movie title
    ✓ should only accept a string as its parameter (3ms)
    ✓ should return a string

--------------|----------|----------|----------|----------|-------------------|
File          |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
--------------|----------|----------|----------|----------|-------------------|
All files     |      100 |      100 |      100 |      100 |                   |
 titlecase.js |      100 |      100 |      100 |      100 |                   |
--------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        0.986s, estimated 1s
Ran all test suites.
✨  Done in 1.41s.
```

```
package size

func Size(a int) string {
    switch {
    case a < 0:
        return "negative"
    case a == 0:
        return "zero"
    case a < 10:
        return "small"
    case a < 100:
        return "big"
    case a < 1000:
        return "huge"
    }
    return "enormous"
}
```

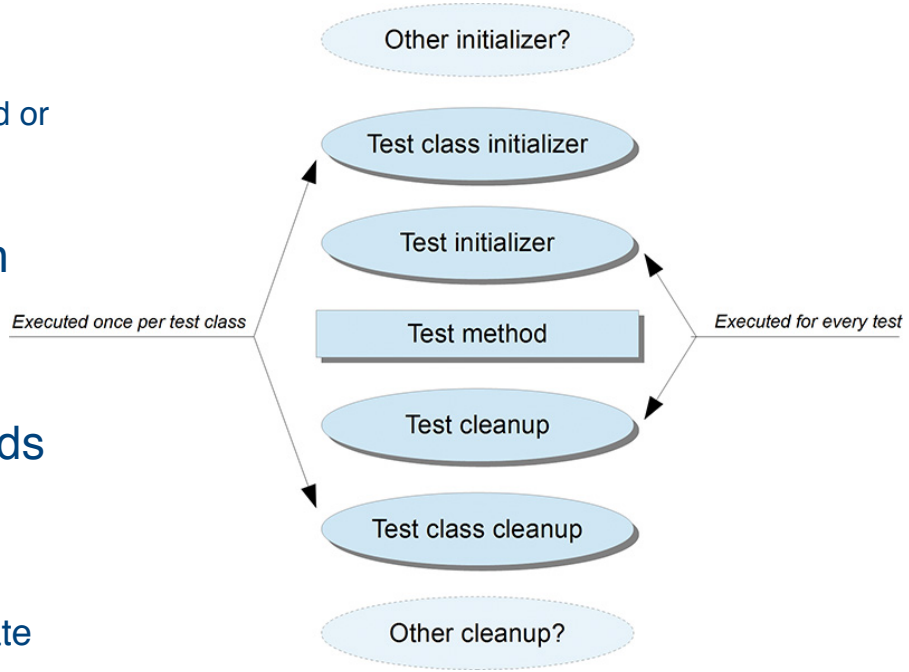


Go Statement Coverage Example

https://blog.golang.org/cover

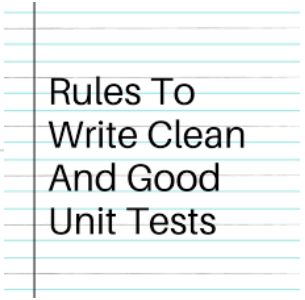
xUnit Tests

- xUnit can be used to test …
  - … an entire object or part of an object i.e., a method or some interacting methods
  - … interaction between several objects
- It is primarily intended for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests



life cycle of a unit testing framework

Good Unit Tests



- Test a **single** logical concept in the system
- Have **full control** over all the pieces running and uses mocks to achieve this isolation as needed
- Are able to be fully **automated** and can be **run in any order**
- Return a **consistent result** for the same test (For example, no random numbers; save those for broader tests)
- Cause **no side effects** and **limits external access** (network, database, file system, etc.)
- Provide trustworthy results such that referring to the code is unnecessary to confirm it
- **Run fast** and are written with readable and maintainable test code, descriptive test names, and is organized or **grouped logically**

References: Adapted from http://artofunittesting.com/definition-of-a-unit-test/

UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

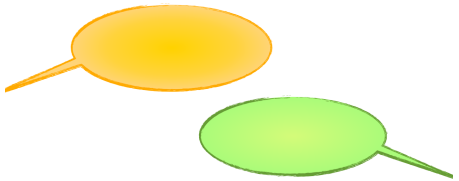
POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

Behavior-Driven Development

Behavior-Driven Development

- Encourages collaboration between Business Analysts, QA Engineers, Developers & Business Owners / Stakeholders
- Conversation focused & driven by business value



- Extends TDD by using **natural language** that non-technical stakeholders can understand
- User Stories & Acceptance Criteria typically defined with business or user focused language
- Developers implement acceptance criteria

BDD Principles

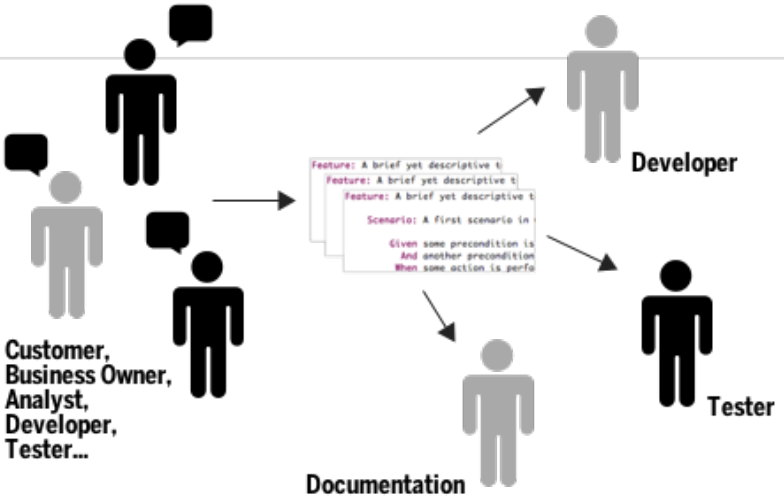
- BDD focuses on:
  - Where to start in the process
  - What to test and what not to test
  - How much to test in one go
  - What to call the tests
  - How to understand why a test fails

BDD provides software development and management teams with shared tools and a shared process to collaborate on software development

- BDD states that tests of any unit of software should be specified in terms of the desired behavior of the unit
  - **Describe** a test **set/suite** for the unit first ;
  - write what 'it', the test does;
  - make the tests fail;
  - implement the unit;
  - finally verify that the implementation of the unit makes the tests succeed
- Similar to TDD (description of desired behavior up front)

BDD

- "TDD Done Right"
- Stories And Specifications
- Ubiquitous Language (Domain Specific Language)
  - Given/When/Then
  - Should/Ensure
  - Expect



Gomega, chaijs.com, cucumber.io, jest.io

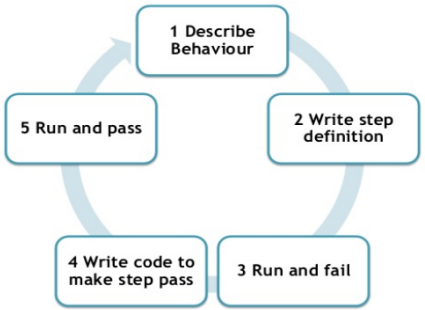
BDD Examples

