Testability in .NET

Jason Bock

Personal Info

- http://www.jasonbock.net
- https://mstdn.social/@jasonbock
- https://www.github.com/jasonbock
- jason.r.bock@outlook.com

Downloads

https://github.com/JasonBock/TestabilityInDotNet
https://github.com/JasonBock/Presentations

Overview

- Motivation
- Benefits
- Techniques
- Conclusion

Remember…
https://github.com/JasonBock/TestabilityInDotNet
https://github.com/JasonBock/Presentations/

Motivation

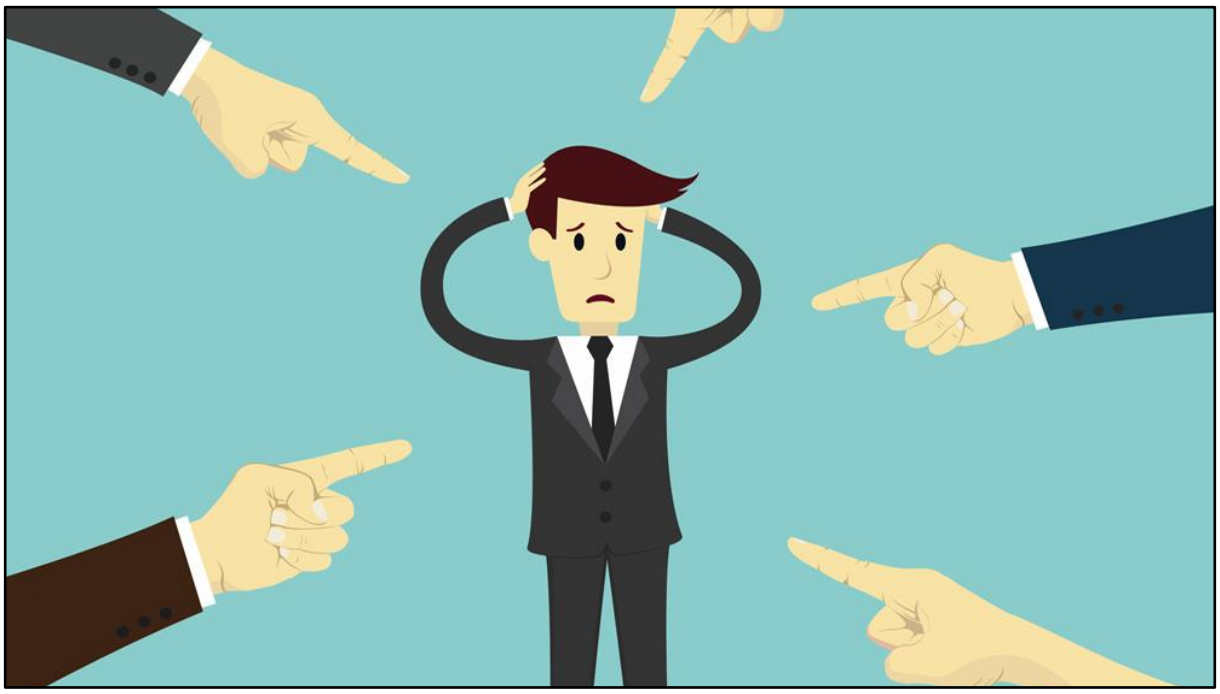
Testability in .NET

What we want is something that works as expected, and is tested to last for a long time. That's where testing comes into play. Making sure you code does what you think it should do makes it easier to maintain as the application changes (as it inevitably will).

https://www.hdwallpaperspulse.com/pearl-bridge-japan.html

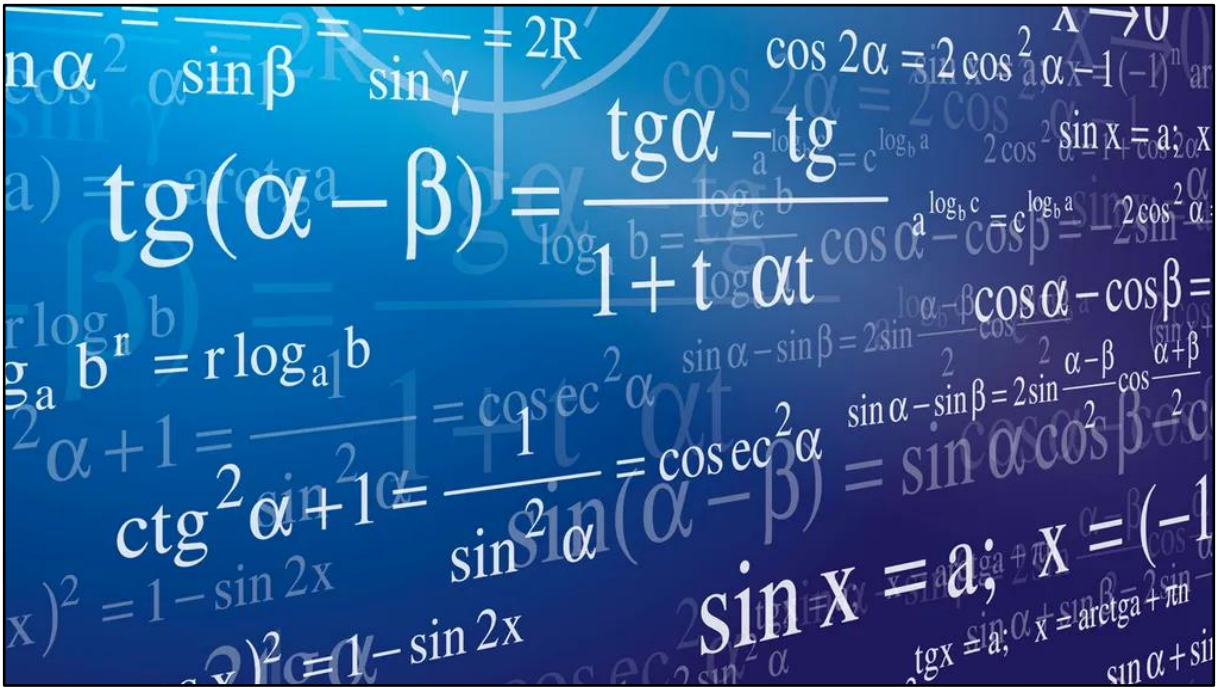
```
public static class Adding
{
  public static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Take a look at this innocent code.



