TDD is not about testing

Bad name for good technique …

GPad

CTO & founder of coders51

Born to be a developer with an interest in distributed system.

I have developed with many languages like C++, C#, js and ruby. I had fallen in love with functional programming, especially with elixir, erlang.

- Twitter: https://twitter.com/gpad619
- Github: https://github.com/gpad/
- Medium: https://medium.com/@gpad

Schedule

What is TDD?

Why TDD?

TDD is not (only) Unit Testing

TDD in OOP vs FP

Examples

What is TDD - History

Wikipedia - TDD

Wikipedia - Extreme Programming

Quora - Why does Kent Beck rediscover TDD

http://wiki.c2.com/?TestingFramework

Wikipedia - C2 System

What is TDD - History

The C3 project started in 1993 [...]. Smalltalk development was initiated in 1994. [...] In 1996 Kent Beck was hired [...]; ***at this point the system had not printed a single paycheck****.*

In March 1996 the development team estimated the system would be ready to go into production around one year later.

In 1997 the development team adopted a way of working which is now formalized as Extreme Programming.

The one-year delivery target was nearly achieved, with actual delivery being a couple of months late;

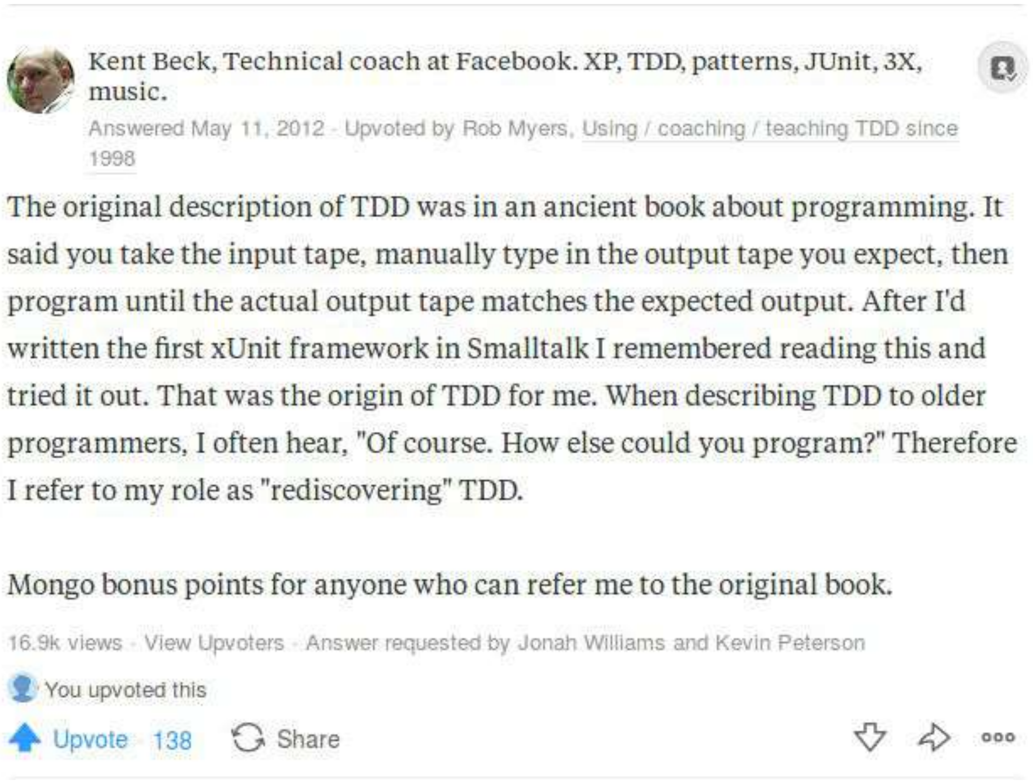
What is TDD - History

It was developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in 1994–1995, with further development led by them through 1996.

In 1997 UML was adopted as a standard by the Object Management Group (OMG) [...].

In 2005 UML was also published by the International Organization for Standardization (ISO) as an approved ISO standard. Since then the standard has been periodically revised to cover the latest revision of UML.

What is TDD - History



Kent Beck, Technical coach at Facebook. XP, TDD, patterns, JUnit, 3X, music.

Answered May 11, 2012 · Upvoted by Rob Myers, Using / coaching / teaching TDD since 1998

The original description of TDD was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output. After I'd written the first xUnit framework in Smalltalk I remembered reading this and tried it out. That was the origin of TDD for me. When describing TDD to older programmers, I often hear, "Of course. How else could you program?" Therefore I refer to my role as "rediscovering" TDD.

Mongo bonus points for anyone who can refer me to the original book.

16.9k views · View Upvoters · Answer requested by Jonah Williams and Kevin Peterson

