Testability + Fakes

Design for testability + Test types

HENRIK BITSCH KIRK
ASSISTANT PROFESSOR

Unit-Test recap

- [TestFixture]
- [TestCase]
  - Assert.That(x, Is.EqualTo(42)

- How should we test the filter class?

Design for Testability - dependencies

"In order to test a functionality of a class, we first need to **detach it from the rest of system in which it is designed to work**. We then need to create an instance from that class, activate the tested functionality and finish by making sure the resulting behavior matches our expectations.

However, u**nless the system is designed specifically to enable this, in most cases, it will not be simple**."

Gil Zilberfeld, http://www.infoq.com/articles/Testability

Towards controlling dependencies

- A testable design allows us to detach a single class (the UUT) from the rest of the system.

- Then, we can control which dependencies the UUT uses, e.g. our “fake” versions of the dependencies

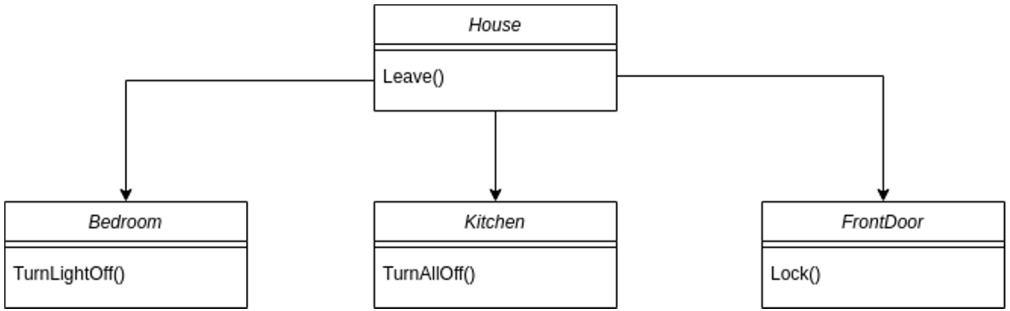
- There are 3 steps towards control (III or Triple-I):
  1. **IDENTIFY** Identify the external dependency
  2. **INTERFACE** Introduce an interface (TAOUT: a seam) at the dependency
  3. **INJECT** Replace (inject) the dependency

1: IDENTIFY the dependencies

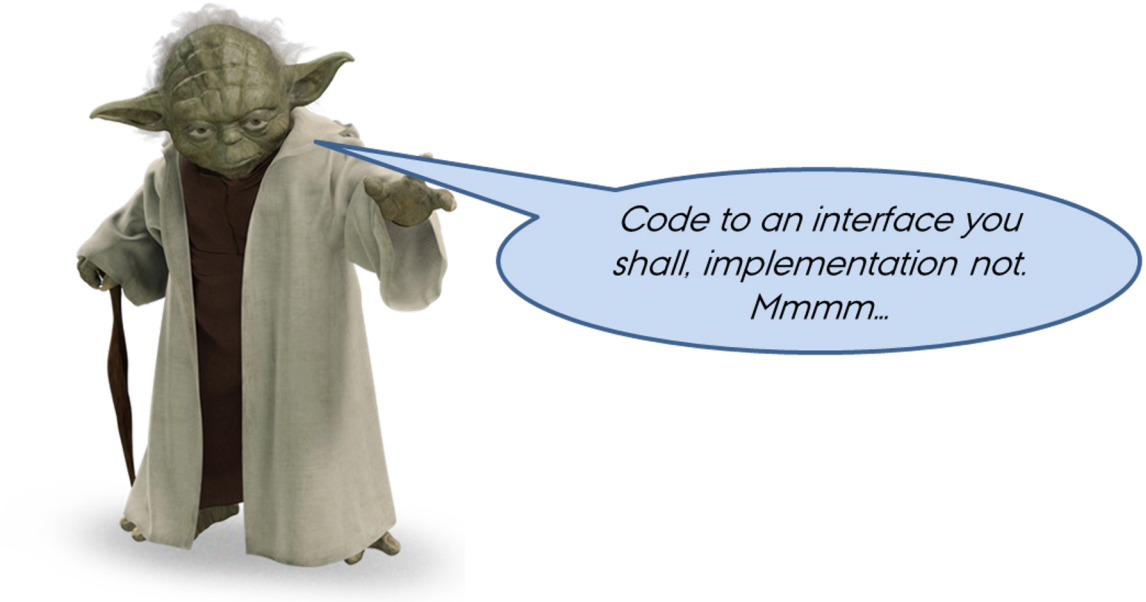
```
class House
{
   private readonly Bedroom _bedroom;
   private readonly Kitchen _kitchen;
   private readonly FrontDoor _door;

   public House()
   {
      _bedroom = new Bedroom();
      _kitchen = new Kitchen();
      _door = new FrontDoor();
   }

   public void Leave()
   {
      _kitchen
         .ShutDownAllAppliances();
      _bedroom.TurnLightOff();
      _door.Lock();
   }
}
```

Where are the dependencies?

