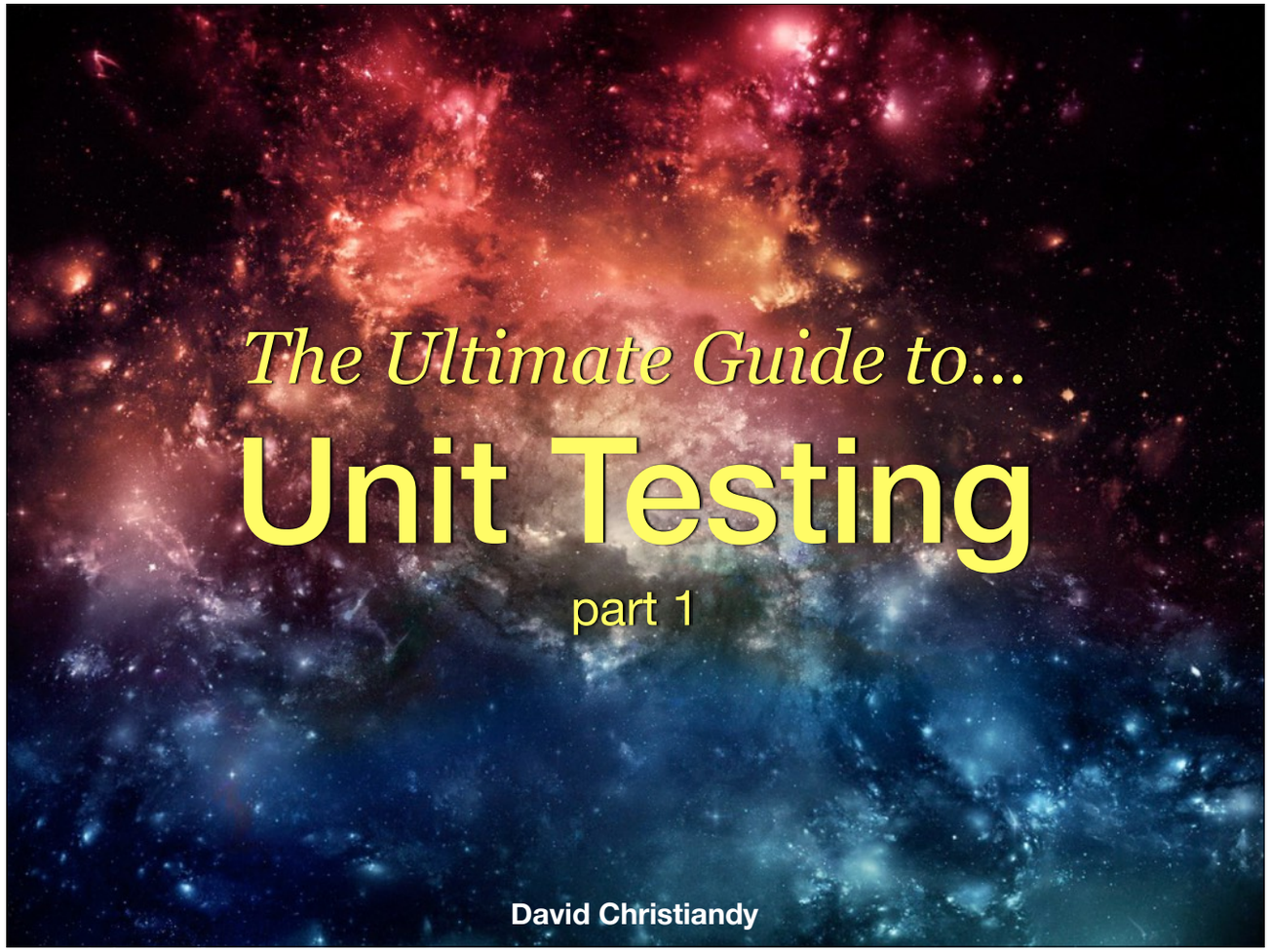

The Ultimate Guide to...
Unit Testing
part 1
David Christiandy

Why?