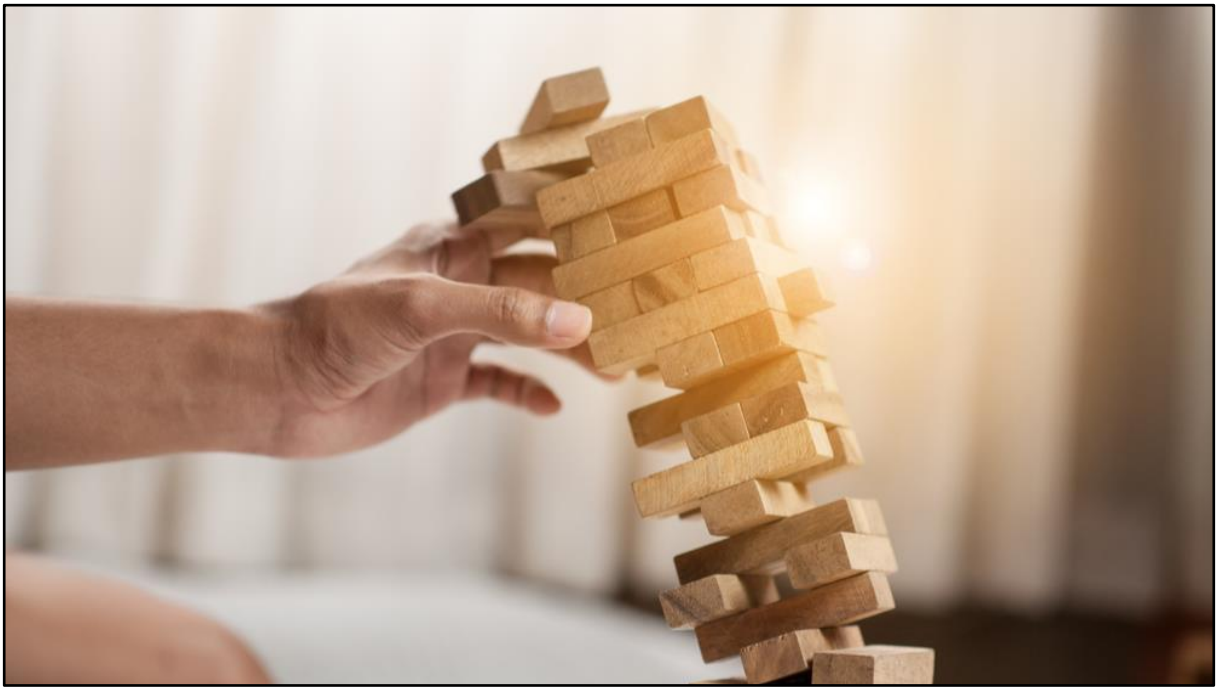
However, other developers will rip it apart (and sometimes their points aren't even valid, like the generics one in this list)

There's no interface!
It's not using generics!
You only take two parameters!
Your naming conventions are horrible!
The performance stinks!
You're not using Code Contracts!
You forgot about overflows!
You shouldn't be allowed near a compiler!



We'd like to be geniuses with our code – we always write perfect code with no issues

https://www.makeuseof.com/tag/solve-math-with-bing/



But it’s easy for us to do things in code that can lead to disasters. (Think of the Zune leap year bug)

https://smallbiztrends.com/2018/04/startup-failure-reasons.html

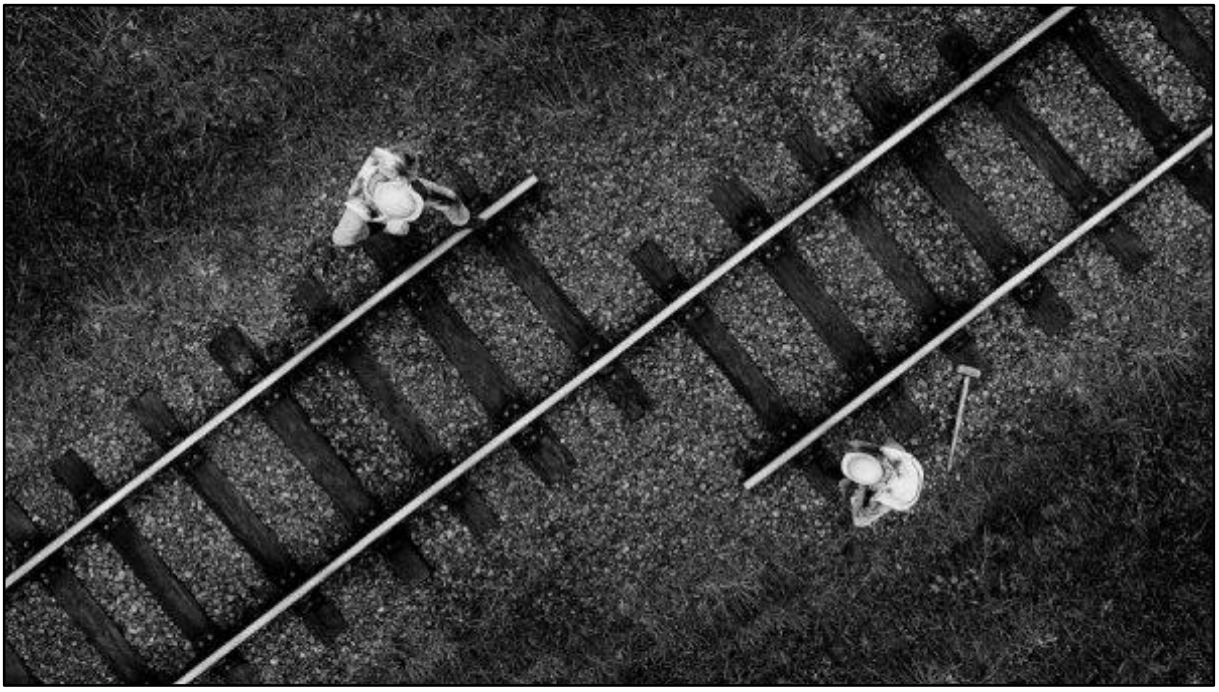
```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
  if (IsLeapYear(year))
  {
    if (days > 366)
    {
      days -= 366;
      year += 1;
    }
  }
  else
  {
    days -= 365;
    year += 1;
  }
}
```

"A bug in the internal clock driver related to the way the device handles a leap year affected Zune users," said the company in a statement. “That being the case, **the issue should be resolved over the next 24 hours as the time change moves to January 1, 2009**."

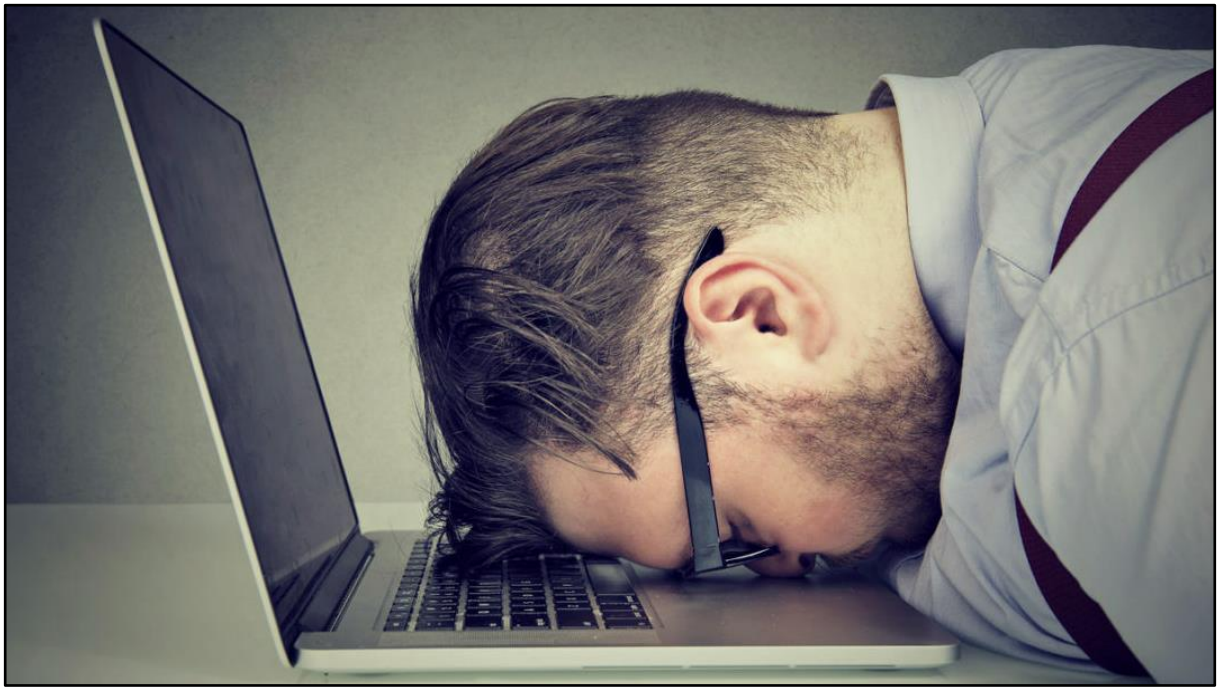
This code is from the Zune. Do you think you could’ve caught the error at first glance?

