How to make your tests keep you focused!

I. About focus

How long does it take to reproduce a bug?

- **fresh dump of live DB every day, instead of you triggering it**
- **profiler**
- **event-sourcing**
- **tools for generating test users (cli)**

“It’s about enabling us to Focus on the things that matter.”

II. About focus

How long does it take to get feedback of my code change?

- **pipelines / tests**
- **continuous delivery, reality check, then refactor**

“If it takes 2h. for your tests to run, you’re esentially unfocusing yourself from the things that matter so that later on you refocus and continue.”

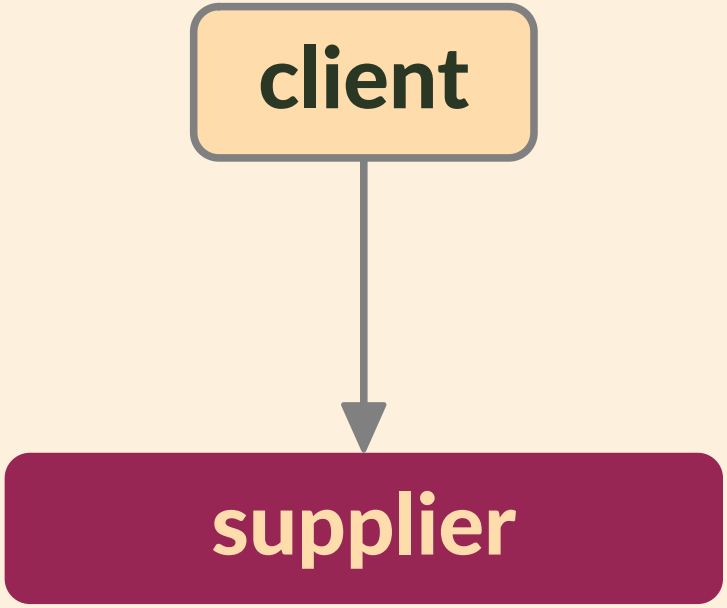
III. About focus

To dive more into this Topic, GeePaw Hill:

- **microtests, TDD**
- **MMMSS methodology:**
  Many Much More
  Smaller Steps

About Tests

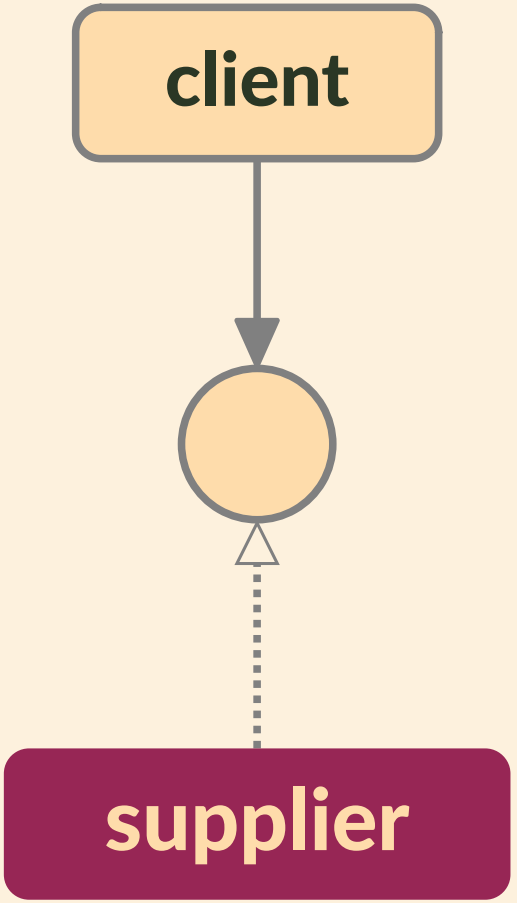
It usually starts like this...


client
supplier

“Don’t refer to volatile concrete classes.

Refer to abstract interfaces instead.”

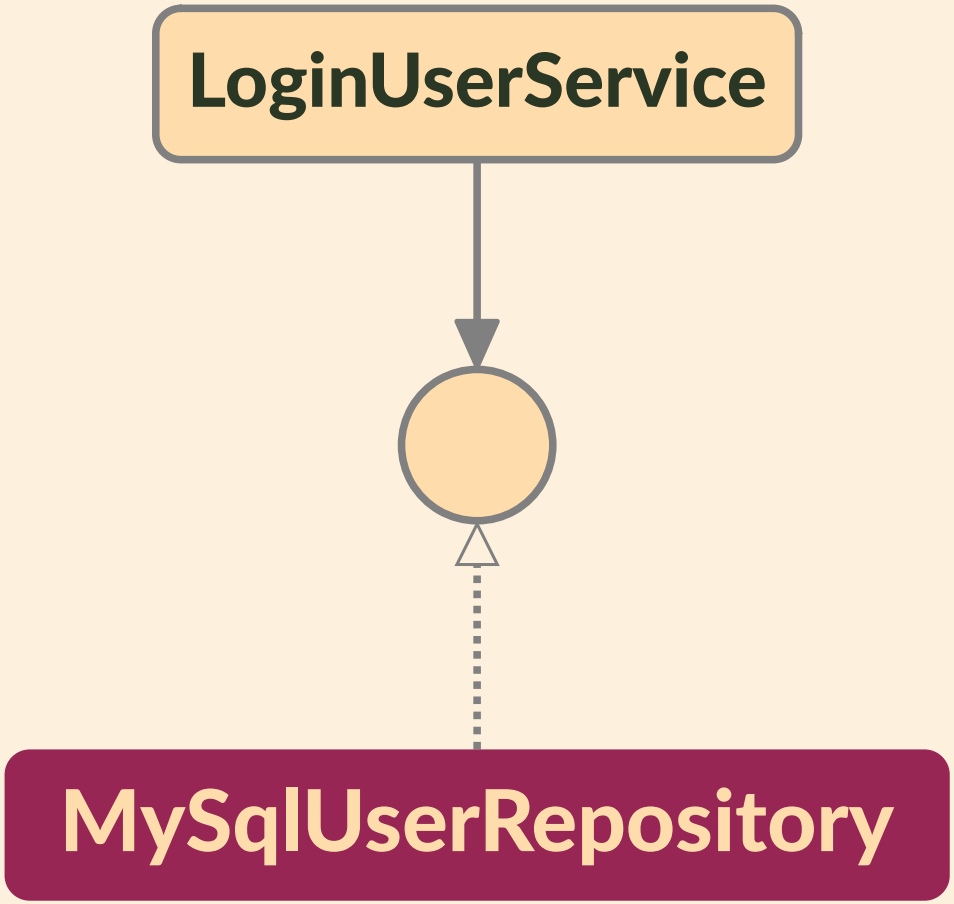
- Clean Architecture (DIP, p. 89)



“Don’t refer to volatile concrete classes.