Why Unit Test?

- Maintain correctness
  - Automatic regression tests.

- Develop faster
  - No need to wait for simulators.
  - Move fast and break ~~things~~ unit tests.
  - Builds developer confidence.
  - Understand context faster by reading expectations/tests.

- Better, faster review
  - If code styling can be handled by linters, then code correctness can be handled by unit tests.
  - More time to focus on design decisions.
  - Verify expected business logic by reviewing the tests.

Principles

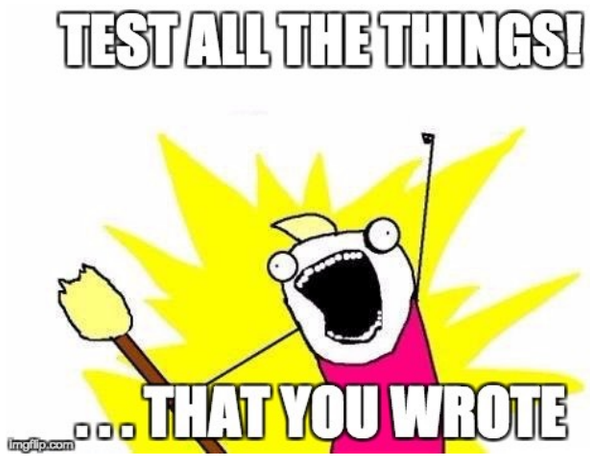
in Unit Testing

TODO (P2): Find a more suitable image.

Principle #1:
Everything you write is testable.

- That includes:

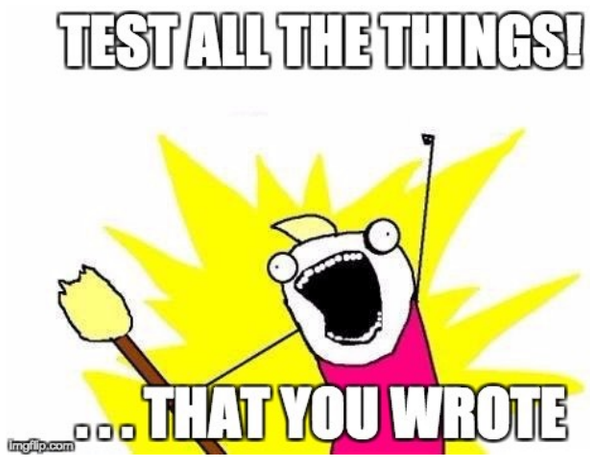
  Models,
  Helpers,
  View controllers,
  Custom objects,
  Library overrides,
  Categories,
  …and so on.



Principle #1:
Everything you write is testable.

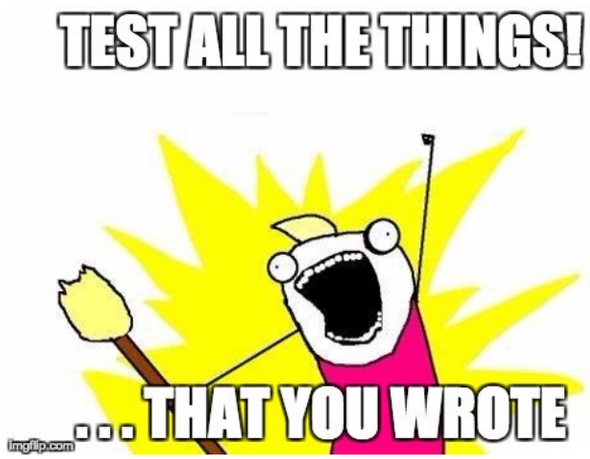
- Not including machine-generated files (plist files, etc.) and XIBs.

  Cover XIB files via UI test.



Principle #1:
Everything you write is testable.