https://www.hanselman.com/blog/back-to-basics-explore-the-edge-cases-or-date-math-will-get-you
https://www.wired.com/2008/12/zune-freeze-res/



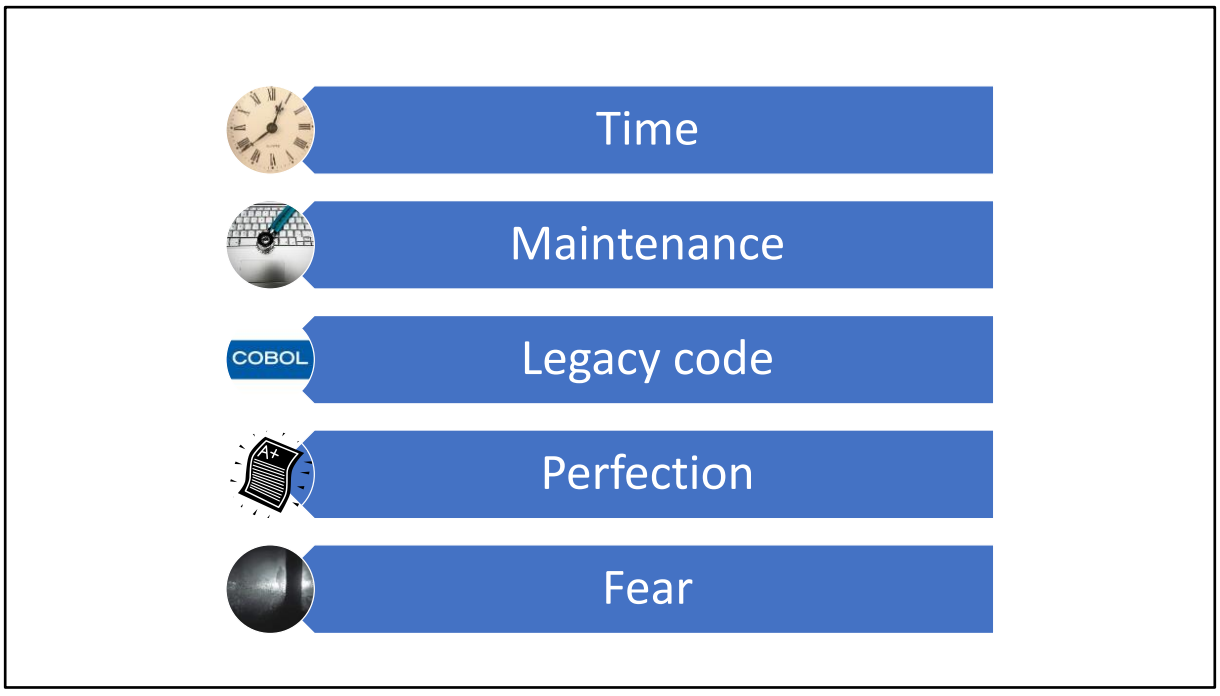
And when we do bad things in code, we end up with something that just don’t work

https://www.heinzmarketing.com/2016/09/why-does-misalignment-exist-between-sales-and-marketing/



And then we end up with frustrated users

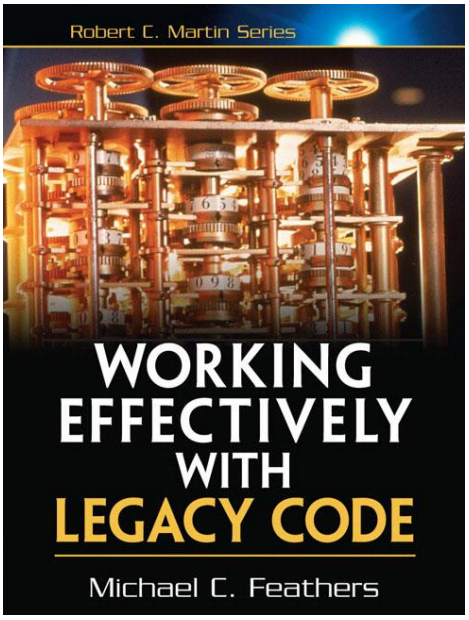
https://dejanmarketing.com/interruptions/



Why don't we test?

Time – it does take more time to write the tests. However, the benefit is you'll (hopefully) catch more defects in your code
Maintenance – if we change our code, we might need to change our tests
Legacy Code – it's too hard to add tests in to our code base. Everything is tied together, and it's too much effort to put tests into place ("Working Effectively with Legacy Code")
Perfection – "I write best code" – if code is written exactly to specs, then why write unit tests at all? (ha!)
Fear – "I've never done this, how long is it going to take for me to get proficient, etc."

In the next set of slides, we'll cover these issues



"Legacy code is simply code without tests." –xvi

So why should you test? You don't want legacy code.

http://www.informit.com/store/working-effectively-with-legacy-code-9780131177055

Benefits

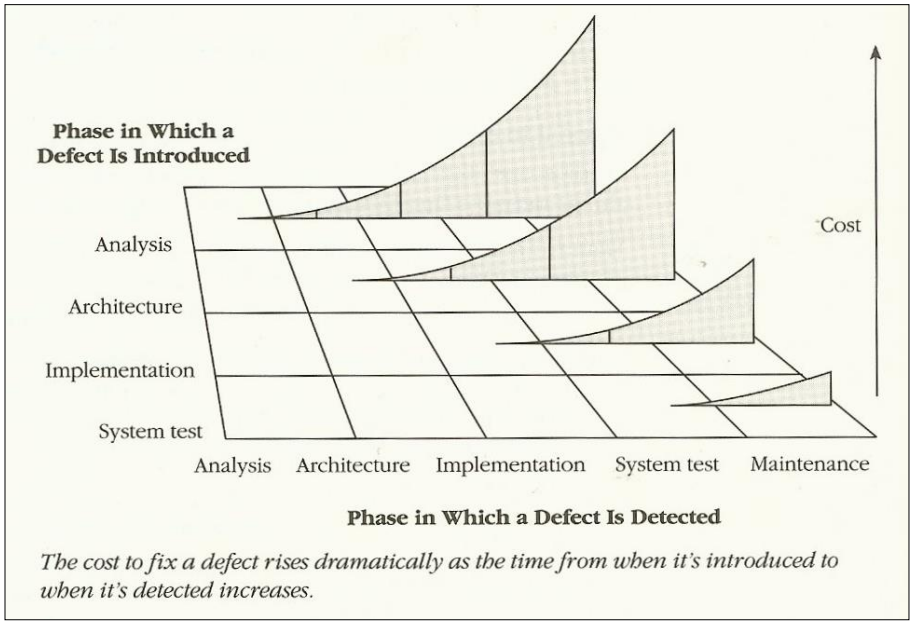
Testability in .NET

Unit testing makes developers dogfood their code. Sometimes that interaction with the API is quite revealing.