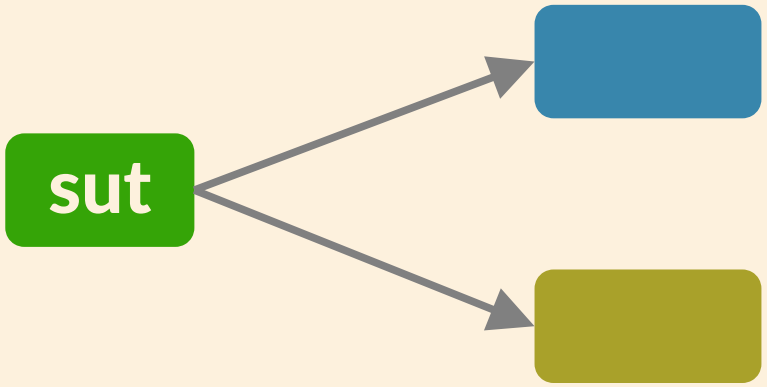
Refer to abstract interfaces instead.”

- Clean Architecture (DIP, p. 89)

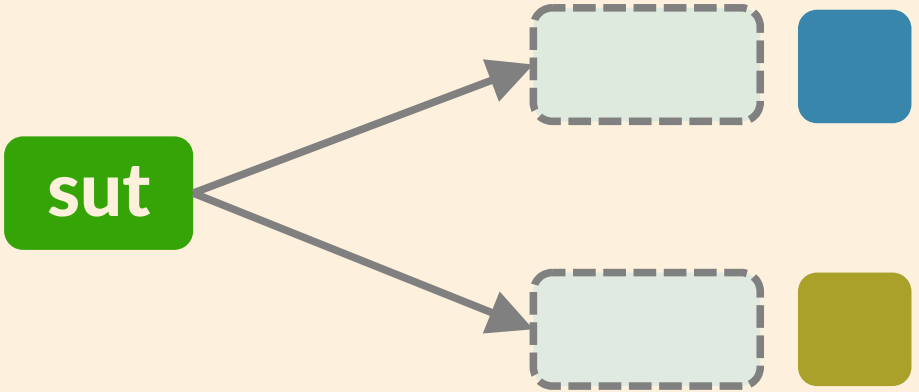


How do we test LoginUserService?

Sociable tests



Solitary tests



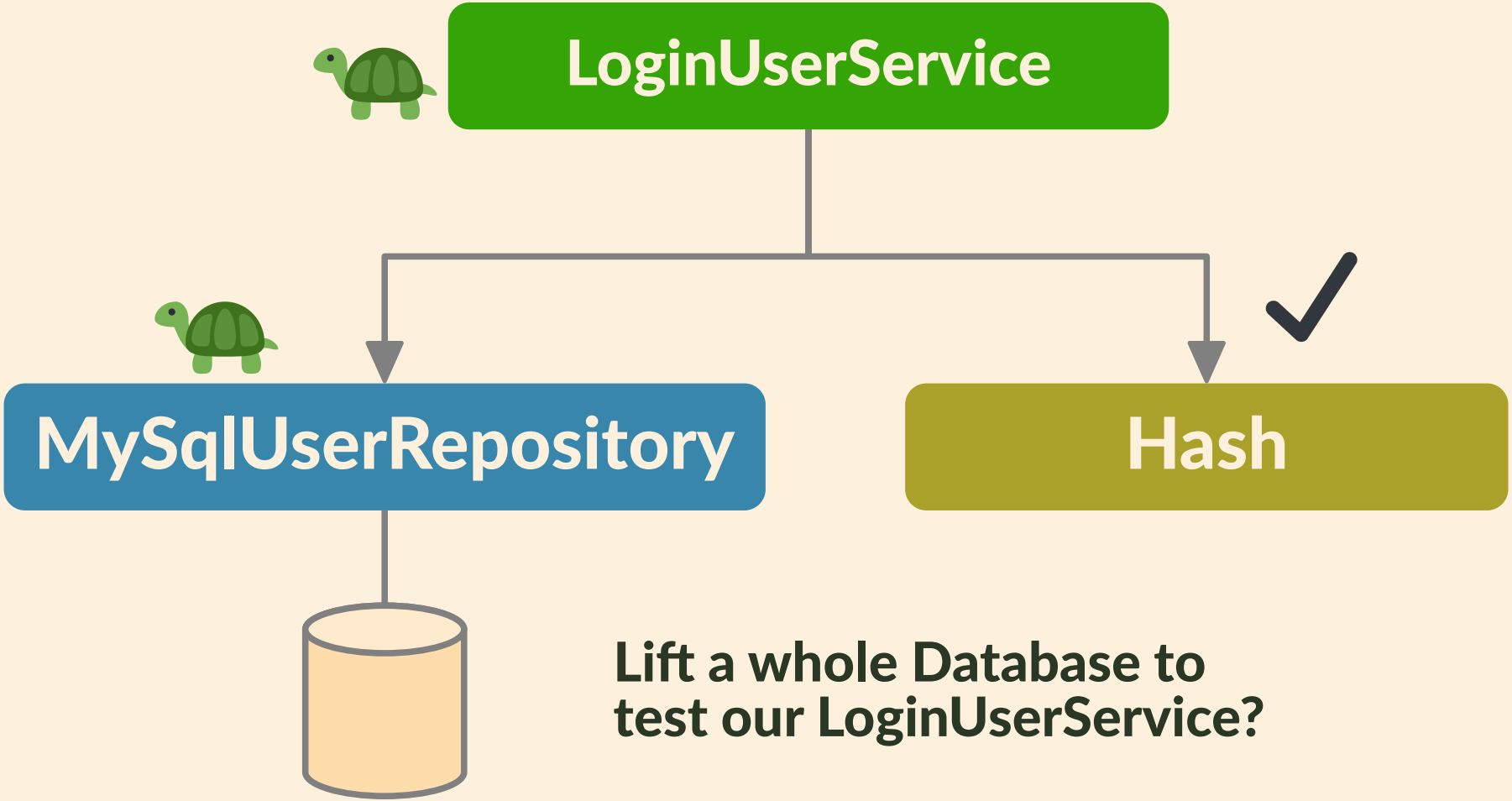
Sociable tests

- Test the integration with collaborators.
- They do state verification.
- They don't know how we use our collaborators.
- Better for refactoring
- May lead to more **flackey** tests.

