Spending less time bug fixing by spending more time unit testing

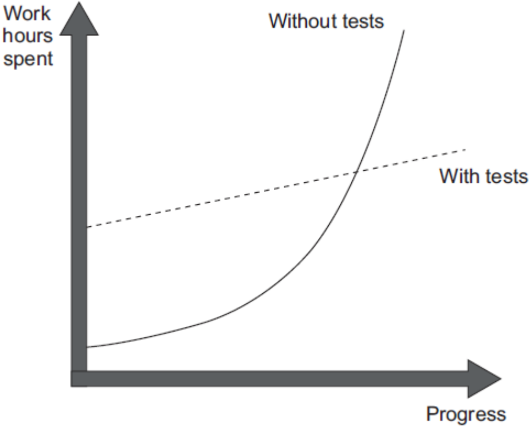
<Name>

There's much more to unit testing than the act of writing tests.

—***Khorikov***, **Unit Testing Principles, Practices, and Patterns**, ***3***

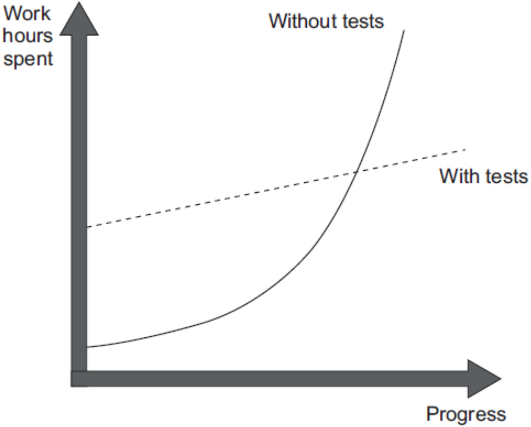
The goal of unit testing

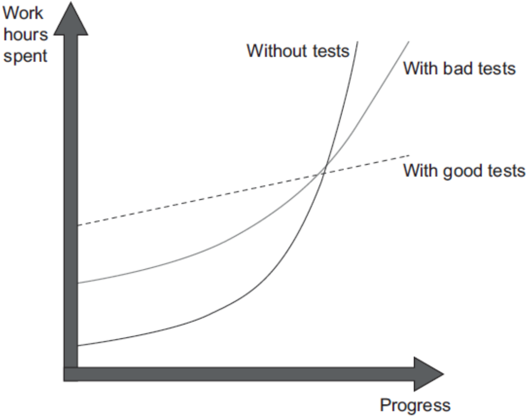
To enable **sustainable** growth of software project.



The goal of unit testing

To enable **sustainable** growth of software project.





Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```python
def is_fizzbuzz(num: int) -> bool:
    if num % 3 and num % 5:
        return True
    return some_var

def test_fizzbuzz():
    result = is_fizzbuzz(3)
    assert result
```

$$\frac{\mathit{Number\ of\ statements\ executed}}{\mathit{Total\ number\ of\ statements}}$$

$\approx 67\%$

Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```python
def is_fizzbuzz(num: int) -> bool:
    return True if num % 3 and num % 5 else some_var

def test_fizzbuzz():
    result = is_fizzbuzz(3)
    assert result
```

$$\frac{Number\ of\ statements\ executed}{Total\ number\ of\ statements}$$

$$= 100\%$$

Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```
def is_fizzbuzz(num: int) -> bool:
    return True if num % 3 and num % 5 else some_var

def test_fizzbuzz():
    result = is_fizzbuzz(3)
    assert result
```

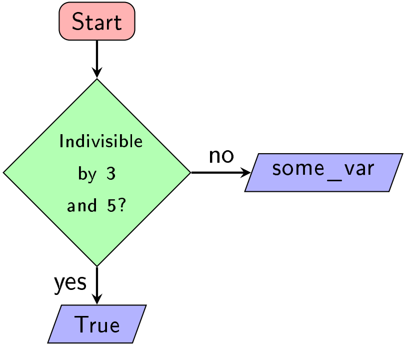
$$\frac{\textit{Branches traversed}}{\textit{Total number of branches}}$$

$$= 50\%$$

Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```python
def is_fizzbuzz(num: int) -> bool:
    return True if num % 3 and num % 5 else some_var

def test_fizzbuzz():
    result = is_fizzbuzz(3)
    assert result
```

