cognizant softvision

Implementing testing pyramid

by Benas Radzevičius

Speed ❤ Quality

Counter intuitive finding - there's no tradeoff between speed and quality

- In order to get **quality** - you need to move quickly in small steps so that we can see the changes, evaluate and understand them
- In order to get **speed** - you need quality, so that when we do make a change we are not chasing defects

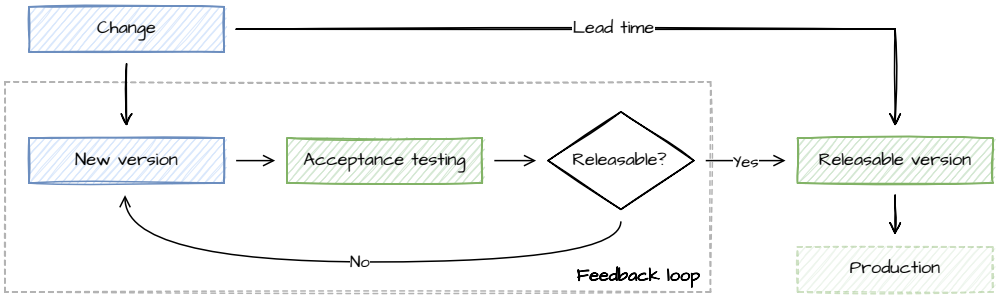
> If we go slower, we build worse software slower, not better software slower
>
> - *Dave Farley*

State of DevOps research (2021)

https://services.google.com/fh/files/misc/state-of-devops-2021.pdf

Continuous Delivery

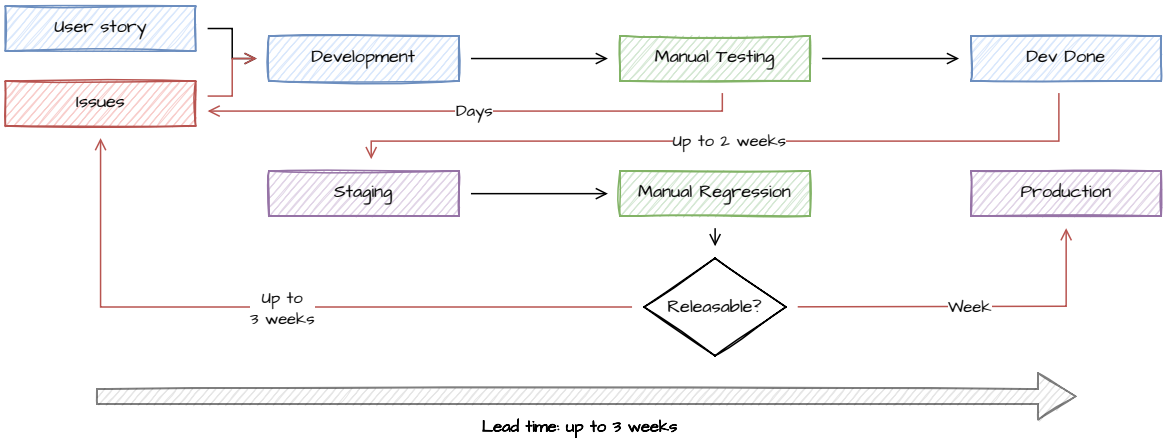
Maintaining software in a releasable state



Let's get on a journey

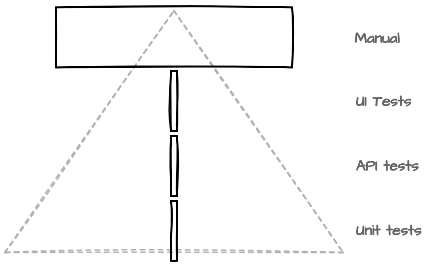
Starting_point

"Thor's hammer"



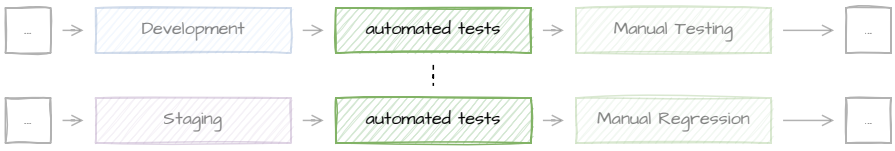
Long lead time creates a lot of inefficiencies:

