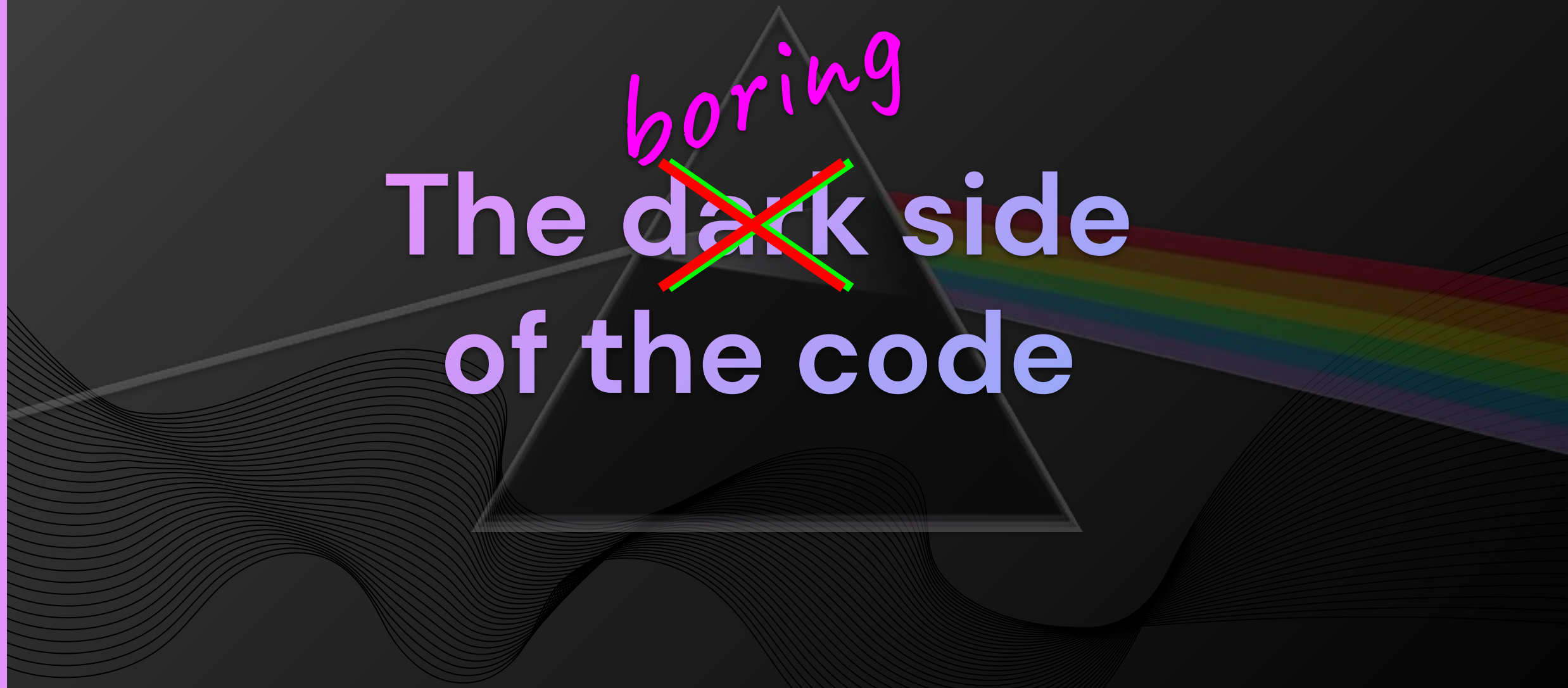

boring
The dark side
of the code

Hi folks!

I'm Mattia Piccinetti
Data Engineer @ Generali Italia
Advanced Analytics Platform

Clean code lover
TDD enthusiast

"I do backend stuff to pay the bills."

