Да, у нас есть тесты. А что?
Хитров Николай
Точка

О себе

- Пишу enterprise приложения
- Продвигаю Elixir и DDD в мире Python
- Помогаю делать конференции и митапы
- Вместо работы выступаю с докладами, как будто я что-то знаю

@nkhitrov_blog

“Да, у нас есть тесты. Мы и unit пишем, и интеграционные. И вообще мы практикуем TDD!”

(с) Какой-то Teamlead
на собеседовании

Продано! Вот моя трудовая!

ПРИКЛЮЧЕНИЯ

Ну ладно, теперь по настоящему

GEEKTRIP

Да, тесты есть, но...

Сложно ориентироваться

- По 1к+ строк кода на модуль
- Фикстур больше, чем блогерок на OnlyFans

Долго проходят

- Пока они проходили, одноклассница уже родила второго
- И еще кофе остыл

Часто падают

- Поменял строчку - упало 20 тестов
- Кнопка рестарта в CI уже как рулетка

Почему так происходит?
И что с этим можно сделать?

О чем сегодня поговорим
AAA
Структура теста
Assert
Проверки
Fixtures
Подготовка данных
Test Types
Виды тестов и их отличия
Test Doubles
Двойники для подмены логики

Начнем с основ

AAA (Arrange, Act, Assert)

- Arrange - подготовка необходимых для сценария данных
- Act — тестируемого сценария
- Assert — проверка того, что тестируемый вызов ведет себя определенным образом

```python
def test_example() -> None:
    # arrange
    user = User(...)
    role = Role(...)

    # act
    user.change_role(role)

    # assert
    assert user.role == role
```

https://github.com/jamescooke/flake8-aaa

AAA. Плохой пример

```python
def test_example(...):
    item_id = ...

    resp = client.put(f".../{item_id}/action")

    assert resp.status_code == 200
    assert resp.json() == ...

    resp = client.put(f".../{item_id}/undo")
    assert resp.status_code == 200
    assert resp.json() == ...

    item_id = ...
    resp = client.put(f".../{item_id}/action")

    assert resp.status_code == 200
    assert resp.json() == ...
```

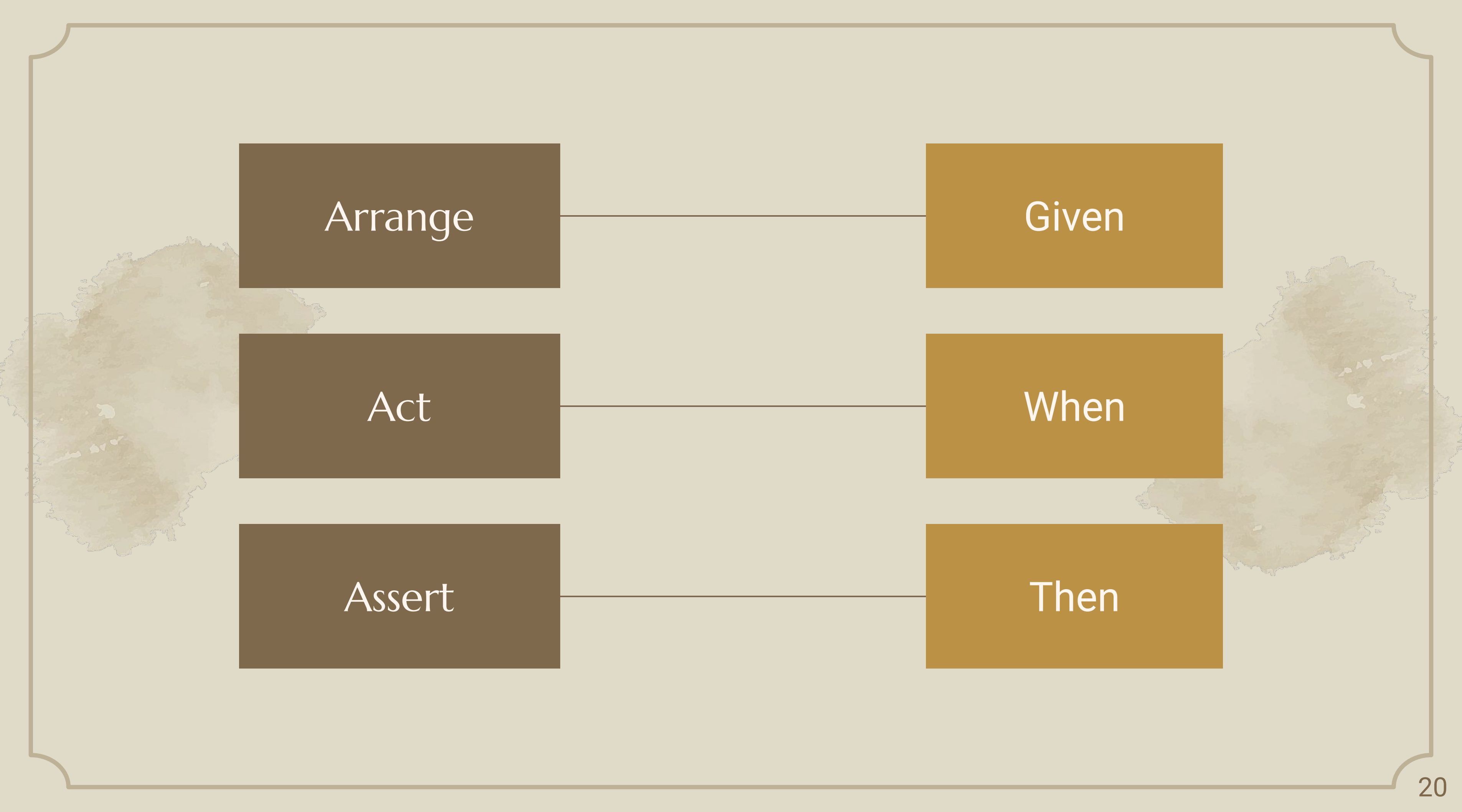
AAA. Плохой пример

```
def test_example(...):
    item_id = ...

    resp = client.put(f".../{item_id}/action")

    assert resp.status_code == 200
    assert resp.json() == ...

    resp = client.put(f".../{item_id}/undo")
    assert resp.status_code == 200
    assert resp.json() == ...

    item_id = ...
    resp = client.put(f".../{item_id}/action")

    assert resp.status_code == 200
    assert resp.json() == ...
```

AAA. Хороший пример

```
def test_...action(...):
    item_id = ...

    resp = client.put(f".../{item_id}/action")

    assert resp.status_code == 200
    assert resp.json() == ...


def test_...undo(...):
    item_id = ...

    resp = client.put(f".../{item_id}/undo")
    assert resp.status_code == 200
    assert resp.json() == ...
```

Describe
&
given-when-then


Arrange
Given
Act
When
Assert
Then

В других языках есть интересные примеры

```
describe("Transfer service", () => {
  describe("When no credit", () => {
    test("Then the response status should decline", () => {});

    test("Then it should send email to admin", () => {});
  });
});
```

```elixir
defmodule TransferServiceTest do
  use ExUnit.Case

  describe "When no credit" do
    setup do
      ...
    end

    test "then the response status should decline" do
      ...
    end

    test "then it should send email to admin" do
      ...
    end
end
```