You upvoted this

Upvote 138 Share

#CodeBEAMSTO

What is TDD - History

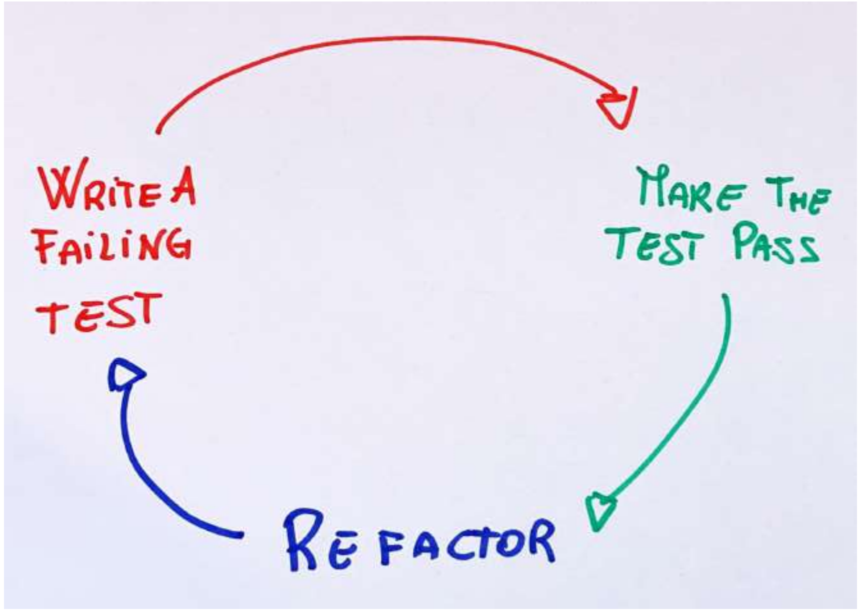
Form Wikipedia:

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only.

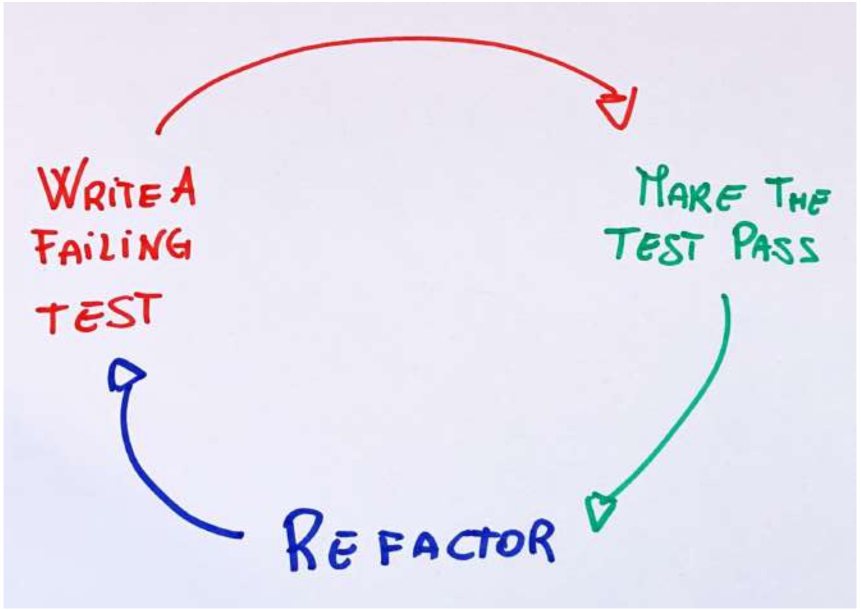
This is opposed to software development that allows software to be added that is not proven to meet requirements.

American software engineer Kent Beck, who is credited with having developed or "rediscovered" the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

What is TDD



What is TDD



1. Add a test
2. Run all tests and see if the new test fails (RED)
3. Write the code (Only to make the test pass!!!)
4. Run tests
5. Refactor code
6. Repeat

Why TDD

Why TDD

Less bugs

I like it …

I feel more comfortable …

…

Helps to create a ***“GOOD”*** design

Why TDD

TDD changes the point of view.

Forces the developer to think about the “behaviour” of the code.

Talks to the developer showing what are the “difficult points” of the code.

If the tests talk, we should listen them … (often we ignore them).

Listen the test

```
defmodule MyModule do
  def postpone(dest, msg) do
    msg = encapsulate(msg)
    time = get_delay()
    Process.send_after(dest, msg, time)
  end

  defp encapsulate(msg) do
    {:postone, msg}
  end

  def get_delay() do
    Application.get_env(:postone, :time, 10000)
  end
end
```

Listen the test

```
defmodule MyModuleTest do
  use ExUnit.Case

  test "postone right" do
    MyModule.postpone(self(), "ciao")

    assert_receive {:postone, "ciao"}, 20_000
  end
end
```

The name doesn't mean anything

Our test is sloooow ...

Listen the test - other smells ...

```
defmodule MyModuleTest do
  use ExUnit.Case

  test "postpone right" do
    Application.put_env(:postone, :time, 1)
    MyModule.postpone(self(), "ciao")

    assert_receive {:postone, "ciao"}
  end

  test "postpone for right time ..." do
    MyModule.postpone(self(), "ciao")

    refute_receive {:postone, "ciao"}, 5000
  end
end
```

Change GLOBAL value!!!!

No More isolated !!!

Listen the test

```
defmodule MyModuleTest do
  use ExUnit.Case, async: false
  import Mock

  test "postone right with mock" do
    with_mock(Process, [:passthrough], send_after: fn dst, msg, _ -> send(dst, msg) end) do
      MyModule.postpone(self(), "ciao")

      assert_receive {:postone, msg}
    end
  end
end
```

No more concurrency

Why this ?!?!

Still doesn't work ...

This is not our module

Listen the test

It isn’t a tool problem.

It’s a design problem …

What is telling (screaming) the test?

Listen the test

```
defmodule MyModuleTest do
  use ExUnit.Case

  test "postone right" do
    MyModule.postpone(self(), "ciao")

    assert_receive {:postone, "ciao"}, 20_000
  end
end
```

The name doesn't mean anything

Our test is sloooow ...

Listen the test

```
defmodule MyModuleTest do
  use ExUnit.Case

  test "postone/1 send delayed message" do
    MyModule.postpone(self(), "ciao", 0)

    assert_receive {:postone, "ciao"}
  end
end
```

Describe the behaviour of method

We can manage the behaviour of our function ...