Let's talk about automated testing

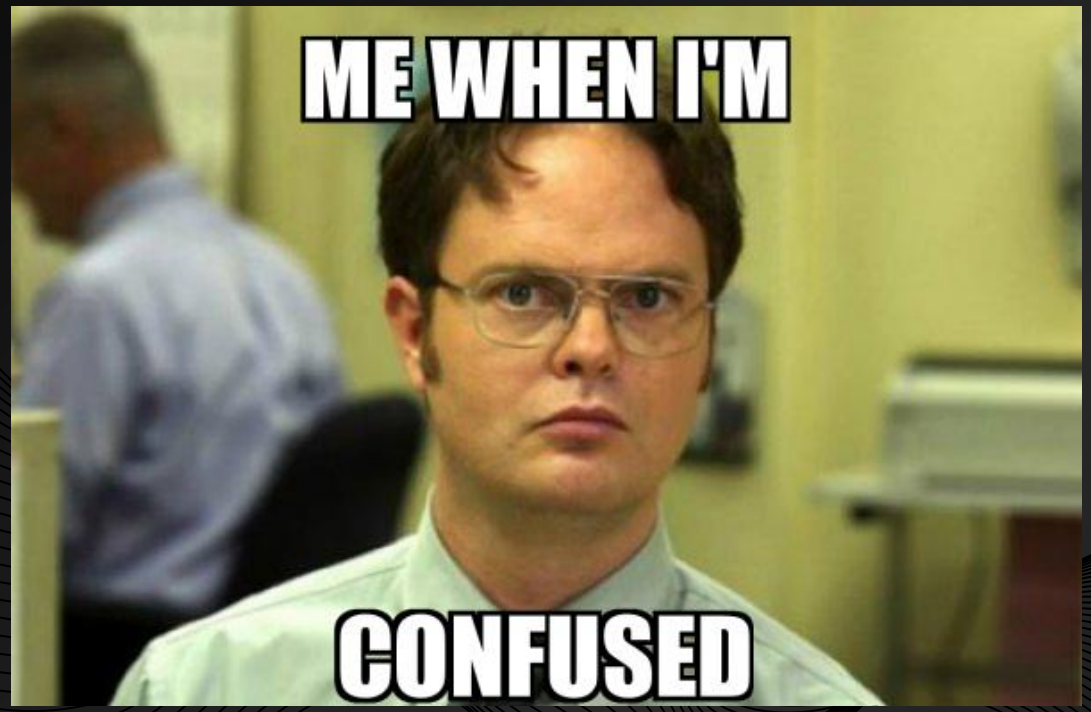
Let's talk about automated testing

Have you ever tried???

What is testing?

“It’s the act of examining the artifacts and the behavior of the software under test by validation and verification.” (Wikipedia)

How many types should I know?



