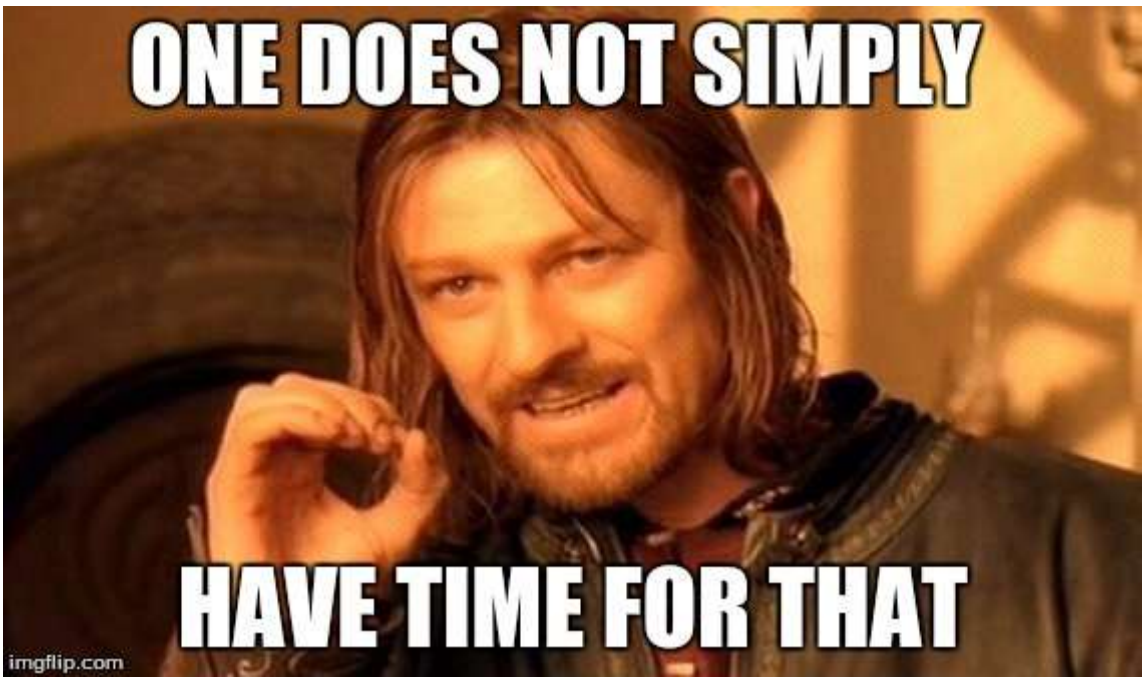
Understand Legacy Code

Change Messy Software Without Breaking It

3 steps to add tests on existing code when you have short deadlines

> *The code requires significant changes to support unit tests. I have deadlines to meet!*



You have that pile of Legacy Code you need to change.

Of course, there are no tests. Deep in your heart, you know that you should add tests before touching this code. People on the Internet told you so. “First, you should add tests” they said.

But you also have deadlines to meet. Time is flying. Maybe the project is already late. You can't afford spending *days* to write the tests that should be here in the first place.

BUT you know that you should. If you don't, you're making things worse! If you don't write the tests, you may even break something and you won't realize!!! 🙀

You want to add tests, but you don't have time to!

If that's you, I do have a solution for you!

It's not a pretty one, but *it works*™. It did save me many times when I was in a hurry. It can help you too.

The 3-steps recipe to add tests when you don't have time to 🚀

What you're looking for is called "*Approval Testing*". Some people call that "*Characterization Testing*".

Here's how it goes:

1. 📸 Generate an output you can snapshot
2. ✅ Use test coverage to find all input combinations
3. 👽 Use mutations to verify your snapshots

Follow that and you'll get your thing under tests *super fast*!

Let's see how you do that in detail.

1. 📸 GENERATE AN OUTPUT YOU CAN SNAPSHOT

That's the most difficult part. When you get that done, you're almost done.

You need to find a way to capture what the code is doing, in a serialized way. Put simply: it should produce some text you can capture.