What is the problem?What is the design flaw?



2: INTERFACE – Loosen the coupling



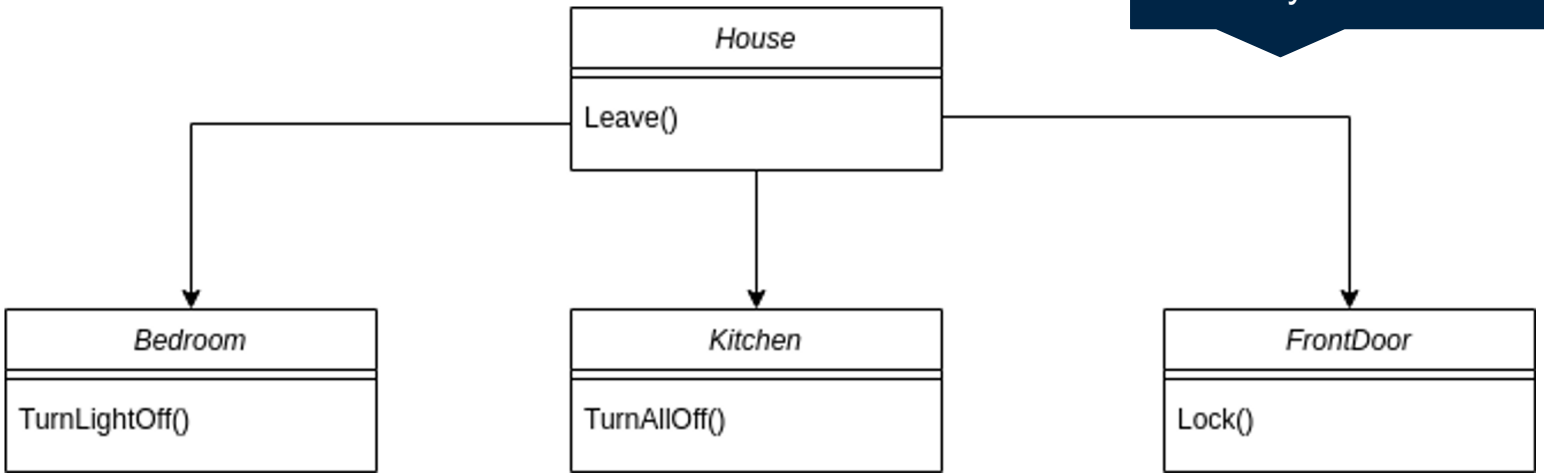
We can loosen the coupling by introducing interfaces to the dependencies