https://unsplash.com/photos/QIbjKOsmIhQ

Developers also want immediate feedback. Turning that feedback up helps in finding problem spots.

https://unsplash.com/photos/GBGcwLwpDCM



The cost to fix a defect rises dramatically as the time from when it's introduced to when it's detected increases.

"Fail fast" – the sooner you fail, the better off you are. "1 hand clapping" vs. thunderous applause

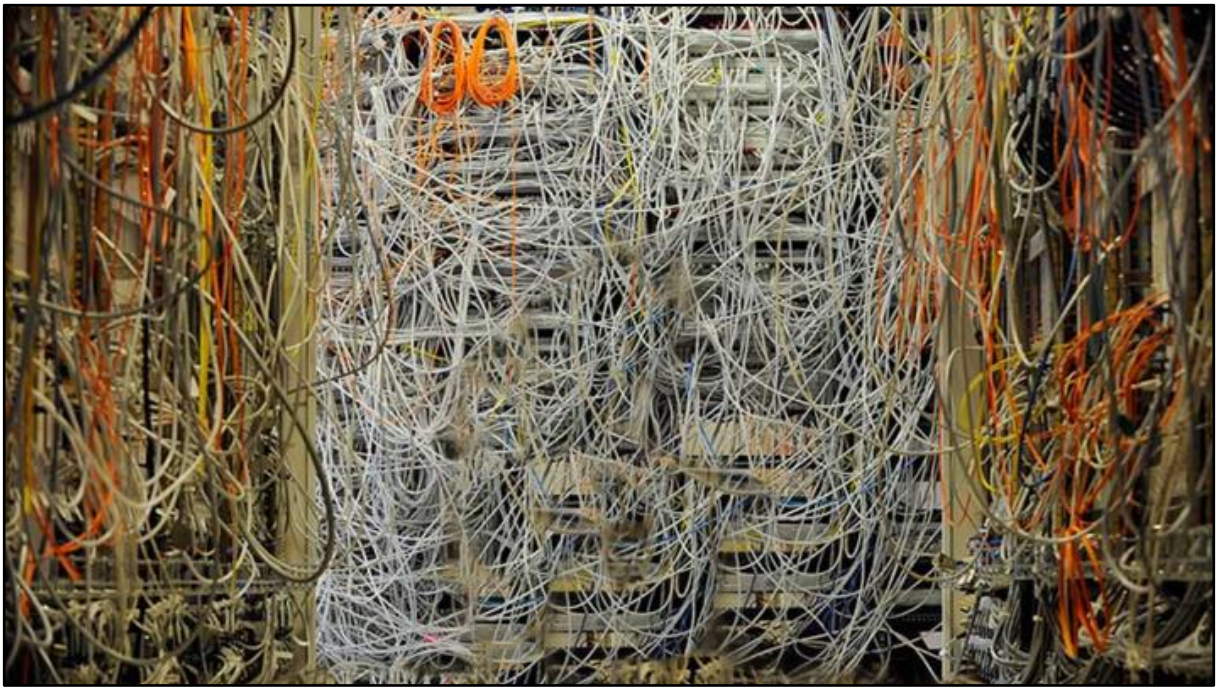
Code Complete

We need clear separations with our layers to make it easier to test our code, no matter how many layers we have

https://d2k9njawademcf.cloudfront.net/indeximages/3794/original/Chocolate_cake_-_for_posting.jpg?1380054214

A big block of chocolate may be tasty, but that’s not what we want in code, because we can’t easily unit test it

https://scratch.typepad.com/.a/6a00e551da83098833012875abba3b970c-800wi

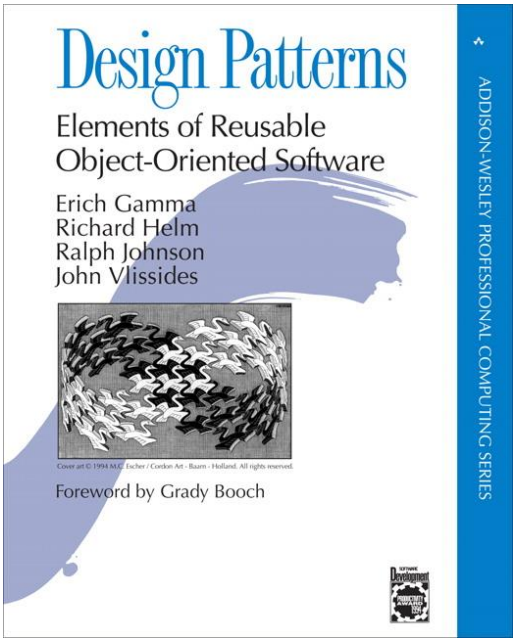


We don’t want a huge mess of interactivity within our application layers. It becomes too hard to test and maintain.

https://leaprofessional.com/blog/cable-management-tips/

Code in isolation, like a sound room. Removing noise helps you focus on the code under test.