Listen the test

```
defmodule MyModule do
  @postpone_time Application.get_env(:postone, :time, 10000)

  def postpone(dest, msg, time \\ @postpone_time) do
    msg = encapsulate(msg)
    Process.send_after(dest, msg, time)
  end

  defp encapsulate(msg) do
    {:postone, msg}
  end
end
```

Can we do better?!?

TDD doesn’t mean Unit Testing …

Type of tests

End to End tests - Tests that work on the entire stack.

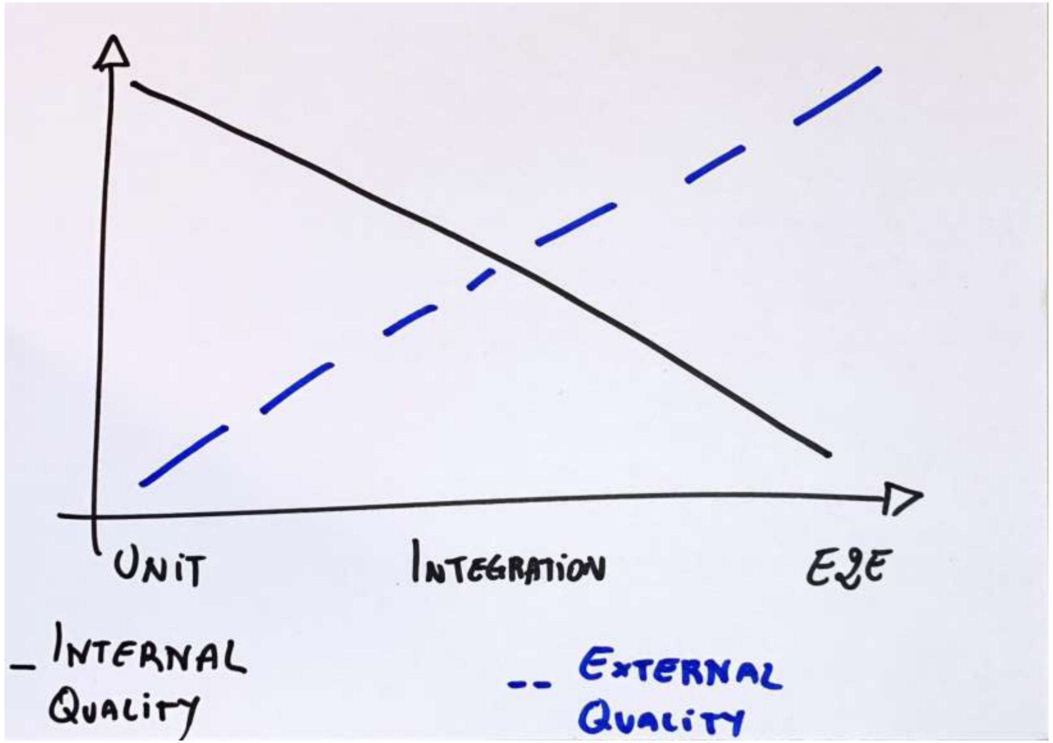
Integration Test - Tests that work on some parts of the application.

Unit Test - Tests that work on a single “module/function”.

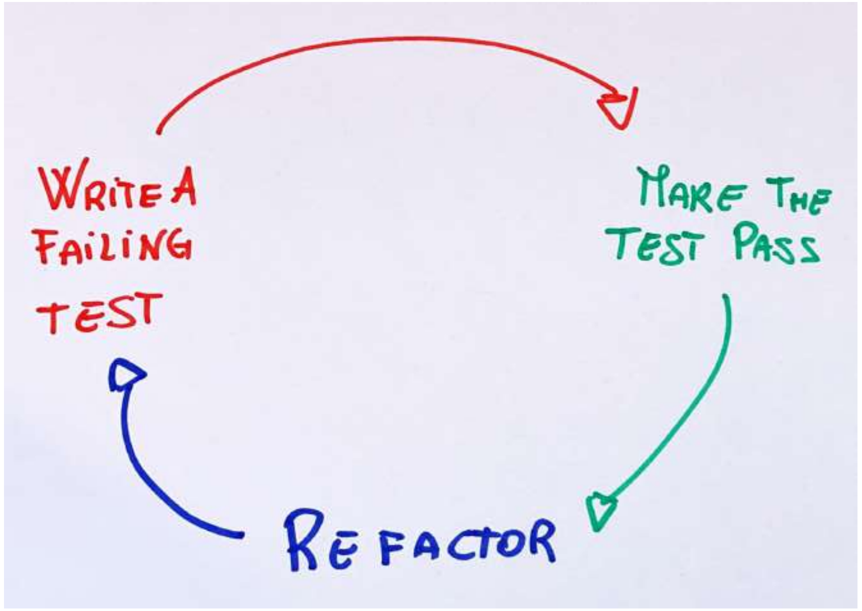
Why is it important to know which type of test we are writing?

Because we get different feedbacks from different types of test.