2: INTERFACE – Loosen the coupling

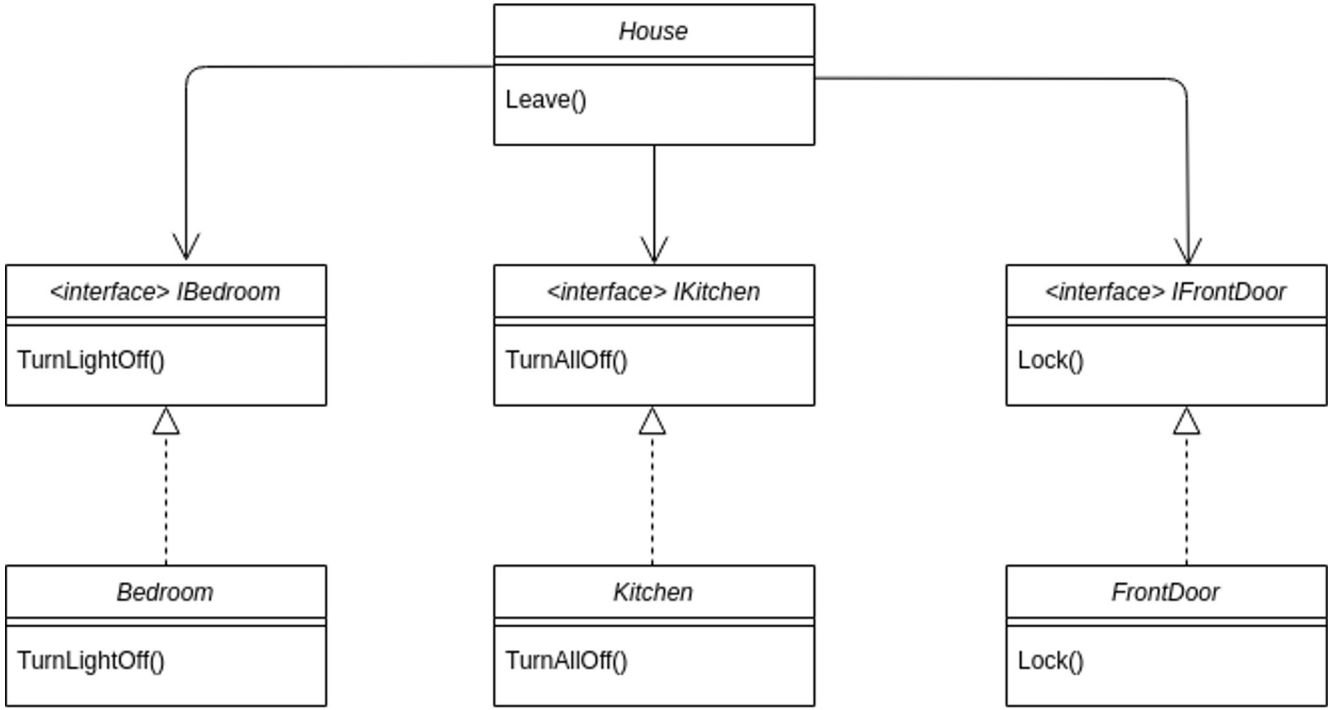
What does House really want to do with the Bedroom?

- “To call TurnLightOff() on its Bedroom object”, or
- “To turn the light off in the bedroom”

How can interfaces help us?
Where do the interfaces go?
How does this further testability?