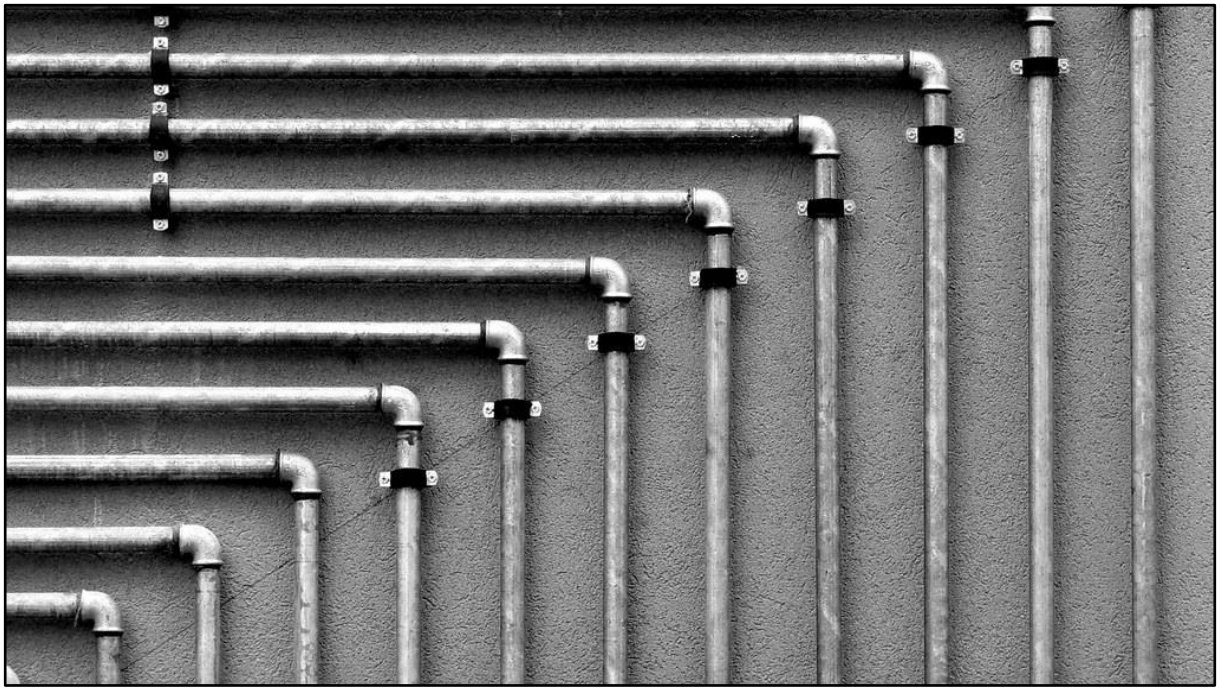
https://news.microsoft.com/stories/building87/audio-lab.php



"Program to an interface, not an implementation" – pg. 18

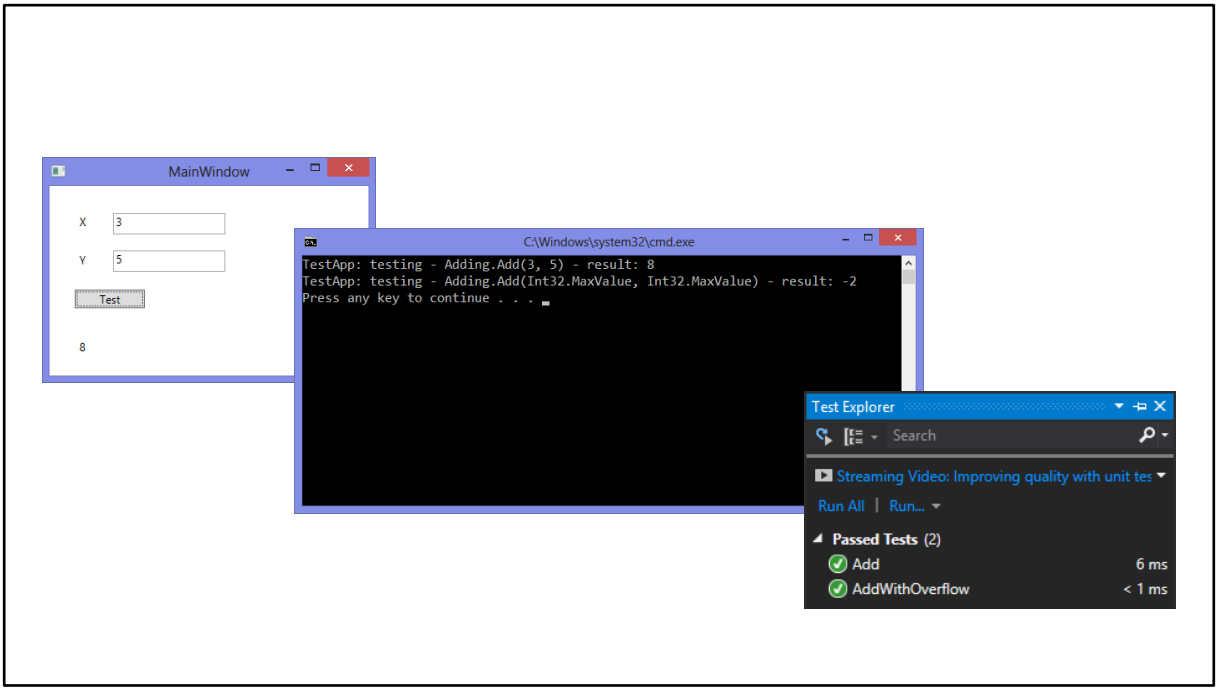
One of the best quotes in Design Patterns

https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-Oriented-Software/PGM14333.html



This can also help if you run tests in parallel. By isolating tests and getting rid of shared state, you can run tests in parallel and not worry about inconsistent failed tests.

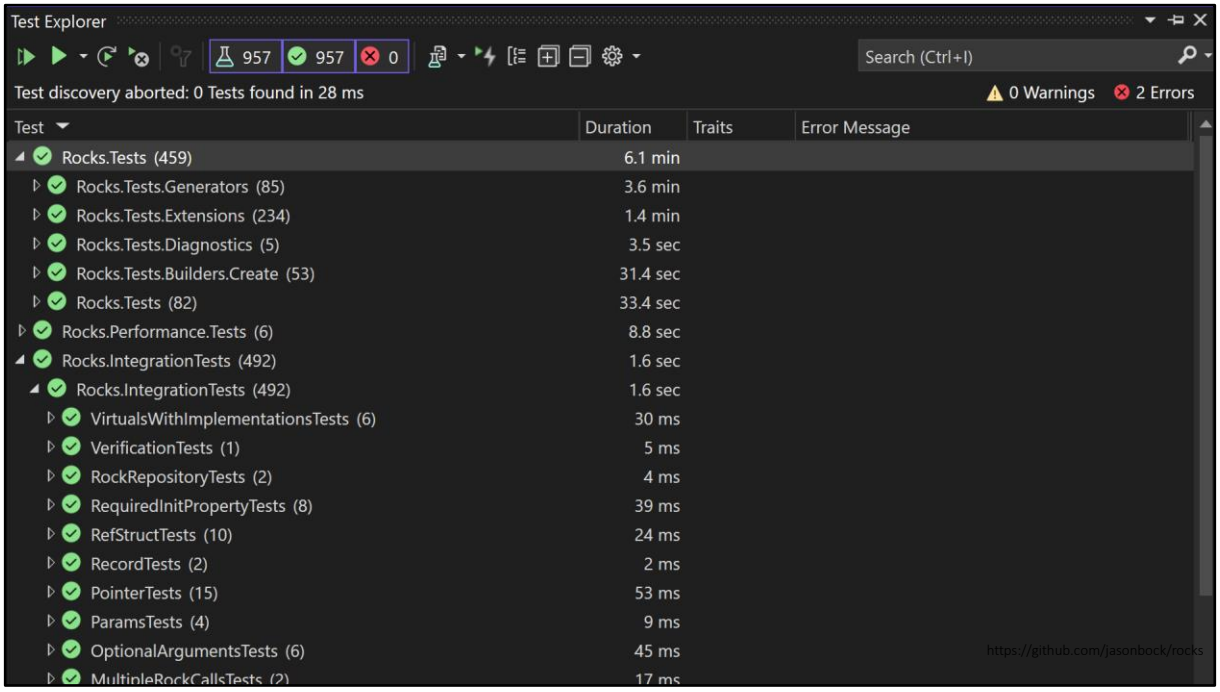
https://www.flickr.com/photos/63477821@N00/2251193780/



You'll always want to do end-to-end testing as a developer, but that's not easily repeatable. Writing apps that test code in a repeatable fashion is a better approach. Using a framework to easily run tests locally and on a build server is the way to go.