Type of tests

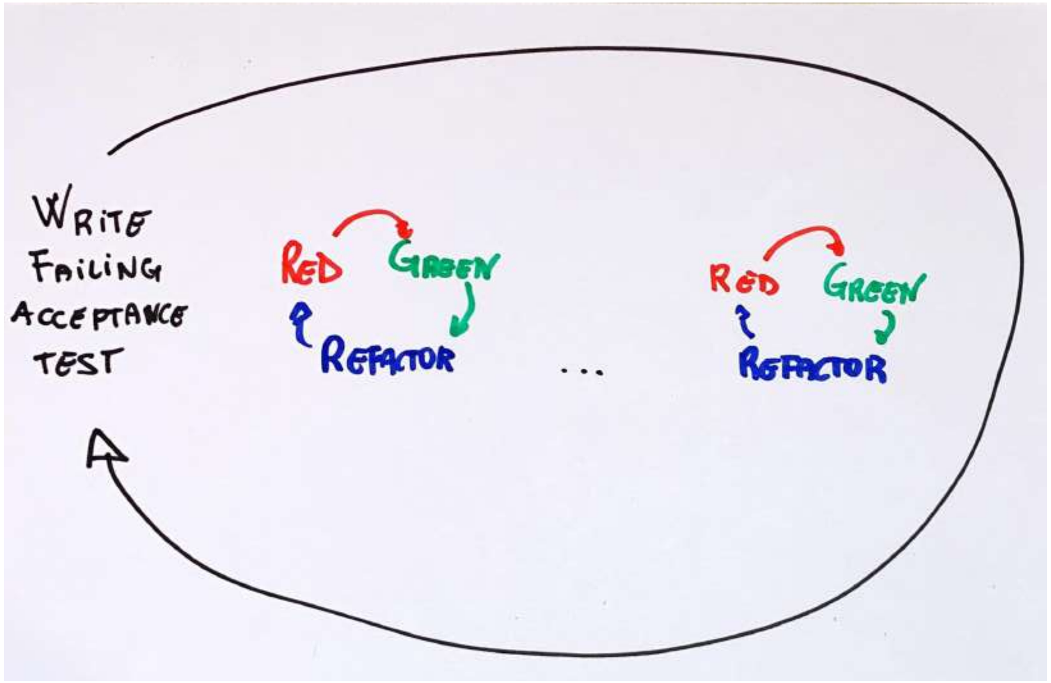


Cycle of TDD ...



1. Add a test
2. Run all tests and see if the new test fails (RED)
3. Write the code
4. Run tests
5. Refactor code
6. Repeat

Cycle of TDD ...


WRITE FAILING ACCEPTANCE TEST
RED
GREEN
REFACTOR
...
RED
GREEN
REFACTOR

End To End Test or Acceptance Test

This type of test exercises the ***entire*** stack of the application.

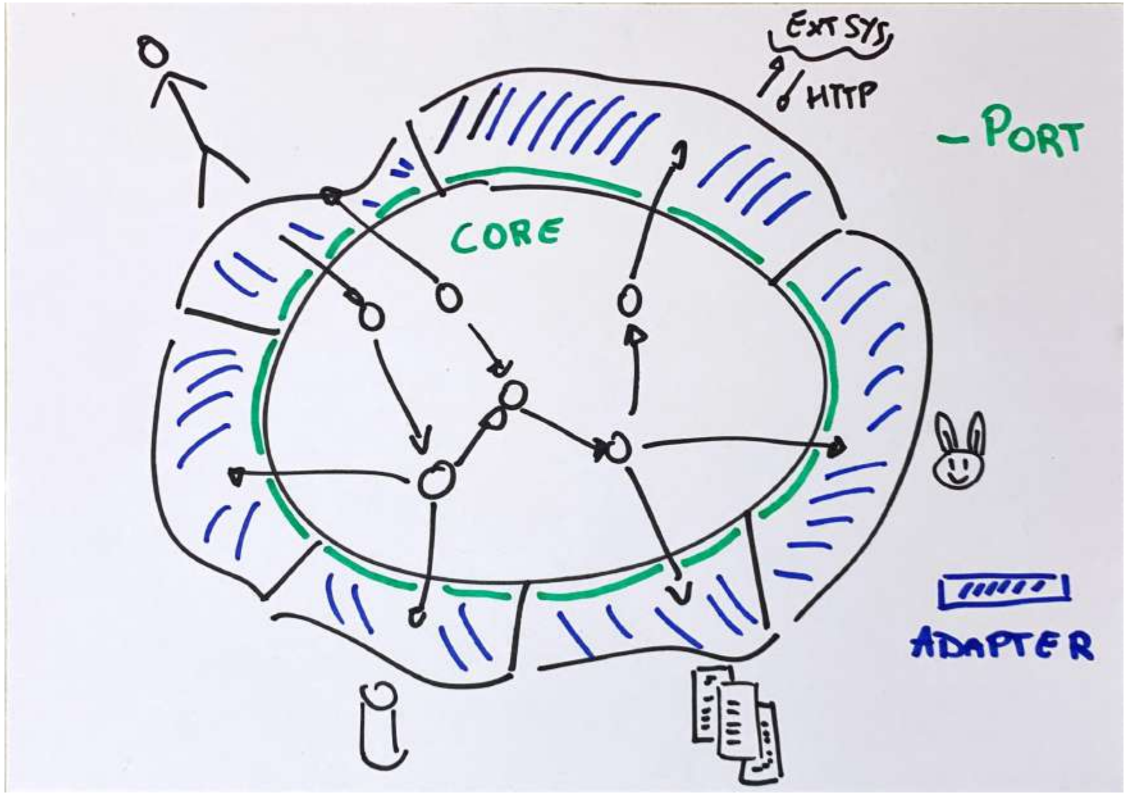
It remains *RED* until the feature is completed.

Don’t write too much E2E.