Call the code from a test file. Provide the simplest inputs you need to get it running.

You can face 3 scenarios:

1. **What you're testing returns a value**. That's your output. Easy!
2. **What you're testing performs side-effects** (e.g. it calls an HTTP endpoint). You should intercept that and capture the parameters that are used. You'll probably need a spy/stub/mock to do so.
3. **What you're testing does both.** That's fine, you can test each side-effect and the return value separately**.**

Go figure out how to get that output. What you need is a string that proves the code has executed.

When you got that string, write it in a file. **That's your snapshot.**

Some testing libraries will give you utilities to do that for you. For example, in JavaScript, *Jest* has a guide to snapshot testing**.**

Here's an example of a snapshot test with Jest:

JS Copy

```
it("should update quality", () => {
  expect(updateQuality("foo", 0, 0)).toMatchSnapshot()
})
```

You get your snapshot? Sweet! You're almost done 👍

2. ☑ USE TEST COVERAGE TO FIND ALL INPUT COMBINATIONS

With your first snapshot test running, you're executing some of the code you want to test.

Probably not all of it.

That's where test coverage is useful: it will tell you **what you're not testing yet**.

Here's an example of a test coverage report:

All files gilded-rose.js

46.67% Statements 14/30 **25.71%** Branches 9/35 **100%** Functions 3/3 **46.67%** Lines 14/30

```
1    export class Item {
2      constructor(name, sellIn, quality) {
3  1x    this.name = name;
4  1x    this.sellIn = sellIn;
5  1x    this.quality = quality;
6      }
7    }
8
9    export class Shop {
10     constructor(items = []) {
11 1x    this.items = items;
12     }
13
14     updateQuality() {
15 1x    for (var i = 0; i < this.items.length; i++) {
16 1x      E if (
17           this.items[i].name != "Aged Brie" &&
18           this.items[i].name != "Backstage passes to a TAFKAL80ETC concert"
19         ) {
20 1x        I if (this.items[i].quality > 0) {
21             if (this.items[i].name != "Sulfuras, Hand of Ragnaros") {
22               this.items[i].quality = this.items[i].quality - 1;
23             }
24           }
25         } else {
26           if (this.items[i].quality < 50) {
27             this.items[i].quality = this.items[i].quality + 1;
28             if (
29               this.items[i].name == "Backstage passes to a TAFKAL80ETC concert"
30             ) {
31               if (this.items[i].sellIn < 11) {
32                 if (this.items[i].quality < 50) {
33                   this.items[i].quality = this.items[i].quality + 1;
34                 }
```

The red lines indicate the code that's not executed.

The "I" and "E" symbols indicate the code inside the "If" or "Else" branch is not executed.

Use this information to guide your next steps. **You want to cover all the code with tests.**

Remember I told you to put the simplest inputs to get your test running? Great.

Now write another test and change one input. Try to find a combination that will exercise a part of the code you're not covering.

This is easier than it sounds. Actually, it can be solved by trying random inputs, if you really have no idea. In practice, you'll certainly make informed guesses of what the next combination should be.

Repeat that until you cover every line.

All files gilded-rose.js

100% Statements 30/30 **68.57%** Branches 24/35 **100%** Functions 3/3 **100%** Lines 30/30

```
1  export class Item {
2    constructor(name, sellIn, quality) {
3  5x  this.name = name;
4  5x  this.sellIn = sellIn;
5  5x  this.quality = quality;
6    }
7  }
8
9  export class Shop {
10   constructor(items = []) {
11 5x  this.items = items;
12   }
13
14   updateQuality() {
15 5x  for (var i = 0; i < this.items.length; i++) {
16 5x  if (
17       this.items[i].name != "Aged Brie" &&
18       this.items[i].name != "Backstage passes to a TAFKAL80ETC concert"
19     ) {
20 3x  if (this.items[i].quality > 0) {
21 2x  E if (this.items[i].name != "Sulfuras, Hand of Ragnaros") {
22 2x  this.items[i].quality = this.items[i].quality - 1;
23       }
24     }
25   } else {
26 2x  E if (this.items[i].quality < 50) {
27 2x  this.items[i].quality = this.items[i].quality + 1;
28 2x  if (
29       this.items[i].name == "Backstage passes to a TAFKAL80ETC concert"
30       ) {
31 1x  E if (this.items[i].sellIn < 11) {
32 1x  E if (this.items[i].quality < 50) {
33 1x  this.items[i].quality = this.items[i].quality + 1;
34       }
```