- **Utilize 80/20 rule.**
  Prioritize which are the most important ones to test first.



Principle #2:
Don’t test other people’s code.

- Always assume that third-party and built-in libraries are **already covered by their owners.**

- Assumption works by mocking implementations of other people’s code.

  Example:
  No need to test whether a certain API returns the correct structure.

"I THINK PEOPLE SHOULD MIND THEIR OWN [redacted] BUSINESS."
- LIL WAYNE

You don’t need to write tests for code that you didn’t (or won’t) write.

Of course, most of the time our code intersects with other components. To write the proper tests for this, we should always assume that other people’s code will work as expected. One way to achieve this is via mock implementations.

Mocking will be explained in later slides.

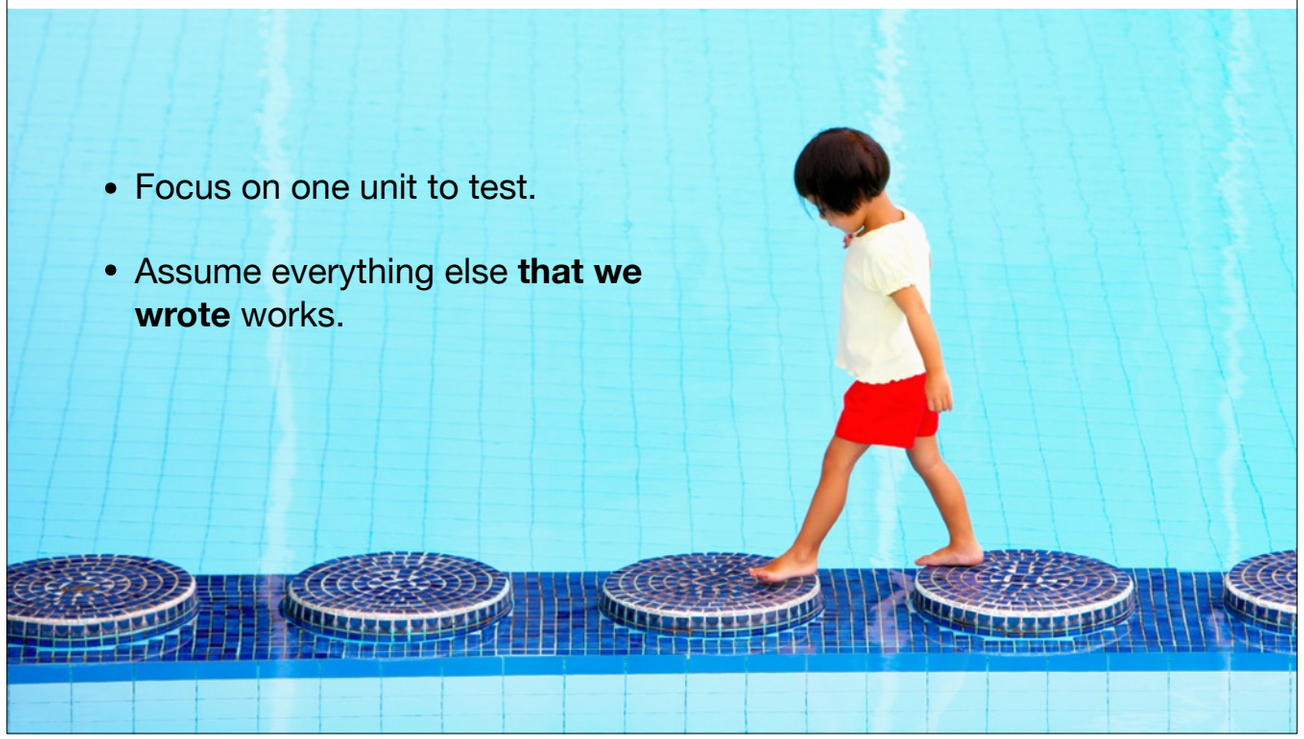
Principle #2:
Don’t test other people’s code.