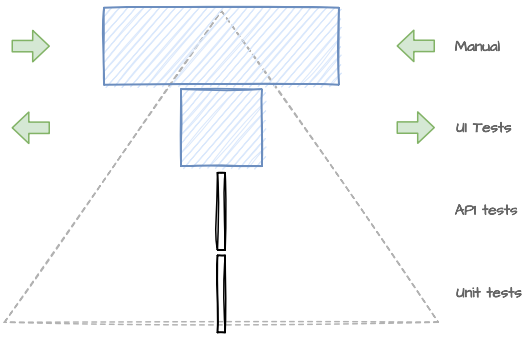
- Hard to triangulate cause of found issues
- Working on multiple sprints at the same time - context switching
- Stress
- etc...



Automation

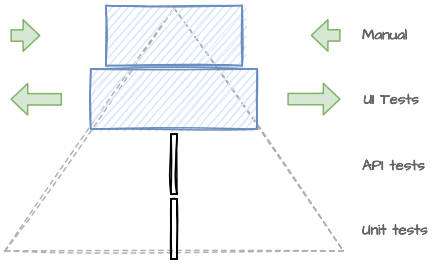
Automating regression through e2e tests



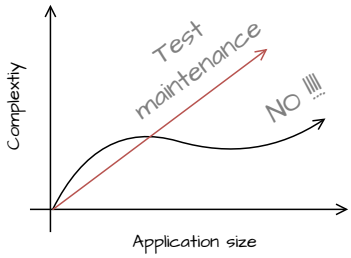


Regression times improved
Faster feedback

But... With more unit tests, we started experiencing issues...



Slow deployments
Hard to maintain
False positives



Great tests are:

Deterministic

No matter the circumstances, test should return the same result every time

Concise and simple

So it's not an overhead to write

Durable

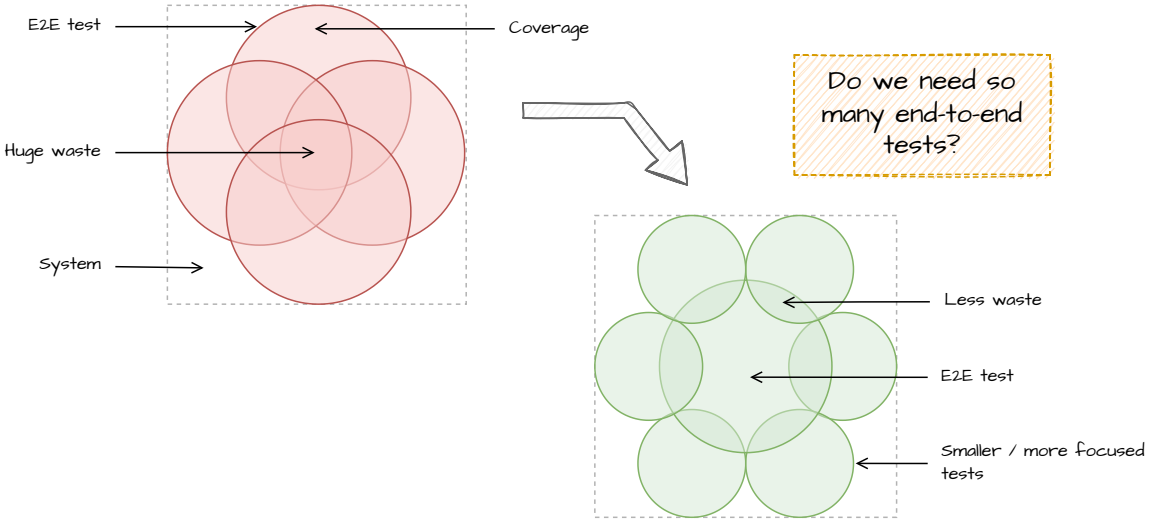
So that tests do not broken easily by changes in the system