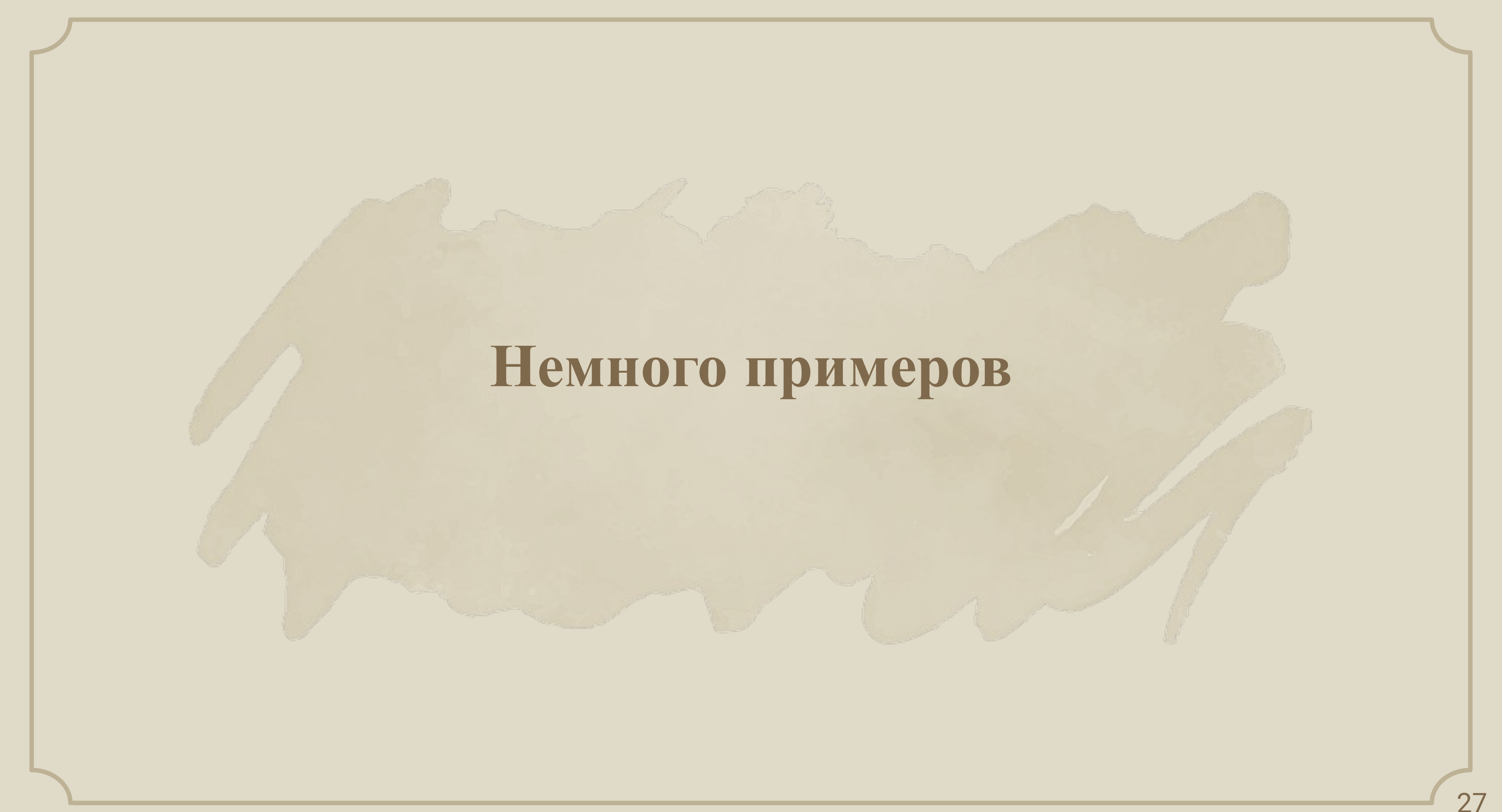
- Вложенность создает уровни и упрощает чтение/поиск
- Название объясняет **логику сценария**, а не детали **реализации**

```
class TransferService:

  def return_400_error(...):
    ...

  def send_email_if_credit_error(...):
    ...


class TransferServiceWhenNoCreditTest:

  def then_response_status_should_decline(...):
    ...

  def then_it_should_send_email_to_admin(...):
    ...
```

Assert
02

Что обычно проверяют тесты?

- **Ответ API** — Дернули **HTTP** метод, проверяем статус, тело…
- **Новое состояние** — Обновили данные в базе или в памяти
- **Внешние вызовы** — Отправили **HTTP** запрос в другой сервис
- **Очереди** — Отправили событие в шину
- **Ошибки** — Не выполнился запрос, откатили транзакцию


Немного примеров

Генерируемые поля в ответе

- Вместо подмены генераторов можно проверять значения по их типу и диапазону
- Можно проверять весь объект через ==, а не поэлементно

https://github.com/samuelcolvin/dirty-equals

```
from dirty_equals import IsNow, IsPositiveInt

def test(...):

    response = client.post(...)
    assert response.json() == {
        "id": IsPositiveInt,
        ...
        "created_at": IsNow(delta=3)
    }
```

Сложные структуры

```
fred = {'first_name': 'Fred', 'last_name': 'Smith'}
bob = {'first_name': 'Bob', 'last_name': 'Barr'}
people = [fred, bob]

assert_that(people).extracting('first_name').is_equal_to(['Fred','Bob'])
assert_that(people).extracting('first_name').contains('Fred','Bob')
```

- Декларативность вместо list comprehension
- Можно добавлять свои обработчики

https://github.com/assertpy/assertpy
https://github.com/hamcrest/PyHamcrest

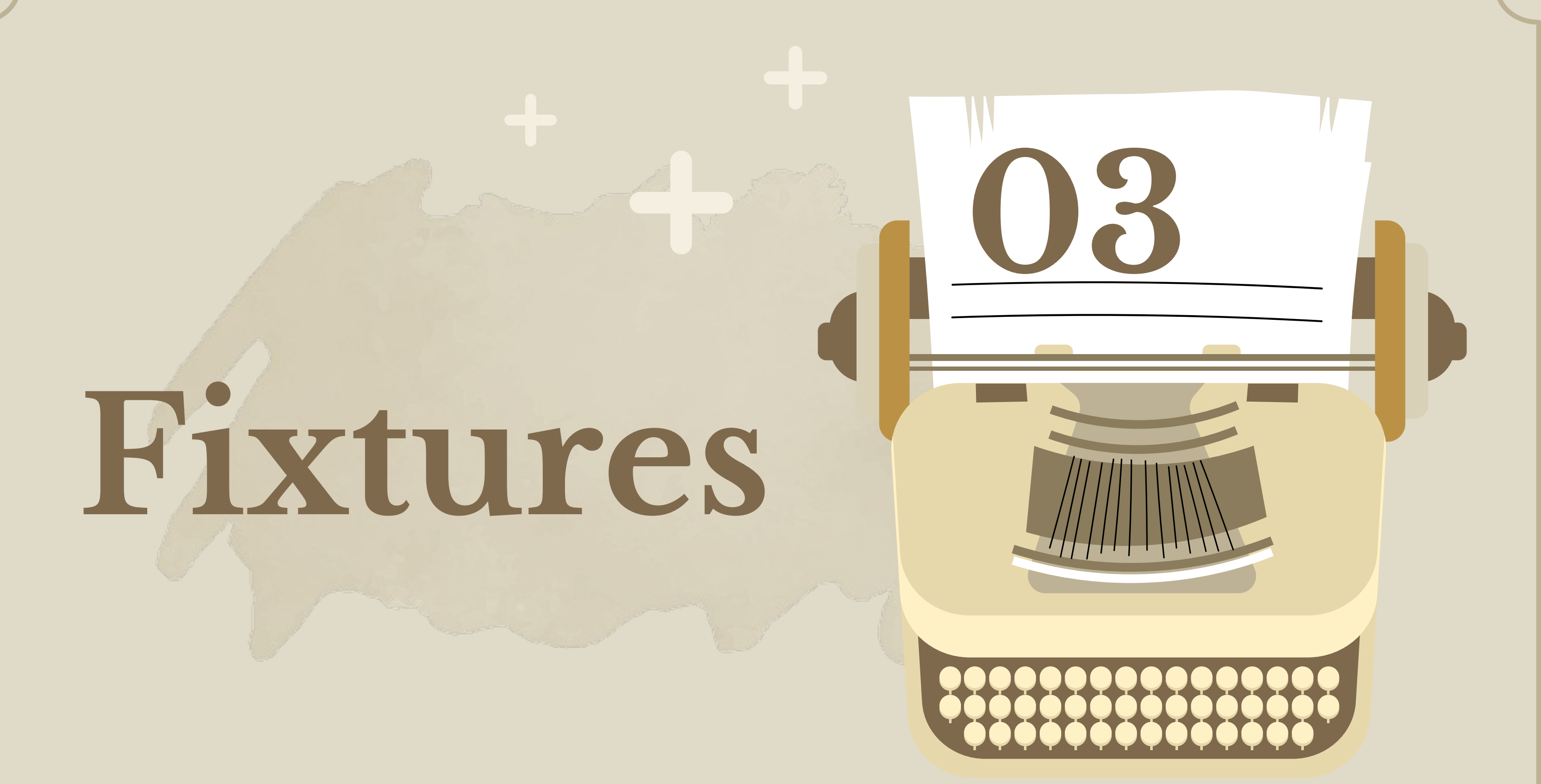
Snapshot asserts

```python
def test_syrupy(snapshot):
    actual = "Some computed value!"
    assert actual == snapshot
```

https://github.com/tophat/syrupy

```python
def test_assertpy():
    actual = "Some computed value!"
    assert_that(actual).snapshot()
```

https://github.com/assertpy/assertpy#snapshot-testing



Fixtures

Рассмотрим простой пример

- Простой класс с бизнес логикой
- Без какой-либо **IO** обвязки
- Без каких-либо ресурсов, которые надо **открыть-закрыть**

```
class SomeObject:

    def do_something():
        ...
        return ...
```

1

```
with SomeObject() as some_object:
    some_object.do_something()
```

2

```
some_object = SomeObject()
some_object.do_something()
```

1

```python
with SomeObject() as some_object:
    some_object.do_something()
```