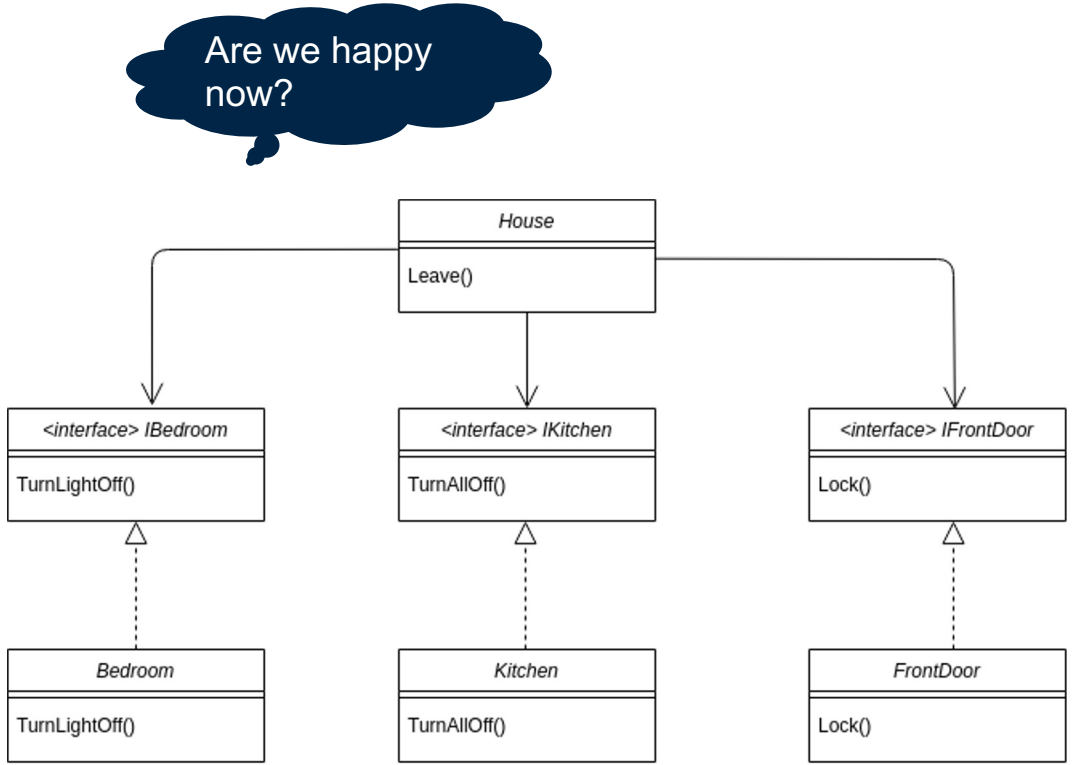


2: INTERFACE – Loosen the coupling



2: INTERFACE – Loosen the coupling

```
class House {
  private readonly IBedroom _bedroom;
  private readonly IKitchen _kitchen;
  private readonly IFrontDoor _door;

  public House() {
     _bedroom = new Bedroom();
     _kitchen = new Kitchen();
     _door = new FrontDoor();
  }

  public void Leavehouse() {
     _kitchen.ShutDownAllAppliances();
     _bedroom.TurnLightOff();
     _door.Lock();
  }
}
```



2: INTERFACE – How to test?

```
class House {
  private readonly IBedroom _bedroom;
  private readonly IKitchen _kitchen;
  private readonly IFrontDoor _door;

  public House() {
     _bedroom = new Bedroom();
     _kitchen = new Kitchen();
     _door = new FrontDoor();
  }

  public void Leavehouse() {
     _kitchen.ShutDownAllAppliances();
     _bedroom.TurnLightOff();
     _door.Lock();
  } }
```


```
class House {
  private readonly IBedroom _bedroom;
  private readonly IKitchen _kitchen;
  private readonly IFrontDoor _door;

  public House() {
     _bedroom = new FakeBedroom();
     _kitchen = new FakeKitchen();
     _door = new FakeFrontDoor();
  }

  public void Leavehouse() {
     _kitchen.ShutDownAllAppliances();
     _bedroom.TurnLightOff();
     _door.Lock();
  } }
```

3: INJECT dependency

- We would like to control the type of dependencies that House uses.
  - Fake or real

- We do this by injecting the dependencies into House.
  - Remember: Since House uses interfaces to its' dependencies, it does not care about their concrete types
  - So instead of letting House construct its own dependencies, we inject them into House
  - This means that we can isolate House from the rest of the system!

Constructor injection

```
class House {
  private readonly IBedroom _bedroom;
  private readonly IKitchen _kitchen;
  private readonly IFrontDoor _door;

  public House(IBedroom b, IKitchen k, IFrontdoor f) {
     _bedroom = b ;
     _kitchen = k;
     _door = f;
  }

  public void Leavehouse() {
     _kitchen.ShutDownAllAppliances();
     _bedroom.TurnLightOff();
     _door.Lock();
  } }
```


```
// Production
class Program {
  static void Main(string[] args) {
     // Use constructor injection to inject reals
     var house = new House(
        new Bedroom(),
        new Kitchen(),
        new FrontDoor());
  }}
```

```
 // Test
 [Test]
 public void House_Ctor_ConstructorInj() {
  // Use constructor injection to inject fakes
  var uut = new House(
     new FakeBedroom(),
     new FakeKitchen(),
     new FakeFrontDoor());
}
```

Property injection

```
class House {
  public IBedroom Bedroom {private get; set}
  public IKitchen Kitchen {private get; set}
  public IFrontDoor Door {private get; set}

  public House() {
    Bedroom = new Bedroom();
    Kitchen = new Kitchen();
    Door = new FrontDoor();
  }

  public void Leavehouse() {
    Kitchen.ShutDownAllAppliances();
    Bedroom.TurnLightOff();
    Door.Lock();
  }
}
```

Remains unchanged

```
// Production
class Program {
  static void Main(string[] args) {
    // Use default properties
    var house = new House();
  }
}
```

```
[Test]
public void House_Ctor_PropertyInj() {
  var uut = new House();
  // Use property injection to inject fakes
  uut.Bedroom = new FakeBedroom();
  uut.Kitchen = new FakeKitchen();
  uut.Door = new FakeFrontDoor();
}
```

Dependency injection – the silver bullet?

