Automated Testing

The agilogy Way

Jordi Pradel

Agile software testing

Goals

Desiderata

Technical concepts

A very gentle introduction to Hexagonal Architecture

Patterns

Strategy

Goals

Goals

- We will focus on immediate goals:
  - Bug detection
  - Bug prevention

Desiderata

agilogy

Principles

- Usable tests make developers and testers test better. Write tests that:
  - Are deterministic 👇
  - Make changes easy to welcome 👇
  - Are user friendly (for developers and testers) 👇
- Automated tests are software, they should have good software **quality**
  - High cohesion & low coupling
- Test economy is of uttermost importance 👇

Deterministic tests

Flaky tests

- **Avoid flaky** tests
  - A flaky tests is a test that sometimes fails but that retried enough times ends up passing
  - They undermine your **confidence** in tests
  - Or, even worse, if you continue to trust them, you may ignore a test failure that is actually signaling a bug

Deterministic tests

Tests with undesired dependencies

- **Isolate tests**
  - Some tests are not flaky in isolation but they fail if they are executed…
    - In a different **order**
    - In **concurrency** with other tests
    - When some **external system** fails or changes its state
    - When **the stars align** (or are misaligned) (hint: the clock)

Tests that (allow you to) welcome change

- Tests should **allow refactoring** to enhance the design
  - Don't test the implementation does *what it does*, but that the **outcome** of what it does is the expected outcome (behavior vs outcome)
  - UI tests are difficult to automate... if you welcome changes to UI
- Tests should allow requirement changes
  - Design tests with high cohesion: There should only be **one reason** why you need to change any single test.

User friendly tests

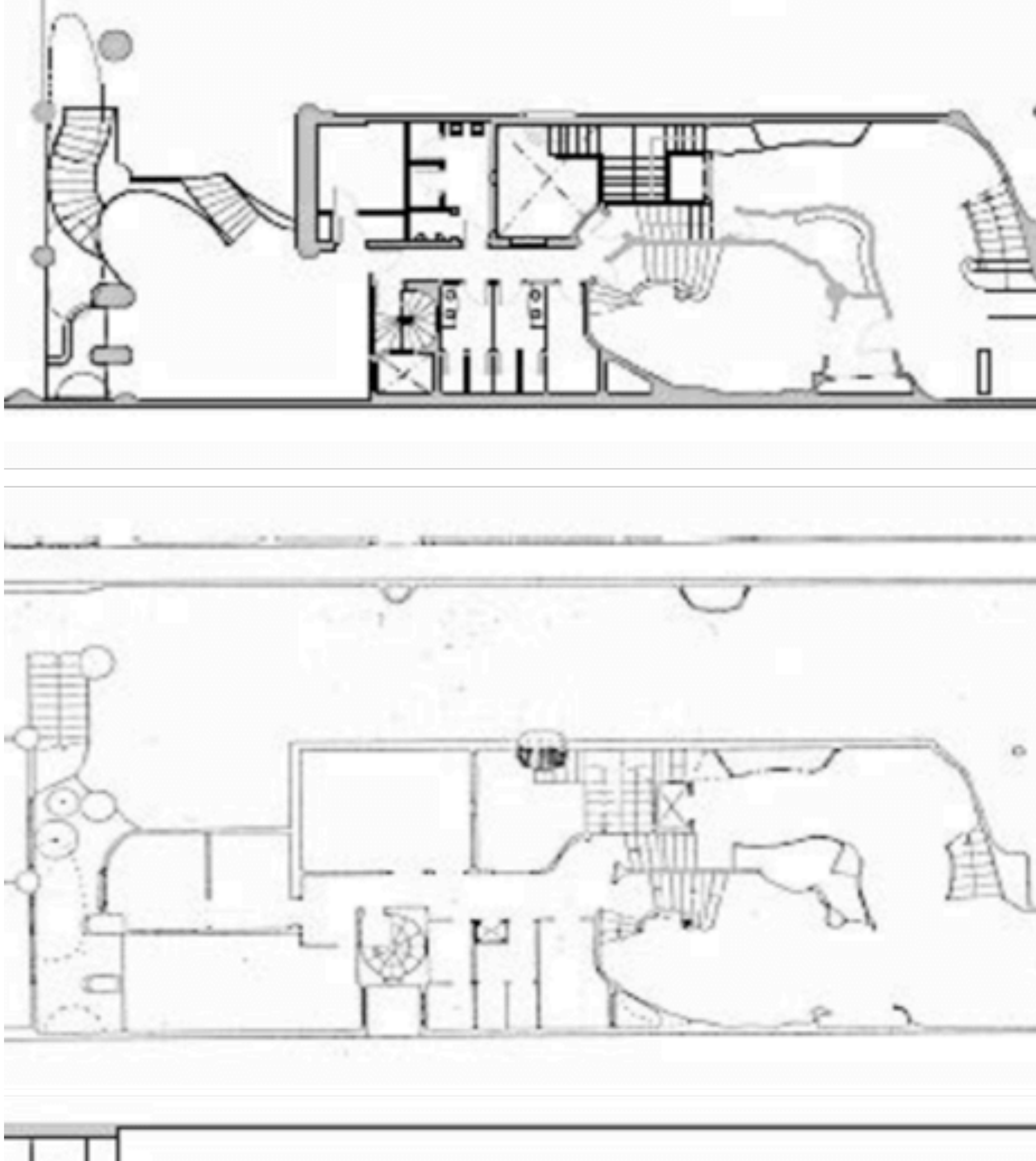
- Clear error **reporting**
- **Fast!**
- **Maintainable** test code

Economics of testing

- **Prioritize** what to test depending on the likeliness of detecting or preventing bugs
  - Even for trivial programs it is impossible or impractical to test every possible test input / initial state
- **Design** software in a way that testing is cheaper
- **Choose** your testing strategy wisely

Technical concepts (aka buzzwords)

- Black box / white box
- Broad stack / component tests
- Test doubles: dummy, fake, mock, stub, spy
- Unit tests
- Integration tests



Broad stack tests vs component tests

- **Broad stack test**: A test that exercises most of the parts of an application.
  - a.k.a. Full-stack tests, end-to-end tests.
- **Component (narrow) test**: A test that limits the scope of the exercised software to a portion of the system under test
  - The component is tested through its interface
  - Components used by the component under test can be replaced with **test doubles**

Test double

- Something (a component, value, etc) that replaces a production element for testing purposes.
  - **Dummy**: Values or components that are never used
  - **Fake**: A component with a working implementation that is not the one used in production (e.g. an in memory test database).
  - **Mocks, stubs and spies**

Test double

Mocks, stubs and spies

- **Mocks vs spies**
  - Mocks: Components that are pre-programmed with expectations which form a specification of the calls they are expected to receive.
  - Spies: Components that record some information based on how they were called, so you can do assertions on the recorded information after the fact.
- **Stubs**: Components that provide canned answers to calls made during the test.
  - Usually both mocks and spies are also stubs