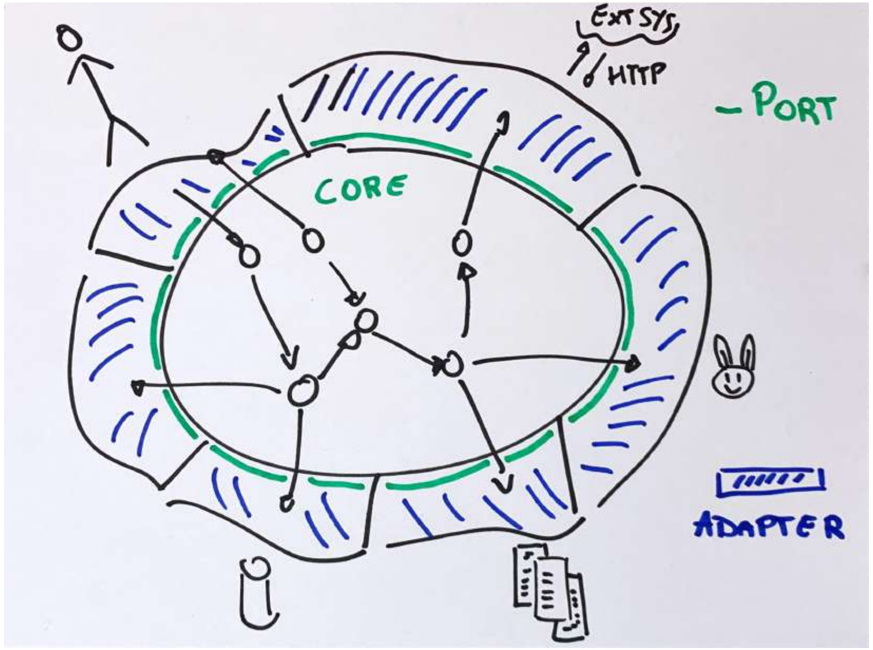
They are slow and fragile.

Where is the design?

How application is done (or should be)



How application is done (or should be)



Try to create a E2E test that interacts with system from the external.

If it’s “impossible” try to move a little inside skipping the adapter.

Are we creating a Java application?

OOP vs FP

OOP - We pass object reference to Object under test!

FP - We have immutability!

Elixir - We have **almost always** immutability!

Immutability means purity.

Purity means no side effects.

OOP vs FP

What are the instructions that make the code impure?

Everytime we make an I/O operation we aren’t pure.

Everytime we send a message.

Everytime we use the time.

OOP vs FP

What’s happen when we lost our purity?

We depend to something “external”.

We are doing at least integration test.

So …

OOP vs FP

In OOP is quite common use mock to *mock* dependencies of objects and to define the relationship between them.

In FP (IMHO) mock should be used to manage the impurity of our language.

Mock

When to use Mock?

When we want to isolate one part from another.

Classic example HTTP connections.

We have to make some HTTP calls and manipulate the results in some ways.

How can we do it?

When to use Mock?

DON’T TRY THIS AT HOME!!

```
test "create data via https", %{id: id} do
  response = create_response(201, %{key: "value"})
  with_mock HTTPoison, [:passthrough], post: fn @url, _, _, _ -> {:ok, response} end do

    {:ok, %{res: value}} = MyModule.execute(id)

    assert value == "value"
    assert called(HTTPoison.post(@url, %{id: id}, :_, :_))
  end
end
```

When to use Mock?

Better solution

Our module

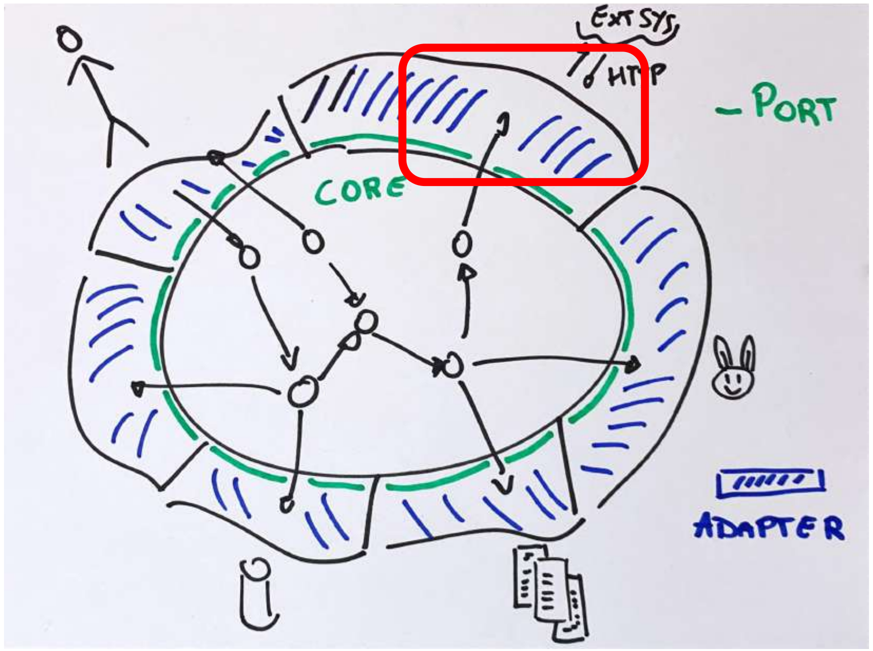
```
test "create data via https", %{id: id} do
  with_mock RemoteData, create: fn %{id: id} -> {:ok, %{key: "value"}} end do

    {:ok, %{res: "value"}} = MyModule.execute(id)

    assert called(RemoteData.create(%{id: id}))
  end
end”
```

More domain related function ...

When to use Mock?



We are at the boundaries of our system and we can use the mock to shape the behavior between the CORE and the adapter that talks with the external system.