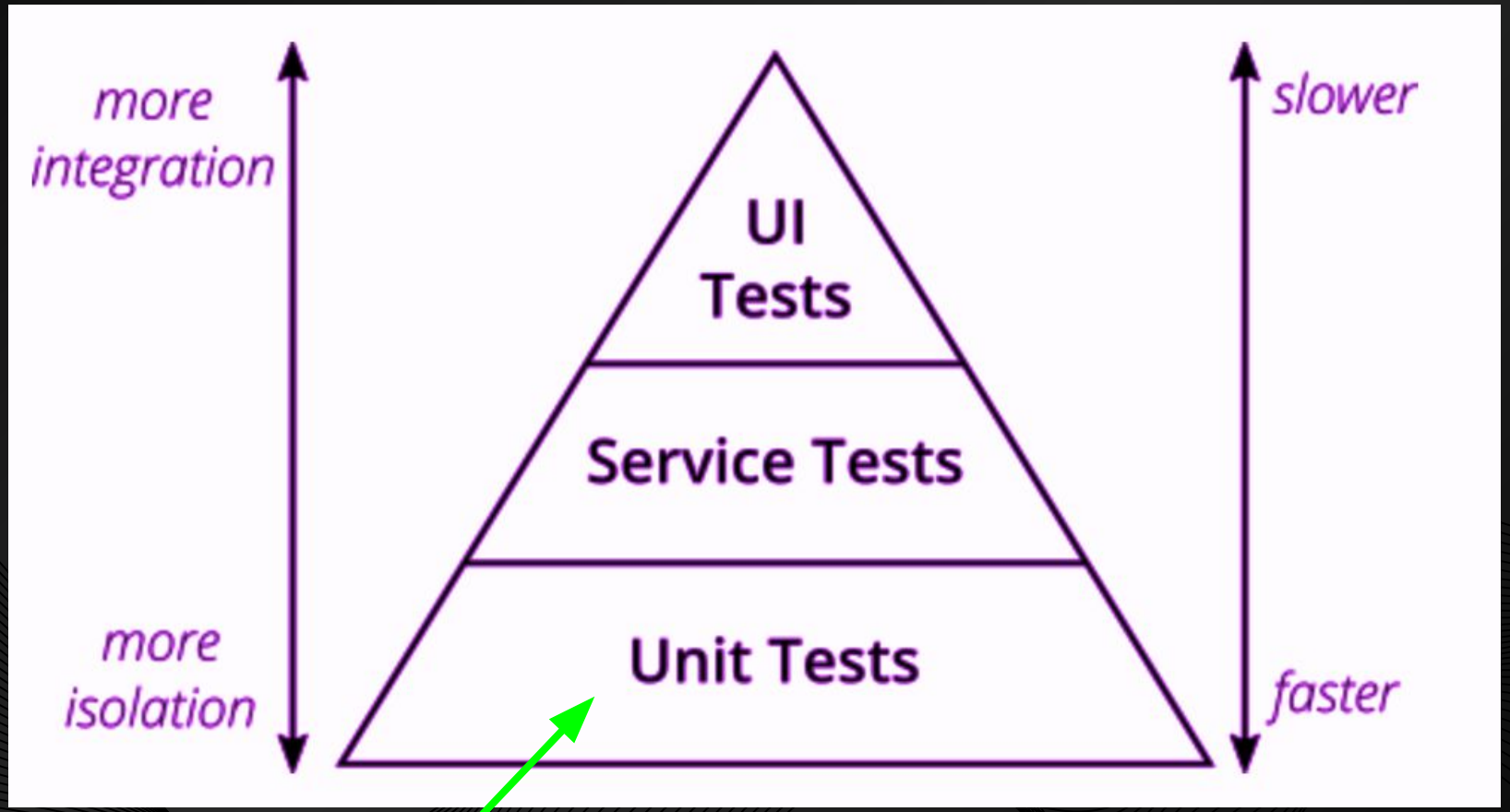
UNIT TESTS
ACCEPTANCE TESTS
PERFORMANCE TESTS
INTEGRATION TESTS
FUNCTIONAL TESTS
END TO END TESTS
SMOKE TESTS

All of them?

Just unit tests

(for today)

Test pyramid



$$$$$$$$
$$$$$$$
$$$$$$
$$$$$
$$$$
$$$
$$
$

WE ARE HERE

Ok, got it but why testing? why should I care?

Less regression errors
Easy changes
Fast debugging
Code quality
Design tool
Documentation

Let's start with the AAA

The AAA pattern

Arrange

Act

Assert

Anatomy of a unit test

```
def test_join_with_dash():
    word_1 = "foo"
    word_2 = "bar"

    actual  = join_with_dash(word_1, word_2)

    assert actual == "foo-bar"
```

Anatomy of a unit test

Anatomy of a unit test

```python
def test_ok():
    word_1 = "foo"
    word_2 = "bar"
    r = join_with_dash(word_1, word_2)
    assert r == "foo-bar"
```

MEH...

Hint: keep the AAA sections separated with blank lines

Anatomy of a unit test

```python
def test_join_with_dash():
    word_1 = "foo"
    word_2 = "bar"

    actual  = join_with_dash(word_1, word_2)

    assert actual == "foo-bar"
```