2

```python
some_object = SomeObject()
some_object.do_something()
```

Поднимете руки, кто за вариант **1**

1

```
with SomeObject() as some_object:
    some_object.do_something()
```

2

```
some_object = SomeObject()
some_object.do_something()
```

А теперь поднимете руки, кто за вариант **2**

1

```
with SomeObject() as some_object:
    some_object.do_something()
```

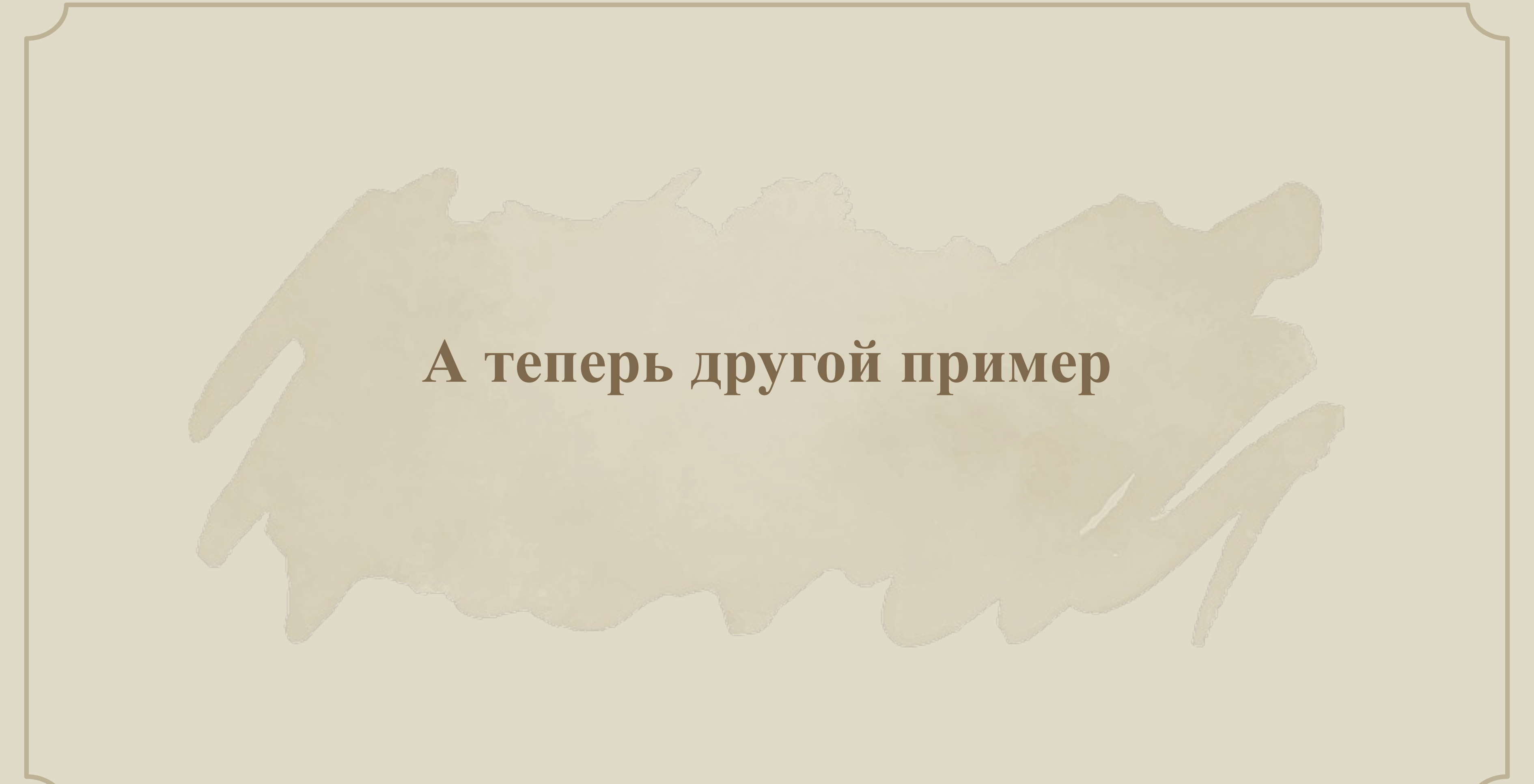
Вряд ли кто-то использует **with** без причины. Верно же?

2

```
some_object = SomeObject()
some_object.do_something()
```

~~Поднимете руки, кто за вариант 1~~

~~А теперь поднимете руки, кто за вариант 2~~


А теперь другой пример

1

```python
@pytest.fixture
def some_fixture():
    return SomeObject()


def test_something(some_fixture):
    ...
```

2

```python
def test_something():

    some_object = SomeObject()
```

1

```python
@pytest.fixture
def some_fixture():
    return SomeObject()


def test_something(some_fixture):
    ...
```

2

```python
def test_something():

    some_object = SomeObject()
```

Поднимете руки, кто за вариант **1**

1

```python
@pytest.fixture
def some_fixture():
    return SomeObject()


def test_something(some_fixture):
    ...
```

2

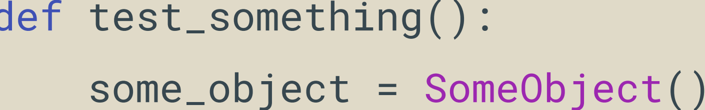
```python
def test_something():

    some_object = SomeObject()
```

~~Поднимете руки, кто за вариант 1~~

А теперь поднимите руки, кто за вариант **2**

```
@pytest.fixture
def some_fixture():
    return SomeObject()


def test_something(some_fixture):
    ...
```

2



```
def test_something():

    some_object = SomeObject()
```

~~Поднимете руки, кто за вариант 1~~

~~А теперь поднимите руки, кто за вариант 2~~

Какие проблемы с таким подходом

- Невозможно прокинуть параметры **из** теста **в** фикстуру без приседаний
- Фикстуры в фикстурах заставляют прибегать к анальной эквилибристике: потоком (flow) управляет фикстура, а не сам тест
- Иногда это приводит к размножению однотипных фикстур с небольшими отличиями или передаче Callable (но зачем?)
- Фикстуры остаются в памяти до завершения, а значит расходуется больше ресурсов (боль в больших проектах)
  https://github.com/pytest-dev/pytest/issues/5642

Тогда когда использовать фикстуры?

- Когда нужно подготовить и после подчистить ресурсы:
  - Внешние (база, кеш)
  - Внутренние (in-memory очереди)
- Когда нужно переопределять фикстуру (event_loop из pytest-asyncio)
- Когда нужно создавать разные версии клиентов (через request.param)

Инструменты

Просмотр графа использованных фикстур и поиск дублей

https://github.com/pytest-dev/pytest-fixture-tools

Фикстуры в виде классов. Куча параметров-фикстур превращается в кучу атрибутов

https://github.com/zmievsa/pytest-fixture-classes

Поиск неиспользованных фикстур. Не забывайте, что надо вызывать команду отдельно!

https://github.com/jllorencetti/pytest-deadfixtures

Flaky Tests

```python
def test_flaky():
    num = random.randint(1,2)
    assert num == 2


def test_default():
    assert 2 == 2
```

```
==================================================================== FAILURES ====
________________________________________________________________ test_flaky[0] _

    def test_flaky():
        num = random.randint(1,2)
>       assert num == 2
E       assert 1 == 2

test_api.py:21: AssertionError
________________________________________________________________ test_flaky[1] _

    def test_flaky():
        num = random.randint(1,2)
>       assert num == 2
E       assert 1 == 2

test_api.py:21: AssertionError
________________________________________________________________ test_flaky[2] _
```

Плагины для борьбы с flaky тестами

Бьем по рукам, если тест поменял значения в os.environ

https://github.com/wemake-services/pytest-modified-env

Вносим хаос в порядок запуска тестов

https://github.com/pytest-dev/pytest-randomly

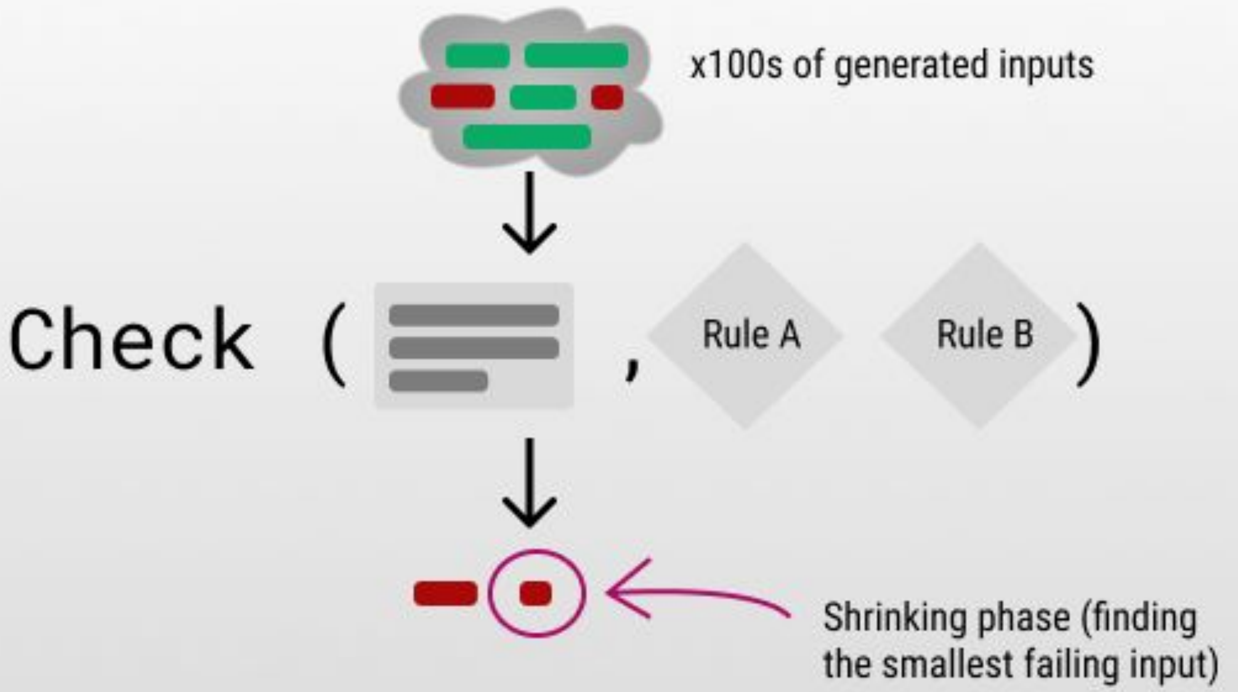
Запускает 100500 раз, чтобы отыскать flaky

