BDD

How to make it work?

Sebastian Buczyński @ EuroPython 2023

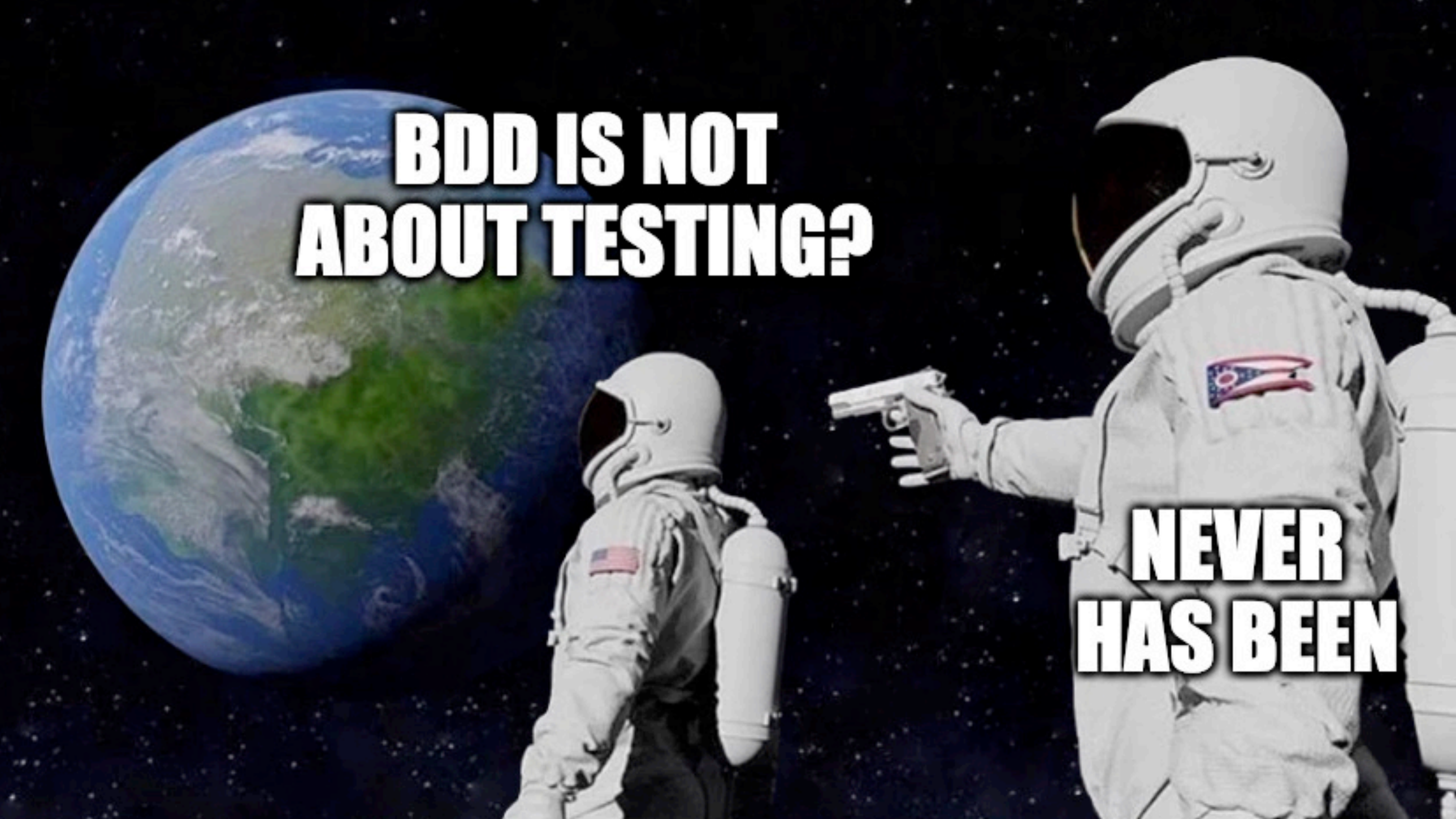
BDD, wow

```
Feature: Subscription Management

  Scenario: Free trial users buys monthly subscription
    Given a user with an expired free trial
    When user purchases monthly subscription
    Then purchase is confirmed
    And the account is upgraded to a monthly subscription
```

BDD, wow

```
from pytest_bdd import scenario, given, when, then


@given("a user with an expired free trial")
def user_with_expired_free_trial():
    pass  # Implement logic here


@then("purchase is confirmed")
def purchase_is_confirmed():
    pass  # Implement logic here
```

```
pip install pytest-bdd
# or
pip install pytest-bdd-ng
# or
pip install behave
```


BDD IS NOT
ABOUT TESTING?
NEVER
HAS BEEN


HOTEL
HOTEL
ALICJA


HOTEL
HOTEL
ALICJA

whoami

Sebatian Buczyński

- Trainer / Consultant @ Bottega IT Minds
- TL @ Sauce Labs
- Evangelist of software engineering in Python
  - Substack - pythoneer.substack.com
- Apprentice of alternative coffee brewing methods

BDD

B is for behaviour

behaviour - whatever is visible on the outside

Behaviour DOs

- ✅ A result can be seen by the user (e.g. on UI/API, report)
- ✅ ...
- ✅ ...
- ✅ ...

```
Scenario: Registered user logs in
    Given user registered with username "foo" and password "bar"
    When user logs in with username "foo" and password "bar"
    Then user is authenticated
```

Behaviour DOs

- ✅ A result can be seen by the user (e.g. on UI/API, report)
- ✅ How it affects other features?
- ✅ ...
- ✅ ...

Behaviour DOs

- ✅ A result can be seen by the user (e.g. on UI/API, report)
- ✅ How it affects other features?
- ✅ Is there some new action available for the user?
- ✅ ...