BETTER

Got it!
Let me add more tests

CASE 1 →

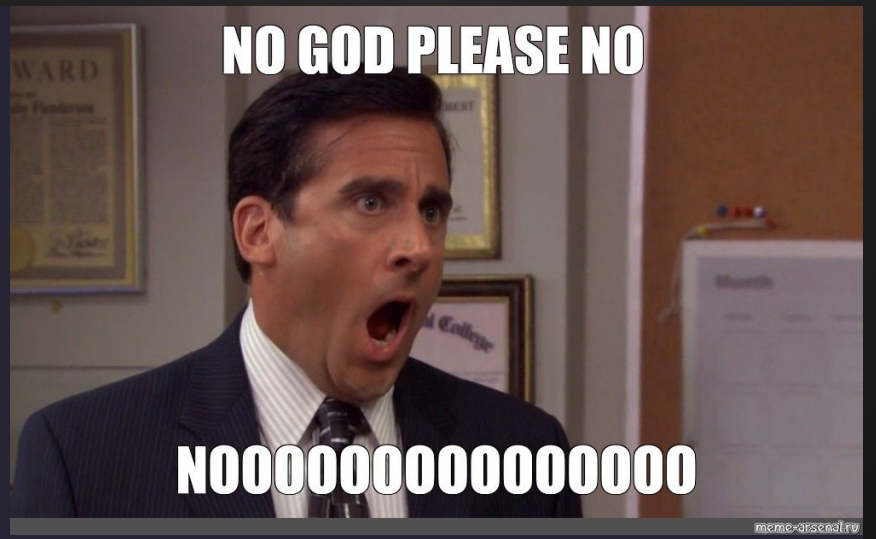
CASE 2 →

CASE 3 →

```python
def test_join_with_dash():
    word_1 = "foo"
    word_2 = "bar"

    actual  = join_with_dash(word_1, word_2)

    assert actual == "foo-bar"

    word_3 = "baz"
    word_4 = "gaglioffo"

    actual  = join_with_dash(word_3, word_4)

    assert actual == "baz-gaglioffo"

    word_empty = ""
    word_none = None

    actual  = join_with_dash(word_empty, word_none)

    assert actual == ""
```

...or worse

```python
def test_join_with_dash():
    word_1 = "foo"
    word_2 = "bar"
    word_3 = "baz"
    word_4 = "gaglioffo"
    word_empty = ""
    word_none = None
    actual  = join_with_dash(word_1, word_2)
    assert actual == "foo-bar"
    actual  = join_with_dash(word_3, word_4)
    assert actual == "baz-gaglioffo"
    actual  = join_with_dash(word_empty, word_none)
    assert actual == ""
```



Hint: keep your tests separated and don’t be redundant

```python
def test_join_with_dash():
    word_1 = "foo"
    word_2 = "bar"

    actual  = join_with_dash(word_1, word_2)

    assert actual == "foo-bar"

def test_join_with_dash_empty_first():
    word_1 = ""
    word_2 = "bar"

    actual  = join_with_dash(word_1, word_2)

    assert actual == "bar"

def test_join_with_dash_none_second():
    word_1 = "foo"
    word_2 = None

    actual  = join_with_dash(word_1, word_2)

    assert actual == "foo"
```

Hint: choose a good name for your tests to explicit the intent

Common mistakes

```python
class ToDoList:
    def __init__(self, tasks: list[str] = None) -> None:
        self.tasks = [] if tasks is None else tasks

    def add_task(self, text: str) -> int:
        self.tasks.append(text)

        return len(self.tasks) - 1
```

```python
todo_list = ToDoList()


def test_tasks_should_be_empty():
    actual = todo_list.tasks

    assert len(actual) == 0


def test_tasks_should_return_1_task():
    todo_list.add_task("go to the doctor")

    actual = todo_list.tasks

    assert len(actual) == 1
    assert "go to the doctor" in actual
```

```python
todo_list = ToDoList()


def test_tasks_should_be_empty():
    actual = todo_list.tasks

    assert len(actual) == 0


def test_tasks_should_return_1_task():
    todo_list.add_task("go to the doctor")

    actual = todo_list.tasks

    assert len(actual) == 1
    assert "go to the doctor" in actual


def test_tasks_should_return_2_tasks():
    todo_list.add_task("go to the doctor")
    todo_list.add_task("go to the pharmacy")

    actual = todo_list.tasks

    assert len(actual) == 2
    assert "go to the doctor" in actual
    assert "go to the pharmacy" in actual
```

LET'S ADD ANOTHER TEST

```
todo_list = ToDoList()


def test_tasks_should_be_empty():
    actual = todo_list.tasks
```

```
assert 3 == 2
 +  where 3 = len(['go to the doctor', 'go to the doctor', 'go to the pharmacy'])
def test_tasks_should_return_2_tasks():
        todo_list.add_task("go to the doctor")
        todo_list.add_task("go to the pharmacy")

        actual = todo_list.tasks

>       assert len(actual) == 2
E       AssertionError: assert 3 == 2
E        +  where 3 = len(['go to the doctor', 'go to the doctor', 'go to the pharmacy'])
```

```
    actual = todo_list.tasks

    assert len(actual) == 2
    assert "go to the doctor" in actual
    assert "go to the pharmacy" in actual
```