https://github.com/dropbox/pytest-flakefinder

Сами говорим, сколько раз хотим запустить тест (для дебага)

https://github.com/pytest-dev/pytest-repeat

Property based


Property-based Testing
x100s of generated inputs
Check
Rule A
Rule B
Shrinking phase (finding the smallest failing input)

```python
def test_example(faker: Faker) -> None:
    name = faker.name()
    user = User(name)

    ...
```

Инструменты (eaze level)

Генерим циферки, буковки, объекты и все такое. Стандарт де-факто

https://github.com/pytest-dev/pytest-faker

Те же генераторы, только в профиль

https://github.com/pytest-dev/pytest-mimesis

Инструменты (hard level)

Генерим тесты для функций на основе указаний контрактов в функциях

https://github.com/life4/deal

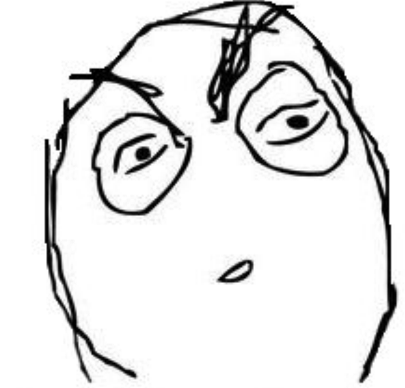
Тоже генерим тесты, но уже указываем в тесте, что хотим

https://github.com/HypothesisWorks/hypothesis

Тот же hypothesis, но настроенный для проверки API

https://github.com/schemathesis/schemathesis + workshop

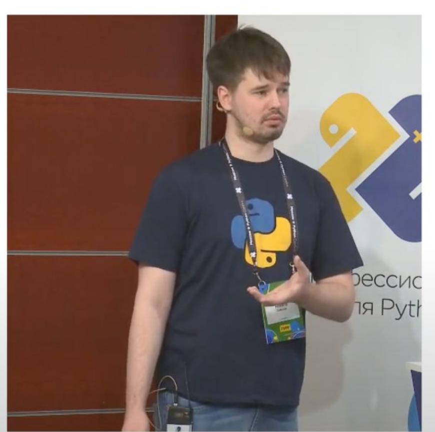
Mutation tests



Убивай мутантов, спаси свой код
Никита Соболев (wemake.services)

```python
from source import WRONG_LETTERS, is_wrong_letter


@pytest.mark.parametrize('letter', WRONG_LETTERS)
def test_is_wrong_letter(letter):
    assert is_wrong_letter(letter) is True
```

85



https://github.com/boxed/mutmut

Кто видит здесь проблему? (c)

Unit vs integration

04

Unit test vs. Integration test

В чем между ними разница?

8.1. Что такое интеграционный тест? 219

8.1. Что такое интеграционный тест?

Интеграционные тесты играют важную роль в проекте. Также важно иметь сбалансированное количество юнит- и интеграционных тестов. Вскоре вы узнаете, что это за роль и как выдерживать этот баланс, но сначала давайте вспомним, чем интеграционные тесты отличаются от юнит-тестов.

8.1.1. Роль интеграционных тестов

Как говорилось в главе 2, юнит-тест удовлетворяет следующим трем требованиям:

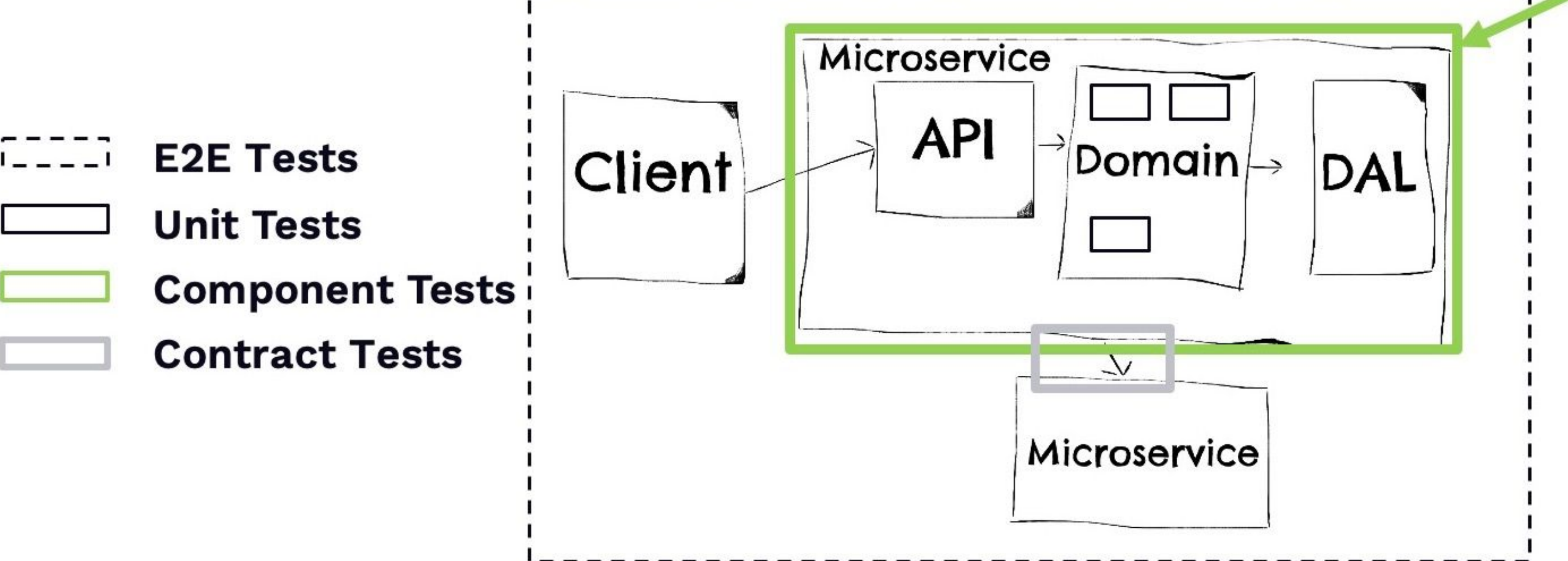
- проверяет правильность работы одной единицы поведения;
- делает это быстро
- и в изоляции от других тестов.

Тест, который не удовлетворяет хотя бы одному из этих трех требований, относится к категории интеграционных тестов. Таким образом, *интеграционным* оказывается любой тест, не являющийся юнит-тестом.