It's the CORE that choose the shape of the data and how the functions should be done.

When to use Mock? - I/O

```
defmodule MyCoreModule do

  def store_valid_data(data, limit, delta) do
    content = data
      |> Enum.filter(fn %{value: value} -> around(value, limit, delta) end)
      |> Enum.map(fn %{value: v, comment: c} -> "value: #{v} - comment: #{c}" end)
      |> Enum.join("\n")

    :ok = File.write(@path, content)
  end
end
```

When to use Mock? - I/O

```
defmodule MyCoreModule do

  def store_valid_data(data, limit, delta, storage \\ FileStorage) do
    content = data
      |> Enum.filter(fn %{value: value} -> around(value, limit, delta) end)
      |> Enum.map(fn %{value: v, comment: c} -> "value: #{v} - comment: #{c}" end)
      |> Enum.join("\n")

    :ok = storage.store(content)
  end
end
```

When to use Mock? - I/O

```
defmodule MyCoreModuleTest do
  use ExUnit.Case

  defmodule MockStorage do
    def store(content) do
      send(self(), {:store, content})
      :ok
    end
  end
```

```
test "store only valid data", %{limit: limit, delta: delta} do
  valid = valid_data(limit, delta)
  invalid = invalid_data(limit, delta)
  content = "value: #{valid.value} - comment: #{valid.comment}"
  MyCoreModule.store_valid_data(
    [valid, invalid], limit, delta, MockStorage)

  assert_received {:store, ^content}
end
```

When to use Mock? - Time

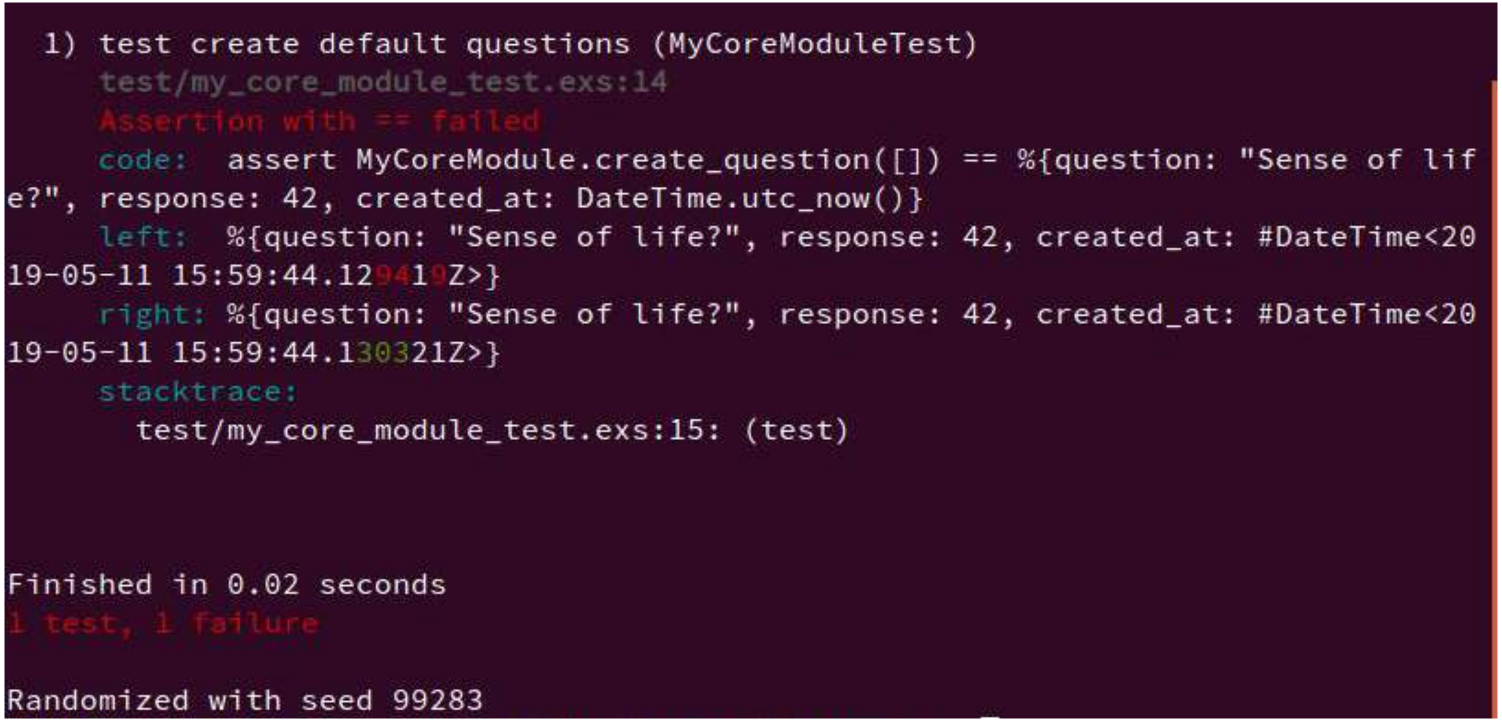
```
defmodule MyCoreModule do
  def create_question(opts) do
    %{
          question: Keyword.get(opts, :q, "Sense of life?"),
          response: Keyword.get(opts, :a, 42),
          created_at: DateTime.utc_now()
    }
  end
end
```

When to use Mock? - Time

```
defmodule MyCoreModuleTest do
  use ExUnit.Case

  test "create questions" do
    assert MyCoreModule.create_question([]) == %{
      question: "Sense of life?",
      response: 42,
      created_at: DateTime.utc_now()
    }
  end
end
```