Fast

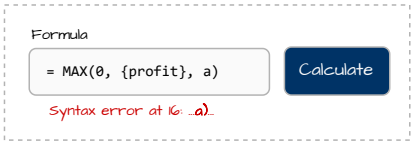
So we could could know faster when something is wrong and focus on the fix

Optimization - shift left

Focusing our testing and reducing waste



Example



Test cases

1. MAX(5, 1) = 5
2. MAX(-5, 1) = 1
3. MAX(1, 1) = 1
4. MAX(1, 5) = 5
5. MAX(-1, -5) = -1
6. MAX() = ERROR at 0
7. MAX(1) = ERROR at 0
8. MAX(a) = ERROR at 4
9. MAX(1, 2, 3, 4) = NO ERROR, ALLOWS MORE THAN 2 ARGS
10. Given {var 1} = 3, then MAX({var 1}, 2) = 3
11. MAX(1+1, 1) = 1
12. 1 + MAX(1, 2) = 3
13. MAX(1, 2) + 1 = 3
14. 1 + MAX(1, 2) + 1 = 4
15. 1 + MAX(a, 2) = ERROR at 8

Tested by

Test 1
1. Login as "admin"
2. Open a calculator
3. Enter a formula "MAX(1, 2)"
4. Result should be "2"

Test 2
1. Login as "admin"
2. Open a calculator
3. Enter a formula "MAX(a, 1)"
4. I see an error "Syntax error at: ...a, 1)..."

...

Test 15
...

Covers

Coverage

1. Authentication works
2. User is able to navigate to the calculator via navigation bar
3. Input allows to enter formulas
4. Submit button correctly sends a request to the backend
5. Error from the backend is shown under the input
6. Syntax error position is bolded

Same thing tested multiple times

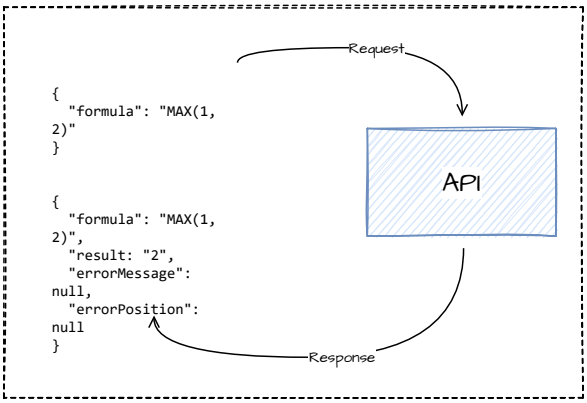
... other examples

1. Syntax error position correctly identified
2. MAX correctly calculates the result
3. Etc...

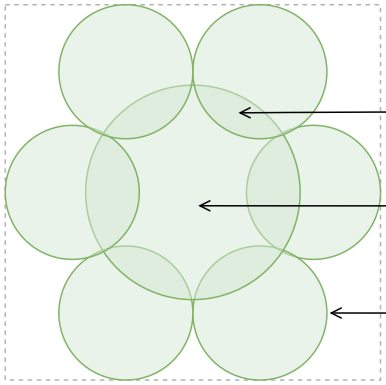
Actually what are we trying to test

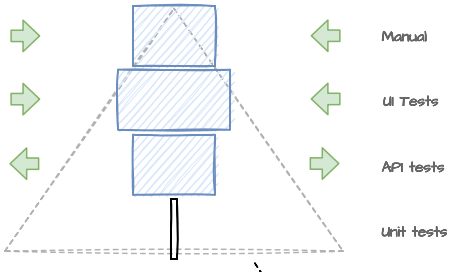
How could we focus our tests on the behavior and avoid unnecessary dependencies?

API Tests !



More efficient coverage





But...