Behaviour DOs

- ✅ A result can be seen by the user (e.g. on UI/API, report)
- ✅ How it affects other features?
- ✅ Is there some new action available for the user?
- ✅ Is there something a user no longer can do?

```
Scenario: Login pattern is banned
  Given "baz*" is banned
  When user registers as "baz astral"
  Then registration attempt is rejected
   And "baz astral" cannot login
```

Behaviour DONTs

- 🚫 Record has been saved in the database
- 🚫 …
- 🚫 …
- 🚫 …

Behaviour DONTs

- Record has been saved in the database
- Overspecification with mocks (Stub versus Mock)
- ...
- ...

```
def test_with_mock_that_should_be_stub():
    external_system_mock.return_value = User( ... )

    ...

    assert response = { ... }  # some User data here
    external_system_mock.assert_called_once_with( ... )
```

Behaviour DONTs

- 🚫 Record has been saved in the database
- 🚫 Overspecification with mocks (Stub versus Mock)
- 🚫 Checking anything that’s not part of the „public API”
- 🚫 …

Behaviour DONTs

- Record has been saved in the database
- Overspecification with mocks (Stub versus Mock)
- Checking anything that's not part of the „public API"
- Avoid technical details (focus on *what,* not *how*)

How to BDD

1. Name the expected behaviour

Gherkin

Human-readable, executable specification

Gherkin is meant to be READABLE

for humans, not [only] computers

```
Scenario: Registered user gets banned
   Given user registered with username "Bob"
     And "Bob" is banned
    When "Bob" logs in
    Then login attempt is rejected
```

```
Scenario: Registered user gets banned
   Given I am logged in as "admin"
   And I am on the "admin" page
   When I follow "Users"
   And I follow "Edit" for user "user"
   And I check "Banned"
   And I press "Update"
   Then I should see "User was successfully updated."
   And I should see "Banned: true"
   And I should see "admin"
   And I should see "user"
```

Writing Gherkin DOs

- ✅ Collaborative way (e.g. Three Amigos - Dev, Tester, PM)
- ✅ ...
- ✅ ...

Writing Gherkin DOs

- ✅ Collaborative way (e.g. Three Amigos - Dev, Tester, PM)
- ✅ Pairing
- ✅ ...

Writing Gherkin DOs

- ✅ Collaborative way (e.g. Three Amigos - Dev, Tester, PM)
- ✅ Pairing
- ✅ Reviewing Gherkin with stakeholders

Writing Gherkin DONTs

- 🚫 Tester / QA department working separately
- 🚫 ...

Writing Gherkin DONTs

- 🚫 Tester / QA department working separately
- 🚫 Business analysts working separately and handing stuff over to teams

Building shared understanding

Between stakeholders and developers


MIND THE GAP


OPEN TO WORK

Shared understanding

AKA Ubiquitous language

Readable Gherkin DOs

- ✅ Use realistic, common scenarios
- ✅ ...
- ✅ ...


BDD
ALL THE THINGS
imgflip.com

Readable Gherkin DOs

- ✅ Use realistic, common scenarios
- ✅ Be specific, avoid expressions like <= 10
- ✅ ...

Specification

Readable Gherkin DOs

- ✅ Use realistic, common scenarios
- ✅ Be specific, avoid expressions like <= 10
- ✅ Put in the spec only what matters [avoid irrelevant information]

```
Feature: Authentication based on username and password
 Scenario: Registered user logs in
    Given user registered with username "foo" and password "bar"
    When user logs in with username "foo" and password "bar"
    Then user is authenticated
```

```
Feature: Banning users based on specific logins and patterns
 Scenario: Registered user gets banned
    Given user registered with username "Bob"
      And "Bob" is banned
     When "Bob" logs in
     Then login attempt is rejected
```