- At some point, we must consider the real world in our system

- Classics are “uncontrollable” dependencies
  - Time, file system, console, randomness, peripherals, …

- We do not omit dependency inclusion, we defer it!
  - During unit testing (and initial integration testing), we isolate the use of these dependencies as much as possible
  - When time comes, we can integrate external dependencies with a high degree of confidence

Recap - Design for testability

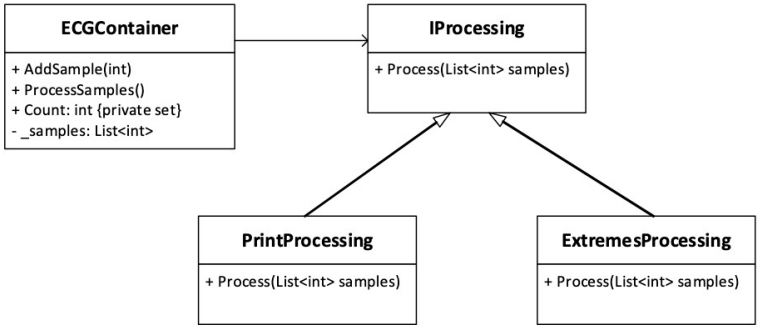
- A sensible design is required for testability
- Primary concern: Dependencies = loss of control
- Gain control by designing for loose coupling - Triple-**I**
  1. **Identify** (the interface of) the dependency
  2. Introduce an **interface** (seam) for the dependency
  3. **Inject** the dependency
- Control dependency type by dependency injection

Exercises

- Start on **ECG Exercise 1. - 3**.
- Continue on **Hospital Bed 1. – 2.**

ECG - Design

- Candidate design



- We want to unit test the class **ECGContainer**. Discuss:
  - What are the dependencies of **ECGContainer** (*Identify*)
  - How can we lower the coupling (*Interface*)
  - How do we get the new dependencies into Control (*Inject*)


FAKES
FAKES EVERYWHERE
makeameme.org

What is a fake?

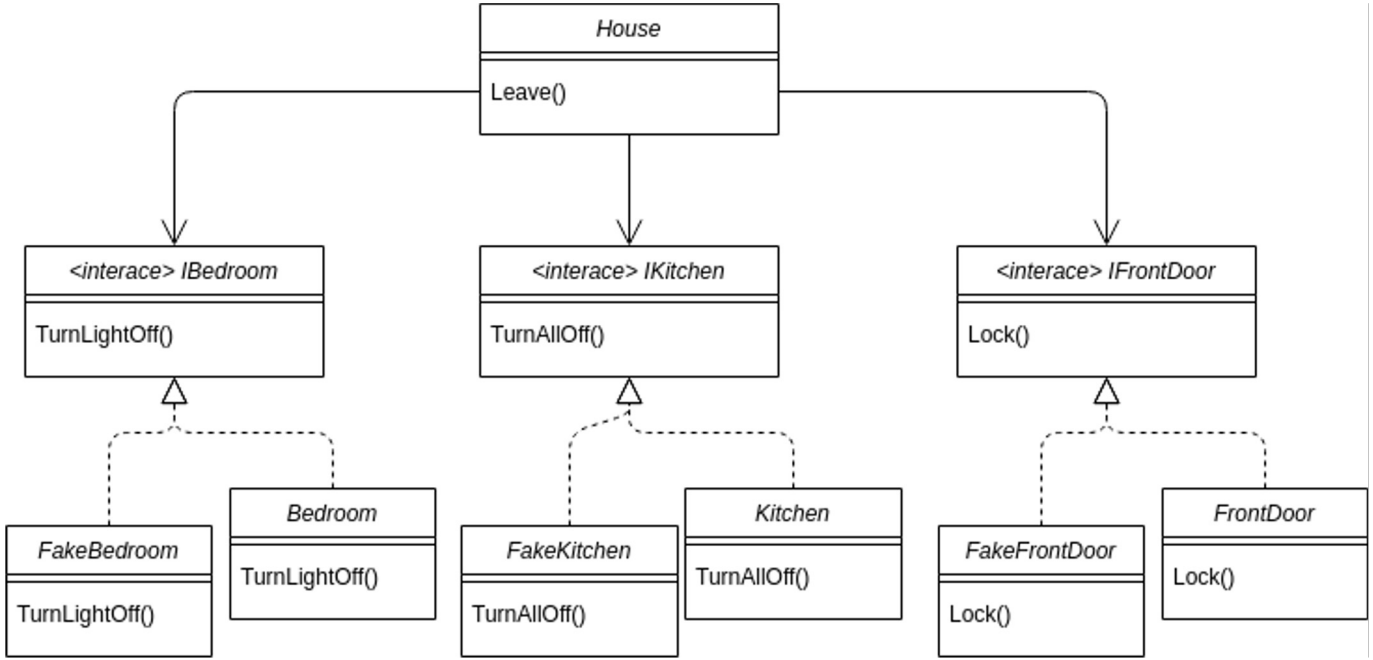
- **WHAT**: A fake is a “fake” version of a class

- **WHY**: We use fakes to substitute a UUT’s “real” dependencies. We write the fakes, so we control their behavior. This means that we can test the UUT in isolation.