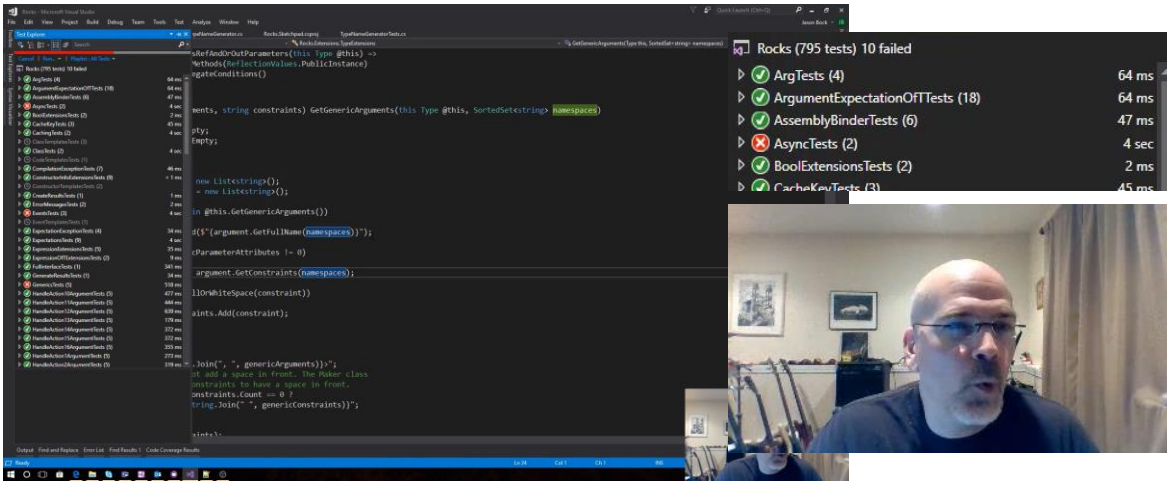
And having tests that run automatically helps with change. Change is inevitable in code. Virtually all the applications I've coded on that have had any life to them will go through change. Features will be added, bugs will be fixed, UIs will be altered – it happens. As developers, to have any hope of assurance that things work as expected is to have a suite of tests that I can fire off and inform me where things are at.

https://jmanandmillerbug.com/wp-content/uploads/2012/02/change.jpg



Here's my own experience. My Rocks library has a fair amount of tests.

https://github.com/jasonbock/rocks



I'm doing some live coding, and one time I made a change and unexpectedly broke tests. You can see my "oh crap!" face show up!

https://youtu.be/XJw-E2QQr-I?t=1581

Having this coverage against change is crucial, which is why you shouldn’t delete tests that are failing.

"Commenting out unit tests is the software equivalent of taking the batteries out of a smoke detector." - https://twitter.com/pottereric/status/1172146329105162240

https://i.ytimg.com/vi/FakT1rgcmrw/maxresdefault.jpg

Techniques

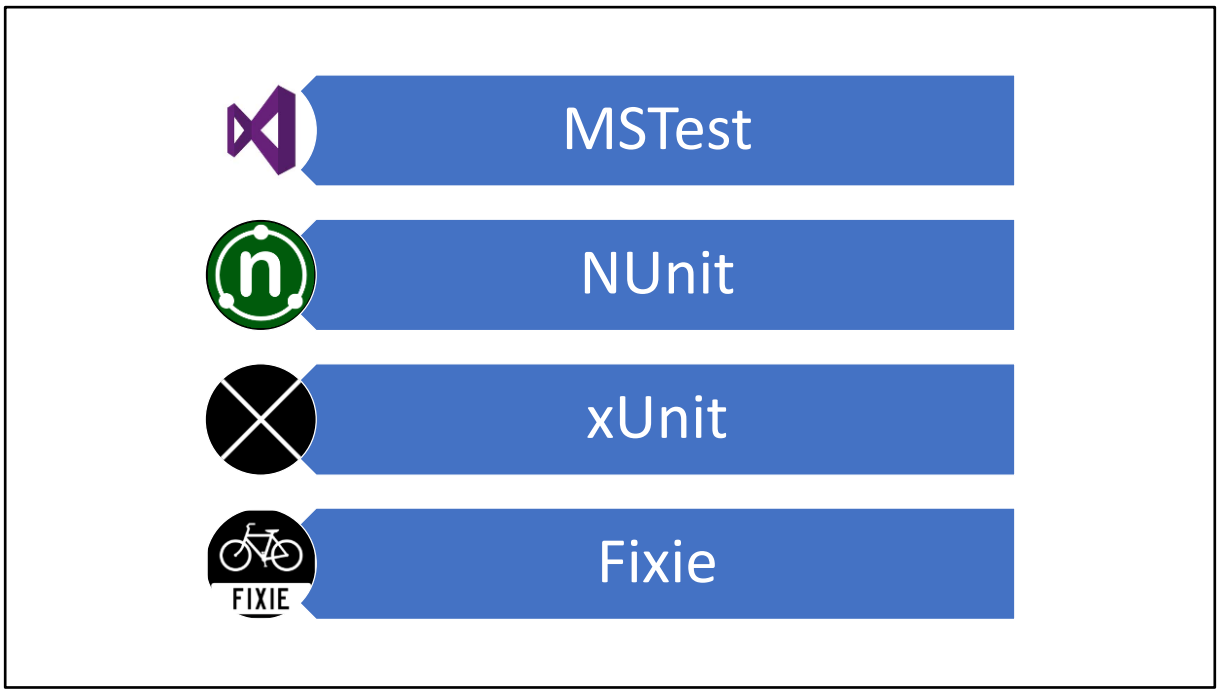
Testability in .NET

TDD,
BDD,
*DD, etc.