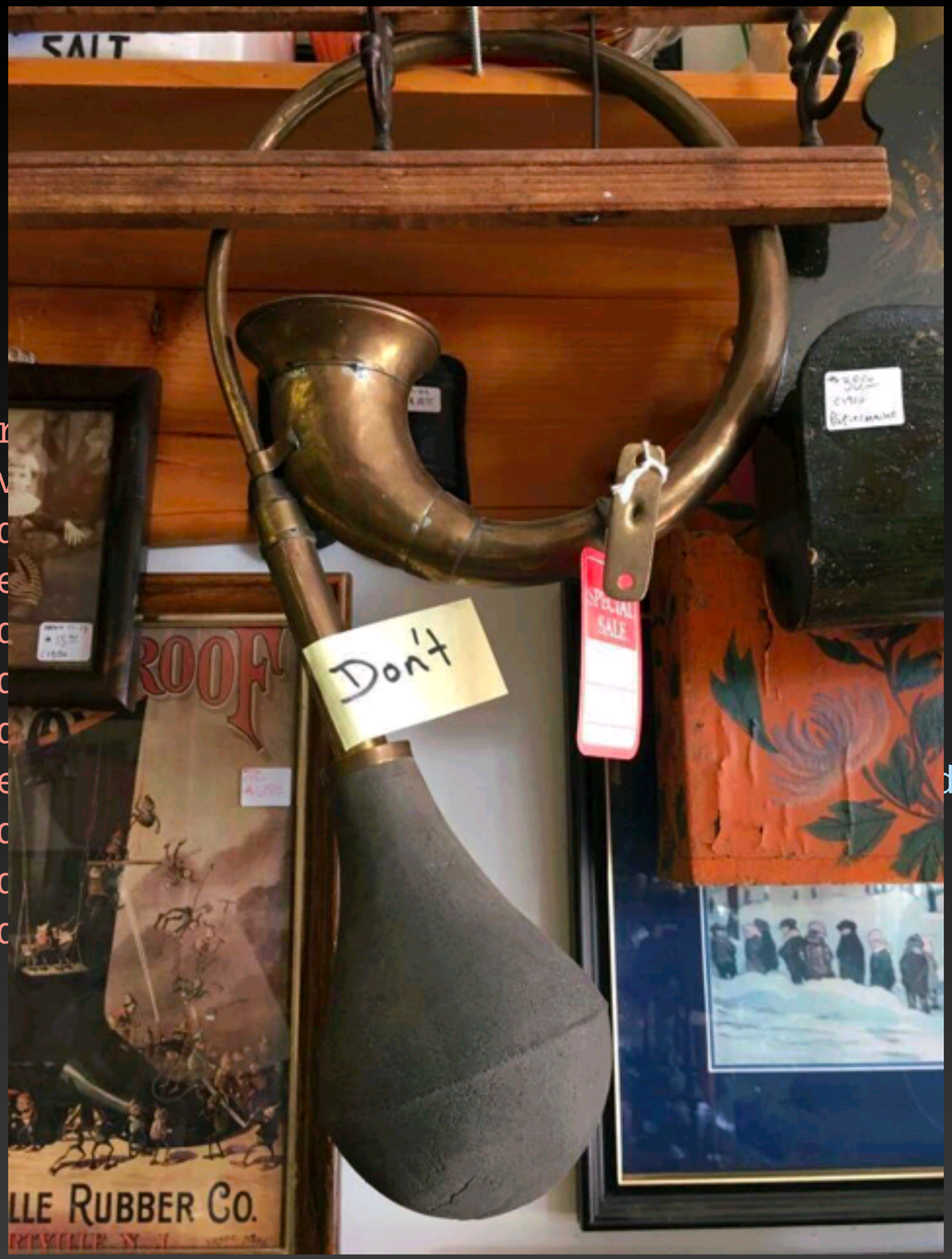
Readable Gherkin DONTs

- 🚫 Don’t write scripts in Gherkin
- 🚫 …
- 🚫 …

Avoid scripts

```
Scenario: Registered user gets banned
   Given I am logged in as "admin"
   And I am on the "admin" page
   When I follow "Users"
   And I follow "Edit" for user "user"
   And I check "Banned"
   And I press "Update"
   Then I should see "User was successfully updated."
   And I should see "Banned: true"
   And I should see "admin"
   And I should see "user"
```

Scenar
Giv
Anc
Whe
Anc
Anc
Anc
The
Anc
Anc
Anc



dated."

Readable Gherkin DONTs

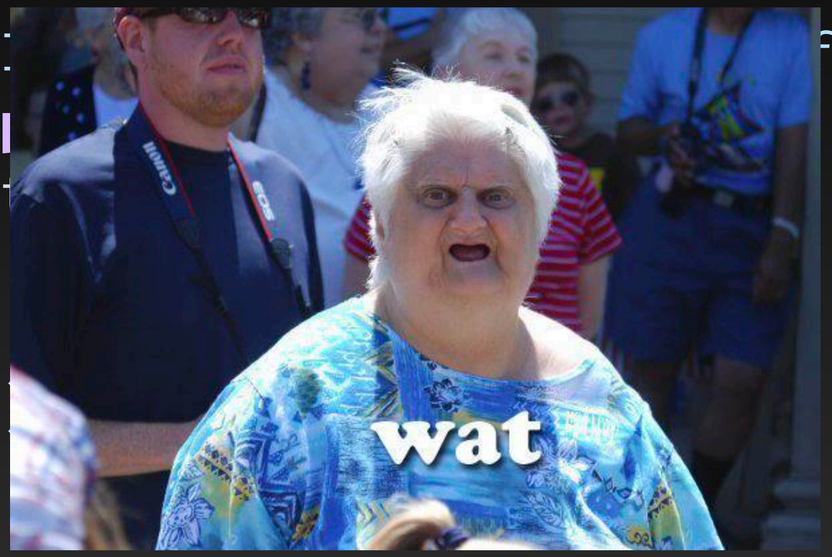
- 🚫 Don’t write scripts in Gherkin
- 🚫 Don’t get lost in UI details
- 🚫 …

Readable Gherkin DONTs

- 🚫 Don’t write scripts in Gherkin
- 🚫 Don’t get lost in UI details
- 🚫 Don’t try to make Gherkin steps generic & reusable

```
@when('I do the same thing as before')
def step_impl(context):
    context.execute_steps('''
        when I press the big red button
         and I duck
    ''')
```

```
@when('                                                    fore')
def step
    cont

                                                           button

    '''
```