- **HOW**: We enable the use of fakes by using Triple-I:
  1. *Identifying* the dependencies
  2. Inserting *interfaces* for them, and
  3. *Injecting* the dependencies into the UUT

What is a fake - example

House has references to interfaces, so it does not know (or care) which concrete implementations it uses



Test types

- Unit tests fall into one of two types:
- *State-based* tests are used to test if the UUT is in the expected *state* after we have acted upon it.
- *Interaction-based* tests are used to test if the UUT has had the expected *interaction* with its dependencies as a result of our acting upon it.
- Each test type uses its own type of fake

State-based tests: Fake is a called STUB

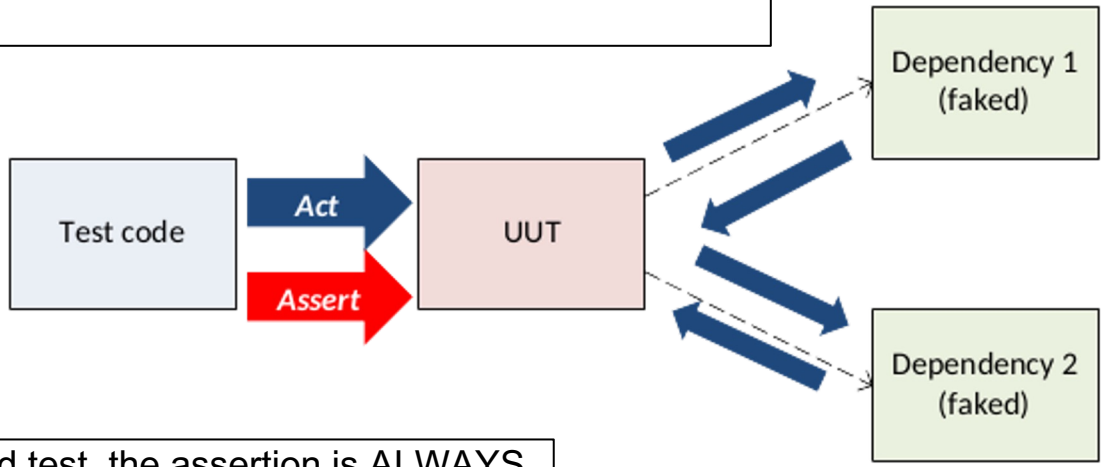
- For state-based testing, the fake is called a stub
  - Stubs exist to provide the UUT with the dependencies it needs to function (i.e. to return a value or just be present)

- The procedure for state-based tests is the usual:
  1. *Arrange* Setup your UUT (and its dependencies)
  2. *Act* Stimulate the UUT
  3. *Assert* That the UUT is in the expected state.

- A stub can never cause a test to fail, since the assertion is on UUT, never on the stub

State-based tests - Assert on UUT

1. *Arrange* Setup your UUT (and its dependencies)
2. *Act* Stimulate the UUT
3. *Assert* That the UUT is in the expected state.



In a state-based test, the assertion is ALWAYS on the UUT, NEVER on the fake(s)