Hint: do not share state between tests, keep them isolated

```python
def test_tasks_should_be_empty():
    todo_list = ToDoList()

    actual = todo_list.tasks

    assert len(actual) == 0


def test_tasks_should_return_1_task():
    todo_list = ToDoList()
    todo_list.add_task("go to the doctor")

    actual = todo_list.tasks

    assert len(actual) == 1
    assert "go to the doctor" in actual


def test_tasks_should_return_2_tasks():
    todo_list = ToDoList()
    todo_list.add_task("go to the doctor")
    todo_list.add_task("go to the pharmacy")

    actual = todo_list.tasks

    assert len(actual) == 2
    assert "go to the doctor" in actual
    assert "go to the pharmacy" in actual
```

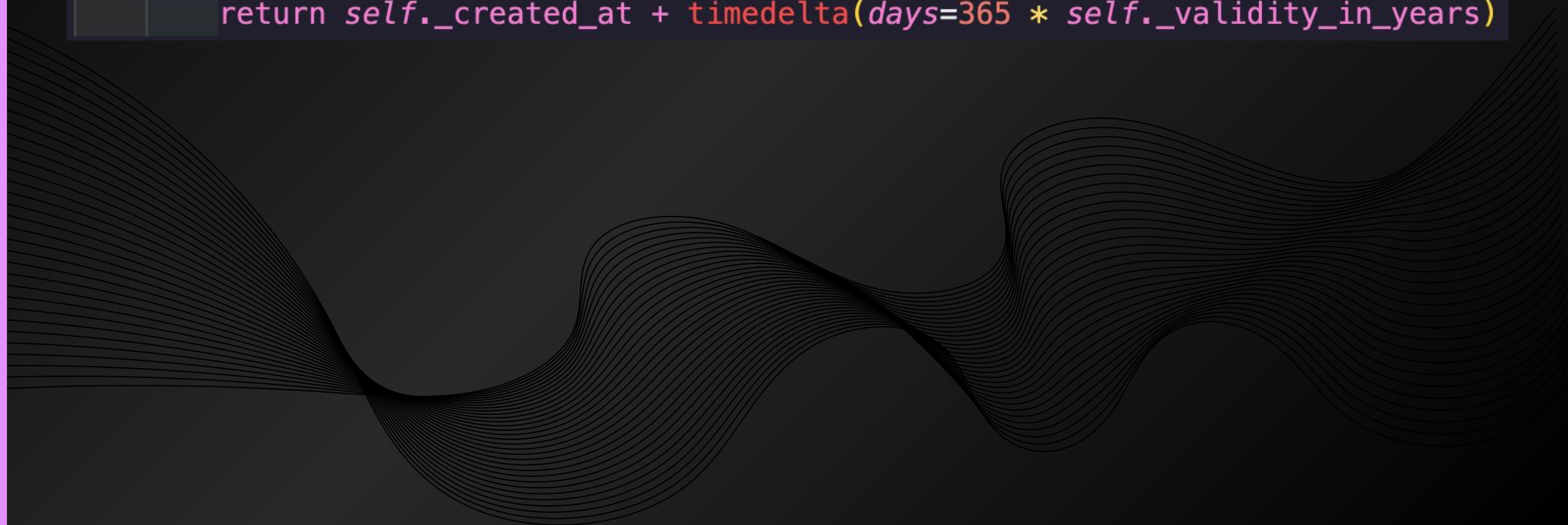
```python
def test_tasks_should_be_empty():
    todo_list = ToDoList()

    actual = todo_list.tasks

    assert len(actual) == 0


def test_tasks_should_return_1_task():
    todo_list = ToDoList()
    todo_list.add_task("go to the doctor")

    actual = todo_list.tasks

    assert len(actual) == 1
    assert "go to the doctor" in actual


def test_tasks_should_return_2_tasks():
    todo_list = ToDoList()
    todo_list.add_task("go to the doctor")
    todo_list.add_task("go to the pharmacy")

    actual = todo_list.tasks

    assert len(actual) == 2
    assert "go to the doctor" in actual
    assert "go to the pharmacy" in actual
```