Solitary tests

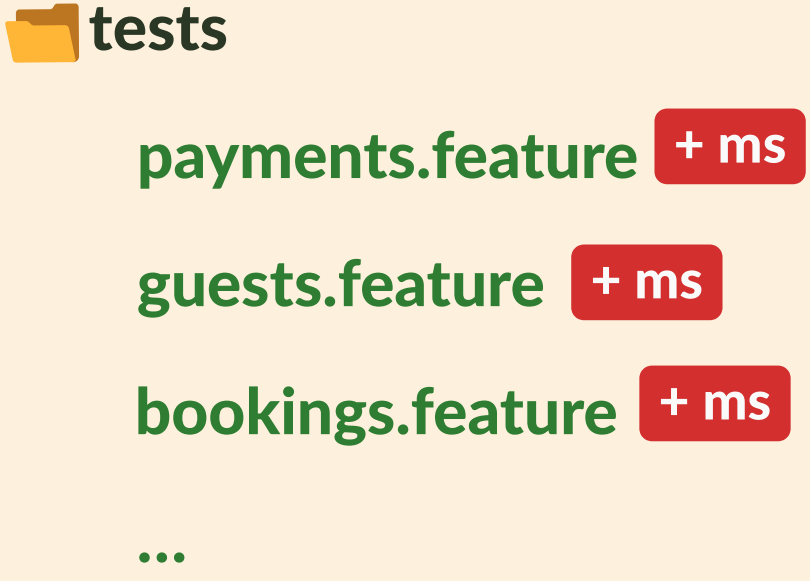
- Use mocks as way to isolate the object.
- They do collaboration verification
- Requires high discipline.
- Tighther to implementation. Leads to more **fragile** tests.

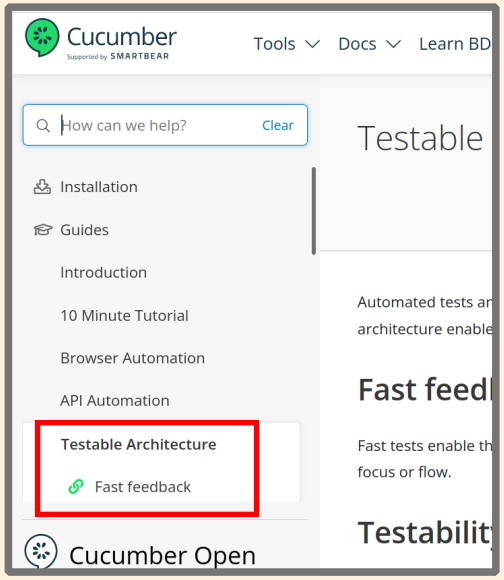
Let’s see which problems do we encounter with each


LoginUserService
MySqlUserRepository
Hash
Lift a whole Database to
test our LoginUserService?

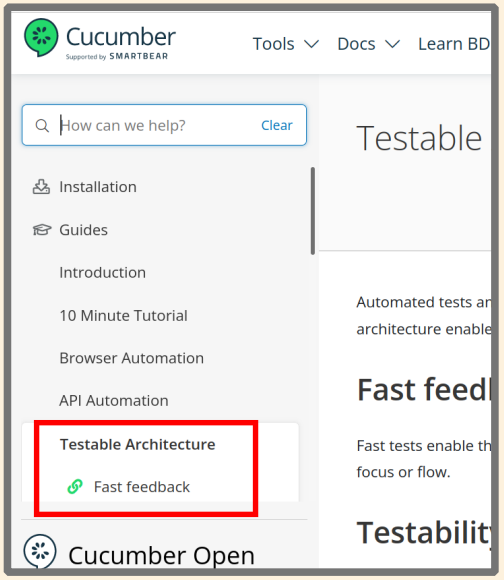
Somebody said Cucumber?

```
Feature: Guess the word

Scenario: Maker starts a game
    When the Maker starts a g
    Then the Maker waits for

Scenario: Breaker joins a gam
    Given the Maker has start
    When the Breaker joins th
    Then the Breaker must gue
```





- **Fast feedback**
- **Testability**
- **Ports and Adapters**
- **Full stack**
- **Test pyramid**



Fast Feedback

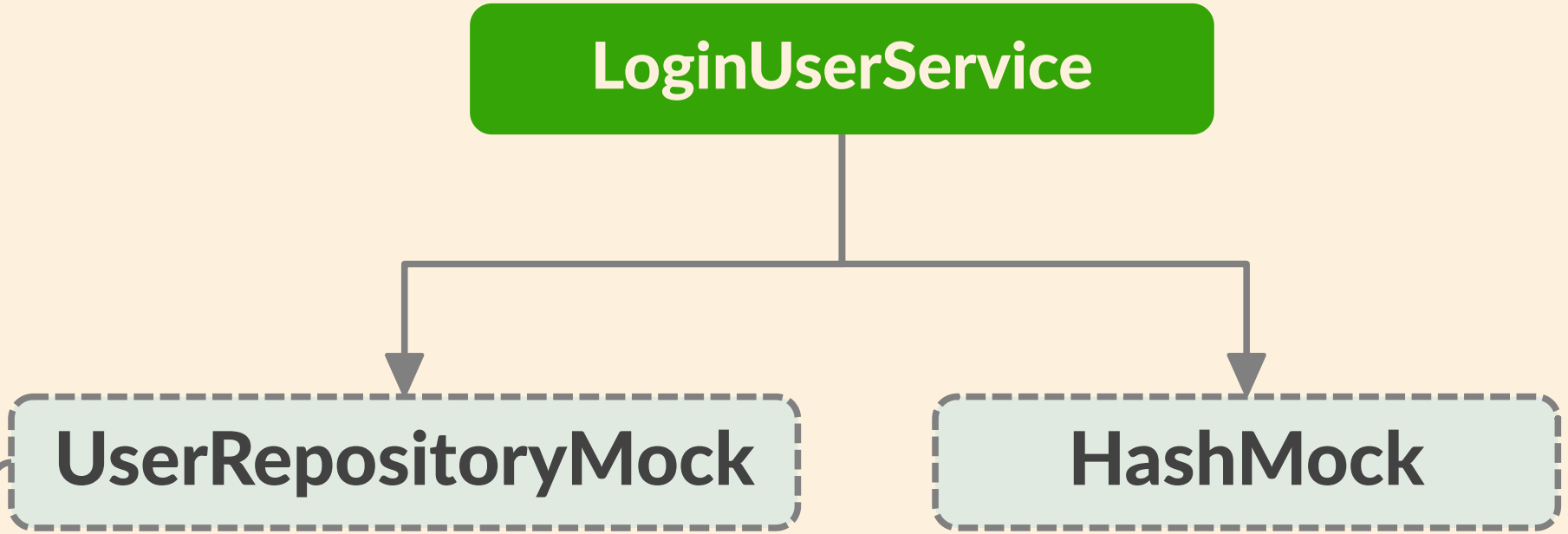
"Fast tests enable the developers to run them frequently to obtain fast feedback on what they are building, without losing focus or flow."