When to use Mock? - Time



```
  1) test create default questions (MyCoreModuleTest)
     test/my_core_module_test.exs:14
     Assertion with == failed
     code:  assert MyCoreModule.create_question([]) == %{question: "Sense of lif
e?", response: 42, created_at: DateTime.utc_now()}
     left:  %{question: "Sense of life?", response: 42, created_at: #DateTime<20
19-05-11 15:59:44.129419Z>}
     right: %{question: "Sense of life?", response: 42, created_at: #DateTime<20
19-05-11 15:59:44.130321Z>}
     stacktrace:
       test/my_core_module_test.exs:15: (test)



Finished in 0.02 seconds
1 test, 1 failure

Randomized with seed 99283
```

#CodeBEAMSTO

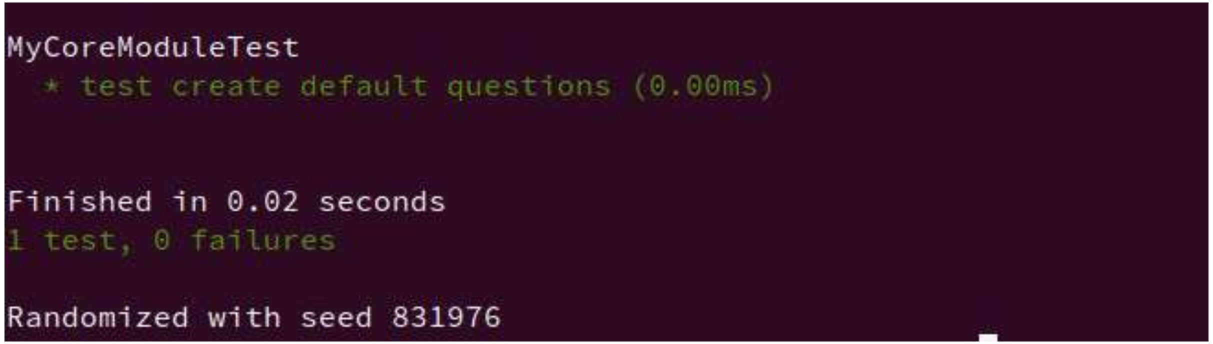
When to use Mock? - Time

```
defmodule MyCoreModule do
 def create_question(opts, clock \\ Clock) do
   %{
     question: Keyword.get(opts, :q, "Sense of life?"),
     response: Keyword.get(opts, :a, 42),
     created_at: clock.utc_now()
   }
   end
end
```

When to use Mock? - Time

```
test "create default questions" do
  assert MyCoreModule.create_question([], MockClock) == %{
    question: "Sense of life?",
    response: 42,
    created_at: MockClock.utc_now()
  }
  end
```

When to use Mock? - Time



```
MyCoreModuleTest
  * test create default questions (0.00ms)


Finished in 0.02 seconds
1 test, 0 failures

Randomized with seed 831976
```

When to use Mock? - Time

How to manage a periodic task?

Divide et impera.

When to use Mock? - Time

```
defmodule MyCoreModuleTest do
  use ExUnit.Case

  test "change state at every tick" do
    prev = MyCoreModule.new()
    next = MyCoreModule.tick(prev)
    assert next.value == prev.value + 1
  end
end
```

When to use Mock? - Time

```
defmodule MyCoreModuleServerTest do
  use ExUnit.Case

  test "call tick when receive tick" do
    with_mock(MyCoreModule, tick: fn s -> s end) do
      pid = MyCoreModuleServer.start_link()

      send(pid, :tick)

      eventually_assert(fn -> assert_called(MyCoreModule.tick(:_)) end)
    end
  end
end
```

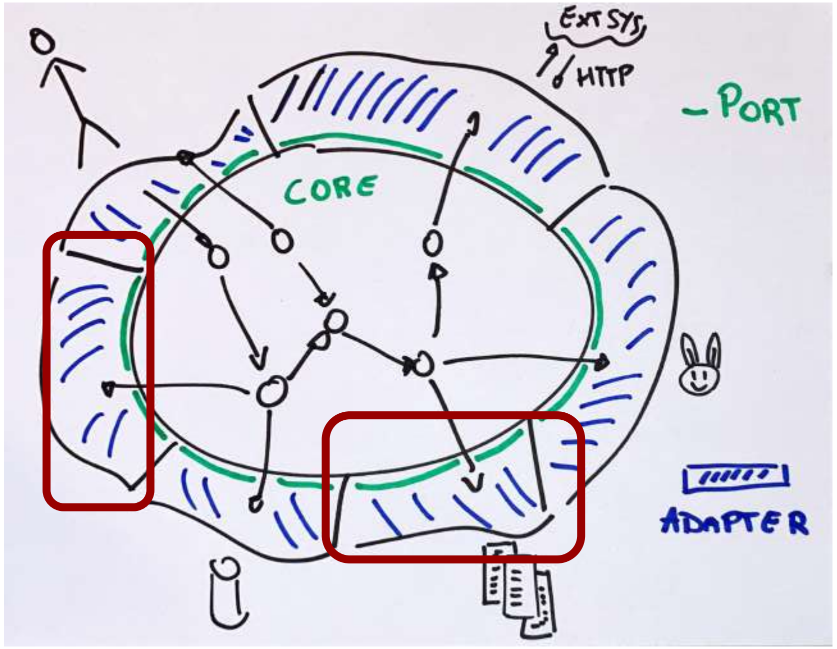
When to use Mock? - Time

```
defmodule MyCoreModuleServer do
  use GenServer

  def init([other_args, duration]) do
    state = calc_state(other_args, duration)
    Process.send_after(self(), :tick, duration)
    {:ok, state}
  end

  def handle_info(:tick, state) do
    new_core = MyCoreModule.tick(state.core)
    Process.send_after(self(), :tick, state.duration)
    {:noreply, %{state | core: new_core}}
  end
```

WHERE to use Mock?