Unit tests

- Slippery word meaning different things to different people:
  - Tests written by developers themselves
  - Focusing on small parts of the system:very **narrow** component tests
  - **Fast**, significantly faster than other kinds of tests

https://martinfowler.com/bliki/UnitTest.html

Integration tests

- Slippery word meaning different things to different people:
  - Originally: Test that separately developed modules worked together properly
  - Today: Test that the system **correctly interacts** with an external service (e.g. a database)
    - Require live versions of the service (e.g. an actual database)
    - Require networking
    - Usually isolated thanks to virtualization (e.g. docker)

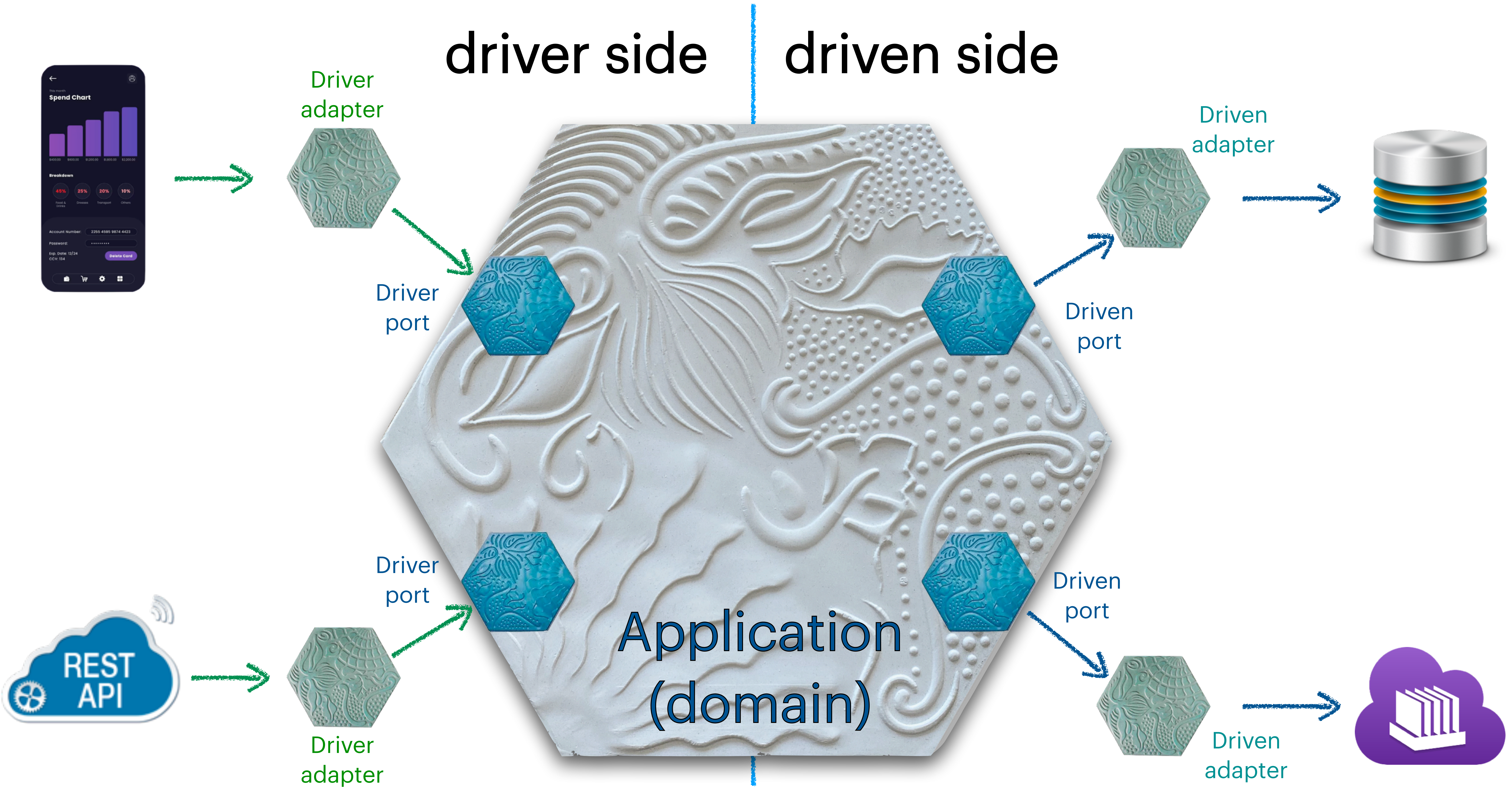
https://martinfowler.com/bliki/IntegrationTest.html

agilogy

A very gentle introduction to Hexagonal Architecture

agilogy

Hexagonal architecture



agilogy

Driver adapters
(aka transport layer)

- Low level implementation of APIs, listeners, etc to get commands, queries or events **from the outside to the driver ports** in the hexagon.

```
@POST
public Response createGroup(@PathParam("name") String name){
    campusApp.createGroup(name);
}
```

Driver ports
(aka interactors)

- **Domain interface** of the app used by the driver adapters to pass commands, queries and events in the language of the domain.

```
public final class CampusApp {

  public void createGroup(final String name) {
    usersRepository.createGroup(name);
  }
```

Driven ports

(aka repositories)

- **Domain abstraction** of the capabilities of external systems used by the application.

```java
public interface UsersRepository {

  void createGroup(final String name);
```

Driven adapters
(aka data sources)

- Low level **implementation** of the interfaces **of driven ports** that use specific external services or technologies (e.g. a PostgreSql database).