OR

```python
@pytest.fixture
def todo_list():
    return ToDoList()


def test_tasks_should_be_empty(todo_list):
    actual = todo_list.tasks

    assert len(actual) == 0


def test_tasks_should_return_1_task(todo_list):
    todo_list.add_task("go to the doctor")

    actual = todo_list.tasks

    assert len(actual) == 1
    assert "go to the doctor" in actual


def test_tasks_should_return_2_tasks(todo_list):
    todo_list.add_task("go to the doctor")
    todo_list.add_task("go to the pharmacy")

    actual = todo_list.tasks

    assert len(actual) == 2
    assert "go to the doctor" in actual
    assert "go to the pharmacy" in actual
```

Let's play with time

```python
class TravelCard:
    def __init__(self, user_id: str, validity_in_years: int) -> None:
        self.user_id = user_id
        self._created_at = datetime.now()
        self._validity_in_years = validity_in_years

    def get_expiration(self) -> datetime:
        return self._created_at + timedelta(days=365 * self._validity_in_years)
```



```python
class TravelCard:
    def __init__(self, user_id: str, validity_in_years: int) -> None:
        self.user_id = user_id
        self._created_at = datetime.now()
        self._validity_in_years = validity_in_years

    def get_expiration(self) -> datetime:
        return self._created_at + timedelta(days=365 * self._validity_in_years)
```

```python
def test_get_expiration():
    travel_card = TravelCard(user_id=42, validity_in_years=4)

    actual = travel_card.get_expiration()

    assert actual == datetime.now() + timedelta(days=365 * 4)
```

```python
class TravelCard:
    def __init__(self, user_id: str, validity_in_years: int) -> None:
        self.user_id = user_id
        self._created_at = datetime.now()
        self._validity_in_years = validity_in_years
```

```
def test_get_expiration():
        travel_card = TravelCard(user_id=42, validity_in_years=4)

        actual = travel_card.get_expiration()

>       assert actual == datetime.now() + timedelta(days=365 * 4)
E       assert datetime.datetime(2026, 9, 24, 16, 14, 40, 499761) == (datetime.datetime(2022, 9, 25, 16, 14, 40, 499777) + datetime.timedelta(days=1460))
E        +  where datetime.datetime(2022, 9, 25, 16, 14, 40, 499777) = <built-in method now of type object at 0x10e99f978>()
E        +    where <built-in method now of type object at 0x10e99f978> = datetime.now
E        +  and   datetime.timedelta(days=1460) = timedelta(days=(365 * 4))
```

```python
    actual = travel_card.get_expiration()

    assert actual == datetime.now() + timedelta(days=365 * 4)
```

Hint: time is a “living” object, take control of it

```python
class TravelCard:
    def __init__(
        self,
        user_id: str,
        validity_in_years: int,
        datetime_fn: Callable[..., datetime],
    ) -> None:
        self.user_id = user_id
        self._created_at = datetime_fn()
        self._validity_in_years = validity_in_years

    def get_expiration(self) -> datetime:
        return self._created_at + timedelta(days=365 * self._validity_in_years)
```

ADD A SIMPLE FUNCTION THAT RETURNS A DATETIME OBJECT

```python
def test_get_expiration():
    now = datetime.now()
    travel_card = TravelCard(user_id=42, validity_in_years=4, datetime_fn=lambda: now)

    actual = travel_card.get_expiration()

    assert actual == now + timedelta(days=365 * 4)
```

hint: “freeze” time by passing a fixed time value

```
class TravelCard:
    def __init__(
        self,
        user_id: str,
        validity_in_years: int,
```

```
@freeze_time()
def test_get_expiration():
    travel_card = TravelCard(user_id=42, validity_in_years=4)

    actual = travel_card.get_expiration()

    assert actual == datetime.now() + timedelta(days=365 * 4)
```

```
    actual = travel_card.get_expiration()

    assert actual == now + timedelta(days=365 * 4)
```