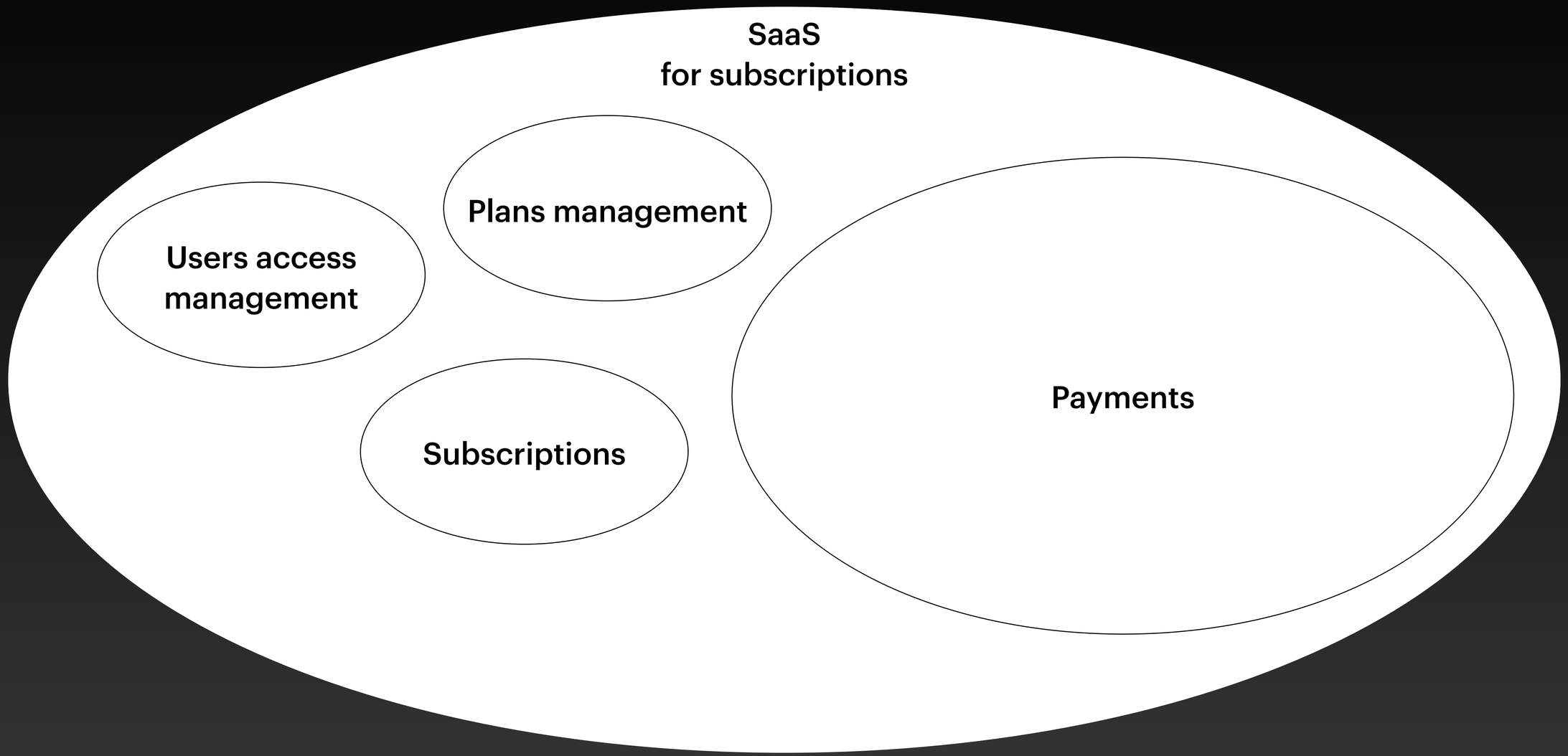
How to BDD

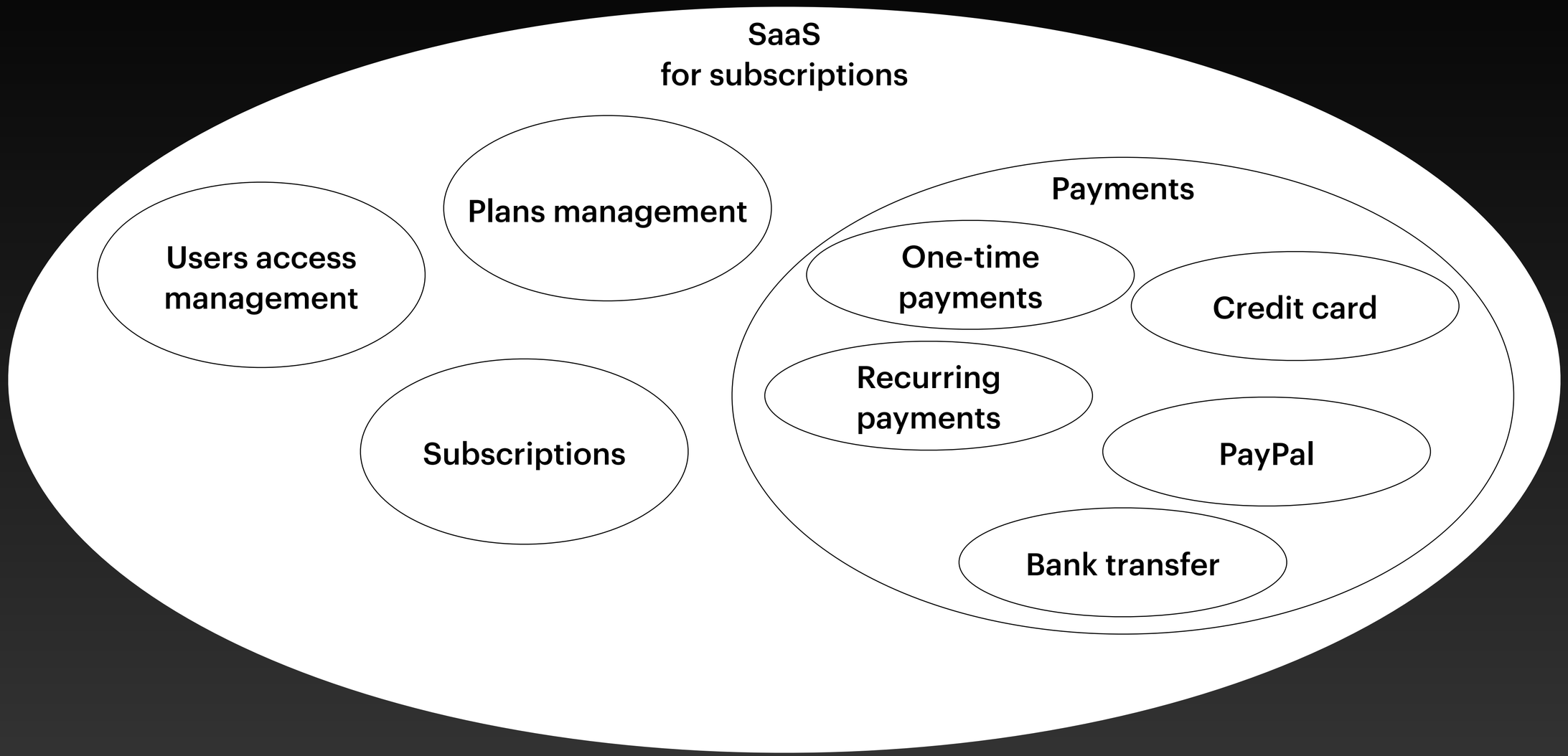
1. Name the expected behaviour
2. Capture it using Gherkin
    1. Use collaborative process
    2. Keep specs short & simple

„Keep it simple!”