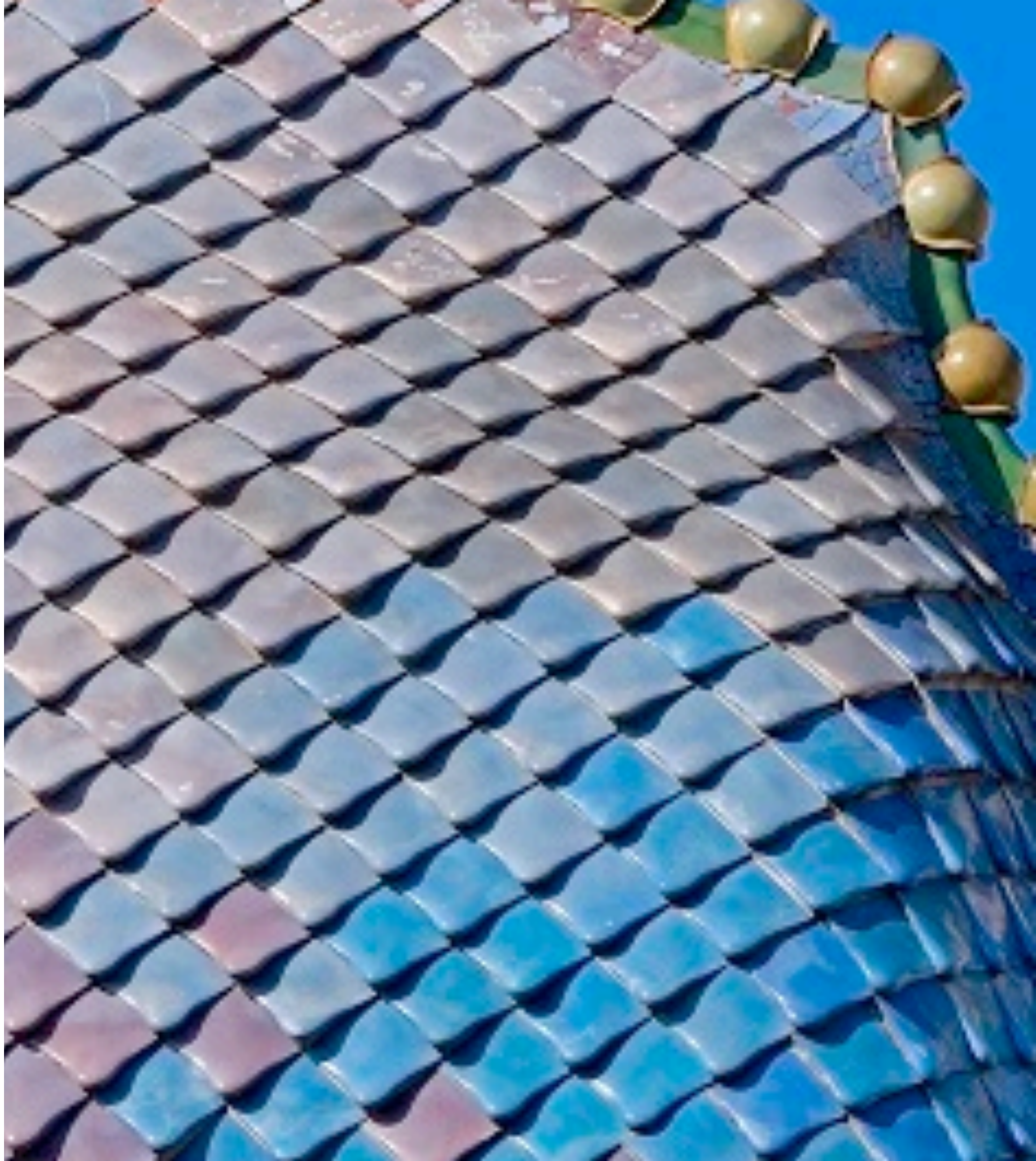
```
public final class PostgreSqlUsersRepository
  implements UsersRepository {

  @Override
  public void createGroup(final String name) {
    db.update("insert into groups (name) values (?)", p(name));
  }
```

agilogy

Patterns

- AAA: Arrange-Act-Assert
- AAA with state
- In-memory production-ready test fakes
- Isolated, production-like external systems



Arrange-Act-Assert (AAA)

- Used to: Test stateless components (e.g. a function that calculates fibonacci)
- Pattern:
  1. **A**rrange: Select the inputs to use and the expected result
  2. **A**ct: Exercise the component and collect any result
  3. **A**ssert: Assert the actual result is the expected result

AAA with state

- Used to: Test stateful components
- Pattern:
  1. Arrange: Select the inputs and initial state to use and **set the initial state**
  2. Act: Exercise the component and collect any result
  3. Assert:
     1. Collect the final state
     2. Assert the actual result is the expected result and **the final state is the expected state**

https://blog.agilogy.com/2022-06-17-testing-and-persistent-state.html

In-memory production-ready test fakes

- Use test fakes (that behave like the **production** component) implemented **in-memory** to avoid the dependency to external systems in tests
- 👍 Make tests run **fast**!
- 👍 Make tests **deterministic and isolated**
- 👎 They may behave **different** than the actual production components
- 👎 Implementation **cost**
- Alternatives:
    A. Test in integration with Isolated, production(-like) external systems
    B. Use mocks and stubs
        - 👎 You test the behavior of the system, not the outcome

Isolated, production(-like) external systems

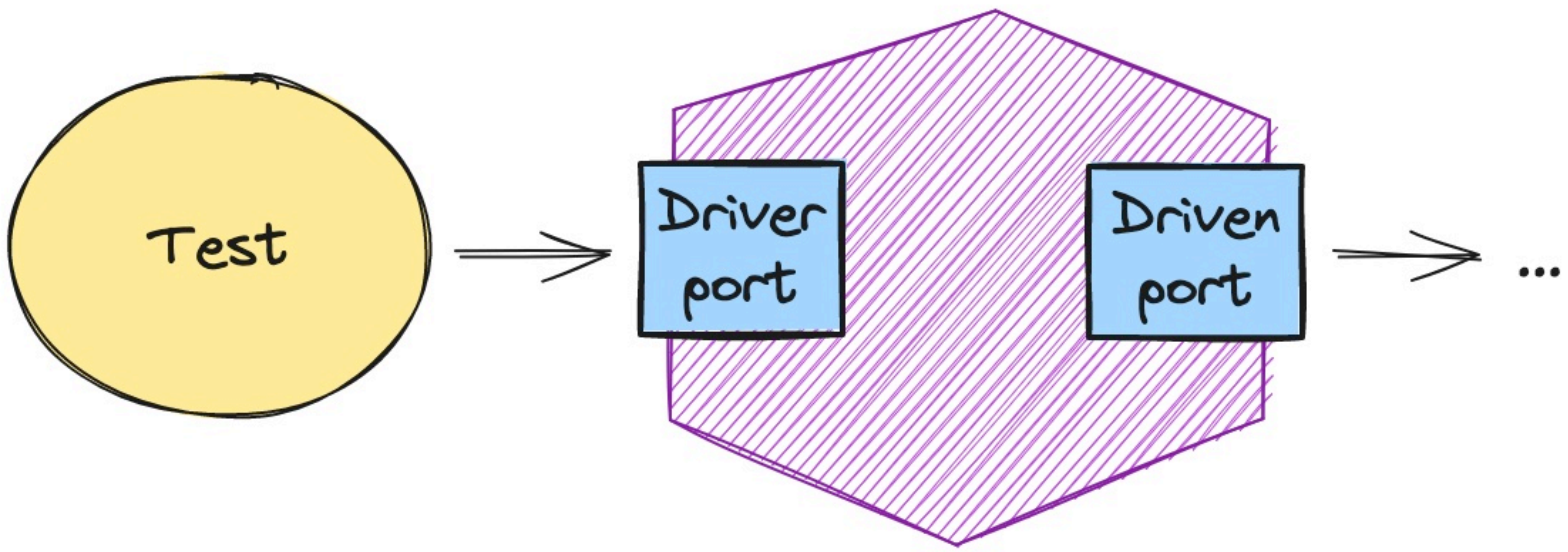
- Run integration tests with isolated versions of the **actual** production external systems
- Isolate tests from other tests, testers from other testers, etc.
- Usually through virtualization (e.g. Docker)
- 👍 You test **the actual component**
- 👎 **Slow**