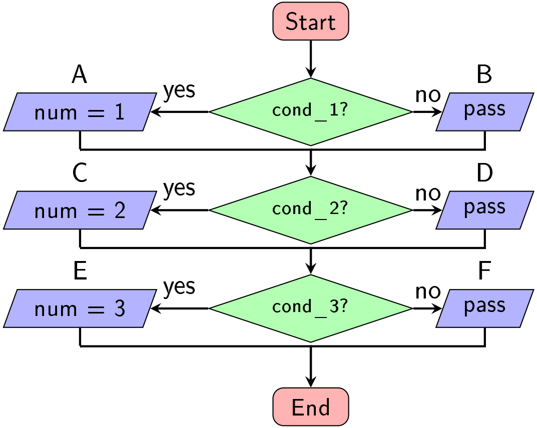


Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```python
def generate_number(
    cond_1: bool = True,
    cond_2: bool = True,
    cond_3: bool = True,
) -> int:
    if cond_1:
        num = 1
    if cond_2:
        num = 2
    if cond_3:
        num = 3
    return num
```

Possible paths:
ACE, ACF, ADE, ADF, BCE, BCF, BDE, BDF



Coverage metrics

Statement vs Branch vs ~~Path~~ vs ~~Condition~~

```
def is_fizzbuzz(num: int) -> bool:
    if num % 3 and num % 5:
        return True
    return some_var

def test_fizzbuzz():
    result = is_fizzbuzz(3)
    assert result
```

num % 3	num % 5	num % 3 and num % 5
True	True	True
True	False	False
False	True	False
False	False	False

> *[C]overage metrics are a good negative indicator, but a bad positive one.*
>
> —***Khorikov***, **Unit Testing Principles, Practices, and Patterns**, ***15***

Definition of a unit test

- Verifies a small piece of code,
- Does it quickly, and
- Does it in an isolated manner.

An integration test is a test that doesn't meet one of these criteria. End-to-end tests are a subset of integration tests and usually include more dependencies.

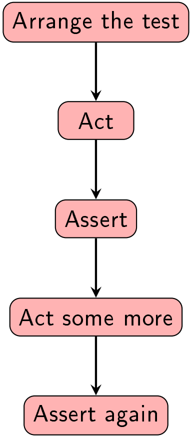
Anatomy of a unit test

The AAA (3A) pattern, also Given-When-Then pattern.

```
# A cohesive set of tests, optional
class TestCalculator:
    # Name of the unit test
    def test_sum_of_two_numbers(self):
        # Arrange
        first = 10
        second = 20
        calculator = Calculator()

        # Act
        result = calculator.sum(first, second)

        # Assert
        assert result == 30
```

- In *Arrange*, bring the system under test (SUT) to the a desired state
- In *Act*, call the method on the SUT, pass the prepared dependencies, and capture the output (if any).
- In *Assert*, verify the outcome. The outcome could be the return value, the final state of the SUT, or the methods the SUT called on its collaborators.

Things to avoid for unit tests


Arrange the test

Act

Assert

Act some more

Assert again

- Avoid multiple arrange, act, and assert sections.

Things to avoid for unit tests

```python
def test_node_with_python_updates(self, req_file):
    with TestCase.assertLogs("...logger") as cap:
        assert check_requirements(
            NODE, req_file
        ) == 2
        for i, rec in enumerate(cap.records):
            idx = int(i / 2)
            if i == 4:
                assert "2 packages updated" in rec.getMessage()
            elif i % 2 == 0:
                assert f"{PY_PKGS[idx]} not found" in rec.getMessage()
            else:
                assert f"pip install {PY_PKGS[idx]}" in rec.getMessage()
```

- Avoid multiple arrange, act, and assert sections.
- Avoid `if` statements.

Naming a unit test