```
describe('Counter', function() {
  it('should increase count by 1 after calling tick', function() {
    var counter = new Counter();
    var expectedCount = counter.count + 1;

    counter.tick();

    assert.equal(counter.count, expectedCount);
  });
});
```

```
it "adds a reminder date when an invoice is created"  do
  current_invoice = create :invoice
  current_invoice.reminder_date.should == 20.days.from_now
end
```

```
1  const titlecase = require('./titlecase');
2
3  describe('titlecase returns the appropriate Title Cased movie title for the movie entered', function() {
4    it('should not capitalize articles found past the first word', () => {
5      const title = 'the young and the restless';
6      expect(titlecase(title)).toBe('The Young and the Restless');
7    });
8
9    it('should capitalize the first letter after a hyphen', () => {
10     expect(titlecase('x-men first-class')).toBe('X-Men First-Class');
11     expect(titlecase('mission-critical EVEnt')).toBe('Mission-Critical Event');
12   });
13   it('should accept a number as a title', () => {
14     expect(titlecase('24')).toBe('24');
15   });
16
17   it('should properly captialize mixed case strings', () => {
18     expect(titlecase('AveNgers eNDgamE')).toBe('Avengers Endgame');
19   });
20
21   it('should capitalize the first letter of each word in a multi-word title', function() {
22     expect(titlecase('superman returns')).toBe('Superman Returns');
23   });
24
25   it('should capitalize a single word movie title', () => {
26     expect(titlecase('it')).toBe('It');
27   });
28
29
30
31
32
33
34
35
36
37
38
39 });
40
41 test('jest framework runs correctly', () => {
42   expect(true).toBeTruthy();
43 });
```

Given-When-Then Example



Earning Frequent Flyer points from flights

In order to encourage travellers to book with
Flying High Airlines more frequently
As the Flying High sales manager

Acceptance Criteria

Earning points from standard flights
Earning extra points based on cabin category

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between <departure> and <destination> is <distance> km
And I am a standard Frequent Flyer member
When I fly from <departure> to <destination>
Then I should earn <points> points
Examples:
| departure | destination | distance | points | comments                                      |
| Sydney    | Melbourne   | 878      | 439    | 1 point per 2 kms for domestic flights        |
| Sydney    | Perth       | 4100     | 2050   |                                               |
| Sydney    | Hong Kong   | 7500     | 5000   | 1 point per 1.5 kms for international flights |
```