- Code that you didn’t or won’t write.
- Code outside your team’s domain.
- Third-party libraries.
  including library that you wrote.
- System libraries.

"I THINK PEOPLE SHOULD MIND THEIR OWN [redacted] BUSINESS."
- LIL WAYNE

How do you know which code belongs to other people?

Principle #3:
Test one thing at a time.



Now that we've taken other people's code out of the way – it's time to focus on the code that we should write tests for.
When testing a class, assume that all of its dependencies just work.
Likewise, when testing a method that calls other method in the same class, assume that other method just works.

Note: The difference with previous principle: this focuses more about limiting the scope of thought when writing tests for the code that you actually wrote / will write.

Some approach to write unit tests

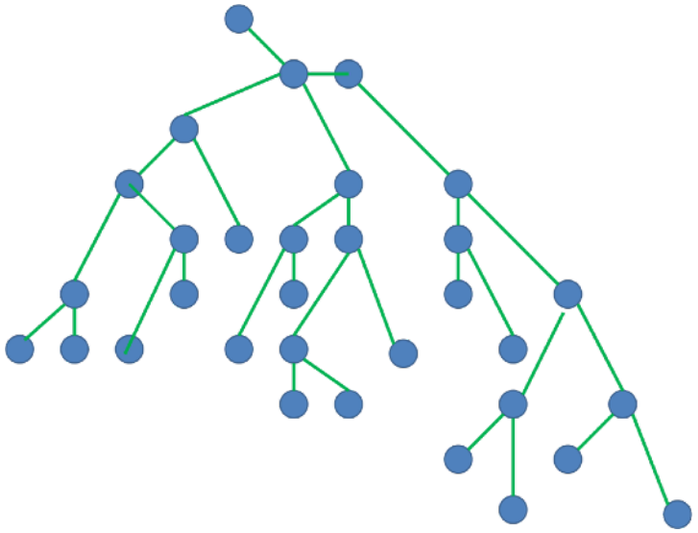
- There's no right or wrong approach.
- The end result should be the same: all components unit tested.
- Which one is easier to think in?

Bottom up: Write tests for smallest units first, and begin moving upwards.
Top down: Write tests for the highest units first, and begin moving downwards.

- There's no right or wrong here – it's all about which one is easier to think in.
- It all comes down to mindset, the end result should look the same: all components unit tested.

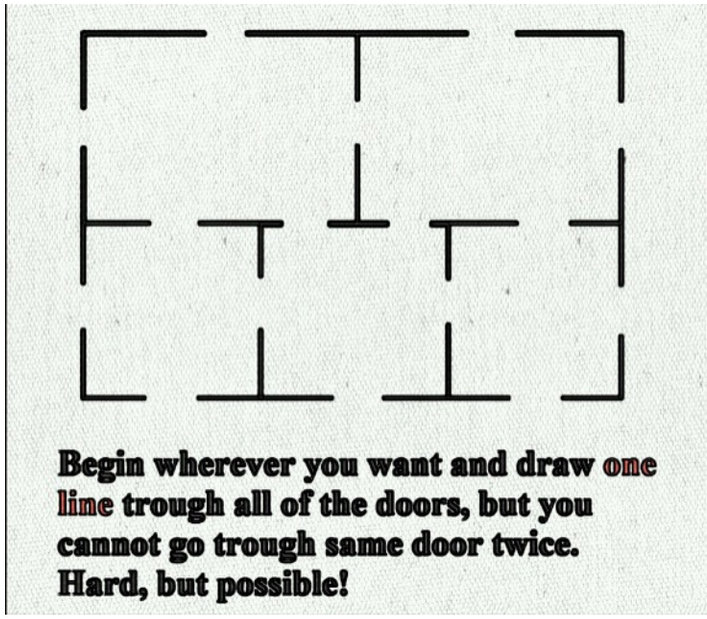
Principle #3:
Test one thing at a time.