Testability

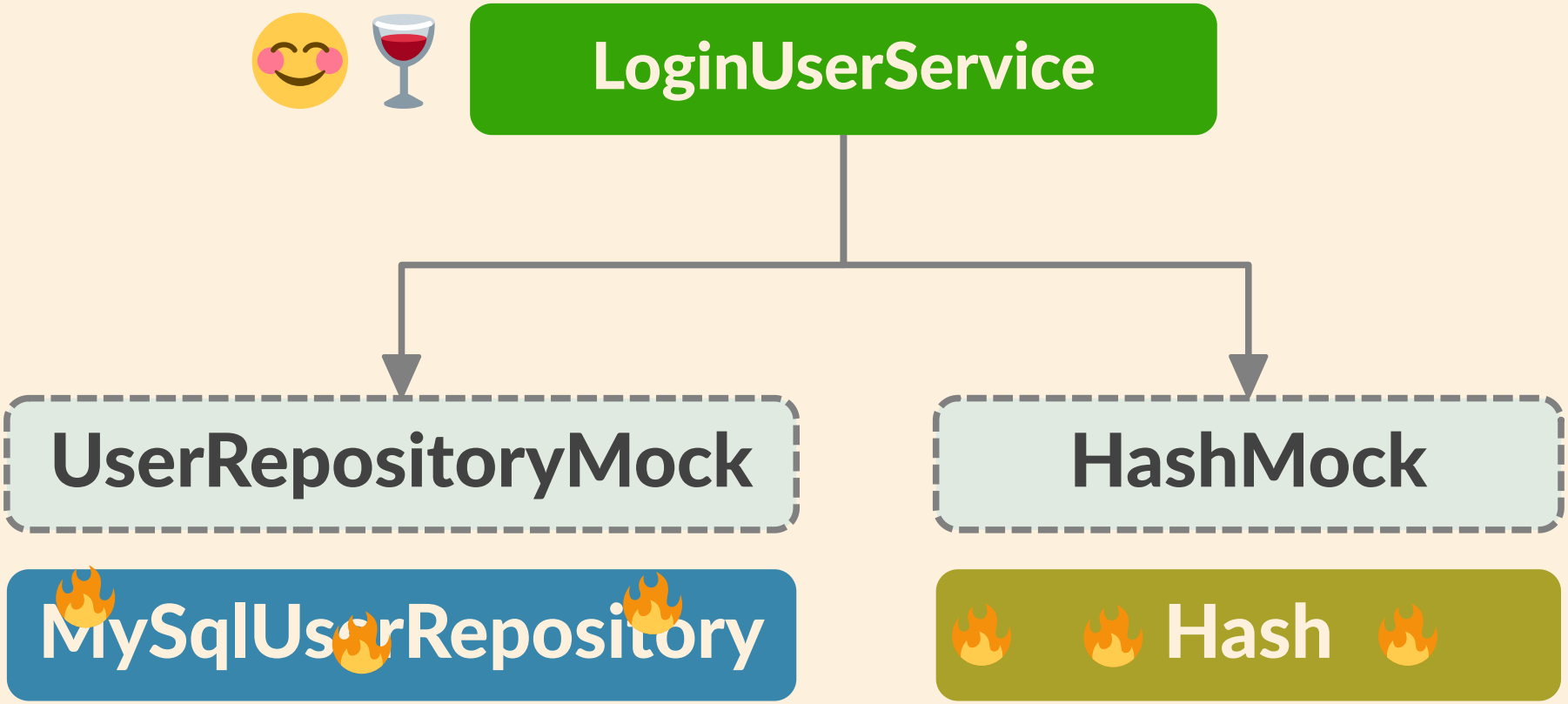
"Ensure that you can run your tests without going through the user interface (UI). They are slow, brittle, expensive, and hard to fix."

https://cucumber.io/docs/guides/testable-architecture/

Let’s not blame technology for using it wrong




```
userRepository→getByUsername(username)
    →shouldBeCalled()
    →willReturn(ANY VALUE MATCHING THE TYPE IS OK);
```