http://www.slideshare.net/wakaleo/bdd-in-action-principles-practices-and-realworld-application

Problems with Unit Testing & TDD pt 1

- **Can seem to double development time (in the short run)**
  - Practicing TDD can feel like it nearly doubles development time because the developer is writing twice the amount of code
  - Long term, however, previously mentioned the benefits of TDD can ultimately save development time
  - This issue is frequently the subject of debate when implementing TDD
- **Difficult to unit test functions with "side effects" or time-delays**
  - Functions that cause irreversible or otherwise un-mockable side effects (such as a imprecise and physical robot arm movement with feedback) may be difficult to unit test
  - To test functions that perform an action but do not return a value, the unit test must check the proper existence/nonexistence of the side effect, which can become difficult (spies as a solution i.e., did the function get called)
- **No tests for the tests**
  - Unit tests themselves are code and can be written incorrectly
  - They might fail at first (as required by TDD), but never become "green" or successful (or successful too early) during the "write code" step because the test itself has one or more faults

Problems with Unit Testing & TDD pt 2

- **TDD doesn't always assist with changes in requirements, system-wide code structure, or the application programming interface (API)**
  - Especially in the early stages of a project, refactoring or other changes can be so extensive that it needs to "break" some tests
  - Those tests need to be modified/rewritten, but the corresponding main code for them might already exist, so the new tests are written without following TDD. Remember that TDD is focused on code base development, not test development
  - Existing unit tests are beneficial during most changes, particularly if a dependency is modified. The tests for each component that relies on that modified dependency can be run the to ensure the change didn't break the higher-level component.
- **Desire to make all tests "green"**
  - The desire to see a passing or "green" result on all tests can cause developers to skip necessary tests or remove broken tests that should instead be fixed

Problems with Unit Testing & TDD pt 3

- **Requires the initial and continuous development of "mocks"**
  - Each external resource or dependency must be mocked to ensure isolation and this process can be time-consuming
  - Unless the mock is built in/with the actual implementation of a dependency, the mock must be constantly updated as the API of the dependency
- **Sometimes difficult to test "private" functions/methods**
  - Sometimes, even when treating an object/class as a "unit," private functions (particularly complex ones) need to be unit tested individually. This means, depending on the language and/or testing framework, that the private function(s) must be exposed in some manner.
- **TDD does not necessarily result in "quality" tests and never guarantees proper code**
  - Developers might skip edge cases, might write an untested branch by accident, etc.
  - TDD is not designed to build the best tests, it's designed as a development process
- **Tedious**
  - Sometimes TDD can feel like it gets in the way of "just coding."
  - **Advice: Try it anyway**

UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

Specification-based Testing

- Okay, I know how to write a unit test, but what should I test?
- How many tests should I write?
- Have I thoroughly tested this unit? How can I show this?
- How can a developer think like a tester (practically vs. theoretically)?

Writing Tests

- **Order of Tests**
  1. **Degenerate case**—Start with a test that operates on an "empty" value like zero, null, the empty string, or the like.
  2. **Happy path tests**—lay the foundation for the implementation while remaining focused on the core functionality
  3. **Provide new information or knowledge**—Try approaching the solution from different angles and exercising different parts
  4. **Error handling and negative tests**—These tests are crucial to correctness - in many cases, they can safely be written last.

- **Red to Green Strategies**
  - **Faking**—This is the simplest way to make a test pass. Just return or do whatever the particular test expects. If the test expects a specific value, then just hand it over
  - **The obvious implementation**—Sometimes beating about the bush just isn't worth it. If we know what to type, then we should type it.
  - **Triangulation** - Some algorithms can be teased out by providing a number of examples and then generalizing

Equivalence Partitioning

- Testing an integer-based calculator— is it meaningful to test whether it can compute the sum of 5 + 5 if it computes the sums of 3 + 3 and 4 + 4 correctly? Or 10,000 + 20,000?