- Focus on one unit, ignore all others.
- Assume each dependencies behave correctly.
- Everyone will eventually get their own turn.



Each node represents one unit. Ignoring its dependencies doesn't mean there's a hole in the unit test; the dependencies, if we wrote them ourselves, will have their own unit tests.

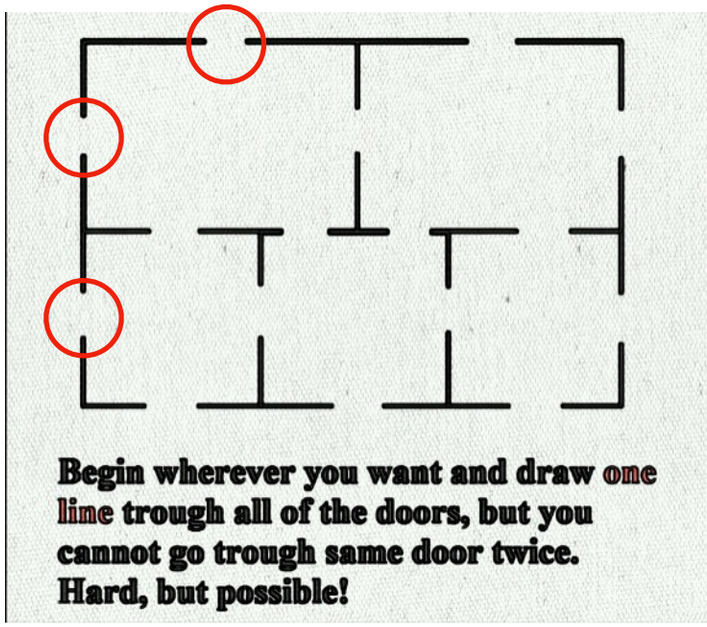
Principle #4:
Test each behavior once.



As analogy, here's a puzzle to better explain this principle.

Principle #4:
Test each behavior once.

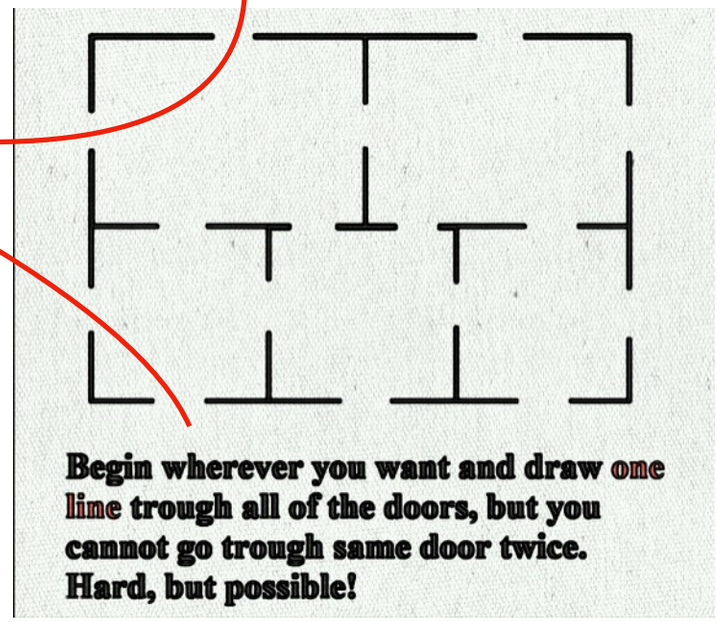
Behaviors /
Code Paths



The doors represent your behaviors (also known as code paths), but we'll get to that later. The objective of the puzzle is to pass each doors with a single line.

Principle #4:
Test each behavior once.

Your Unit Tests



The line represents the unit tests. Try to pass all the doors (behaviors) once. More than once is considered redundant (will discuss more on later slides).