```python
def test_is_delivery_valid_invalid_date_returns_false():
    sut = DeliveryService()
    past_date = datetime.today() - timedelta(days=1)
    delivery = Delivery(date=past_date)

    is_valid = sut.is_delivery_valid(delivery)

    assert not is_valid
```

- A rigid convention such as `<method>_<scenario>_<expected>` isn't as helpful as plain English

Naming a unit test

```python
def test_is_delivery_valid_invalid_date_returns_false():
    sut = DeliveryService()
    past_date = datetime.today() - timedelta(days=1)
    delivery = Delivery(date=past_date)

    is_valid = sut.is_delivery_valid(delivery)

    assert not is_valid

def test_delivery_with_past_date_should_be_considered_invalid():
    ...
```

- A rigid convention such as `<method>_<scenario>_<expected>` isn't as helpful as plain English
- Should not be too verbose

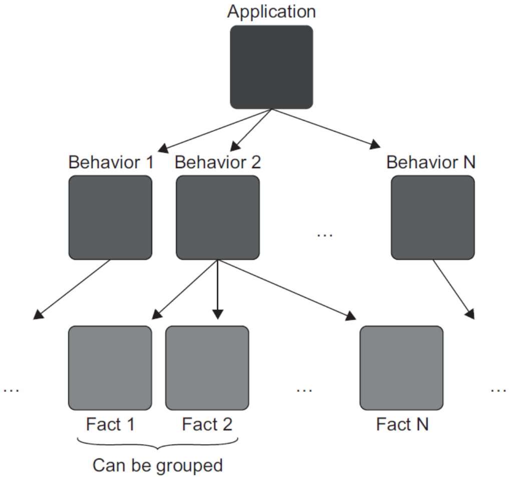
Naming a unit test

```python
def test_is_delivery_valid_invalid_date_returns_false():
    sut = DeliveryService()
    past_date = datetime.today() - timedelta(days=1)
    delivery = Delivery(date=past_date)

    is_valid = sut.is_delivery_valid(delivery)

    assert not is_valid

def test_delivery_with_past_date_should_be_considered_invalid():
    ...

def test_delivery_with_a_past_date_is_invalid():
    ...
```

- A rigid convention such as `<method>_<scenario>_<expected>` isn't as helpful as plain English
- Should not be too verbose

Parametrizing tests

Parametrization, also spelled parameterization, parametrisation or parameterisation, is the process of defining or choosing parameters. — Wikipedia



- The number of tests can become unmanageable if each component/behavior of the application is tested with its own test.
- Some (similar) behaviors can be grouped into a single test using parametrization.

Parametrizing tests

Behavior: The soonest allowed delivery date is two days from now.

In addition to test_delivery_with_a_past_date_is_invalid, we need to add three more:

```
def test_delivery_for_today_is_invalid():
    ...

def test_delivery_for_tomorrow_is_invalid():
    ...

def test_the_soonest_delivery_date_is_two_days_from_now():
    ...
```

This would result in four test methods, with the only difference between them being the delivery date.

Parametrizing tests

Behavior: The soonest allowed delivery date is two days from now.

```python
@pytest.mark.parametrize(
    "days_from_now,expected",
    [(-1, False), (0, False), (1, False), (2, True)],
)
def test_can_detect_an_invalid_delivery_date(
    days_from_now, expected
):
    sut = DeliveryService()
    delivery_date = datetime.today() + timedelta(days=days_from_now)
    delivery = Delivery(date=delivery_date)

    is_valid = sut.is_delivery_valid(delivery)

    assert is_valid == expected
```

- Significantly reduce the amount of test code

Parametrizing tests (meaningfully)

Behavior: The soonest allowed delivery date is two days from now.

```
@pytest.mark.parametrize("days_from_now", [-1, 0, 1])
def test_detects_an_invalid_delivery_date(days_from_now):
    ...

    assert not is_valid


def test_the_soonest_delivery_date_is_two_days_from_now():
    ...

    assert is_valid
```

- Significantly reduce the amount of test code
- Do not “over parametrize” if the scenarios are complicated

Using an assertion library (optional)

An assertion library like `assertpy` can improve test readability by making the assert section read like plain English.