Hint: “freeze” time using FreezGun or similar libraries

Let's interact with the external world

```
class UserNormalizer:
    def __init__(self) -> None:
        pass

    def normalize(self, user_id: int) -> tuple[int, Optional[str]]:
        client = UserServiceClient()
        response = client.get_user_by_id(user_id)

        normalized_user = None if response[1] is None else response[1].lower()

        return (user_id, normalized_user)
```

```python
_USER_DATABASE: dict[int, str] = {
    0: "Ray Proctor",
    1: "Ila Harrison",
    2: "Leo Brock",
    3: "Jon Carrillo",
    4: "Liz Shepherd",
    5: "Von Hayden",
    6: "Son Fuller",
    7: "Ann Crawford",
    8: "Ken Mack",
    9: "Bud Wagner",
}


class UserServiceClient:
    def __init__(self) -> None:
        pass

    def get_user_by_id(self, id: int) -> tuple[int, Optional[str]]:
        return _long_http_call(id)


def _long_http_call(id: int) -> tuple[int, Optional[str]]:
    import time

    time.sleep(5)

    return (id, _USER_DATABASE.get(id))
```

FAKE DATABASE

FAKE PROCESSING TIME

Let's test it

```python
def test_normalize_should_return_lowercase_user():
    user_normalizer = UserNormalizer()

    actual = user_normalizer.normalize(2)

    assert actual == (2, "leo brock")
```

It works but...

```python
def test_normalize_should_return_lowercase_user():
    user_normalizer = UserNormalizer()

    actual = user_normalizer.normalize(2)

    assert actual == (2, "leo brock")
```

- IT DEPENDS FROM AN EXTERNAL SERVICE
- HIGHLY COUPLED WITH THE CLIENT
- TOO SLOW
- WHO THE F**K IS LEO BROCK?

It works but...

Let's use the dependency injection

BEFORE

```python
class UserNormalizer:
    def __init__(self) -> None:
        pass

    def normalize(self, user_id: int) -> tuple[int, Optional[str]]:
        client = UserServiceClient()
        response = client.get_user_by_id(user_id)

        normalized_user = None if response[1] is None else response[1].lower()

        return (user_id, normalized_user)
```

AFTER

```python
class UserNormalizer:
    def __init__(self, client: UserServiceClient) -> None:
        self._client = client

    def normalize(self, user_id: int) -> tuple[int, Optional[str]]:
        response = self._client.get_user_by_id(user_id)

        normalized_user = None if response[1] is None else response[1].lower()

        return (user_id, normalized_user)
```

Hint: provide your object as constructor argument instead of creating it

```python
def test_normalize_should_return_lowercase_user():
    fake_client = FakeClient()
    user_normalizer = UserNormalizer(fake_client)

    actual = user_normalizer.normalize(2)

    assert actual == (2, "john doe")


class FakeClient(UserServiceClient):
    def get_user_by_id(self, id: int) -> tuple[int, Optional[str]]:
        return (id, "John Doe")
```

CREATE A FAKE CLASS OVERRIDING THE METHODS YOU NEED TO CONTROL

Hint: get control of your dependencies using fake objects

```python
def test_normalize_should_return_lowercase_user():
    fake_client = FakeClient()
    user_normalizer = UserNormalizer(fake_client)

    actual = user_normalizer.normalize(2)

    assert actual == (2, "john doe")


class FakeClient(UserServiceClient):
    def get_user_by_id(self, id: int) -> tuple[int, Optional[str]]:
        return (id, "John Doe")
```

CREATE A FAKE CLASS OVERRIDING THE METHODS YOU NEED TO CONTROL

OR USE THE MOCK OBJECT TO STUB THE RETURN VALUE OF YOUR METHODS

```python
def test_normalize_should_return_lowercase_user():
    client = UserServiceClient()
    client.get_user_by_id = Mock(return_value=(2, "John Doe"))
    user_normalizer = UserNormalizer(client)

    actual = user_normalizer.normalize(2)

    assert actual == (2, "john doe")
```