1. Usually endpoints have more complex behaviors
2. Might have complex contracts - making it hard to write
3. API requests more optimal but still expensive when doing thousands of them
4. Small changes to contracts brake a lot of tests, even if behavior did not change

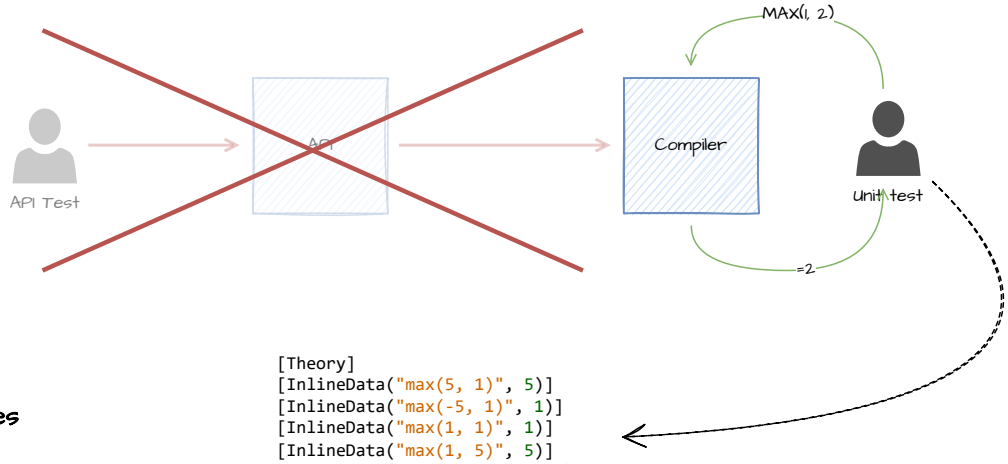
Request

```
{
  "cases": [{
    "forecasts": [{
      "year": "2022-01-01T00:00:00Z",
      ...
    }],
    ...
  }],
  "dataModel": {
    "equity": "{initial equity} *
{interests}^{years}",
    "payout": "{shareholding %} * {equity}"
  }
}
```

Response

```
{
  "cases": {
    "low case": {,
      "forecasts": {
        "2022-01-01": {
          "equity":
150000.00,
          ...
        },
        ...
      }
      ...
    }
  }
}
```

Could we do better?

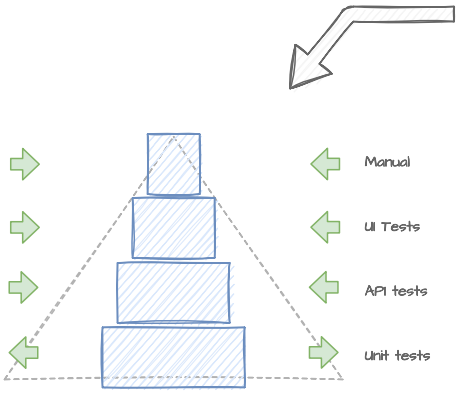


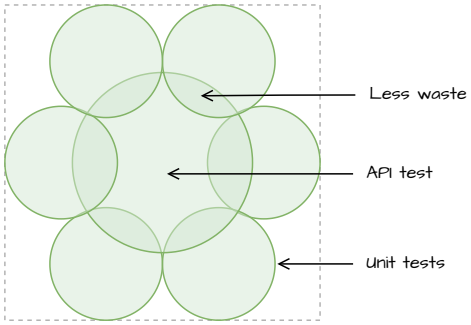
Positives

- Stability
- Speed
- Simpler to write
- Easily put system in a desired state, thus testing edge cases becomes possible

```
[Theory]
[InlineData("max(5, 1)", 5)]
[InlineData("max(-5, 1)", 1)]
[InlineData("max(1, 1)", 1)]
[InlineData("max(1, 5)", 5)]
[InlineData("max(-1, -5)", -1)]
public void ReturnsMaxNumber_OfGivenArguments(string expression, double
result)
  => new CalculationTestBuilder(_compiler) ...;

[Theory]
[InlineData("<syntax_error>max()</syntax_error>")]
[InlineData("<syntax_error>max(1)</syntax_error>")]
[InlineData("max(1, 2)")]
[InlineData("max(1, 2, 3, 4)")]
public void RequiresMoreThan1Argument(string expression)
  => new CalculationTestBuilder(_compiler) ...;
```





But...