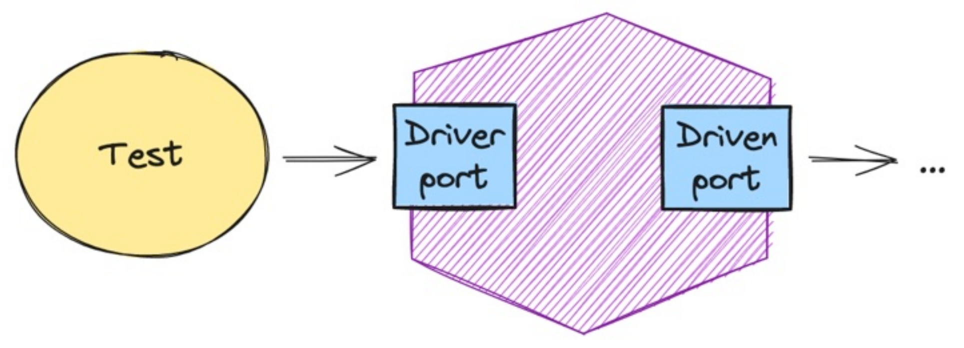
agilogy

Strategy

- Test all the domain
- Narrow tests of complex logic
- Integration tests of driven adapters
- Test driver adapters
- Test the assembly end-to-end
- Test test fakes

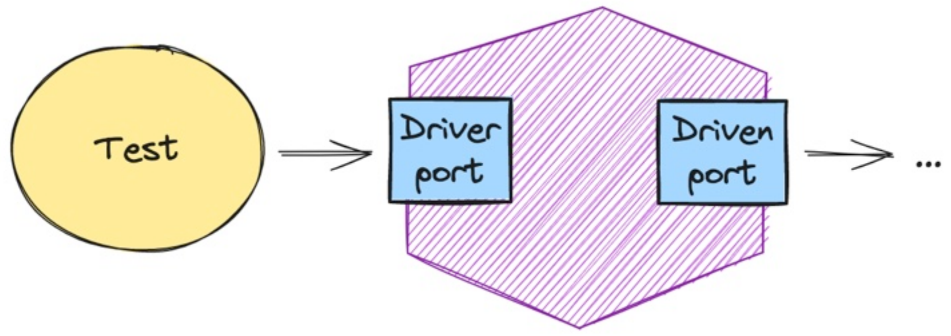
Test all the domain





Test all the domain

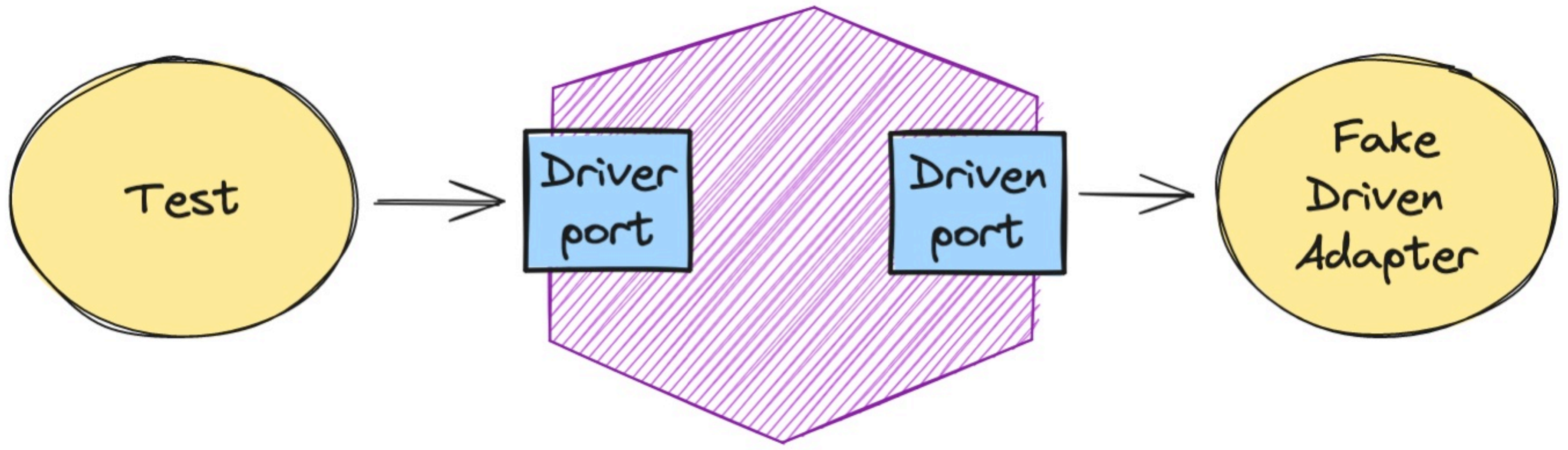
- **Broad** test of all the components that form the **domain**
  - Write tests of the whole domain of your system, including all domain components
  - Exercise it through the appropriate (driver) **ports**
  - Use AAA with state