Many different techniques and ideas exist for unit testing, and they all have something to bring to the table

Arrange
Act
Assert

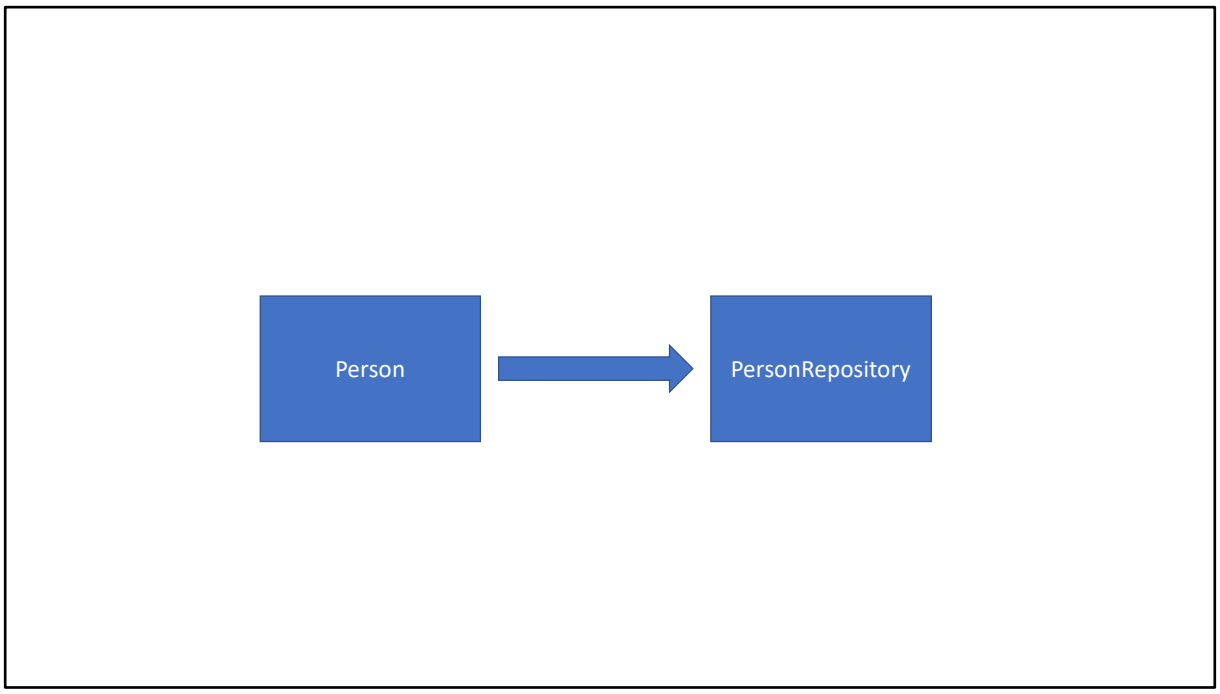
AAA: good approach to structuring unit testing code



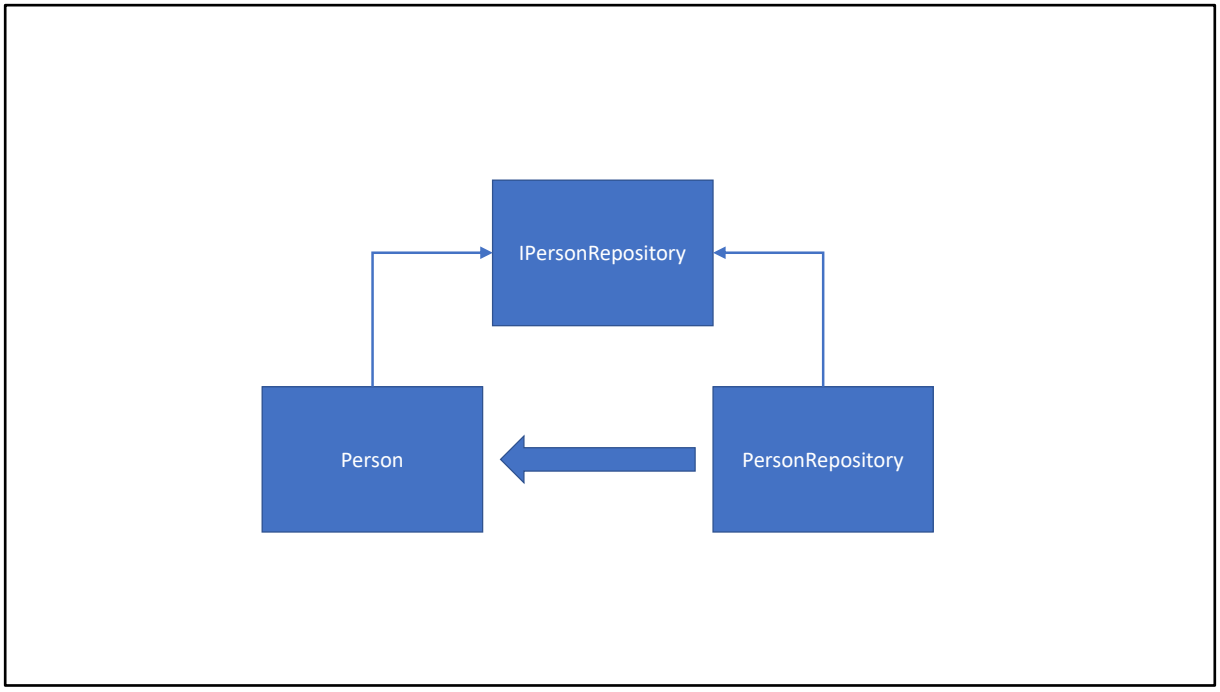
Here's a list of unit testing libraries

Using mocking libraries help in isolating code under test. Think of the purpose of a spare tire.

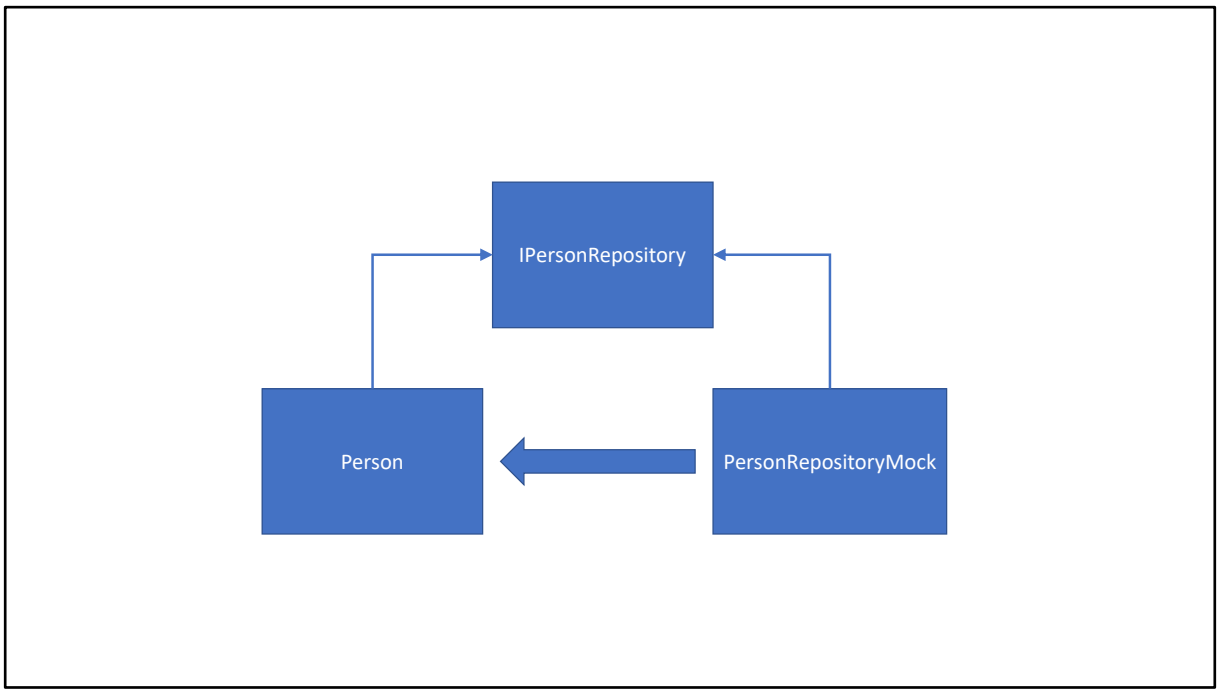
https://www.pexels.com/photo/spare-tire-near-a-car-door-9996404/



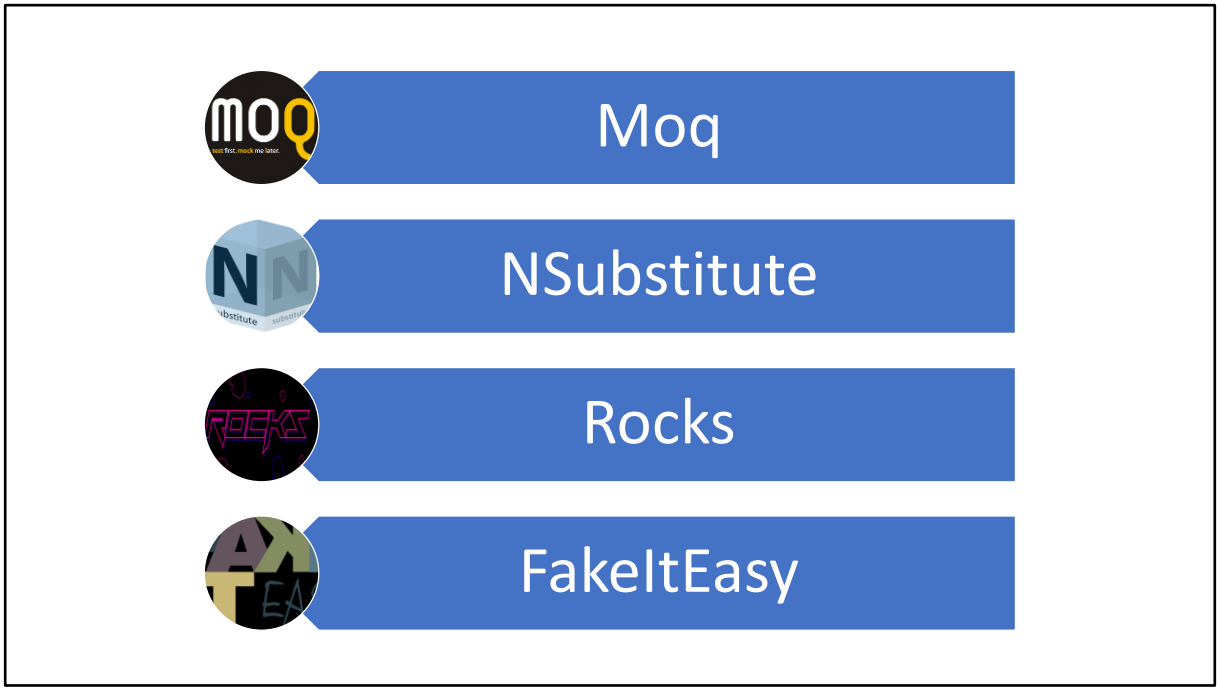
Here’s how it works. Let’s say Person has a reference to PersonRepository. This is strong coupling and can make testing difficult.



The way to fix this is to create an abstraction that Person references and PersonRepository implements. Then an instance of PersonRepository is injected to Person.



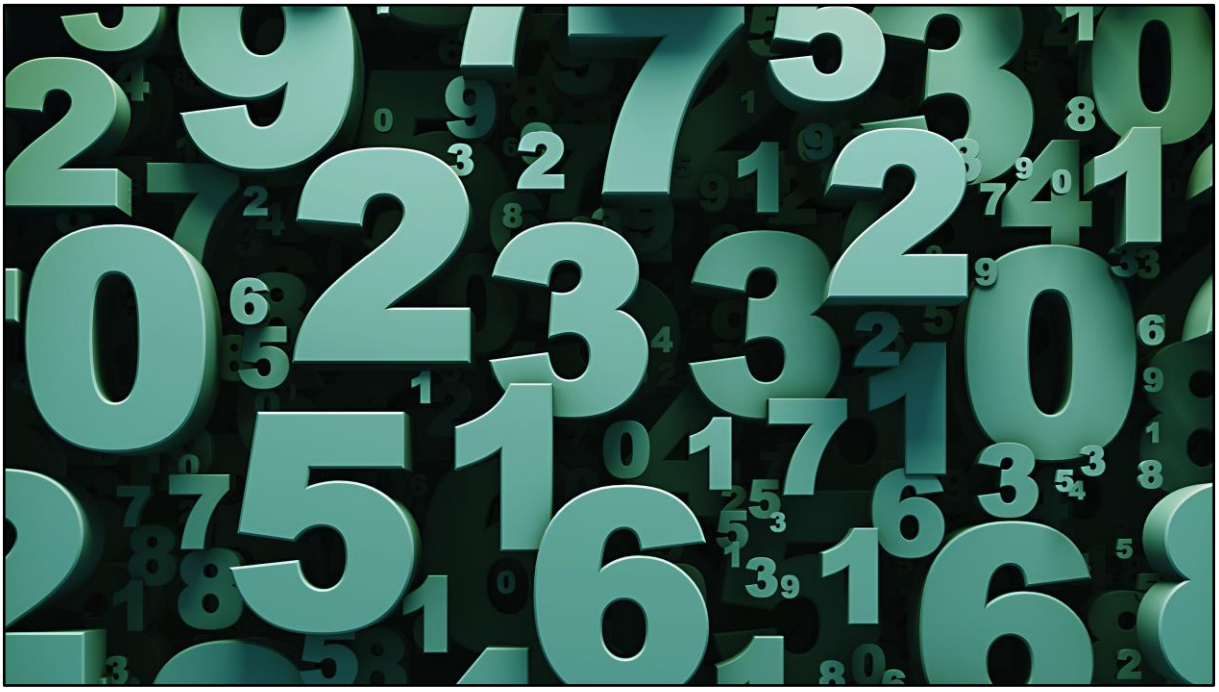
In a test, you can create a mock of PersonRespository, either by hand or using a framework, that has a specific setup that you can verify in a test that it was used correctly.



And here's some mocking libraries

Code coverage is a good metric to use as long as you understand what it's measuring. It simply informs you of code that hasn't been executed during a test. It's not a measure of test quality. Mutation testing is an area to look at.

https://unsplash.com/photos/PqqJ340jrow



Also…."how many tests should we have?" gets asked too. It's really not about many you have, it's about how effective they are. That said, in general, the more code you have, the "more" tests you should have as well, but IMO there is no hard and fast rule.