- Further from the actual user - harder to understand whats being tested
- Closer to implementation - **harder for QA to write**

A lot of implementation detail knowledge required

```
[Fact]
0 references | 0 changes | 0 authors, 0 changes
private void MaxFunction_ReturnsCorrectly_WhenAllParametersAreNegative()
{
    var name = "equity";

    var _inputs = new List<Input>() {
        new Input(new VariableName("first"), InputValue.Number(-1)),
        new Input(new VariableName("second"), InputValue.Number(-5))
    };

    var variables = new Dictionary<string, string>() { { name, "MAX({first}, {second})" } };

    var compiledVariables = variables.Select(x =>
    {
        var variable = new Variable<ParsedFormula>(new VariableName(x.Key), ParsedFormula.Create(x.Value));
        return variable.Transform(formula => new CompiledFormula(formula, _compiler.Compile(variable.Formula.SyntaxTree)));
    });

    var compiledDataModel = DataModel<CompiledFormula>.Create(compiledVariables.ToArray());

    var dataContext = DataContext.Create(
        new ProbableValue<Guid>(GuidProvider.CreateGuid(), Proportion.Create(1)),
        new ProbableValue<Guid>(GuidProvider.CreateGuid(), Proportion.Create(1)),
        new ProbableValue<Guid>(GuidProvider.CreateGuid(), Proportion.Create(1)));
    _inputs.ForEach(x => dataContext.Set(x.Name, x.Value));

    var allContexts = new[] { new ProbableValue<DataContext>(dataContext, Proportion.Create(1)) };

    var evaluationContext = EvaluationContext.Create(compiledDataModel, new[] { dataContext });

    var compiledVariable = evaluationContext.Model.Variables[new VariableName(name)];

    var evaluatedResult = evaluationContext.Evaluate(new VariableName(name));
    var expectedResult = Value.Number(compiledVariable.Formula.SyntaxTree, -1);

    RuntimeValueTester.EnsureEqual(evaluatedResult, expectedResult);
}
```

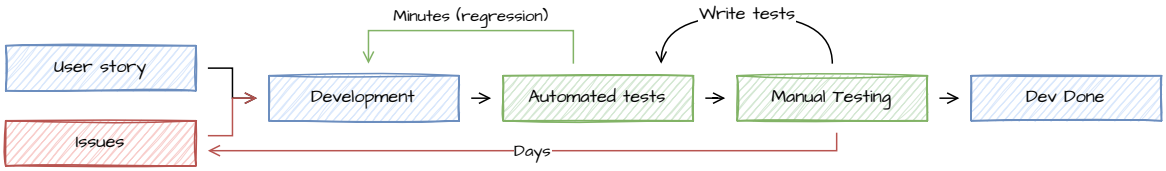
Good engineers should always cover their work with tests - best engineers practice "Test Driven Development"

But then, how can we have confidence of what's covered?

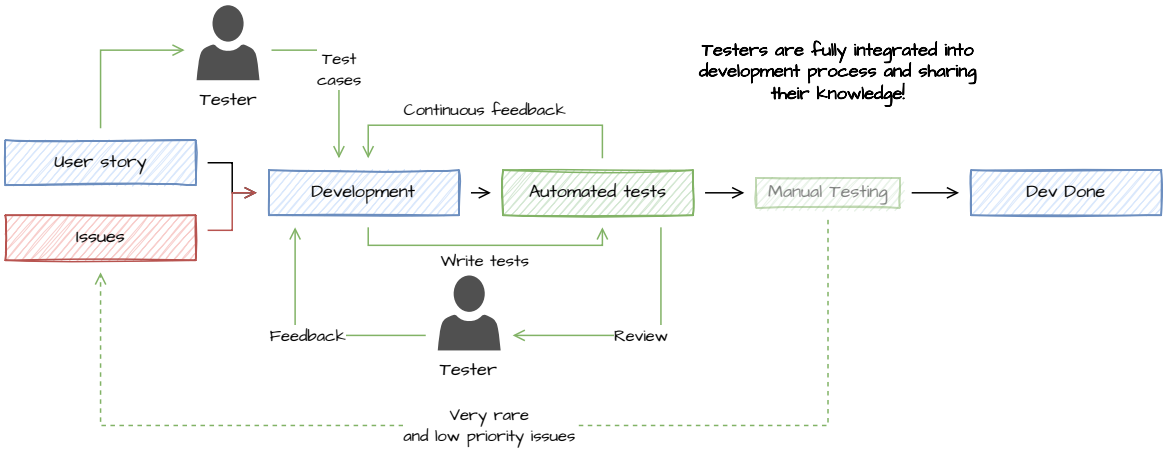
P.S. We, developers, love continuous feedback