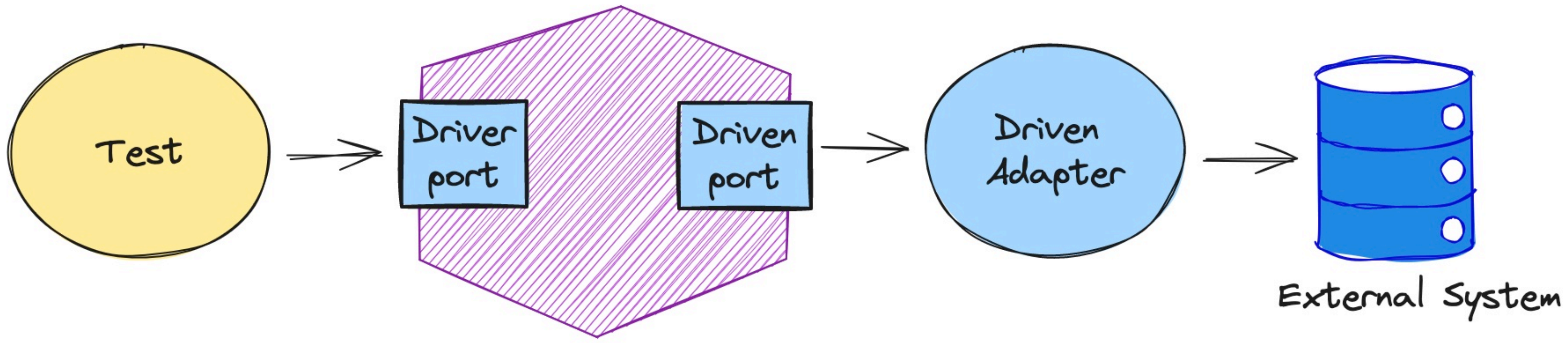
Test all the domain

Two alternative strategies

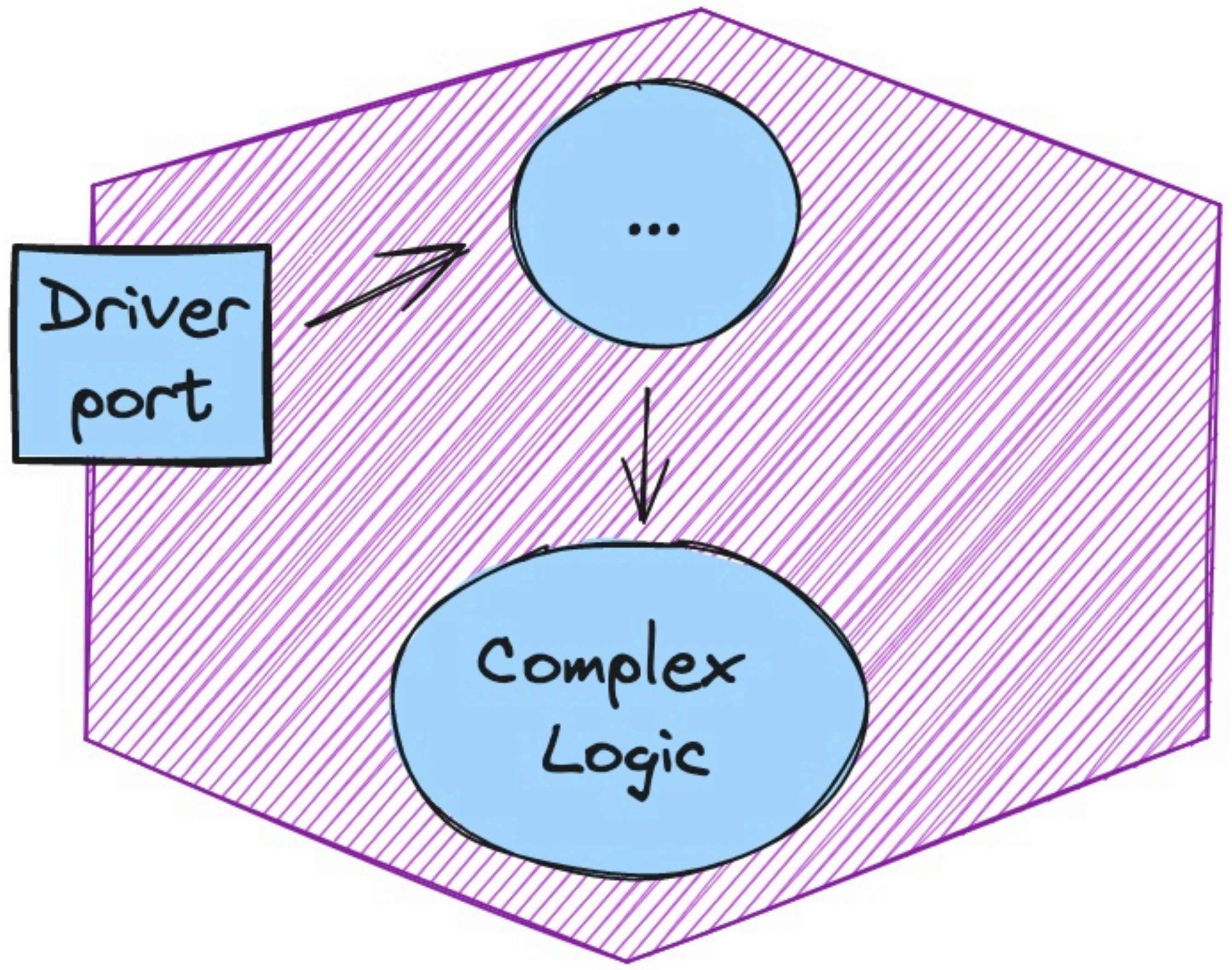
A. Test with In-memory, production-ready test fakes (**preferred**)

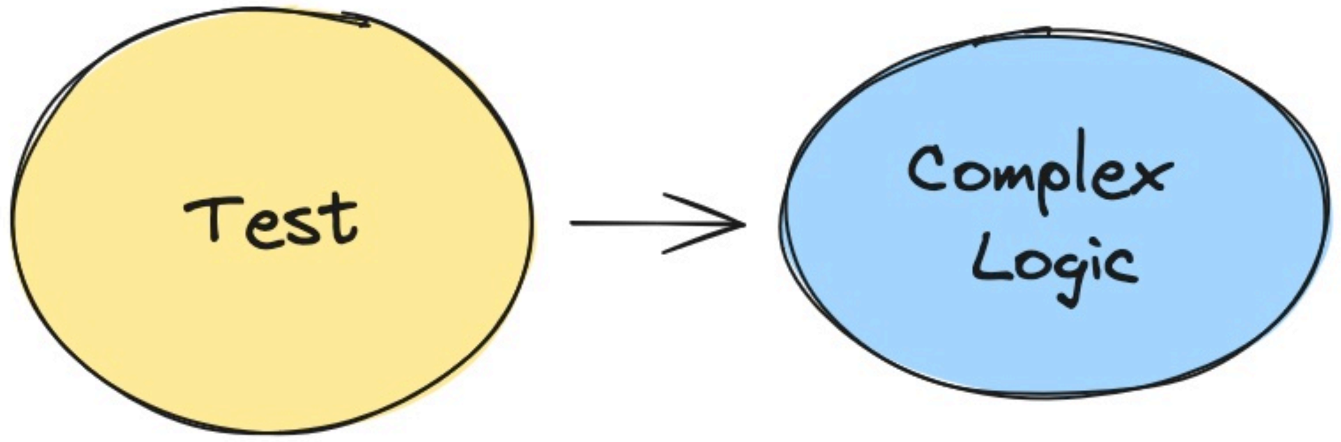


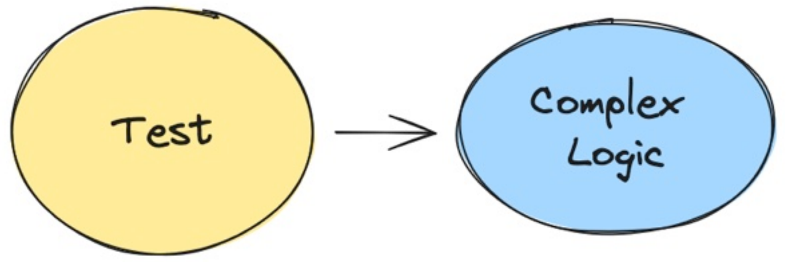
B. Test in integration with Isolated, production(-like) external systems (slower)