...but getting to „simple” is not easy

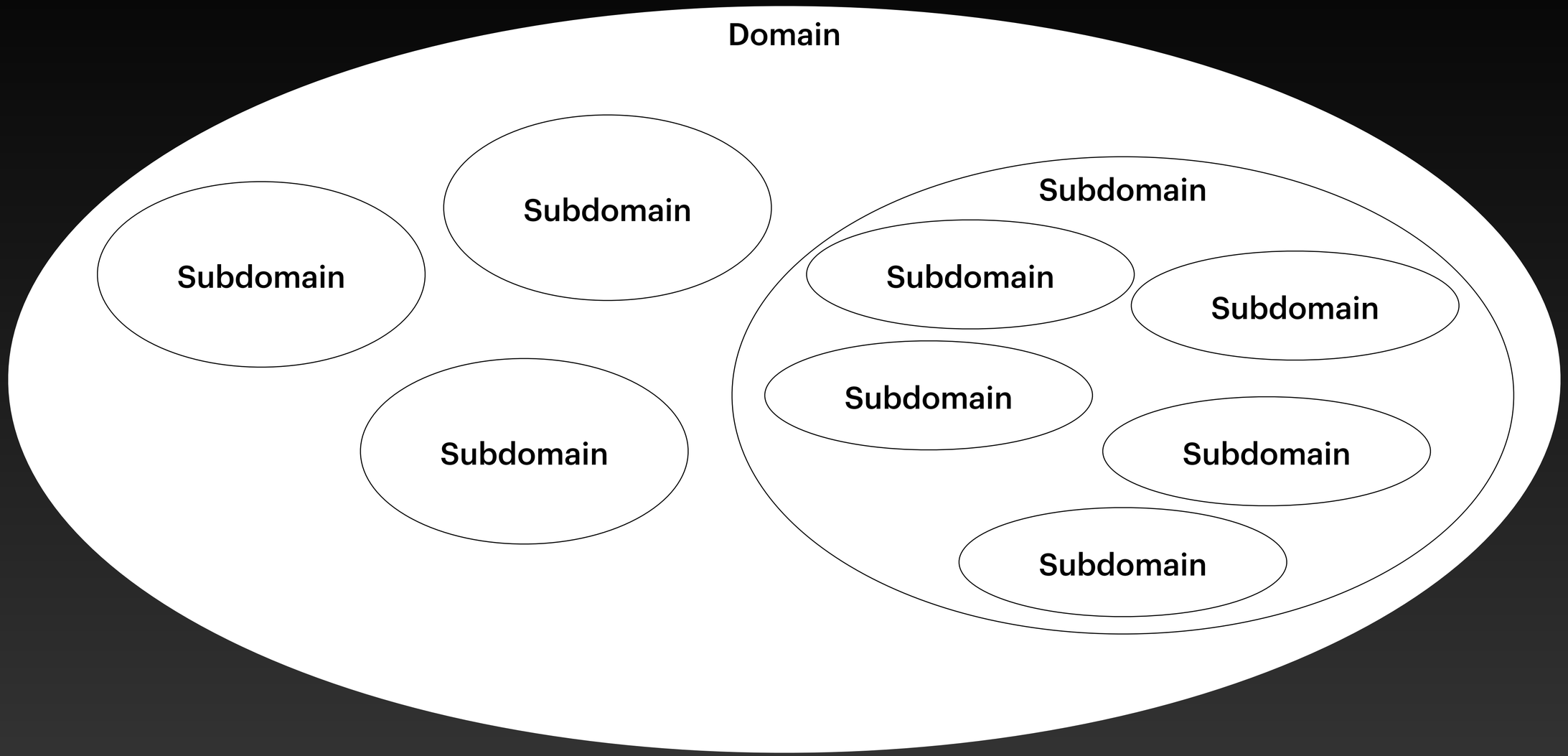
SaaS
for subscriptions


SaaS
for subscriptions
Users access management
Plans management
Subscriptions
Payments


SaaS
for subscriptions
Users access management
Plans management
Subscriptions
Payments
One-time payments
Credit card
Recurring payments
PayPal
Bank transfer

DDD

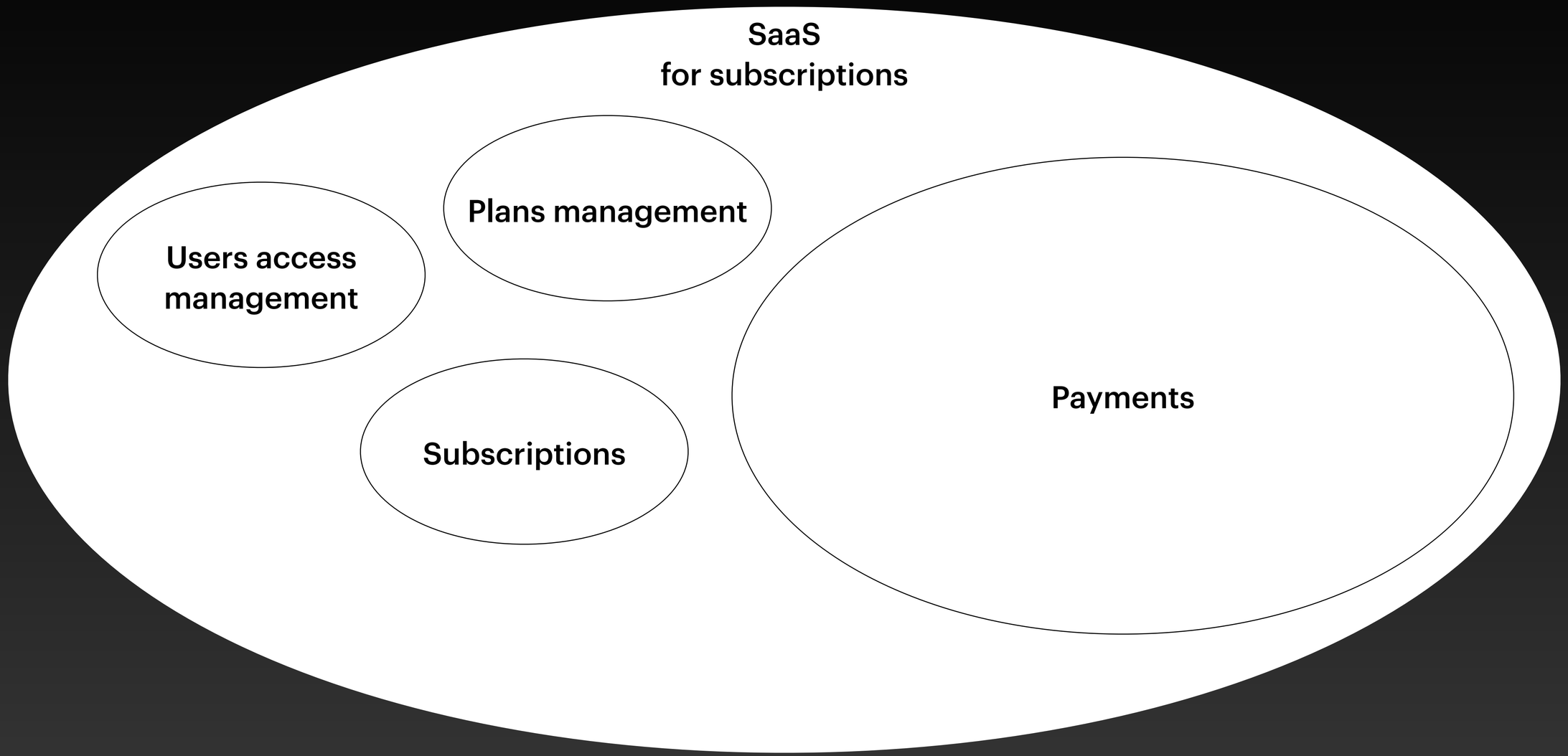
Domain-driven design


Domain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain
Subdomain

Limit the scope of a BDD spec

e.g. it doesn’t matter for payments if user authenticates using username & password or social login