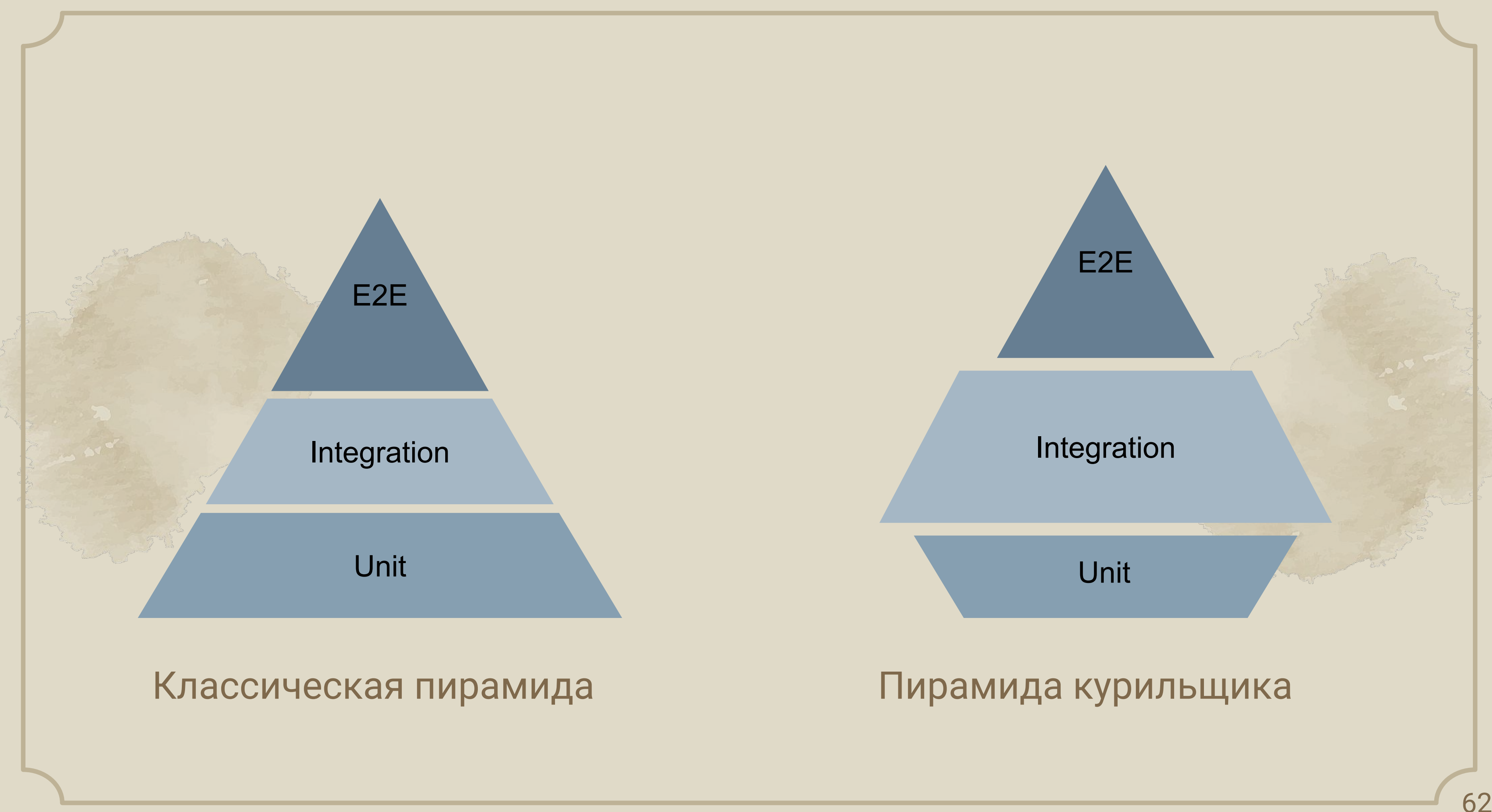
На практике интеграционные тесты почти всегда проверяют, как ваша система работает в интеграции с внепроцессными зависимостями. Другими словами, эти тесты покрывают код из четверти контроллеров (за информацией о классификации кода обращайтесь к главе 7). Диаграмма на рис. 8.1 представляет типичные обязанности юнит- и интеграционных тестов. Юнит-тесты покрывают доменную модель (модель предметной области), тогда как интеграционные тесты проверяют код, связывающий доменную модель с внепроцессными зависимостями.

Рис. 8.1. Интеграционные тесты покрывают контроллеры, а юнит-тесты покрывают модель предметной области и алгоритмы. Тривиальный и переусложненный коды тестироваться не должны