Narrow tests of complex logic



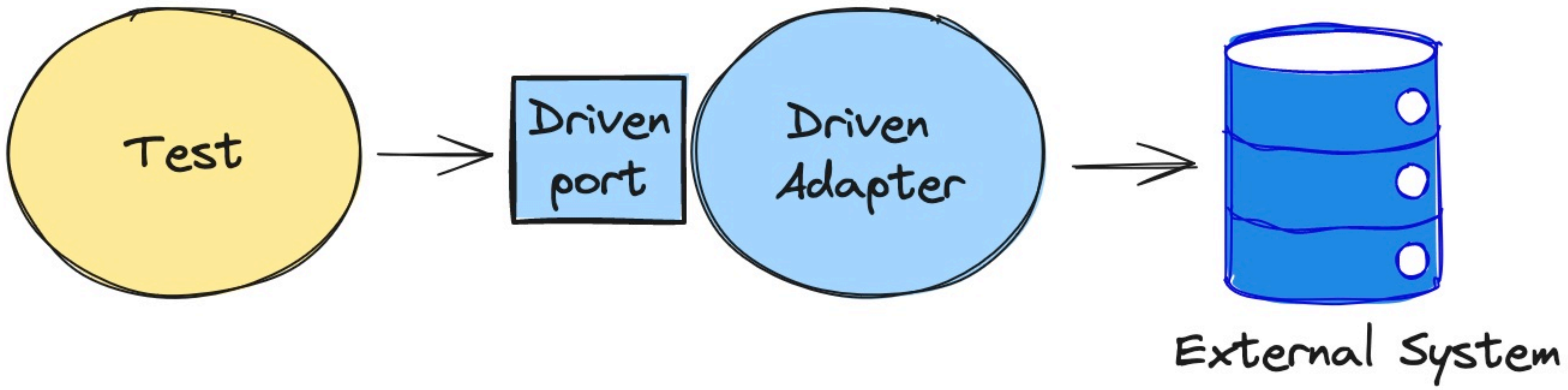


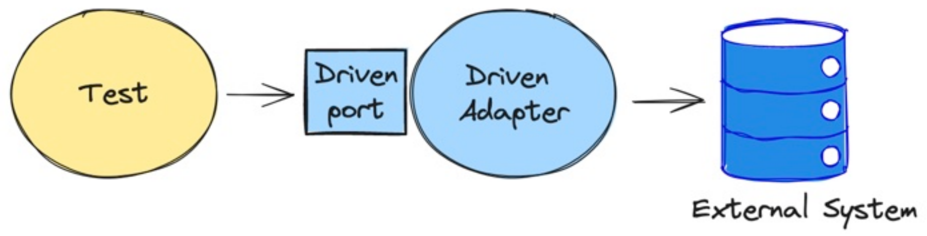


Narrow tests of complex logic

- Test complex logic inside your domain with **narrow** tests
  - Write a classical unit test to test some function or algorithm
  - Usually tests stateless components (AAA), but may test stateful components too (AAA with state)
  - Take into account they may be fragile to refactoring
  - Use really few of these

Integration tests of driven adapaters

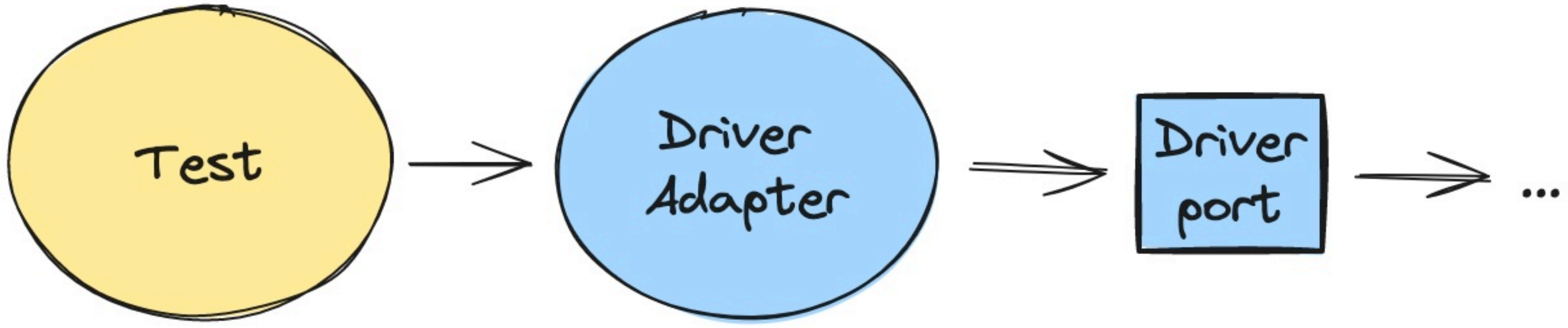




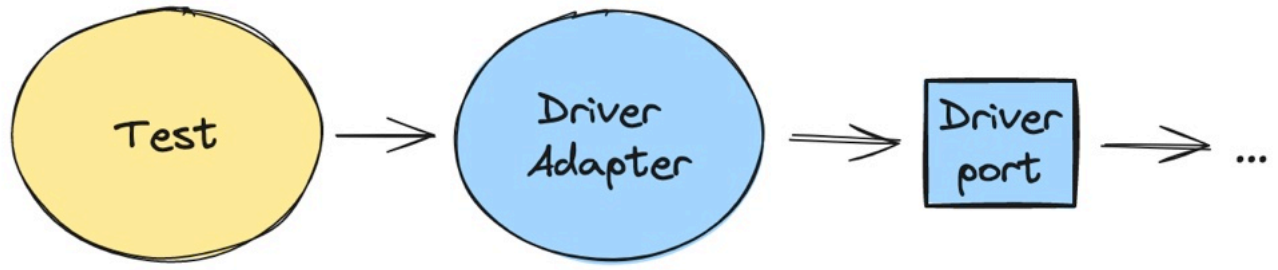
Integration tests of driven adapaters

- Test components interacting with external services in integration with the actual external services (e.g. databases, message queues, etc)
  - Test your driven adapters with integration tests (with Isolated, production-like external systems)
  - Test them in isolation
  - Use AAA with state