Modularized software makes BDD simple


SaaS
for subscriptions
Users access management
Plans management
Subscriptions
Payments

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

To simplify, we need to limit the scope

```
Feature: Subscribing

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    And plan "monthly" with a price "9.99" USD and "fast support" as benefits
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD and "fast support" as benefits

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user should have "fast support" in their benefits
    And user's next bill date should be a month from now
```

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    When user subscribes to "monthly" plan
    Then user should be subscribed to plan "monthly"
    And user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    Given user registered with username "foo"
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

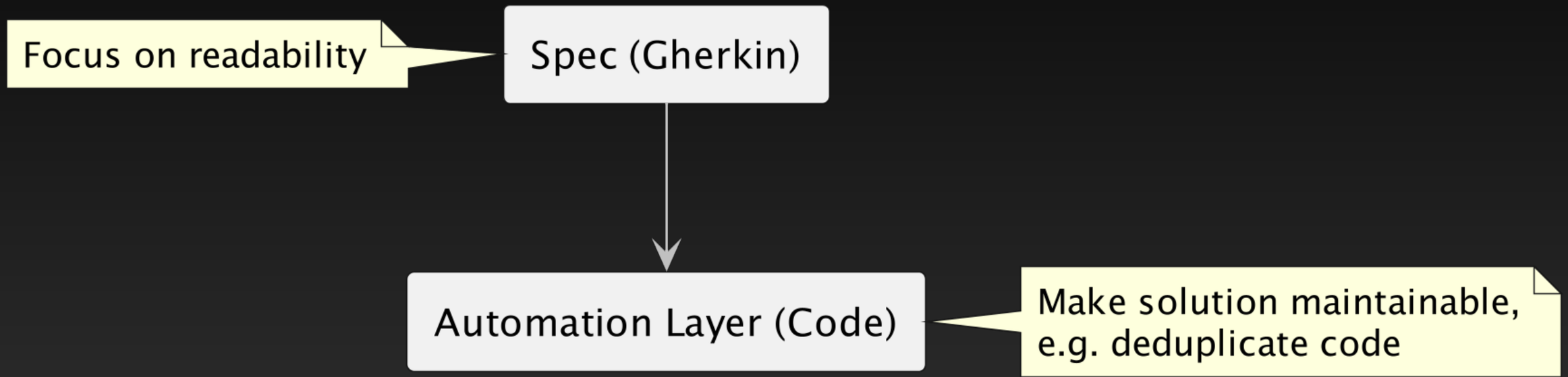
```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to monthly plan
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

Simplifying means limiting number of details

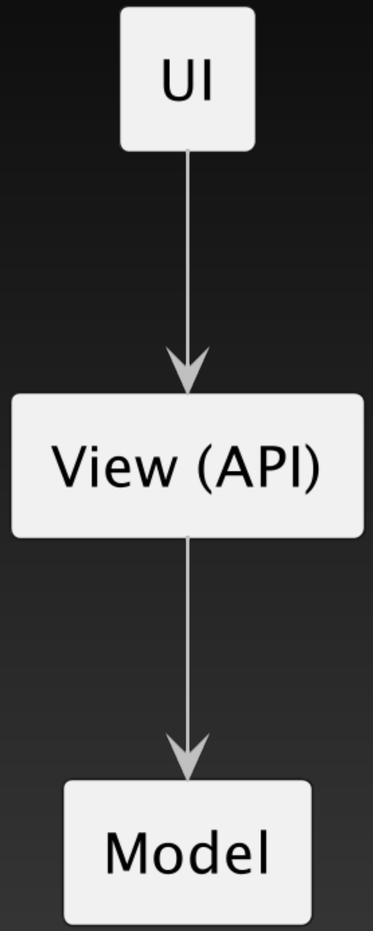
How to BDD

1. Name the expected behaviour
2. Capture it using Gherkin
    1. Use collaborative process
    2. Keep specs short & simple
3. Limit the scope of a single scenario (not too many subdomains at once)

We made it simple...
but where do dragons 🐉🐉🐉 live, then?


Focus on readability
Spec (Gherkin)
Automation Layer (Code)
Make solution maintainable,
e.g. deduplicate code

Should automation layer operate on UI or API?



~140 HTTP Requests

(that’s how many Facebook sends on load on my profile & computer)

UI will *generally* have more dependencies

Hence it’s harder to setup for BDD and will be *generally* slower, less stable.

API level should be considered first*

*go up or down if it makes sense

Let's automate THIS

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

```python
@when(parsers.parse('user subscribes to "{plan_id}" plan'))
def user_subscribes(test_client: TestClient, plan_id: str) -> None:
    ...

    response = test_client.post(
        "/subscriptions/",
        json={"plan_id": plan_id},
        headers={"Authorization": f"Bearer {token}"},
    )
    response.raise_for_status()
```

```python
class AppClient:
    def __init__(self, test_client: TestClient) -> None:
        self._test_client = test_client

    def subscribe(self, plan_id: str) -> None:
        response = self._test_client.post(
            "/subscriptions/",
            json={"plan_id": plan_id},
            headers={"Authorization": f"Bearer {token}"},
        )
        response.raise_for_status()
```