```python
def test_sum_of_two_numbers():
    ...

    assert result == 30
```

```python
def test_sum_of_two_numbers():
    ...

    assert_that(result).is_equal_to(30)
```

- Introduces additional dependencies

Using an assertion library (optional)

An assertion library like assertpy can improve test readability by making the assert section read like plain English.

```python
def test_sum_of_two_numbers():
    ...

    assert result == 30

def test_sum_of_two_numbers():
    ...

    assert_that(result).is_equal_to(30)
```

Bonus: Chai assertion library

```javascript
describe("Calculator", () => {
    it("computes the sum of two number", () => {
        const calculator = new Calculator();

        const result calculator.sum(10, 20);

        expect(result).to.be.equal(30);
    });
});
```

- Introduces additional dependencies

Recognizing a good unit test

The four pillars of a good unit test

- Protection against regression
  - Amount of code executed during the test
  - Complexity of that code
  - The code's domain significance
- Resistance against refactoring
- Fast feedback
  - "Fast enough"
  - Can be run more often to detect regressions
- Maintainability
  - How hard is it to understand the test: Test code quality matters as much as production code
  - How hard is it to run the test

The second pillar: Resistance to refactoring

Refactoring means changing existing code without modifying its observable behavior.

The second pillar: Resistance to refactoring

Refactoring means changing existing code without modifying its observable behavior.

Scenario: You developed a new feature and everything works great. The feature is working as intended and all the tests are passing.

You decide to clean up the code before submitting the PR. Some refactoring here and there, and the code ends up looking better than before.

Except one thing — the tests are failing. But the feature is still working perfectly, just as before. Turns out the tests are written in such a way that they fail with any modifications to the underlying code.

This situation is a *false positive*.

The second pillar: Resistance to refactoring

Refactoring means changing existing code without modifying its observable behavior.

Scenario: You developed a new feature and everything works great. The feature is working as intended and all the tests are passing.

You decide to clean up the code before submitting the PR. Some refactoring here and there, and the code ends up looking better than before.

Except one thing — the tests are failing. But the feature is still working perfectly, just as before. Turns out the tests are written in such a way that they fail with any modifications to the underlying code.

This situation is a *false positive*.

Why is this so important that it deserves its own slide?

- Enable sustainable project growth
- Provide early warning to regressions
- Give confidence that code changes won't lead to regressions

How to avoid false positives?

Number of false positives is directly related to how the test is structured.

- The more the test is coupled to the implementation detail, the more false positives it generates.

Solution: Verify the end result (observable behavior) the SUT delivers, not the steps it takes to do that. The best way is to structure the test to tell a story about the problem domain.

Example: Testing MessageRenderer

```
@dataclass
class Message:
    header: str
    body: str
    footer: str

class IRenderer(ABC):
    @abstractmethod
    def render(self, message: "Message") -> str:
        """Renders the provided message."""

class MessageRenderer(IRenderer):
    def __init__(self) -> None:
        self.sub_renderers = [HeaderRenderer(), BodyRenderer(), FooterRenderer()]

    def render(self, message: "Message") -> str:
        return "".join(renderer.render(message) for renderer in self.sub_renderers)

class HeaderRenderer(IRenderer):
    def render(self, message: "Message") -> str:
        return f"<head>{message.header}</head>"

...
```

Example: Testing MessageRenderer

```
def test_message_renderer_uses_correct_sub_renderers():
    sut = MessageRenderer()

    sub_renderers = sut.sub_renderers

    assert len(sub_renderers) == 3
    assert isinstance(sub_renderers[0], HeaderRenderer)
    assert isinstance(sub_renderers[1], BodyRenderer)
    assert isinstance(sub_renderers[2], FooterRenderer)
```

Example: Testing MessageRenderer