Test driver adapters

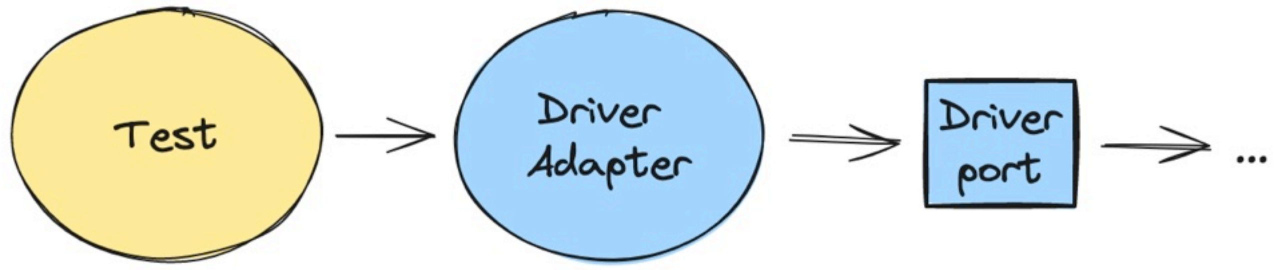


agilogy



Test driver adapters

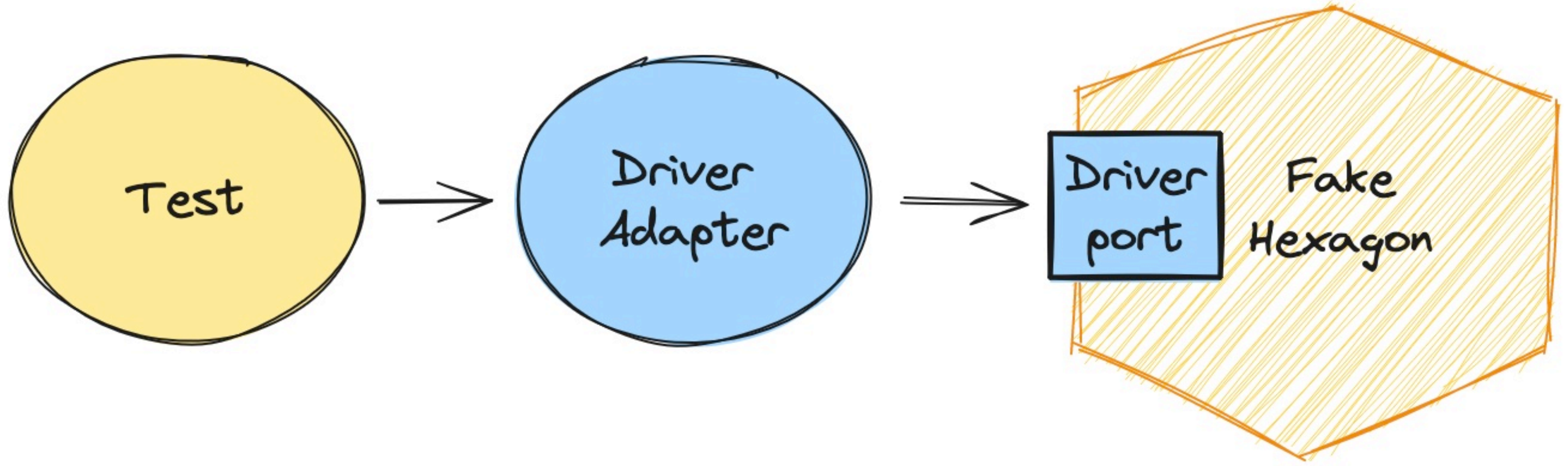
- Tests driver adapters: APIs, GUIs and other components driver the system
  - Two alternatives:
    A. Test their **behavior** with Mocks and stubs
    B. Test their **outcome** with the actual domain (and tests doubles for driven adapters)



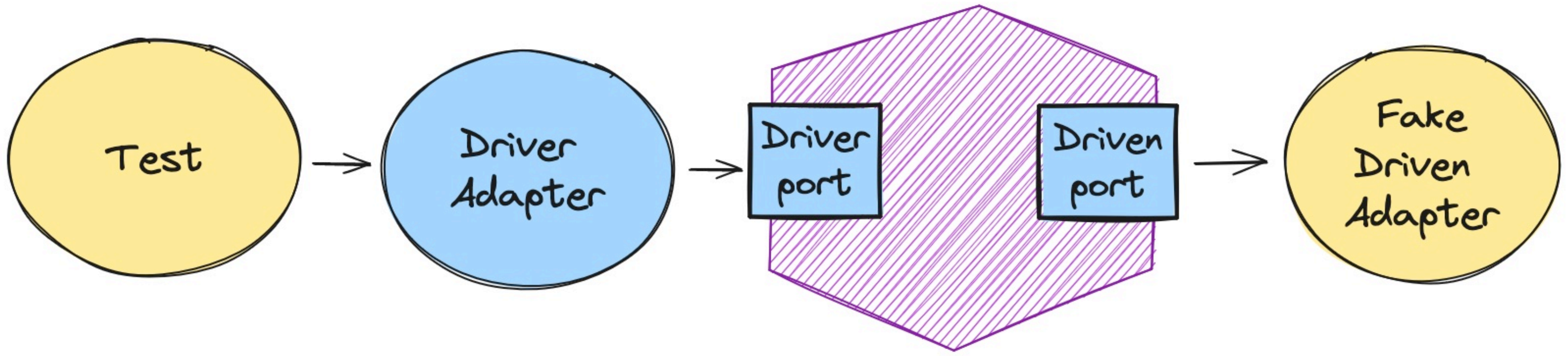
Test Driver Adapters

Two alternative strategies

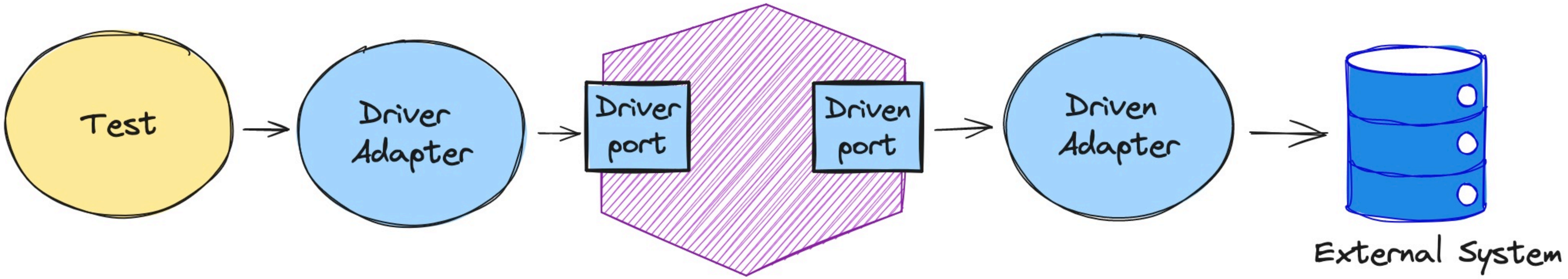
A. Test their **behavior** with <u>Mocks and stubs</u>