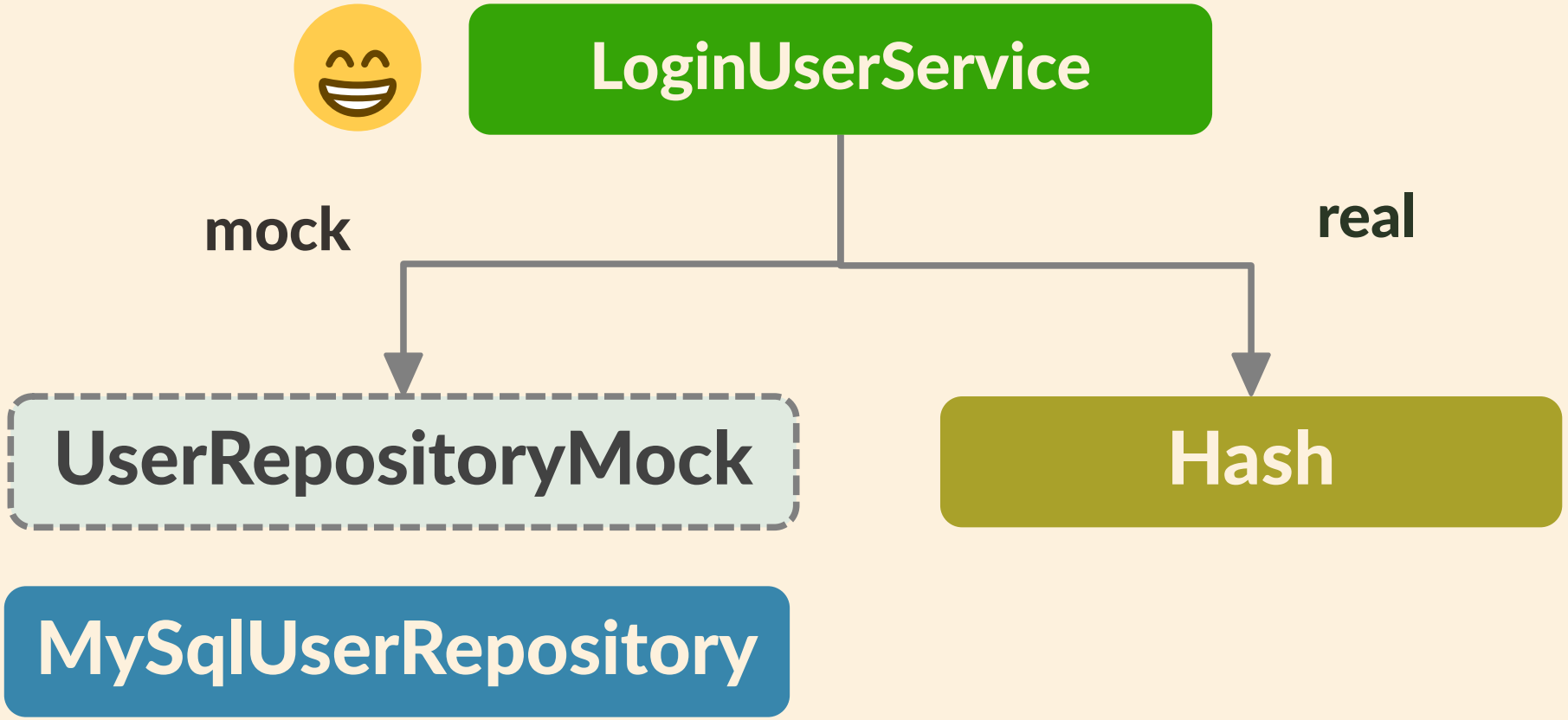

LoginUserService
UserRepositoryMock
HashMock
MySqlUserRepository
Hash

```
userRepository->getByUsername('admin')
    ->shouldBeCalled()
    ->willReturn(Maybe::just(new User('admin')));
```

```
getByUsername(string username): Maybe
{
    if (username === 'admin') {
        return Maybe::nothing();
    }
}
```

- **Test Theatre:** activities that give the illusion of testing while generating no useful information.

But then, what actually happens is...


LoginUserService
mock
real
UserRepositoryMock
Hash
MySqlUserRepository

So… when something is fast and reliable we actually prefer it? 😏

Contracts to the rescue!

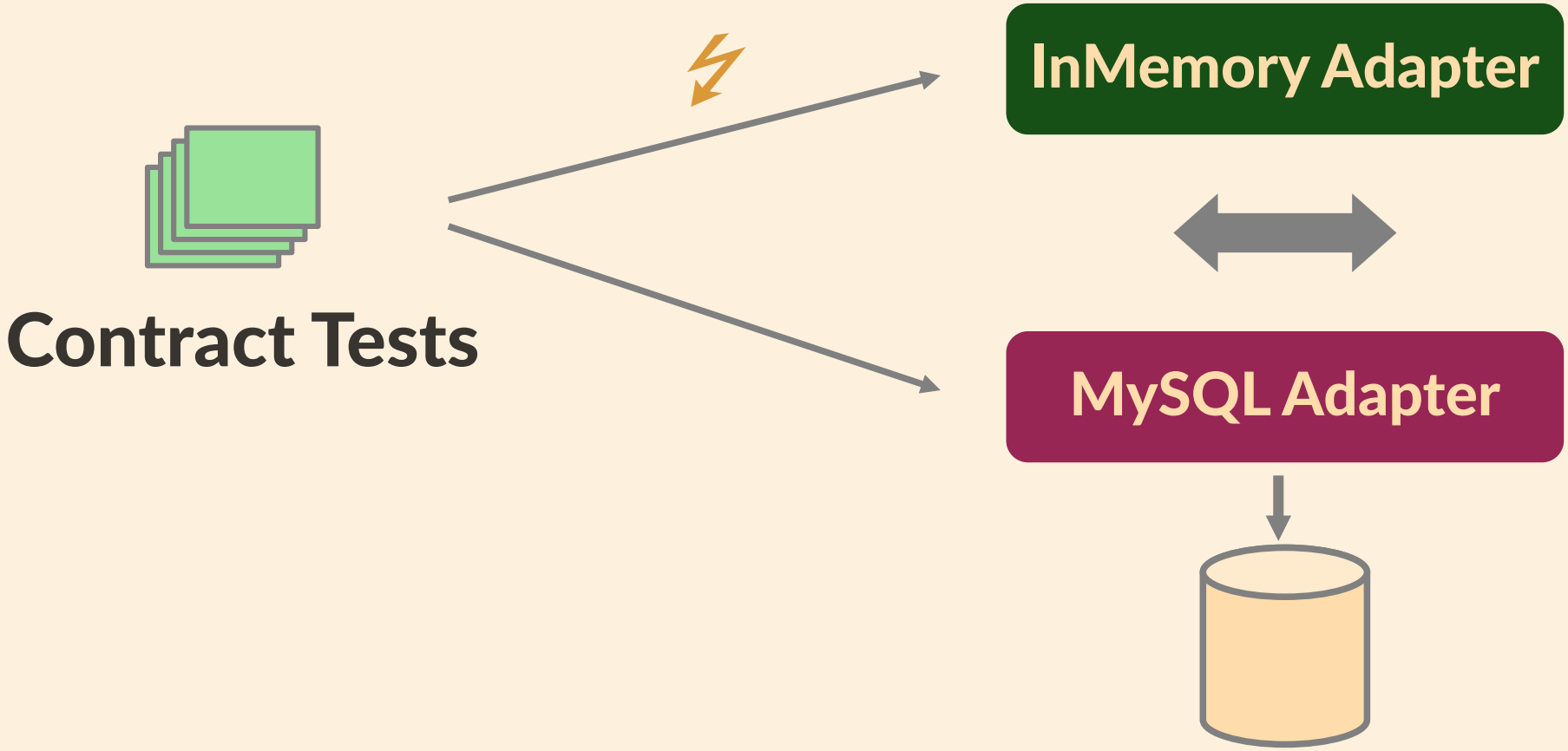
But first... do we know what a Contract is?

Interface vs Contract

What is a Contract Form in Eiffel lang? (DbC)

“The Contract Form […] enables client authors to reuse software elements without having to read their source code. This is a crucial requirement in large-scale industrial developments.”

“Contract Form is **truly abstract [set of tests]**, free from the implementation details of what it describes and concentrating instead on its **functionality**.”