```python
class AppClient:
    def __init__(self, test_client: TestClient) -> None:
        self._test_client = test_client

    def subscribe(self, plan_id: str) -> None:
        response = self._test_client.post(
            "/subscriptions/",
            json={"plan_id": plan_id},
            headers={"Authorization": f"Bearer {token}"},
        )
        response.raise_for_status()


@fixture
def app_client(test_client: TestClient) -> AppClient:
    return AppClient(test_client=test_client)
```

```python
@when(parsers.parse('user subscribes to "{plan_id}" plan'))
def user_subscribes(app_client: AppClient, plan_id: str) -> None:
    app_client.subscribe(plan_id=plan_id)
```

We abstract communication protocol away with AppClient

```
def user_subscribes(app_client: AppClient, plan_id: str) → None:
    app_client.subscribe(plan_id=plan_id)
```

```python
class AppClient:
    def __init__(self, test_client: TestClient, user_token: str) -> None:
        self._test_client = test_client
        self._user_token = user_token

    def subscribe(self, plan_id: str) -> None:
        response = self._test_client.post(
            "/subscriptions/",
            json={"plan_id": plan_id},
            headers={"Authorization": f"Bearer {self._user_token}"},
        )
        response.raise_for_status()


@fixture
def app_client(test_client: TestClient, user_token: str) -> AppClient:
    return AppClient(test_client=test_client, user_token=user_token)
```

```python
class AppClient:
    def __init__(self, test_client: TestClient, user_token: str) -> None:
        self._test_client = test_client
        self._user_token = user_token

    def subscribe(self, plan_id: str) -> None:
        response = self._test_client.post(
            "/subscriptions/",
            json={"plan_id": plan_id},
            headers={"Authorization": f"Bearer {self._user_token}"},
        )
        response.raise_for_status()


@fixture
def app_client(test_client: TestClient, user_token: str) -> AppClient:
    return AppClient(test_client=test_client, user_token=user_token)
```

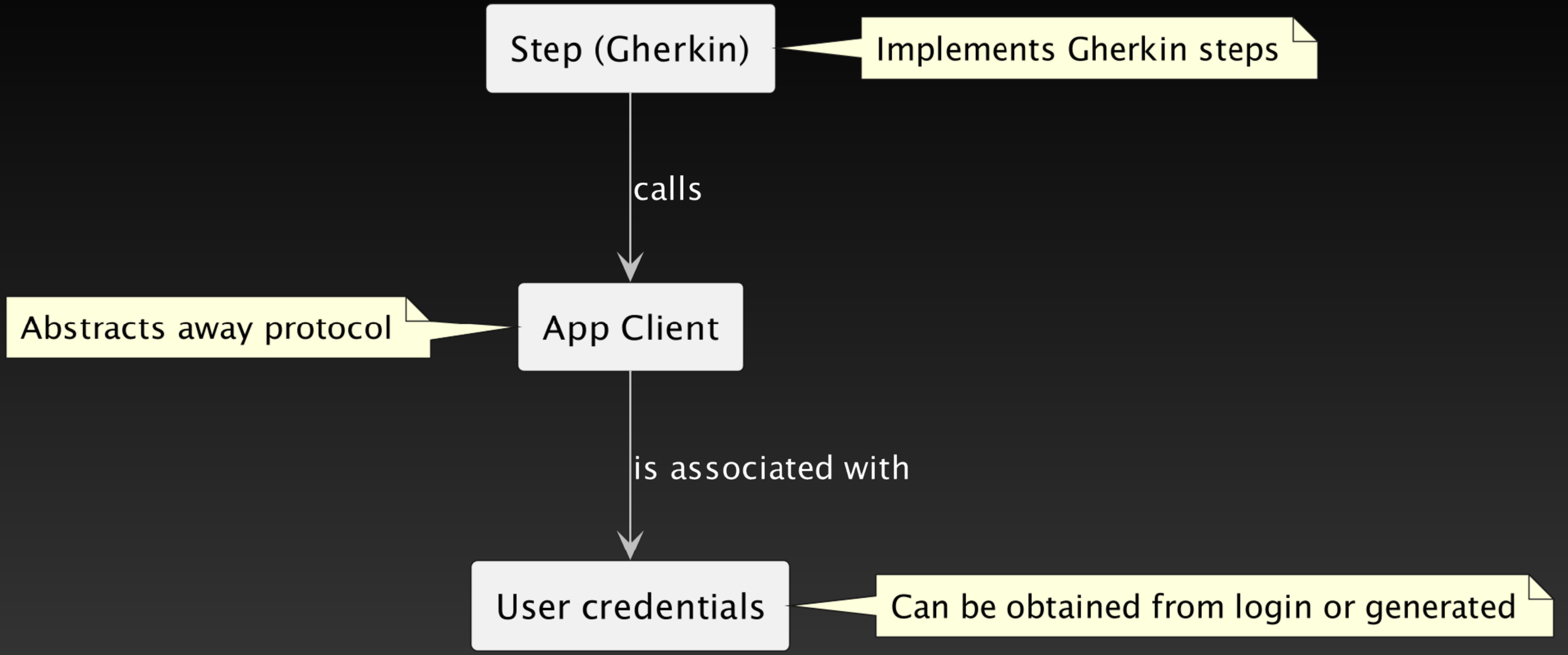
```python
class AppClient:
    def __init__(self, test_client: TestClient, user_token: str) -> None:
        self._test_client = test_client
        self._user_token = user_token

    def subscribe(self, plan_id: str) -> None:
        response = self._test_client.post(
            "/subscriptions/",
            json={"plan_id": plan_id},
            headers={"Authorization": f"Bearer {self._user_token}"},
        )
        response.raise_for_status()


@fixture
def app_client(test_client: TestClient, user_token: str) -> AppClient:
    return AppClient(test_client=test_client, user_token=user_token)
```

The usual way

```
@fixture
def user_token(test_client: TestClient) → str:
    register_response =  ...
    login_response = test_client.post(
        "/login", json={"username": "ep", "password": "2023"}
    )
    login_response.raise_for_status()
    return login_response.json()["token"]
```

The shortcut

```
users_ids = iter(range(1, 10_000))