Use the mock at the external of CORE/Domain (adapter).

Try to keep the CORE pure so we don't have "side-effect" inside it.

When not to use Mock?

Don’t use mock on Module that don’t own.

```
test "create data via https", %{id: id} do
  response = create_response(201, %{key: "value"})
  with_mock HTTPoison, [:passthrough], post: fn @url, _, _, _ -> {:ok, response} end do

    {:ok, %{res: value}} = MyModule.execute(id)

    assert value == "value"
    assert called(HTTPoison.post(@url, %{id: id}, :_, :_))
  end
end
```

When not to use Mock?

Don’t use mock because otherwise “you aren’t doing unit test”

```
defmodule MyCoreModule1 do
  def execute(data) do
    data
    |> Enum.map(fn -> {data.id, data.value, data} end)
    |> MyCoreModule2.filter(data.filter)
    |> MyCoreModule3.map(data.type)
  end
end
```

When not to use Mock?

```
defmodule MyCoreModule1Test do
  use ExUnit.Case

  test "execute filter and map data" do
    with_mocks [
      {MyCoreModule2, [], filter: fn data, _level -> data end},
      {MyCoreModule3, [], map: fn data, _type -> data end}
    ] do
        MyCoreModule1.execute([%{id: 12, value: 1, filter: 10, type: OtherModule}])
        assert called(MyCoreModule2.filter(:_, :_))
        assert called(MyCoreModule3.map(:_, :_))
      end
  end
end
```

TDD - Mix Everything together

What is the first test to do?

We could start from an E2E test to enter ***inside*** our application.

Create at least one E2E for every ***User Story.***

Don’t create too much E2E they are slow and fragile.

TDD - Mix Everything together

```
defmodule MayApp.EndToEnd.GamesTest do
  use MayApp.ConnCase, async: false

  describe "As Logged User" do
    test "I would see my active games", %{conn: conn} do
      {:ok, game} = create_active_game()
      conn = get(conn, "/api/games")
      assert json_response(conn, 200) == expected_response_for(game)
    end

    ...
```

TDD - Mix Everything together

```
defmodule MayApp.EndToEnd.GamesTest do
  use MayApp.ConnCase, async: false

  describe "As Logged User" do
    ...
    test "I would create a new games", %{conn: conn} do
      conn = post(conn, "/api/games")
      res = json_response(conn, 200)
      assert %{"id" => _, "status" => "active"} = res
    end
  end
end
```

TDD - Mix Everything together

The E2E remains RED until all the cycle is completed.

After that we have written the E2E we go inside the CORE and start to create some **unit tests**.

The Unit Test should be ***PURE.***

TDD - Mix Everything together

```
defmodule MayApp.MyCoreModuleTest do
  use ExUnit.Case

  test "at every tick the count is incremented" do
    state = MyCoreModule.new(MockClock)

    new_state = MyCoreModule.tick(state)

    assert new_state.count == state.count + 1
    assert new_state.last_updated_at == MockClock.utc_now()
  end


  ...
end
```

TDD - Mix Everything together

```
defmodule MayApp.MyCoreModuleTest do
  use ExUnit.Case

  test "at every tick the count is incremented" do
    state = MyCoreModule.new()

    new_state = MyCoreModule.tick(state, now)

    assert new_state.count == state.count + 1
    assert new_state.last_updated_at == now
  end


  ...
end
```

TDD - Mix Everything together

After we have written the unit tests for the CORE we could move to the boundaries where we should write tests for the adapter parts.

The test for storage part should be written using the DB. IMHO they are more integration than unit.

TDD - Mix Everything together

```
defmodule MayApp.GameServerTest do
  use ExUnit.Case, async: false

  test "at every tick the state is changed" do
    id = 123
    game = %{id: id, state: :ready}
    with_mocks [{GameStorage, load: fn ^id -> game end},
      {MyCoreModule, tick: fn ^game, _ -> game end}] do
      {:ok, pid} = GameServer.start_link(id, tick: :manual)
      GameServer.tick(pid)
    assert_called(MyCoreModule.tick(game, :_))
  end
end
```

TDD - Mix Everything together

```
defmodule MayApp.GameServerTest do
  use ExUnit.Case, async: false

  test "at every tick the new game is stored" do
    game = Game.new()
    GameStorage.save_game(game)
    {:ok, pid} = GameServer.start_link(game.id, tick: :manual)

    GameServer.tick(pid)

    eventually_assert(fn ->
      new_game = GameStorage.load(game.id)
      assert new_game.count == game.count + 1
    end)
  end
end
```

TDD - Mix Everything together

Writing these tests ***BEFORE*** the implementation we are doing ***DESIGN.***

We are shaping the production code.

The code became more “composable”.

It’s more clear where are side effects (I/O, Time).

It’s more clear what are the different parts of our applications.

Recap

TDD is not a Silver Bullet.

TDD doesn’t give us a “good” design if we are not able to do it.

TDD can help us to find some issues in our design.

Listen the test, often they are screaming in pain ...

Reference

GOOS

Clean Architecture

Unit Test in Elixir

Mocks and explicit contracts

Property based testing with PropEr, Erlang and Elixir

Testing Elixir

End - Thanks !!!