InMemory Adapter
Contract Tests
MySQL Adapter

```php
trait DatabaseContractTests
{
    protected abstract function getInstance(): Database;

    public function test_when_I_store_something_I_can_retrieve_it(): void
    {
        $database = $this->getInstance();

        $database->insert('reservations', ['id' => 1, 'username' => 'John']);

        $result = $database->find('reservations', ['id' => 1]);

        assertEquals($result, ['id' => 1, 'username' => 'John']);
    }
}
```

```php
class InMemoryDatabaseTest extends TestCase
{
    use DatabaseContractTests;

    protected abstract function getInstance(): Database
    {
        return new InMemoryDatabase();
    }
}
```

Real implementation

In Memory implementation

When non-reuse is in fact preferable

Ariane 5's first test flight on 4 June 1996 failed with the rocket self-destructing 37 seconds after launch because of a malfunction in the control software.

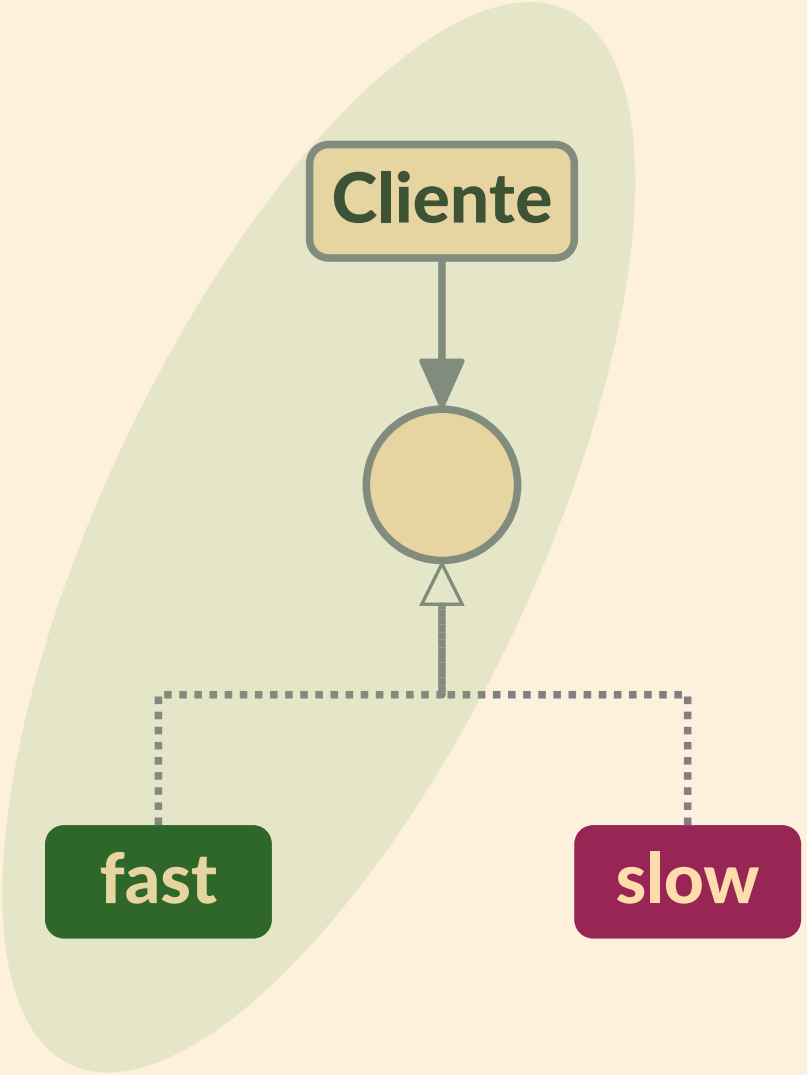
$500-million crash due to the incorrect reuse of a software module from the Ariane-4 project, **reuse without a contract documentation is the path to disaster. Non-reuse would, in fact, be preferable. .**

Note

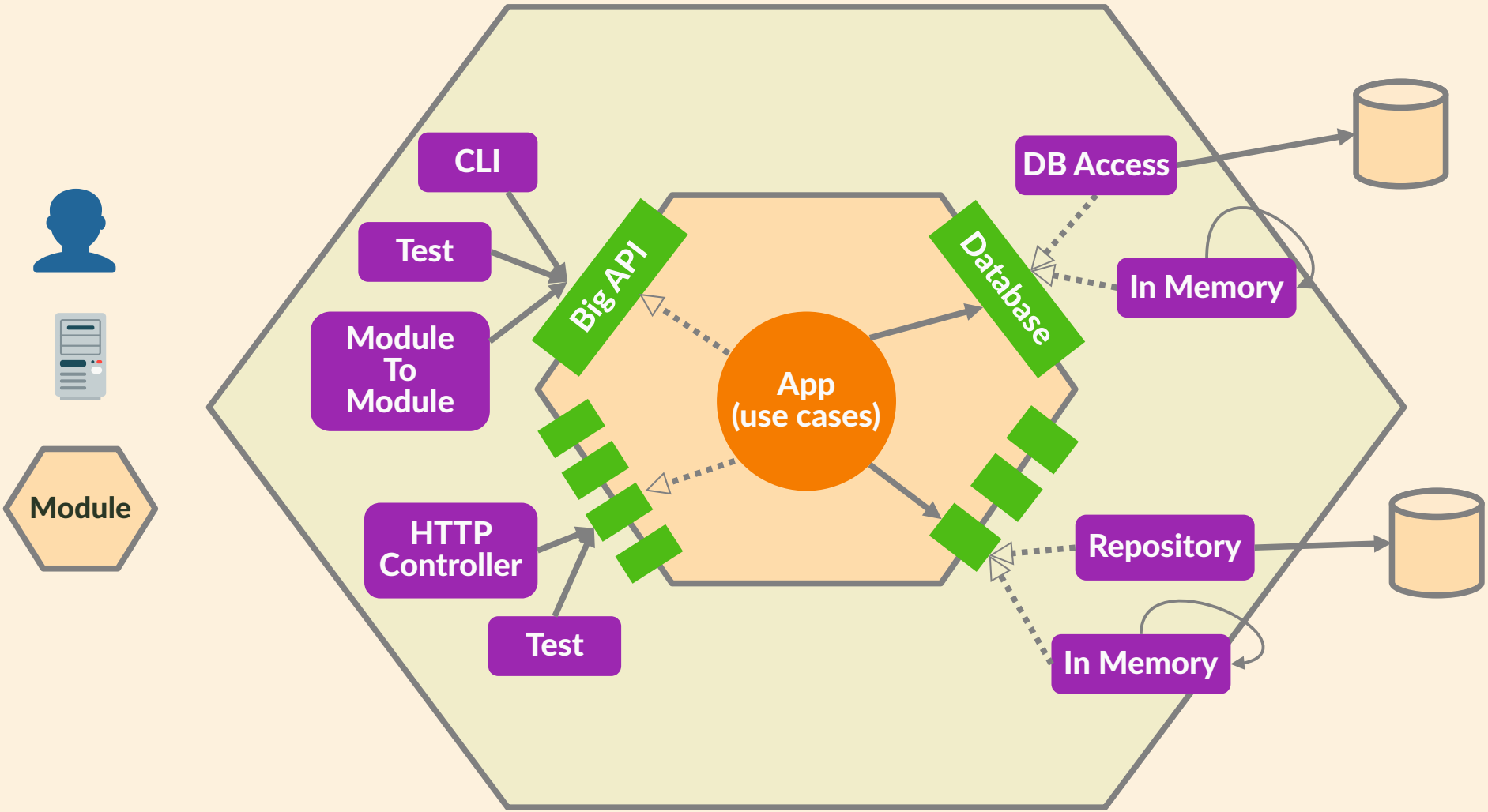
- Try not to depend on your Infrastructure tech if you don’t need to.
  - Use your Database as a Database, not as a place to put domain logic in.