You might end up with a test looking like that:

JS

Copy

```
it("should update quality", () => {
  expect(updateQuality("foo", 0, 0)).toMatchSnapshot()
  expect(updateQuality("foo", 0, 1)).toMatchSnapshot()
  expect(updateQuality("foo", 0, 2)).toMatchSnapshot()
  expect(updateQuality("Aged Brie", 0, 1)).toMatchSnapshot()
  expect(updateQuality("Aged Brie", 0, 50)).toMatchSnapshot()
  expect(updateQuality("Sulfuras", 0, 1)).toMatchSnapshot()
  expect(updateQuality("Sulfuras", -1, 1)).toMatchSnapshot()
})
```

If you're using Jest, you can use jest-extended-snapshot. It's a free library I created to simplify previous code into:

JS Copy

```
it("should update quality", () => {
  expect(updateQuality).toVerifyAllCombinations(
    ["foo", "Aged Brie", "Sulfuras"],
    [-1, 0],
    [0, 1, 2, 50]
  )
})
```

3. 👽 USE MUTATIONS TO VERIFY YOUR SNAPSHOTS

Everything is now covered with snapshots.

But 100% test coverage doesn't mean you actually test the code.

There's a simple way to verify you're safe: introduce mutations!

And by "introduce mutations" I really mean:

deliberately change each line of code to introduce a silly mistake (I like to comment out the code)

verify a test is failing

revert the silly mistake

celebrate internally (yay!)

For every line you mutate and see a failing snapshot, your confidence in the tests grows.

If no test fails, you need to add other input combinations. Revert the silly mistake, find the correct combination that will exercise this line and try again. You want to see a failing test.

When you reach the end of the code, you're done!

Why is it the fastest technique to add tests?

Because you don't have to write comprehensive unit tests of the existing code.

Instead, you take it as a black box. You execute the code, whatever it does. And you capture the output.

It's the fastest way to put regression tests on existing code.

With this technique, I already put monstrous lumps of code under tests within a couple of hours.

Why don't we *always* use that kind of test?

As sexy as this approach is, it has downsides:

- you capture existing behavior, bugs included
- tests will fail whenever you change the behavior, even if it's intended
- you can't read the tests and understand what the code does

These kind of tests can be really noisy. If you notice people just update them when they fail, they don't provide any value. **Delete them.**

It's fine to delete useless tests afterwards. They were useful when you had to change the code. They're not here to replace proper tests in your codebase!

That's my secret weapon to add tests when we're in a hurry. That's a pragmatic compromise. But it's a temporary solution until we have time to write better, helpful tests on the code.

It's your secret weapon too now. Use it wisely!

I've got your back! Every Wednesday I share **practical tips** to help people work with Legacy Code. Get 'em for FREE.

Your first name

Your email address

I want these, get me on the list!

No spam. No sharing of your email. Unsubscribe at any time.

Kudos!

Written by **Nicolas Carlo** who lives and works in Montreal, Canada
He founded the Software Crafters Montreal community which cares about building maintainable softwares.

Similar articles that will help you...

Approval Tests, TCR, and Menders

5 great talks on Legacy Code that I the pleasure to host. Learn how to test existing code and approach unfamiliar codebases.

7 techniques to regain control of your Legacy codebase

Working with Legacy Code is no fun… unless you know how to approach it and get your changes done, safely.

A quick way to add tests when code has database or HTTP calls

You need to fix a bug, but you can't spend a week on it? Here's a technique to isolate problematic side-effects and write tests within minutes.

5 coding exercises to practice refactoring Legacy Code

Feeling overwhelmed by your legacy codebase? These katas will help you learn how to tackle it.

← Find more tips to work with Legacy Code