Continuous testing

From fast to continuous feedback



Still, days until we get feedback about the feature...



Also drives automated test quality

	Institution	Management	Co-investor	Other
Invested amount	**50000.00**	**30000.00**	**20000.00**	**0.00**
Holding	50%	30%	20%	0%

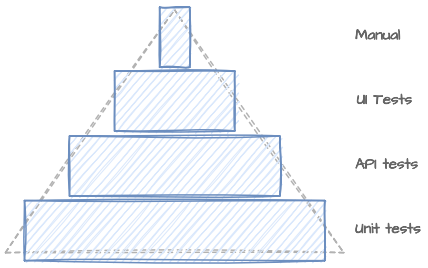
All scenarios must be duplicated for preference shares and shareholder loan notes instrument types

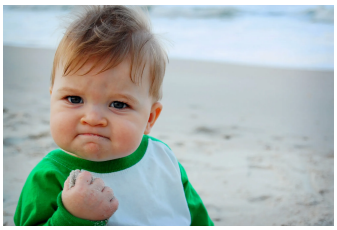
Forecast calculation with start date before valuation date

Coupon	10%
Coupon start date	2022-06-01
Valuation date	2023-01-01
Forecasts	2023-01-01
Year frac	0.58630
Closing balance total	105747.12967
Institution	52873.56484
Management	31724.13890
Co-investor	21149.42593
Other	0.00000

```
[Theory]
[InlineData(InstrumentType.ShareholderLoanNotes)]
[InlineData(InstrumentType.PreferredShares)]
0 references | benas.radzevicius, 17 hours ago | 1 author, 1 change
public void Forecasts_WhenCouponStartDate_IsBeforeFirstForecastYearDate(InstrumentType instrumentType)
{
    var result = Instrument(instrumentType,
        couponStartDate: new DateOnly(2022, 6, 1),
        coupon: 10m,
        investments: new[]
        {
            new Investment(OwnerType.Institution, 50000, NumberOfShares_Any),
            new Investment(OwnerType.Management, 30000, NumberOfShares_Any),
            new Investment(OwnerType.CoInvestor, 20000, NumberOfShares_Any),
            new Investment(OwnerType.Other, 0, NumberOfShares_Any),
        })
        .GetForecast(forecastYear: new DateOnly(2023, 1, 1));

    result.Should()
        .BeEquivalentTo(new Forecast(
            Total: 105747.12967m,
            new Dictionary<OwnerType, decimal>()
            {
                { OwnerType.Institution, 52873.56484m },
                { OwnerType.Management, 31724.13890m },
                { OwnerType.CoInvestor, 21149.42593m },
                { OwnerType.Other, 0m },
            }),
            (options) => options.Using<decimal>(o => o.Subject.Should()
                    .BeApproximately(o.Expectation, 0.00001m)).WhenTypeIs<decimal>());
}
```





Summary

- **There is no tradeoff** between delivery speed and quality - you can't choose one over the other
- **Optimize your test suite** form End-to-end tests towards unit tests by writing more focused and cheaper tests
- Having multiple tests testing partially the same thing is a **great indicator** for opportunity of optimization
- **Shift left quality evaluation** - move it earlier in development process until you reach continuous testing, work together with developers
- Focus on quality and speed will **naturally steer you towards correct test type distribution** marketed by a testing pyramid

Thank you!

Benas Radzevičius
Principal Software engineer
benas.radzevicius@devbridge.com