- Contracts is what ensure you safe-replaceability, not interfaces ;)

Fast Collaboration Tests



Ports & Adapters


Module
CLI
Test
Module
To
Module
HTTP
Controller
Test
Big API
App
(use cases)
Database
DB Access
In Memory
Repository
In Memory

How Thick a Port should be?

“What exactly a port is and isn't **is largely a matter of taste**. At the one extreme, every use case could be given its own port, producing hundreds of ports for many applications. Alternatively, one could imagine merging all primary ports and all secondary ports so there are only two ports, a left side and a right side.”

“It doesn’t appear that there is any particular damage in choosing the “wrong” number of ports, **so that remains a matter of intuition**.”

- Alistair Cockburn

Incoming Ports

```php
// very thin left-side Port
// ./src/App/Ports/Incoming
interface ForCancelingReservations
{
    public function cancelReservation(int $id): void;
}

// thick left-side Port
// (suitable for cases where you want to have a "module", "sdk" kind of Hexagon)
// ./src/App/Ports/Incoming
interface ForManagingReservations
{
    public function cancelReservation(int $id): void;

    public function changeReservationDate(int $id, DateTime $date): void;

    public function addGuestToReservation(int $id, Guest $guest): void;

    // ...
}
```

Outgoing Ports

```php
// thin right-side Port
// ./src/App/Ports/Outgoing
interface ForStoringReservations
{
    public function save(Reservation $reservation): void;
}

// thick right-side Port
// could be a wrapper for your favorite DB library
// ./src/App/Ports/Outgoing
interface Database
{
    public function select(): Query;

    public function insert(): Query;

    public function update(): Query;

    public function delete(): Query;
}
```

```php
// ./src/App/Repositories
class ReservationsRepository implements ReservationsRepositoryInterface
{
    public function __construct(
        private Database $database
    ) {}

    public function getById(int $id): Reservation
    {
        $result = $this->database->select('reservations')
            ->where('id', $id)
            ->getOne();

        return new Reservation($result['id']);
    }
}
```

Repository using a Thick Port

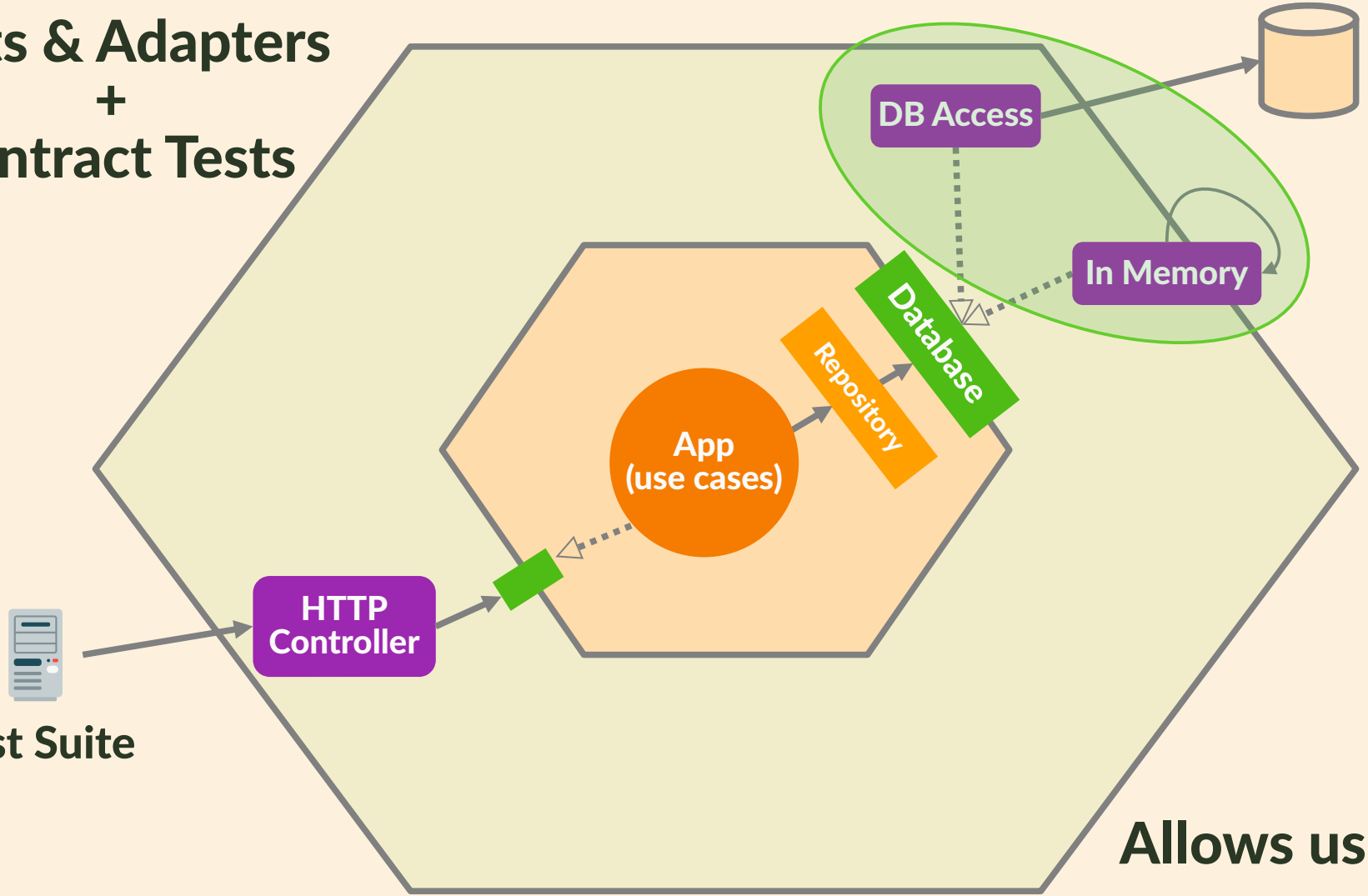
Exposing one Use case

```php
// step 1: instantiate right-side adapters
$outgoingAdapter = new Database();

// step 2: instantiate your app / module / use-cases
$repo = new ReservationsRepository($outgoingAdapter);
$useCase = new ForCancelingReservations($repo);

// step 3: instantiate left-side adapters
$incomingAdapter = new CancelReservationsController($useCase);
$incomingAdapter->handleRequest();
```