Note: this puzzle is actually mathematically proven impossible, though!

Code Path Explained

- Code path represents a single, unique run.
- How many code paths does this method have?

```
- (NSString *)booleanToString:(BOOL)input {
    if (input) {
        return @"true";
    }
    else if (!input) {
        return @"false";
    }
    else {
        return @"nil";
    }
}
```

Now we're getting into code paths, or behaviors.
A code path essentially represents a single unique run. Can you guess the number of code path in this method?

Code Path Explained

- Number of code paths determine number of test cases.
- Craft your input to reach the desired code path.

```
[self booleanToString:YES];

- (NSString *)booleanToString:(BOOL)input {
    if (input) {
        return @"true";
    }
    else if (!input) {
        return @"false";
    }
    else {
        return @"nil";
    }
}
```

Example objective:

- Show what a code path is.
- Show how not to do redundant testing

Code Path Explained

- Number of code paths determine number of test cases.
- Craft your input to reach the desired code path.

```
[self booleanToString:NO];

- (NSString *)booleanToString:(BOOL)input {
    if (input) {
        return @"true";
    }
    else if (!input) {
        return @"false";
    }
    else {
        return @"nil";
    }
}
```

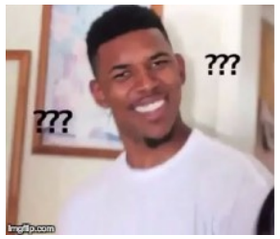
Example objective:

- Show what a code path is.
- Show how not to do redundant testing

Code Path Explained

- Helps us to think about our code.
- If code path is unreachable, **can we remove the code?**

```
- (NSString *)booleanToString:(BOOL)input {
    if (input) {
        return @"true";
    }
    else if (!input) {
        return @"false";
    }
    else {
        return @"nil";
    }
}
```



Of course, the example is oversimplified and the real case might not be as obvious. But how else would you realize a certain code path is unreachable unless you write a unit test for it?

Why do we need to keep a code that would never get executed?

Code Path Explained

- It's not just if/else.
- Method parameter adds to the code path variation.

```
- (BOOL)stringToBool:(NSString *)inputString {
    if (inputString.length > 5) {
        return YES;
    }
    else {
        return NO;
    }
}
```

```
[self stringToBool:@"Objective-C"];
[self stringToBool:@"Foo"];
[self stringToBool:nil];
```

How many code paths are in this method?

Redundancy in Unit Tests

- Tests are redundant if they go through exactly the same code path.
- Quality over quantity.

```
- (BOOL)stringToBool:(NSString *)inputString {
    if (inputString.length > 5) {
        return YES;
    }
    else {
        return NO;
    }
}

[self stringToBool:@“Traveloka"];
[self stringToBool:@"Objective-C"];
```

These 2 method calls go through exactly the same code paths!

Writing redundant test code wastes time on:

- Writing tests,
- Adjusting for behavior changes.

Principle #4:
Test each behavior once.

- Prioritize quality (code coverage %) over quantity (# of tests).
  Code Coverage = Tested LoC / Total LoC
- Do it once, do it right.
  Avoid writing redundant tests

To recap:

- Redundant tests have no benefits. Also, if your code changes, you'll also have to change all your redundant tests.

Principle #5:
Test for negative cases.

“Anything that can go wrong, *will* go wrong.”
Murphy’s Law

- Test possible negative cases (remember last principle!)
- Tighter code leads to less unit tests (maybe next slide).
- Refer to Rob’s talk on UIKonf 2017.

Principle #5:
Test for negative cases.

- Happy examples so far.

```
[self stringToBool:@"Objective-C"];
[self stringToBool:@"Foo"];
```

- We also need to pay attention to negative cases.

```
[self stringToBool:nil];
```

```
- (BOOL)stringToBool:(NSString *)inputString {
    if (inputString.length > 5) {
        return YES;
    }
    else {
        return NO;
    }
}
```

Most of the examples we've seen are considered "happy cases", although there's one example that tests negatively. Let's revisit the stringToBool example.