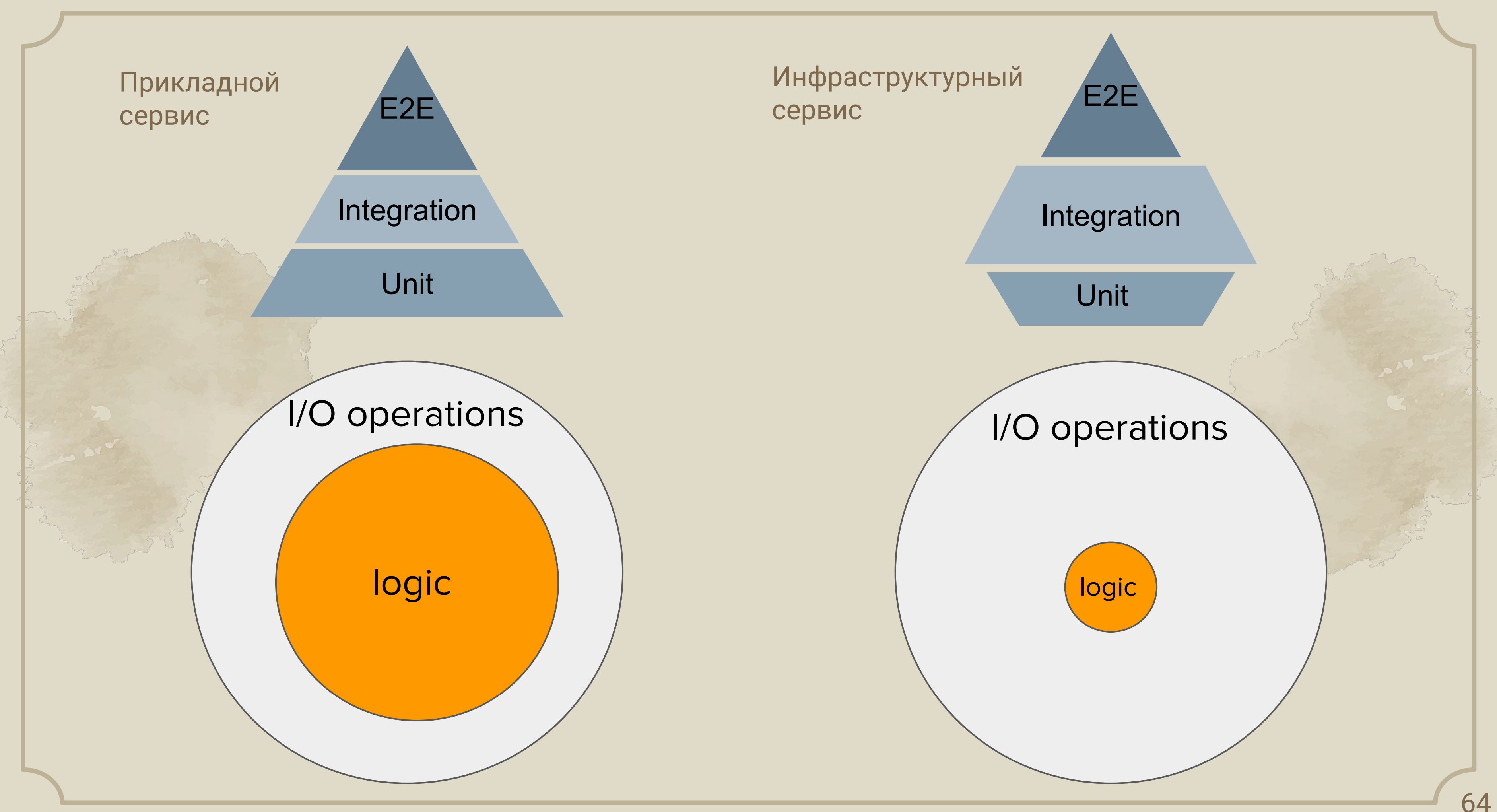
Integration Component Tests

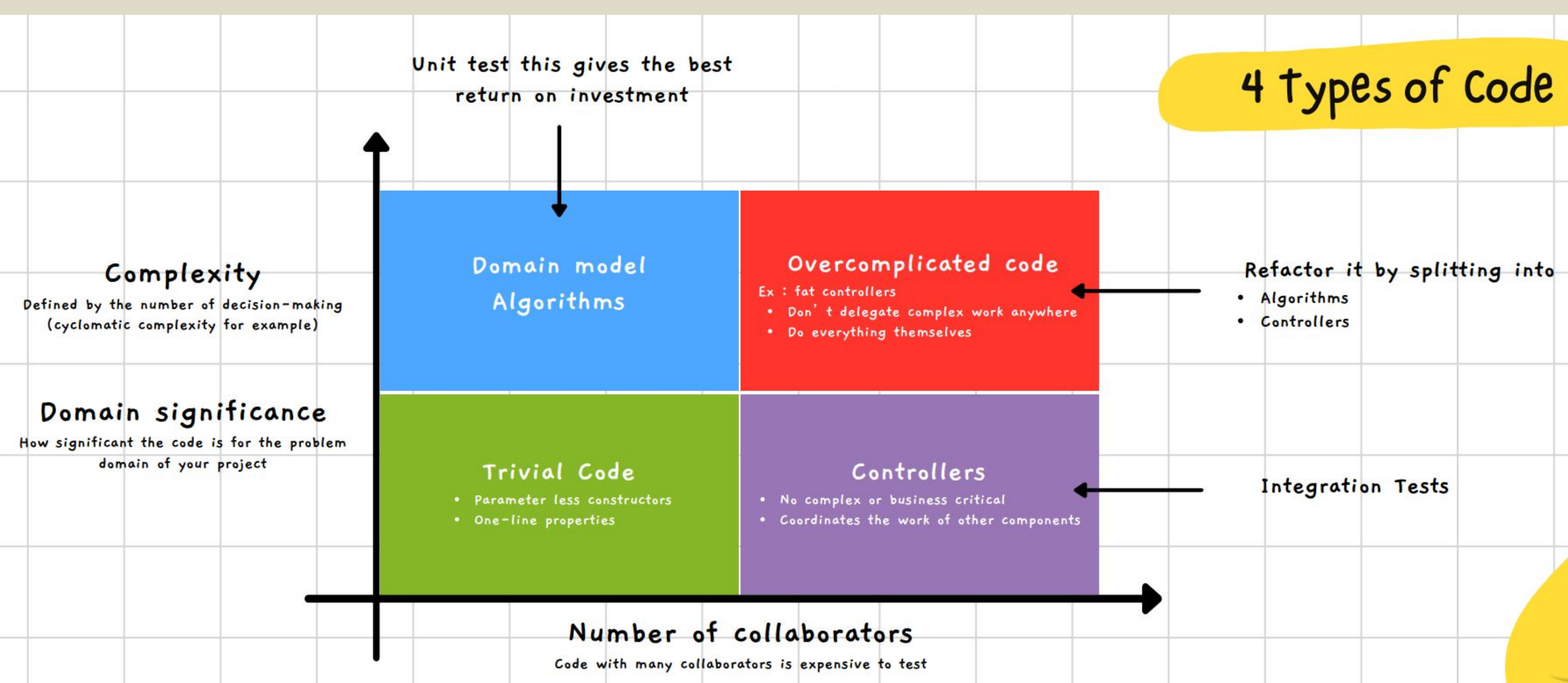


Каких тестов должно быть больше?


E2E
Integration
Unit
Классическая пирамида
E2E
Integration
Unit
Пирамида курильщика

Но как всегда, единственно верного ответа не существует


Прикладной
сервис
E2E
Integration
Unit
I/O operations
logic
Инфраструктурный
сервис
E2E
Integration
Unit
I/O operations
logic


4 Types of Code
Unit test this gives the best return on investment
Complexity
Defined by the number of decision-making (cyclomatic complexity for example)
Domain significance
How significant the code is for the problem domain of your project
Domain model
Algorithms
Overcomplicated code
Ex : fat controllers
• Don't delegate complex work anywhere
• Do everything themselves
Trivial Code
• Parameter less constructors
• One-line properties
Controllers
• No complex or business critical
• Coordinates the work of other components
Refactor it by splitting into
• Algorithms
• Controllers
Integration Tests
Number of collaborators
Code with many collaborators is expensive to test

Ответы на частые вопросы про тесты

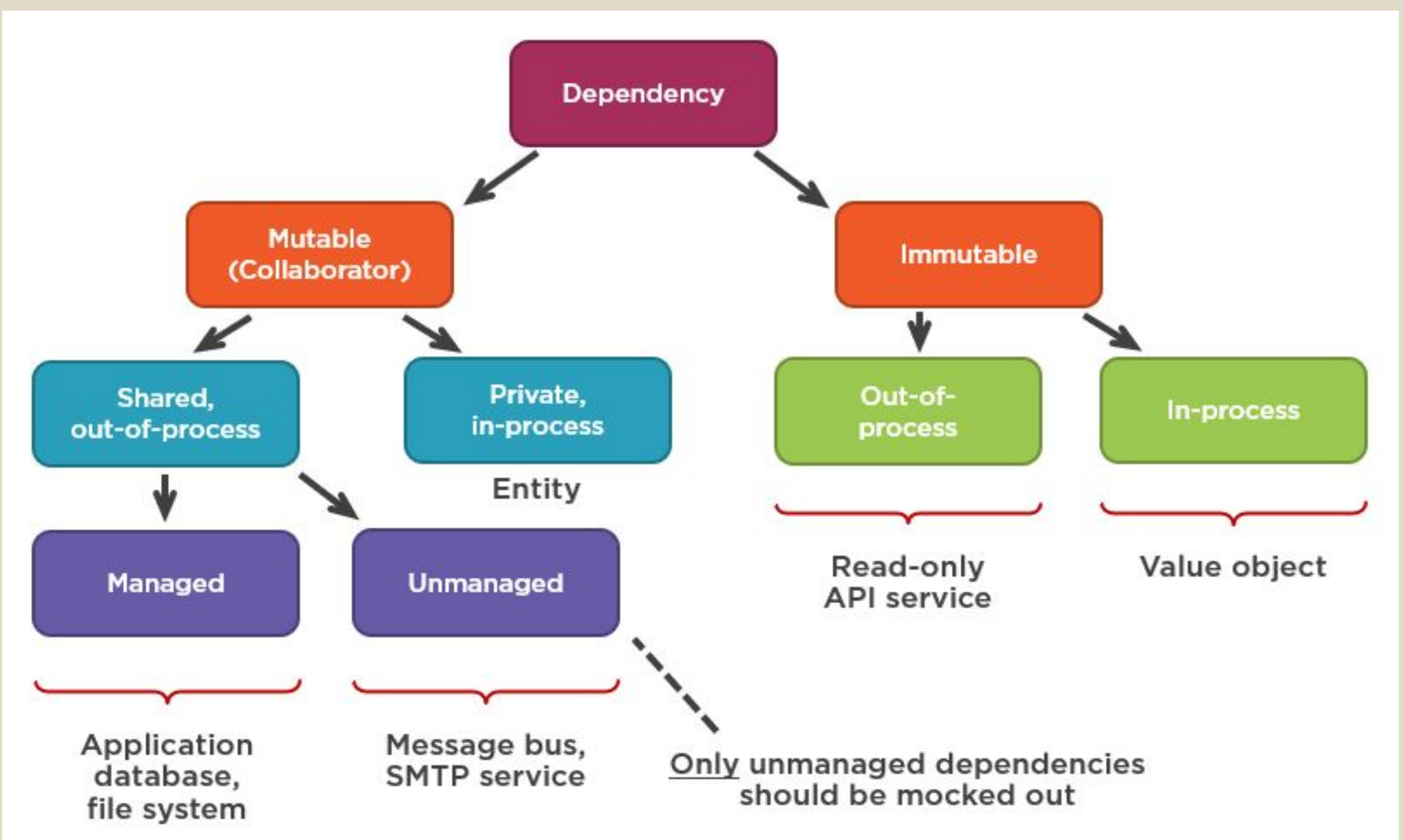
- Чем сложнее будет бизнес логика, тем больше вы захотите затащить BDD

- Если логика простая или инвариантов мало, можно писать как обычно
- Интеграционные тесты дешевле в плане поддержки, т.к. они покрывают фичи, а не функции/классы
- В новом проекте проще начинать с интеграционных тестов
- Юнит тесты стоит писать ради ускорения на чистые (без IO) части системы

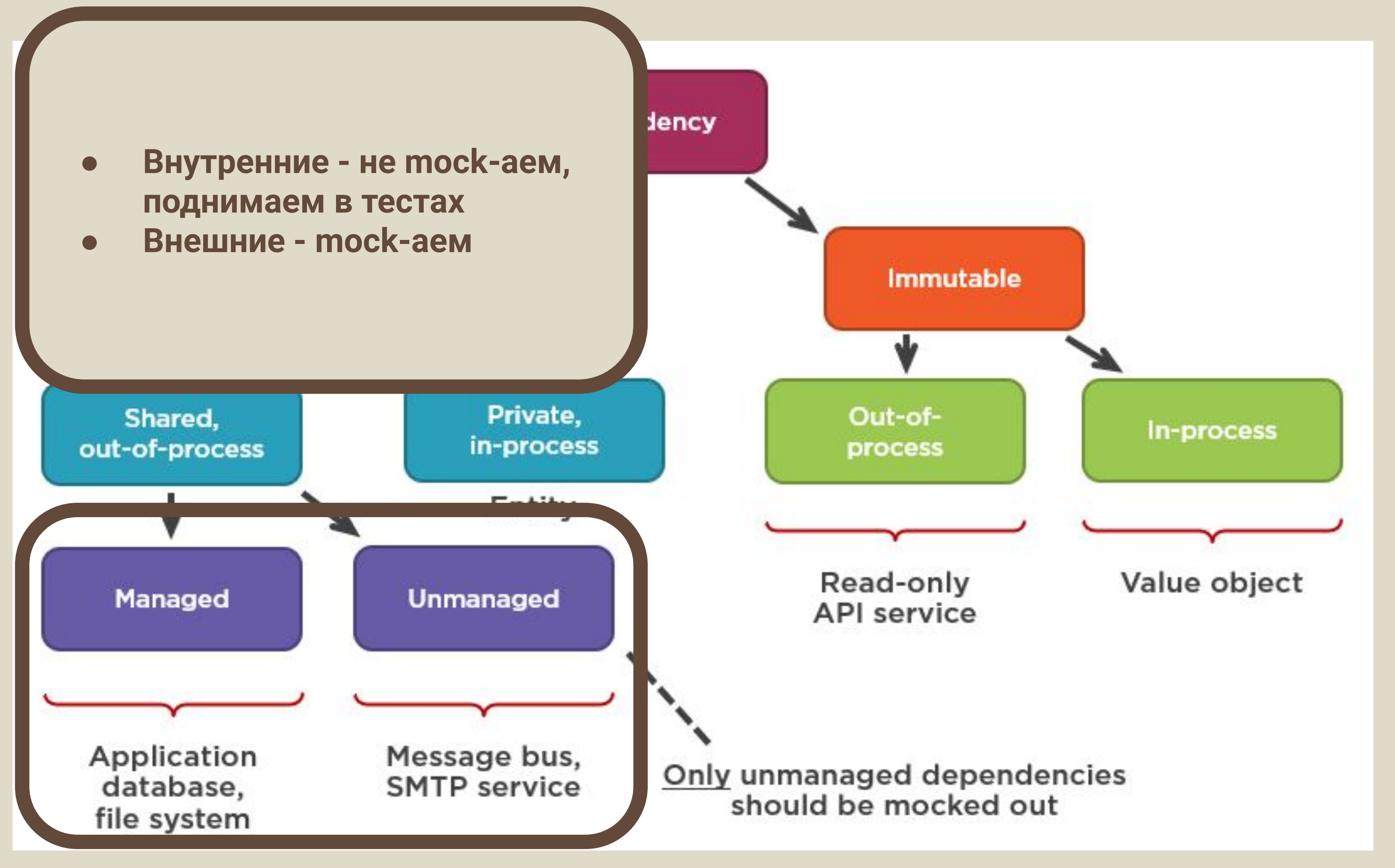
Подождите, но в этом списке нет ответа на самый главный вопрос!