@fixture
def user_token(test_client: TestClient) -> str:
    next_free_user_id = next(users_ids)
    return auth.token_for(user_id=next_free_user_id)
```


Step (Gherkin)
Implements Gherkin steps
calls
Abstracts away protocol
App Client
is associated with
User credentials
Can be obtained from login or generated

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

```python
@fixture(autouse=True)
def patched_payments_provider() -> None:
    with patch.object(Adyen, „....payments_api.payments") as payments:
        yield payments


@then(parsers.parse('user should be charged "{amount}" USD'))
def user_charged(patched_payments_provider: Mock, amount: str) -> None:
    patched_payments_provider.assert_called_once_with( ... )
```

```
@fixture(autouse=Tru
def patched_payments
    with patch.objec                                    nts") as payments:
        yield paymen


@then(parsers.parse(                                    }" USD'))
def user_charged(pat                                    ount: str) → None:
    patched_payments                                    h( ... )
```

```python
@fixture(autouse=True)
def patched_payments_provider() → None:
    with patch.object(Adyen, „....payments_api.payments") as payments:
        yield payments
```

Don’t mock what you don’t own

```python
@then(parsers.parse('user should be charged "{amount}" USD'))
def user_charged(patched_payments_provider: Mock, amount: str) → None:
    patched_payments_provider.assert_called_once_with( ... )
```

```python
@fixture(autouse=True)
def patched_payments_provider() → None:
    with patch.object(Adyen, „....payments_api.payments") as payments:
        yield payments
```

Concrete provider belongs to payments module

```python
@then(parsers.parse('user should be charged "{amount}" USD'))
def user_charged(patched_payments_provider: Mock, amount: str) → None:
    patched_payments_provider.assert_called_once_with( ... )
```

Facade design pattern - to create API over component

```
class PaymentsFacade:
    def create_recurring_payment(
        self, payment_id: PaymentId, amount: Money, term: Term
    ) → None:
        pass

    def cancel_recurring_payment(self, payment_id: PaymentId) → None:
        pass
```

Facade design pattern - to create API over component

```python
# subscriptions/payments/facade.py
def create_recurring_payment(
    payment_id: PaymentId, amount: Money, term: Term
) -> None:
    pass


def cancel_recurring_payment(payment_id: PaymentId) -> None:
    pass
```

Use mocks on stable API

```python
@fixture(autouse=True)
def create_recurring_payment_mock() -> None:
    with patch.object(PaymentsFacade, "create_recurring_payment") as mock:
        yield mock


@then(parsers.parse('user should be charged "{amount}" USD'))
def user_charged(create_recurring_payment_mock: Mock, amount: str) -> None:
    create_recurring_payment_mock.assert_called_once_with(
        payment_id=ANY,
        amount=Money("9.99", "USD"),
        term=ANY,
    )
```

```
Feature: Subscribing
  Background:
    Given plan "monthly" with a price "9.99" USD

  Scenario: Subscribing to a monthly plan
    When user subscribes to "monthly" plan
    Then user should be charged "9.99" USD
    And user's next bill date should be a month from now
```

```python
class PlansFacade:
    def create_plan(
        self,
        user_id: UserId,
        plan_id: str,
        price: Money,
        benefits: list[Benefit],
    ) -> None:
        pass

    def get_plan(self, plan_id: PlanId) -> PlanOut:
        pass


@dataclass
class PlanOut:
    plan_id: PlanId
    price: Money
    benefits: list[Benefit]
```

```python
class PlansFacade:
    def create_plan(
        self,
        user_id: UserId,
        plan_id: str,
        price: Money,
        benefits: list[Benefit],
    ) -> None:
        pass

    def get_plan(self, plan_id: PlanId) -> PlanOut:
        pass
```

```python
@dataclass
class PlanOut:
    plan_id: PlanId
    price: Money
    benefits: list[Benefit]
```

```python
@given(parsers.parse('plan "{plan_id}" with a price "{price}" USD'))
def plan_exists(plan_id: str, price: str) -> None:
    plan = PlanOut(
        plan_id=PlanId(plan_id),
        price=Money(amount=price, currency="USD"),
        benefits=[],
    )

    with patch.object(PlansFacade, "get_plan") as get_plan_mock:
        get_plan_mock.return_value = plan
        yield
```

```python
@given(parsers.parse('plan "{plan_id}" with a price "{price}" USD'))
def plan_exists(plan_id: str, price: str) -> None:
    plan = PlanOut(
        plan_id=PlanId(plan_id),
        price=Money(amount=price, currency="USD"),
        benefits=[],
    )

    with patch.object(PlansFacade, "get_plan") as get_plan_mock:
        get_plan_mock.return_value = plan
        yield
```

```python
class PlanOutFactory(factory.Factory):
    class Meta:
        model = PlanOut

    price = Money("1.99", "USD")
    benefits = factory.LazyFunction(lambda: [])


@given(parsers.parse('plan "{plan_id}" with a price "{price}" USD'))
def plan_exists(plan_id: str, price: str) -> None:
    plan = PlanOutFactory.build(
        plan_id=PlanId(plan_id),
        price=Money(amount=price, currency="USD"),
    )

    with patch.object(PlansFacade, "get_plan") as get_plan_mock:
        get_plan_mock.return_value = plan
        yield
```

How to BDD

1. Name the expected behaviour
2. Capture it using Gherkin
    1. Use collaborative process
    2. Keep specs short & simple
3. Limit the scope of a single scenario (not too many subdomains at once)
4. Leverage modularization of your architecture and use various patterns for automation layer

Verify behaviour

Not implementation.

Build shared understanding

Ubiquitous Language

Further reading

- [book] Specification By Example by Gojko Adzic
- [book] BDD in Action by John Fergusson Smart
- [blog post] Whose domain is it anyway? by Dan North
- [blog post] People behaviour and unit testing by Vladimir Khorikov

Q & A

pythoneer.substack.com

Resources used

- https://commons.wikimedia.org/wiki/File:Hotel_%27Alicja%27_and_EC2_cooling_tower,_%C5%81%C3%B3d%C5%BA_Politechniki_Avenue.jpg [no changes]
- https://www.booking.com/hotel/pl/alicja.pl.html [official gallery of Hotel's profile on booking.com]
- https://pixabay.com/pl/photos/londyn-metro-underground-2768732/