Interaction-based tests: Fake is called a Mock

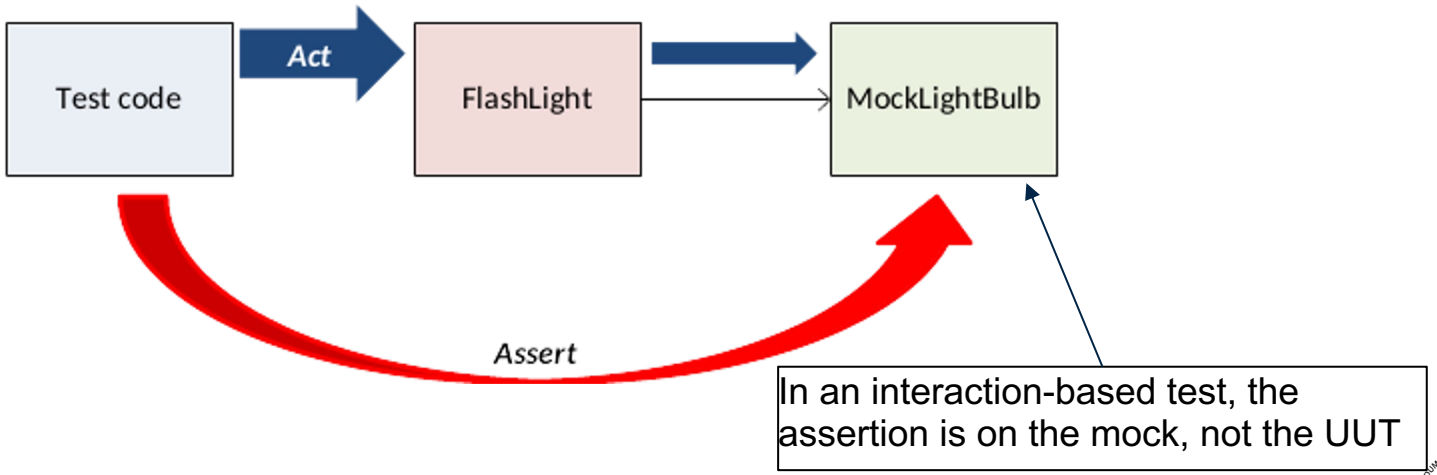
- For interaction-based testing, the fake is called a mock
  - We wish to test that the UUT had the expected interactions with the dependencies, so the mock must "*record*" if the interaction took place

- The procedure for interaction-based tests is the usual:
  a. *Arrange* Setup your UUT (and its dependencies)
  b. *Act* Stimulate the UUT
  c. *Assert* That the mock received the expected interactions with the UUT.

- The mock *can* cause a test to fail since the assertion is on the mock.

Interaction-based tests: Assert on Mock

- In interaction-based tests, we also follow the Arrange-Act-Assert strategy.
- Assertions are on the mock object – *not* the UUT

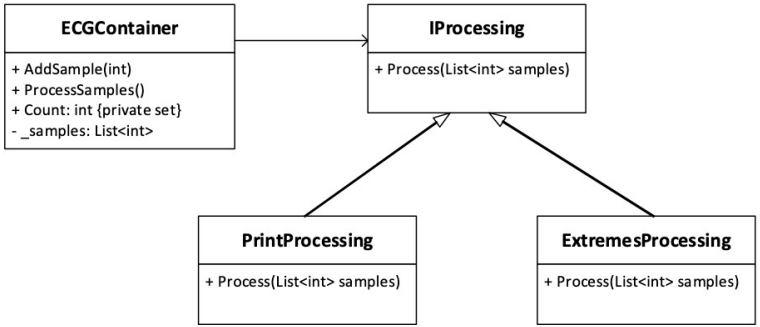


Exercises

- ~~Start on **ECG Exercise 1. - 3**.~~
- Continue on **ECG Exercise 4. - 6**.
- Continue on **Hospital Bed 1. – 3.**

ECG - Fakes

- Candidate design



- We want to unit test the class **ECGContainer**. Discuss:
  - ~~What are the dependencies of **ECGContainer**~~ (~~*Identify*~~)
  - ~~How can we lower the coupling~~ (~~*Interface*~~)
  - ~~How do we get the new dependencies into Control~~ (~~*Inject*~~)
  - What test cases do we need, to test `Count`(a property)?
  - What test cases do we need, to test `ProcessSample`()?
  - Are they state-based or interaction-based?

Caution: Avoid complex mocks

- To allow assertions to be made, the mock will to store….what?
  - The fact that a method was called
  - Number of times a method was called
  - Parameters
  - Call sequence

- This typically make mocks more complicated than stubs
  - Take longer to write, harder to re-use, more error-prone
  - Soon, you will have to test the mocks :(

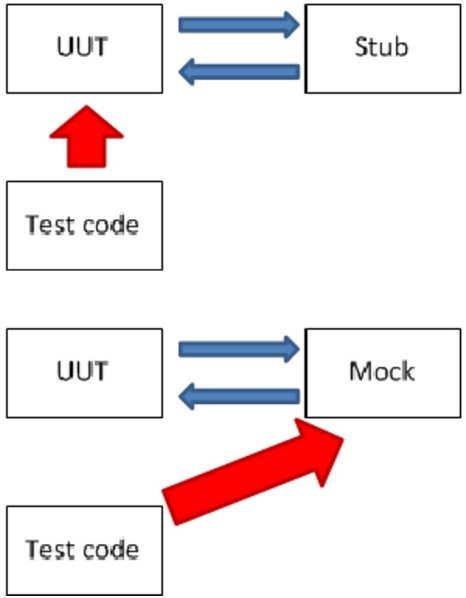
- Protip: Strive to keep your mocks simple