Подождите, но в этом списке нет ответа на самый главный вопрос!

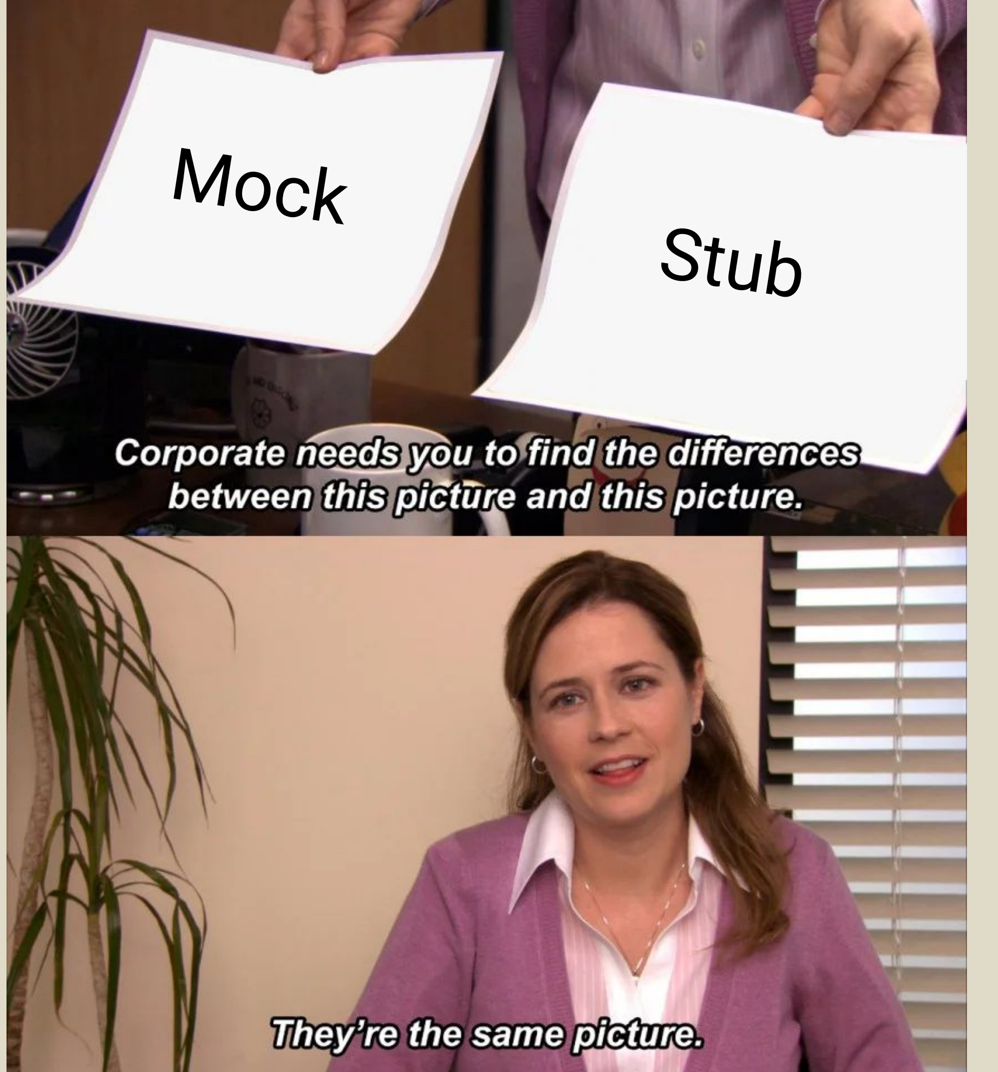
А базу то можно mock-ать или как?

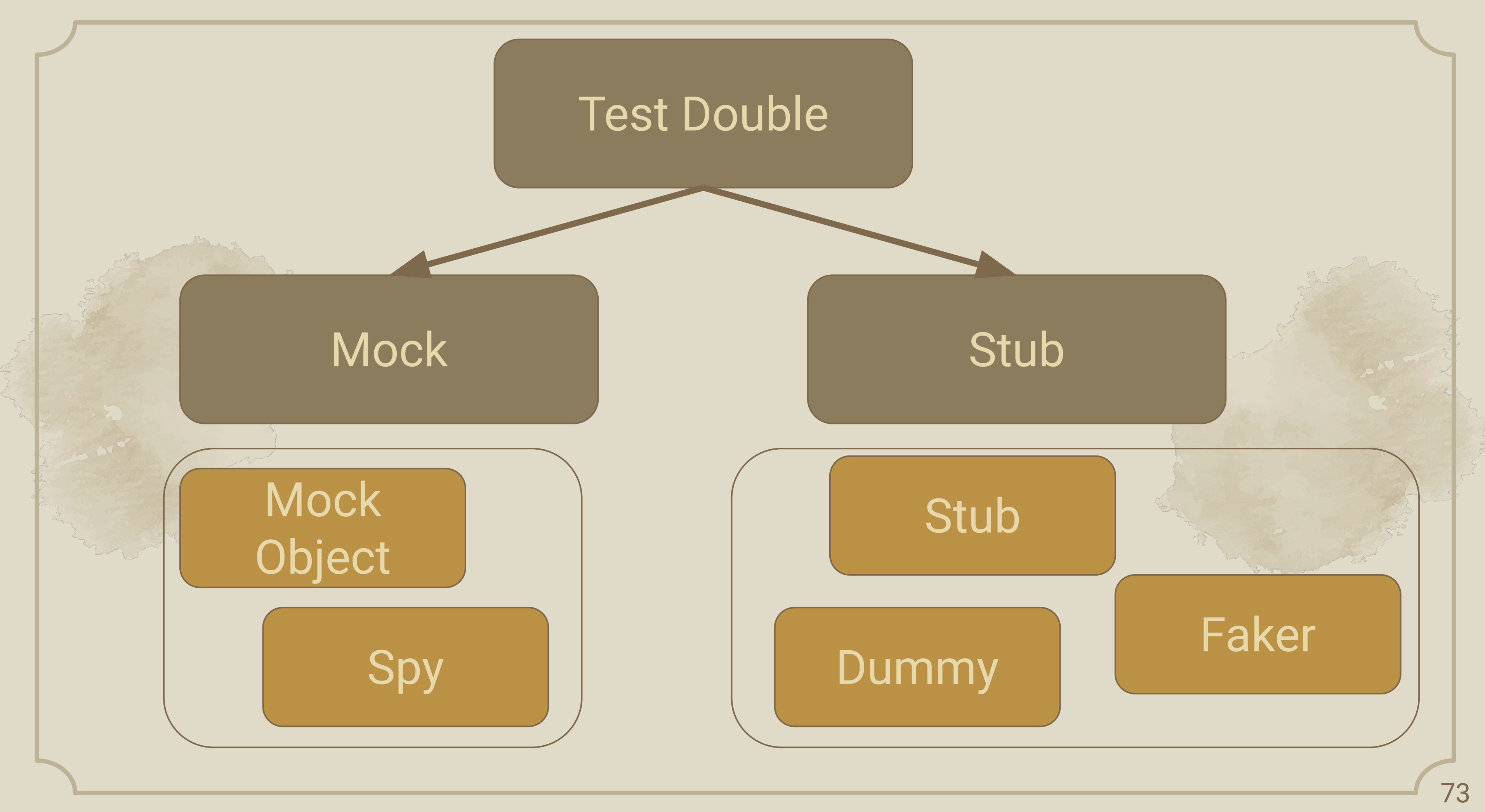

Dependency
Mutable
(Collaborator)
Immutable
Shared,
out-of-process
Private,
in-process
Out-of-
process
In-process
Entity
Managed
Unmanaged
Read-only
API service
Value object
Application
database,
file system
Message bus,
SMTP service
Only unmanaged dependencies
should be mocked out