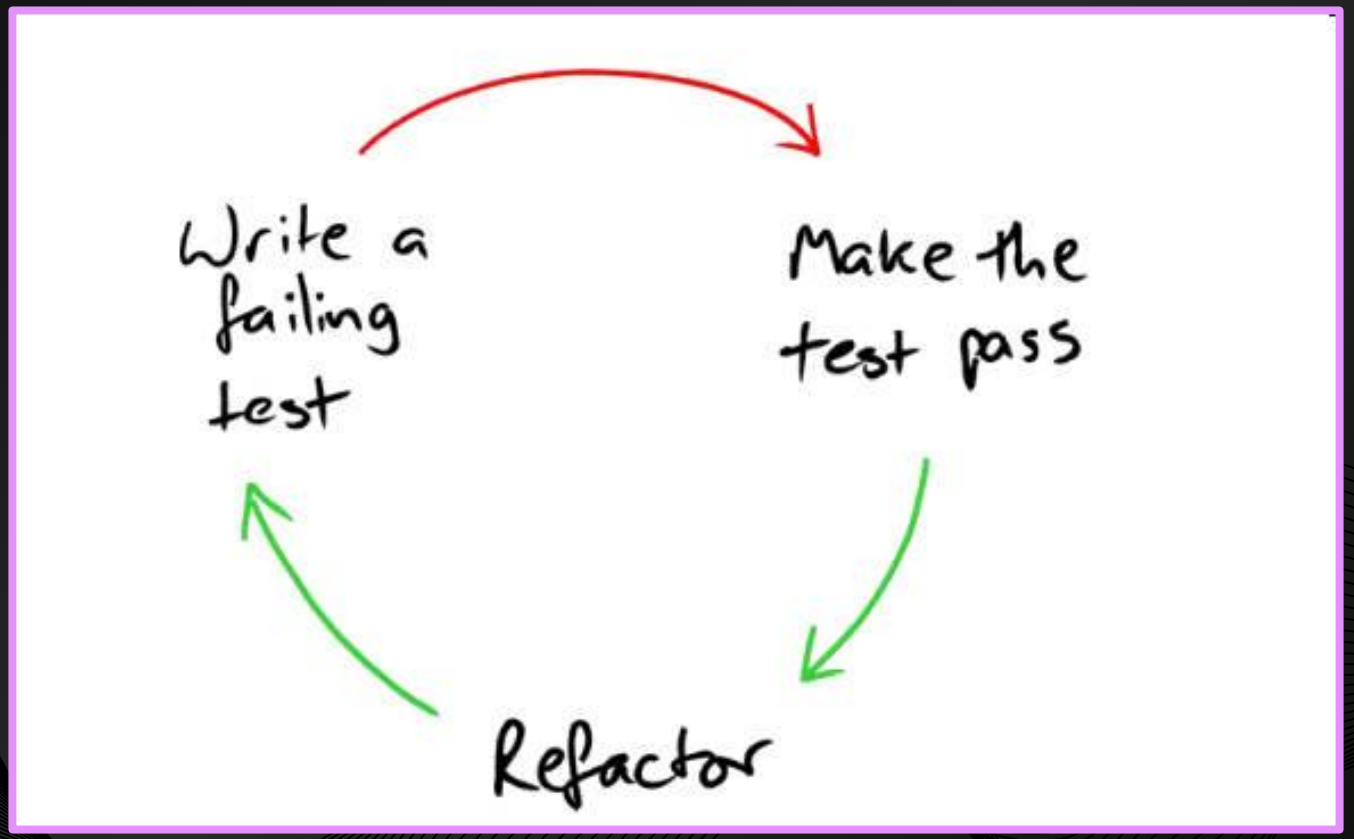
Hint: get control of your dependencies using fake objects

A bit of TDD

A bit of TDD

“Test Driven Development is a software development process relying on requirements being converted to test cases before software is fully developed.” (Wikipedia)

The TDD Mantra



Let's get practical!

Summing up

- Test one feature at time expliciting the intent
- Follow the AAA
- Keep your tests fast and isolated
- Take control of your living object
- Mock your dependencies
- Do not unit-test external services
- Refactoring is part of the process
- If it’s hard to test it could be bad design

Funny excuses

“Not all code is testable”
“The code I write is not testable”
“Writing tests slows down development and delivery”
“My tests are too slow”
…

Thank you!

Any questions?