https://origin.pymnts.com/wp-content/uploads/2014/09/numbers2.jpg

Demo: Writing Tests in .NET

Testability in .NET

Conclusion

Testability in .NET

Making code maintainable is an investment in time (and money). It may seem daunting, especially if you haven't used the techniques and tools used in the presentation. But if you're willing to invest, your application will be looking down at the others!

https://unsplash.com/photos/cqbLg3lZEpk

But if you're willing to invest, your application will be looking down at the others!

https://unsplash.com/photos/J7fxkhtOqt0

Testability in .NET

Jason Bock

Remember…

- https://github.com/JasonBock/TestabilityInDotNet
- https://github.com/JasonBock/Presentations
- References in the notes on this slide

References

* Guidelines for Better Unit Tests - https://www.infoq.com/news/2009/07/Better-Unit-Tests
* Test Experience Improvements - https://blogs.msdn.microsoft.com/visualstudio/2017/11/16/test-experience-improvements/
* Should Code Review Include Manual Testing? Depends! - https://exceptionnotfound.net/should-code-review-include-manual-testing-depends-2/
* Unit Testing: Basics and Best Practices - https://www.daedtech.com/unit-testing-basics-best-practices/
* How Much Unit Testing is Enough? - https://www.daedtech.com/unit-testing-enough/
* Common Excuses Why Developers Don't Test Their Software - https://hackernoon.com/common-excuses-why-developers-dont-test-their-software-908a465e122c