All of the tests are performing as *expected.*

Principle #5:
Test for negative cases.

```
+ (NSInteger)boolToInteger:(BOOL)input {
    return (NSInteger)input;
}
```

```
[self boolToInteger:YES];
[self boolToInteger:NO];
```

```
[self boolToInteger:(unsigned char)100];
```

objc.h

```
#if defined(__OBJC_BOOL_IS_BOOL)
    // Honor __OBJC_BOOL_IS_BOOL when available.
#   if __OBJC_BOOL_IS_BOOL
#       define OBJC_BOOL_IS_BOOL 1
#   else
#       define OBJC_BOOL_IS_BOOL 0
#   endif
#else
    // __OBJC_BOOL_IS_BOOL not set.
#   if TARGET_OS_OSX || (TARGET_OS_IOS && !__LP64__ && !__ARM_ARCH_7K)
#      define OBJC_BOOL_IS_BOOL 0
#   else
#      define OBJC_BOOL_IS_BOOL 1
#   endif
#endif
```

BOOL is signed char on everything, except 64-bit iOS and ARMv7k (watch).

Go deeper:
https://reviews.llvm.org/D26234
https://reviews.llvm.org/D28349
https://reviews.llvm.org/D29768

(Yes, LLVM uses Phabricator)

Let's try another one. Consider a very simple boolToInteger function. Maybe this is in a helper function somewhere.

There's no end to this since we can go deeper and try to look for limitations. But I hope you get the point.

Principle #5:
Test for negative cases.

- The rabbit hole goes deeper.
- Prioritize. Remember the 80/20 rule.

```
BOOL state = (BOOL)256;
if (state) {
    NSLog(@"YES");
}
else {
    NSLog(@"NO");
}
```

YES on 32-bit iOS,
NO on 64-bit iOS. 

Most of the examples we've seen are considered "happy cases", although there's one example that tests negatively. Let's revisit the stringToBool example.

All of the tests are performing as *expected.*

Principle #5:
Test for negative cases.

- Prefer nonnull over nullable.
- Avoid loose types.
- Be specific with your types.
- Be careful with your inputs!

Here's a good rule of thumb when writing negative cases:

- Prefer nonnull over nullable as it can reduce your number of test cases.
- Talking about id, Class, NSObject, NSDictionary (refer to Rob's talk on UIKonf 2016).
- If we can pass the object, why pass an NSDictionary?

Principle #6:
Mock things that takes time.

- We want unit tests to be as fast as possible.
- Things that should be mocked: network requests, database operations, file operations, UIViewController lifecycle, etc.