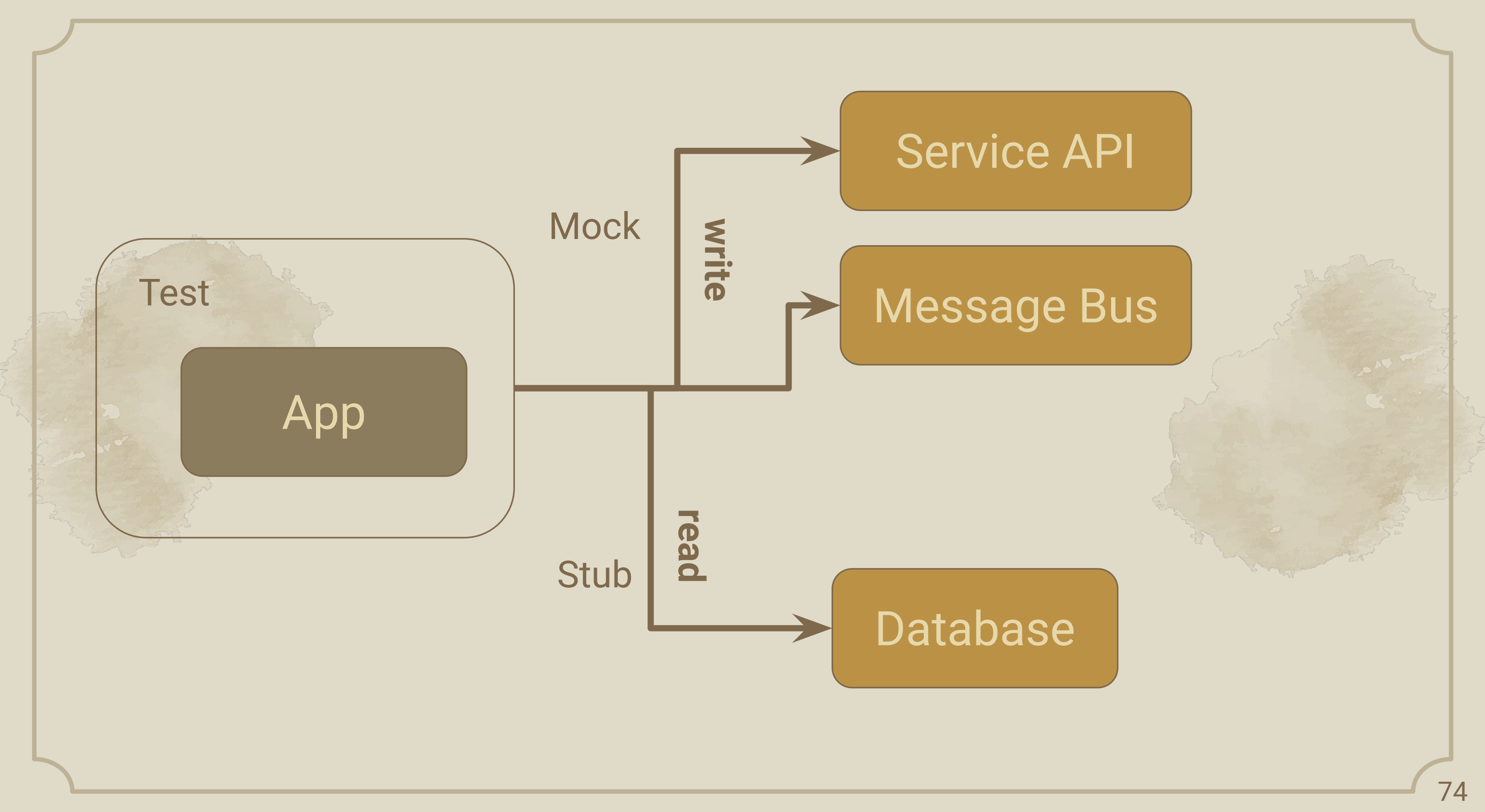
- **Внутренние - не mock-аем, поднимаем в тестах**
- **Внешние - mock-аем**



05
Test Doubles


Mock
Stub
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.


Test Double
Mock
Stub
Mock Object
Spy
Stub
Dummy
Faker


Service API
Message Bus
Mock
write
Test
App
read
Stub
Database

Ну это понятно,
а разница то в чем?

Stub

- Отдает заготовленные ответы вместо вызова реального запроса
- Обычно простой и тупой объект, отдает хардкод или параметр, заданный из теста

```
class ClientStub:

    def get_something(...):
        return {"value": 42}
```

Spy

- Не подменяет, а дополняет логику объекта
- Отслеживает, как был вызван тестируемый объект. Например, сколько раз, с какими данными

```
def test_example(caplog):
    assert 'warning text' in caplog.text
```

Faker

- “Умный”, есть своя реализация
- Подменяет поведение
- Есть интерфейс, чтобы проверить данные
- Иногда можно использовать как более быструю замену долгим объектам (in memory база и пр.)

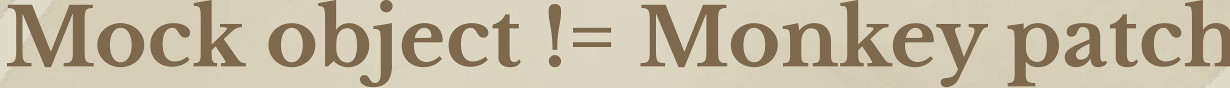
```
class FakeSMTPClient(Client):

    def send_message(...):
        ...
        self.messages.append(msg)


def test_example(fake_smtp_client, ...):
    response = ...

    assert response ...

    assert fake_smtp_client.messages == [...]
```

Mock Object

- Простой объект
- Не имеет реализации
- Не наследуется от кода приложения
- Проверяет что и как именно было вызвано
- Без проверок на вызовы можно использовать и как stub

```
mock = Mock(name='Thing', return_value=None)
mock(1, 2, 3)
mock.assert_called_once_with(1, 2, 3)
mock(1, 2, 3)
mock.assert_called_once_with(1, 2, 3)
```

https://docs.python.org/3/library/unittest.mock.html#create-autospec
https://docs.python.org/3/library/unittest.mock.html#autospeccing
https://hynek.me/articles/what-to-mock-in-5-mins/


Mock object != Monkey patch

Mock Action (monkey patch)

- Подменяет объект глобально
- Есть несколько нюансов с импортами, которые не так очевидны
- Сильно привязывает тест к деталям реализации
- Зато работает с любым “грязным” кодом

```
# product.py
from os import listdir

def my_function():
    files = listdir(some_directory)
    # ... use the file names ...

# test.py
def test_it():
    with mock.patch("product.listdir") as listdir:
        listdir.return_value = ['a.txt', 'b.txt']
        my_function()
```

https://nedbatchelder.com/blog/201908/why_your_mock_doesnt_work.html#h_mock_it_where_its_used
https://nedbatchelder.com/blog/201206/tldw_stop_mocking_start_testing.html

Performance & Memory

15 jobs for `master` in 36 minutes and 23 seconds (queued for 2 seconds)

`8557d956`

No related merge requests found.

83

No related merge requests found.

84

Инструменты (perfomance)

Раскидываем тесты по воркерам в CI
https://docs.gitlab.com/ee/ci/yaml/#parallel
https://github.com/jerry-git/pytest-split

Запускаем тесты на измененный код (чекаем по coverage)
https://github.com/tarpas/pytest-testmon
https://github.com/joeyespo/pytest-watch

Тайминги на тесты и другие этапы
https://github.com/pytest-dev/pytest-timeout
https://github.com/Scony/pytest-timeouts

Отчет по времени выполнения
https://github.com/miketheman/pytest-execution-timer
https://github.com/koaning/pytest-duration-insights

Метрики по ЦПУ, ОЗУ с экспортом в sqlite или на сервер
https://github.com/CFMTech/pytest-monitor

Нашумевший memray для pytest
https://github.com/bloomberg/pytest-memray

Упомянутая раннее либа для контрактов. Вроде как умеет в поиск утечек
https://deal.readthedocs.io/details/tests.html?highlight=leak#finding-memory-leaks

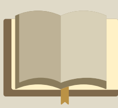
Старая либа. Требует дебаг версии интерпретатора, но пусть будет
https://github.com/abalkin/pytest-leaks

И тут пора напомнить, что у любой истории есть конец

fin.

Сюда сказать автору, где он не прав
@nkhitrov

Сюда лайк, подписка, репост
@nkhitrov_blog



Полезные ссылки

P.S. А мы с Денисом пока ответим на ваши вопросы