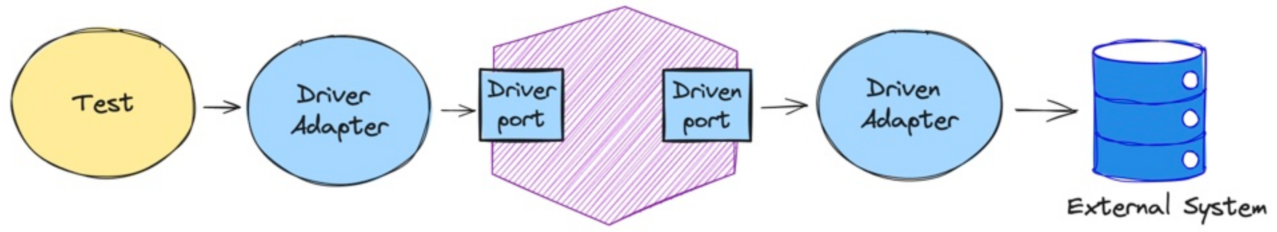


B. Test their **outcome** with the actual domain (and tests doubles for driven adapters)



Test the assembly with end-to-end tests

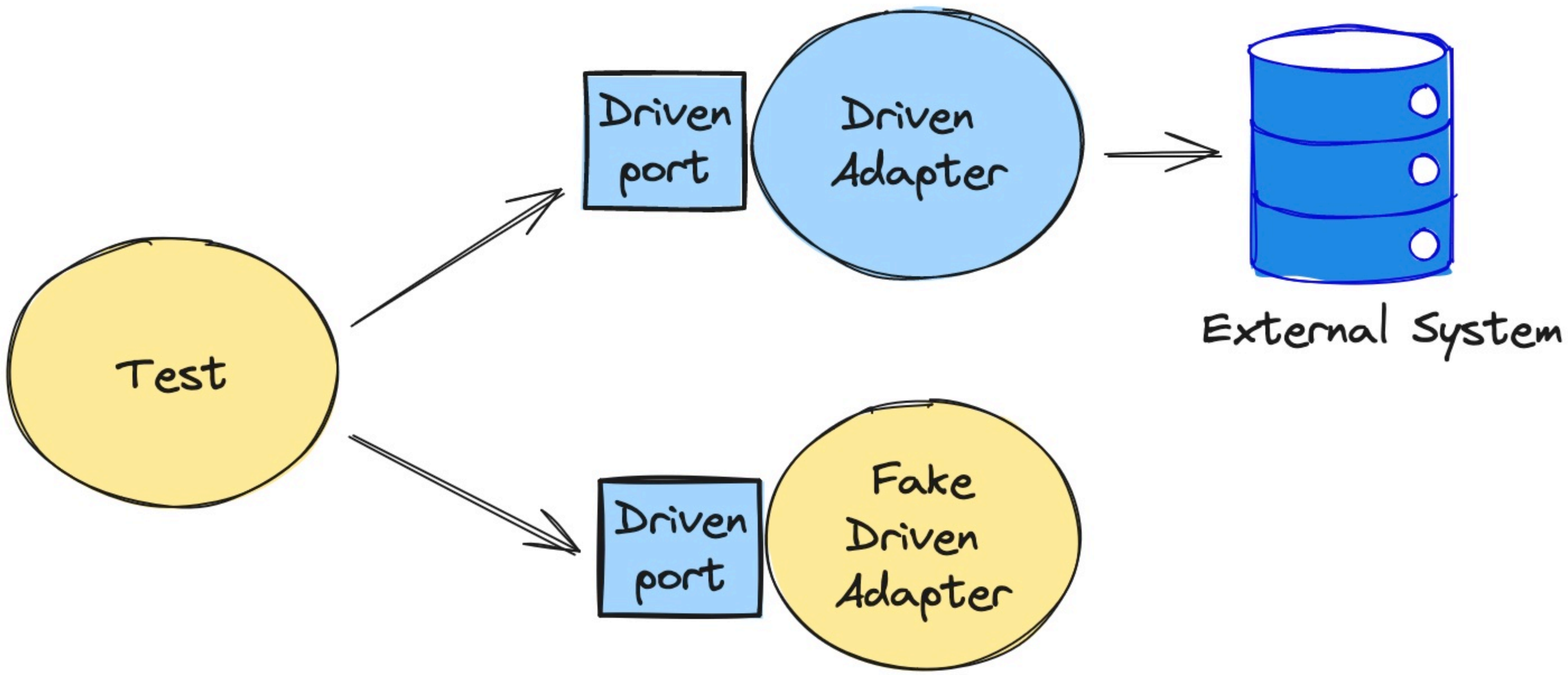




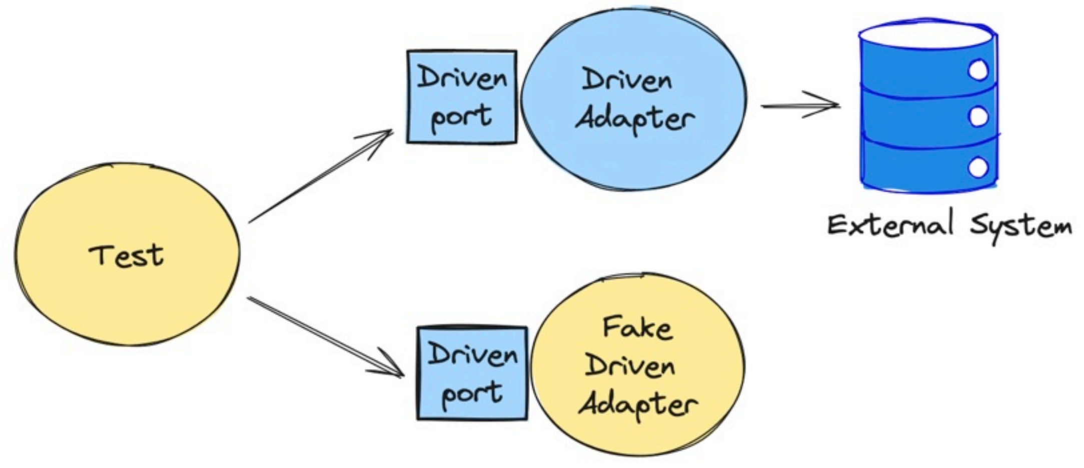
Test the assembly with end-to-end tests

- Write a few tests that check all the components are correctly assembled
  - Don't test whatever can be tested with the previous strategies
  - Use AAA with state and Isolated, production(-like) external services

Test test fakes



Test test fakes



- Many of the tests so far depend on In-memory, production-ready tests fakes to actually behave like production components.
- Make them actually production-ready by testing them like production components:
  - **Parameterize** the Integration tests of driven adapters so that they test whatever adapter of such ports they receive
  - Run the **same test suite for both** the production driven adapters and their test fakes to guarantee both behave exactly equal