```
def test_message_renderer_uses_correct_sub_renderers():
    sut = MessageRenderer()

    sub_renderers = sut.sub_renderers

    assert len(sub_renderers) == 3
    assert isinstance(sub_renderers[0], HeaderRenderer)
    assert isinstance(sub_renderers[1], BodyRenderer)
    assert isinstance(sub_renderers[2], FooterRenderer)
```

Does this really verify `MessageRenderer`'s observable behavior?

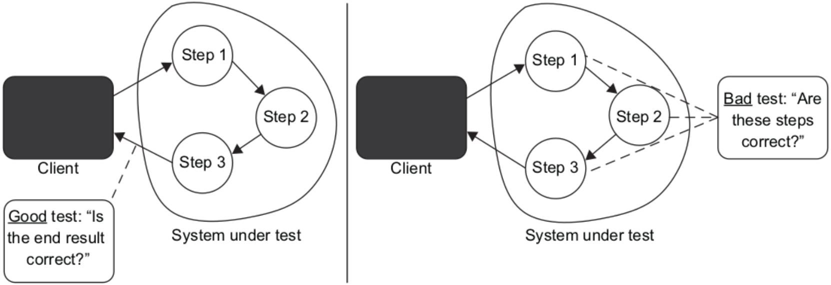
- What if you rearrange the sub-renderers?
- What if you replace one of the sub-renderers?
- What if you stop using sub-renderers and implement the rendering directly?

Does this affect the rendered HTML document? Does the test fail?

Example: Testing MessageRenderer

```
def test_message_renderer_renders_message():
    sut = MessageRenderer()
    message = Message("h", "b", "f")

    html = sut.render(message)