- Probably not, but why?

Equivalence Partitioning

- Concept from partition-based testing - basic ideas:
  - Members in equivalence class are treated as **“equivalent”**
  - Defining meaningful partitions — sampling from partitioned subsets for different types of partitions
  - Coverage of partitions: uniform
- Ex: Integer-based calculator - all integers are in the same equivalence partition or ***equivalence class***, are subsets of data in which all values are equivalent to each other

Testing for Partition Coverage

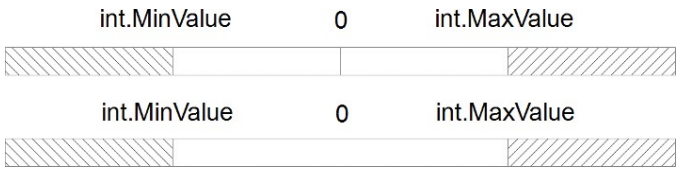
Sensitize test cases

- i.e., defining specific input variables and associated values to exercise certain parts of the program in the white-box view or to perform certain functions in the black-box view
- e.g., function `add(int a, int b)`
- considering valid/invalid input values of *a* and *b*
- How many cases are in exhaustive test?

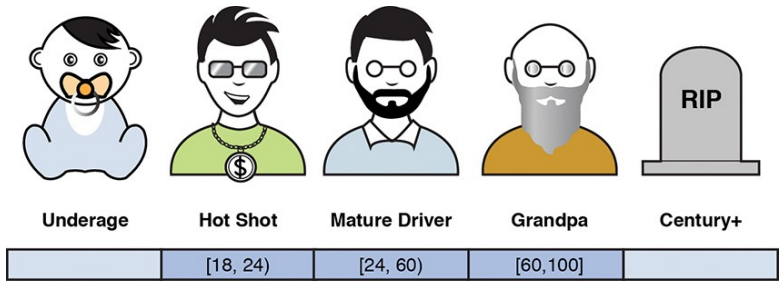
Test Case	Condition		Input	
	int a	***int b***	***a***	***b***
1	False	False	3.2	-0.4
2	False	True	"MSU"	2
3	True	False	7	3/4
4	True	True	-9	-2

Domain Equivalence

- A **developer** would know the **range** of a data type and base the partitioning on that…
- Whereas the **tester** would probably think more about the **domain** and partition from that viewpoint. This could lead to different partitioning
- Near endless possibilities, and it's the **specification** and **test scenario** that should guide the choice of relevant equivalence partitions




```
public double getPremiumFactor(int age) {
   if (age >= 18 && age < 24) {
      return 1.75;
   } else if (age >= 24 && age < 60) {
      return 1;
   } else if (age >= 60) {
      return 1.35;
   }
}
```

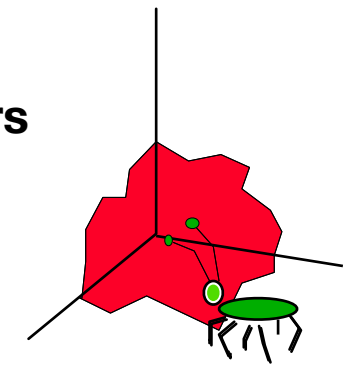


Boundary Value Analysis

- Sub-domain coverage is inadequate when dealing with problems involving **numeric input variables**
- Many problems are commonly observed at the boundaries

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



Source: *Google Images*

Boundary Value Analysis

- ***Values that occur at the boundary of an equivalence partition***
- If no equivalence partitions have been identified, think of boundary values as occurring at the edges of **the domain of *allowed* input**
  - often called **edge cases**
- **Derived**: from the spec; the size of a data type; and/or common sense
- Developers have both access to a **specification** and **knowledge** about the **ranges of the data types** they are using.
  - no excuses for not checking potential edge cases

Edge Cases

Strings:
empty string “”
null, nil, undefined
optionals
UTF-8 Character encodings?
string.length()

Integers: Input range —> m-n
check m – 1, m, n, and n + 1
may also try m + 1, and n – 1
value 0 might or might not be a boundary value, but it’s usually a good idea to investigate what happens around it

Collections / Objects:
empty collections
null, nil, undefined collections
memory allocation

Dates & Times: ...