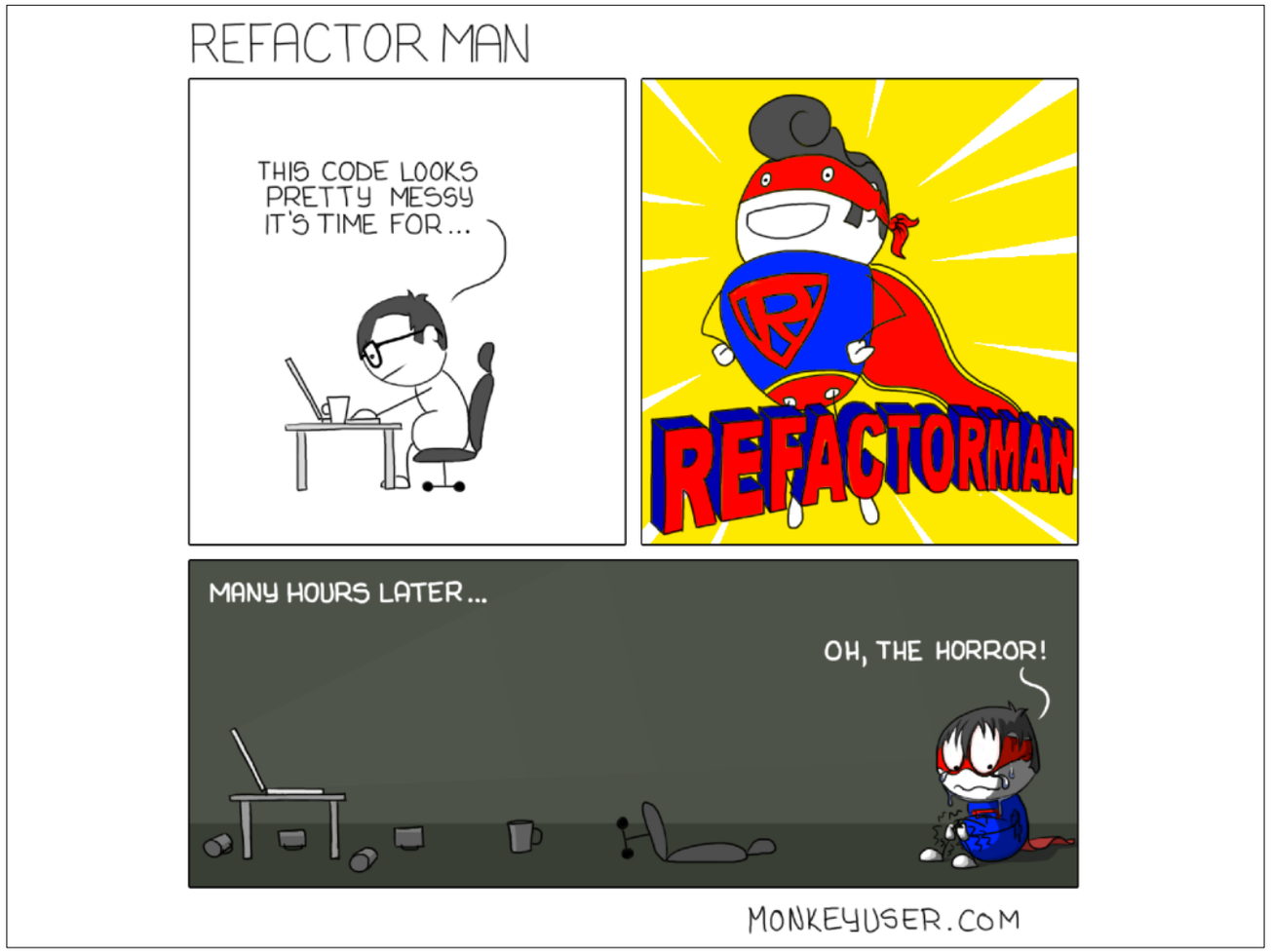
- Techniques used here should also be applicable to principle #2
- Cue the testing pyramid?

Principle #6:
Mock things that takes time.

```
+ (void)setLocalCurrency:(NSString *)currency {
    [[NSUserDefaults standardUserDefaults] setObject:currency forKey:PREFS_CURRENCY];
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

- No need to verify that the currency is actually saved in NSUserDefaults.

Simple method that saves to NSUserDefaults.


REFACTOR MAN
THIS CODE LOOKS PRETTY MESSY IT'S TIME FOR...
REFACTORMAN
MANY HOURS LATER...
OH, THE HORROR!
MONKEYUSER.COM

Principle #7: Refactor as needed

- Unit testing drives you to design loosely-coupled objects.
- Focus more on how components interact.
- The more bloated your class, the more painful it is to unit test.
- **Don't be Refactorman.** Be mindful of the amount you refactor, and don't tangle yourself!

To Recap:

- Everything you write is testable.
- Don’t test other people’s code.
- Test one thing at a time.
- Test each behavior once.
- Consider negative cases.
- Mock things that takes time.
- Refactor as needed.


Thanks!
David Christiandy