Exposing multiple use-cases

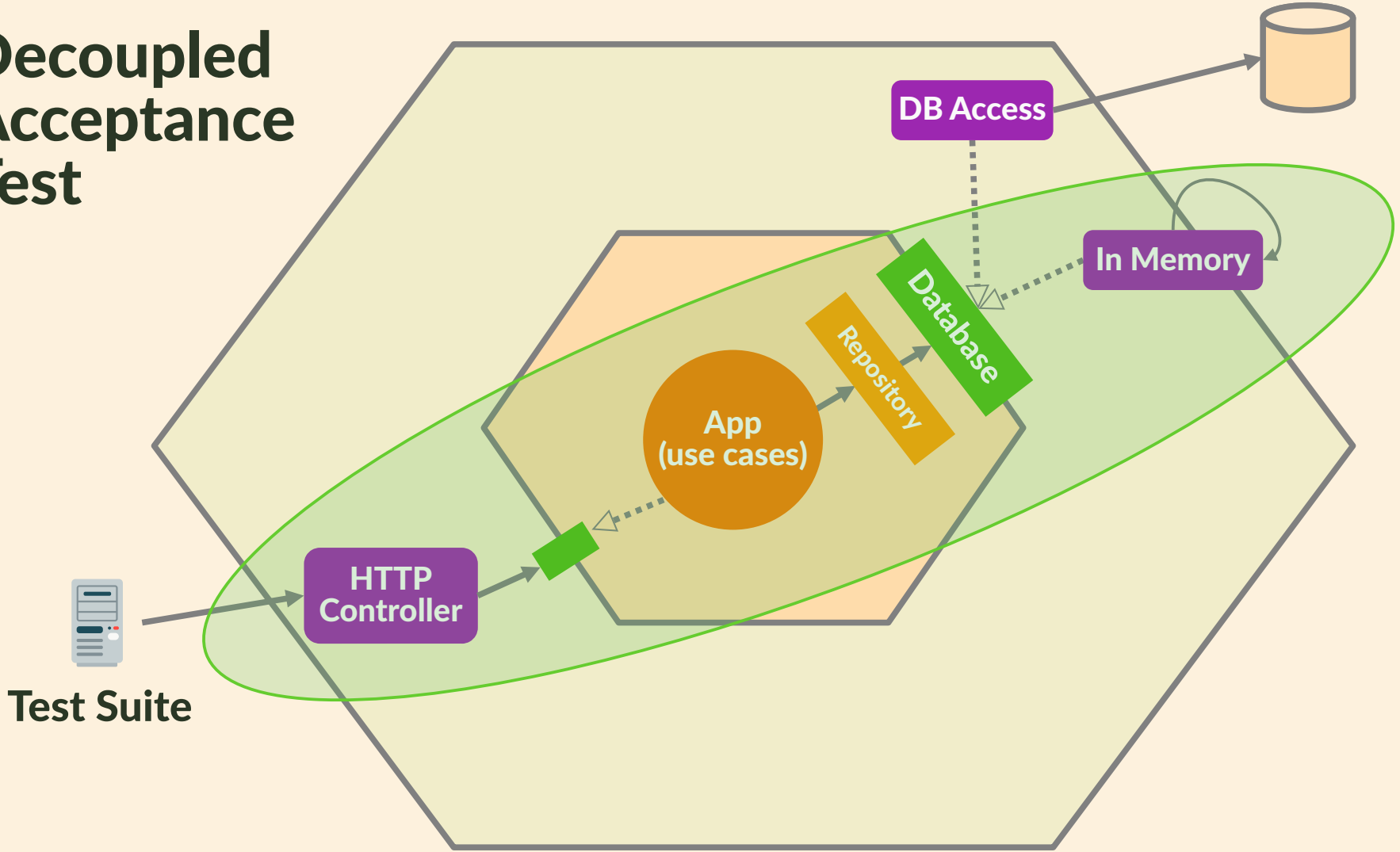
```php
// step 1: instantiate right-side adapters
$outgoingAdapter = new Database();

// step 2: instantiate your app / module / use-cases
$repo = new ReservationsRepository($outgoingAdapter);
$reservations = new Reservations($repo);

// step 3: instantiate left-side adapters
$incomingAdapter = new ReservationsController($app);
$incomingAdapter->handleRequest();
```

Ports & Adapters
+
Contract Tests



Allows us to ...

Decoupled Acceptance Test



```php
// step 1: instantiate right-side adapters
$outgoingAdapter = new InMemoryDatabase();

// step 2: instantiate your app / module / use-cases
$repo = new ReservationsRepository($outgoingAdapter);
$reservations = new Reservations($repo);

// step 3: instantiate left-side adapters
$incomingAdapter = new ReservationsController($app);

$httpRequest = new Request('POST', '/reservations/1/cancel');
$httpResponse = $incomingAdapter->handle($httpRequest);

assertEquals($httpResponse->getStatusCode(), 200);
assertEquals($httpResponse->getJson(), [...]);
```

Take aways

- Contract Tests are the supporting Mechanism that allows us to run tests in a more **efficient** way, at least at scale, and it does require actual work.
- Care about your test suite as you do for the production code.
- Don’t let your Software turn into Hardware.

We’re hiring!

I’m Filis, you can find me at:

@filisdev

TREKK SOFT