    assert html == "<head>h</head><body>b</body><foot>f</foot>"
```

This test treats `MessageRenderer` as a black box and is only interested in its observable behavior.



Dynamics between the first and second pillar

		Functionality is	
		Correct	Broken
Test result	Passes	Correct inference (TN)	Type II error (FN)
	Fails	Type I error (FP)	Correct inference (TP)

Protection against regression

Resistance to refactoring

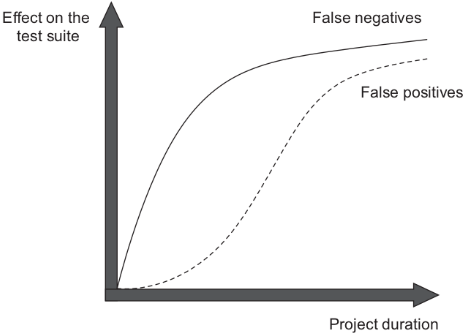
Dynamics between the first and second pillar

		Functionality is	
		Correct	Broken
Test result	Passes	Correct inference (TN)	Type II error (FN)
	Fails	Type I error (FP)	Correct inference (TP)

Protection against regression

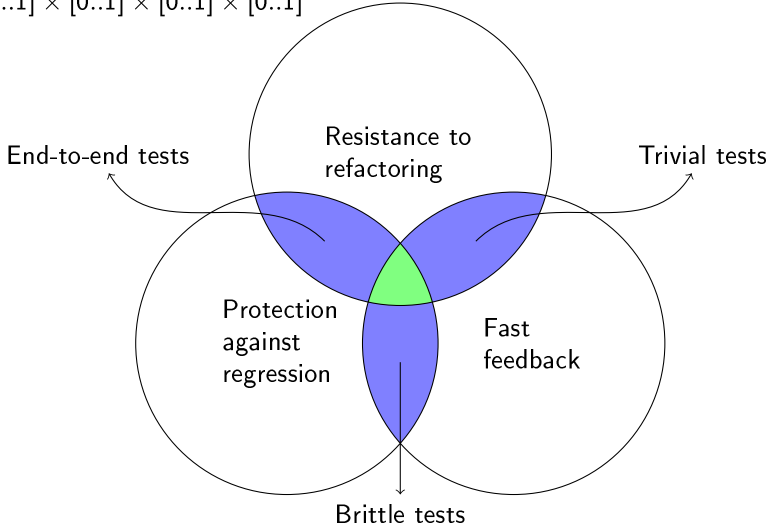
Resistance to refactoring

- How good the test is at indicating the presence of bugs: Protection against regression
- How good the test is at indicating the absence of bugs: Resistance of refactoring



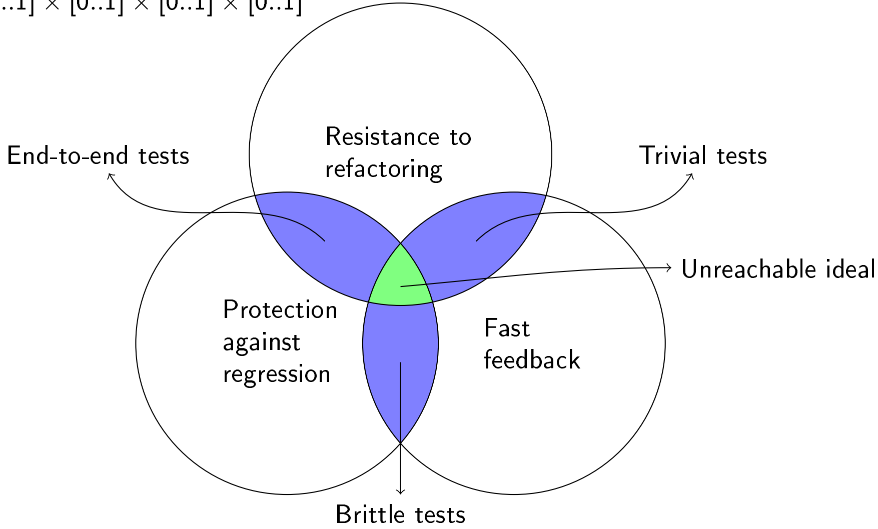
In search of an ideal test

$\mathit{Value} = [0..1] \times [0..1] \times [0..1] \times [0..1]$

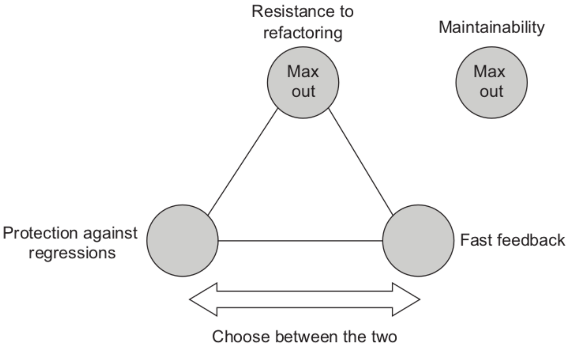


In search of an ideal test

$Value = [0..1] \times [0..1] \times [0..1] \times [0..1]$



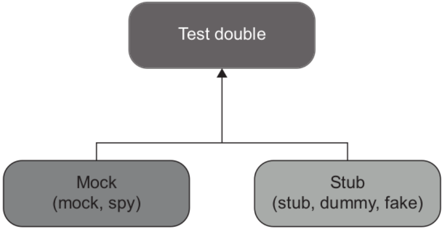
In search of an ideal test



- *Resistance to refactoring* is non-negotiable: Almost a binary choice
- Test automation concepts can be traced back to the four pillars:
  - The Test Pyramid
  - White-box versus black-box testing

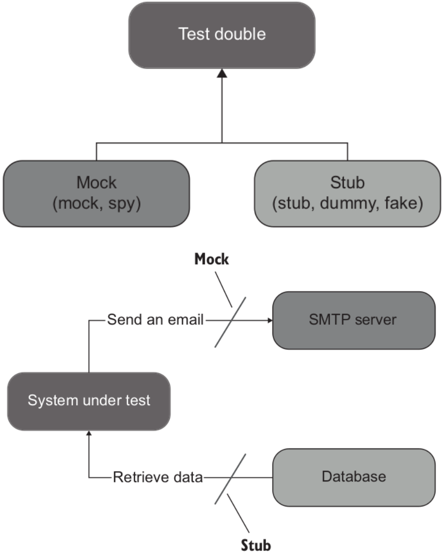
Mocks and test fragility

Differentiating mocks from stubs



- Test doubles can be grouped into two types: mocks and stubs

Differentiating mocks from stubs



- Test doubles can be grouped into two types: mocks and stubs
- Mocks emulate and examine *outcoming* interactions
- Stubs emulate *incoming* interactions