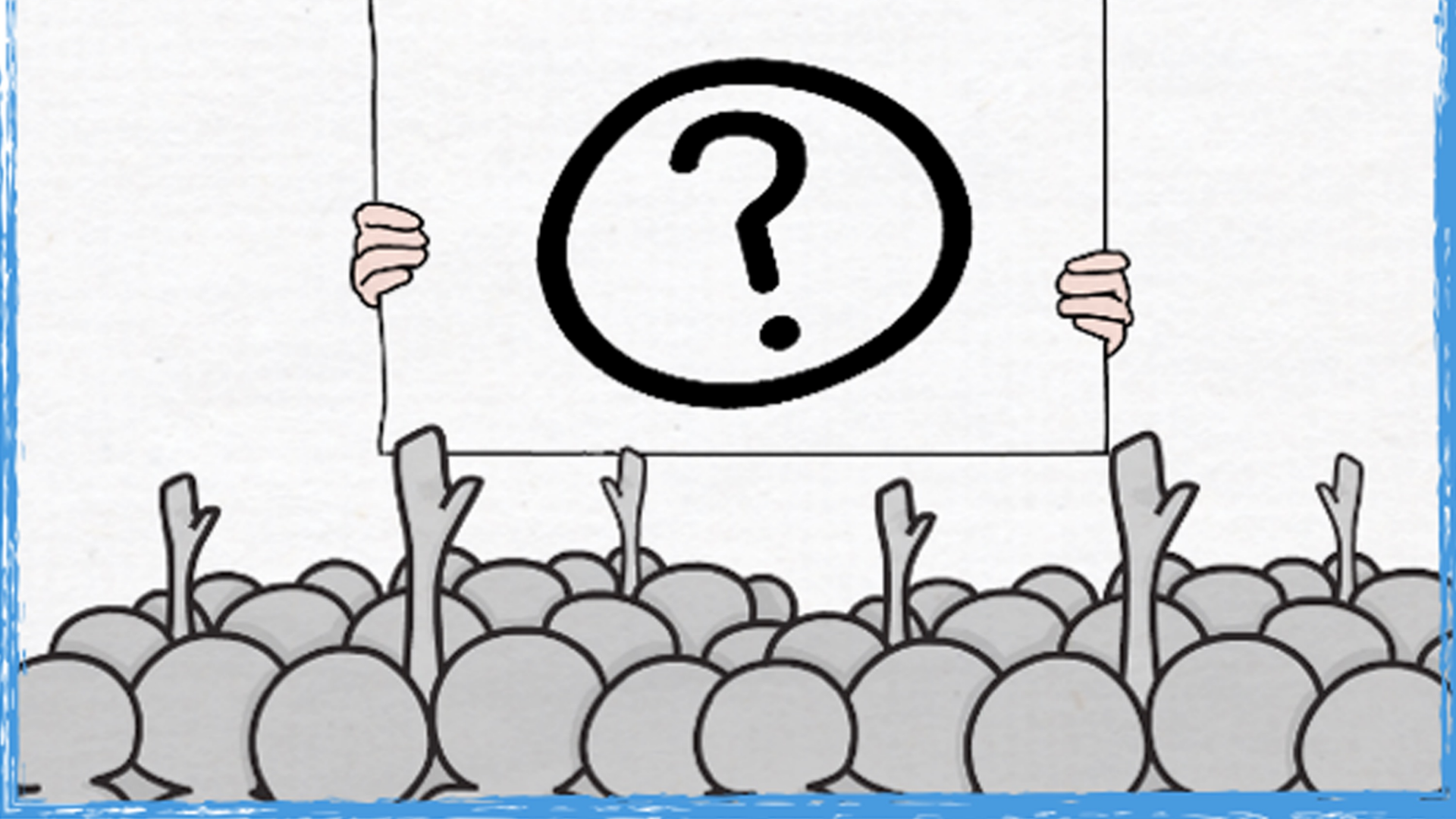
Recap - test types

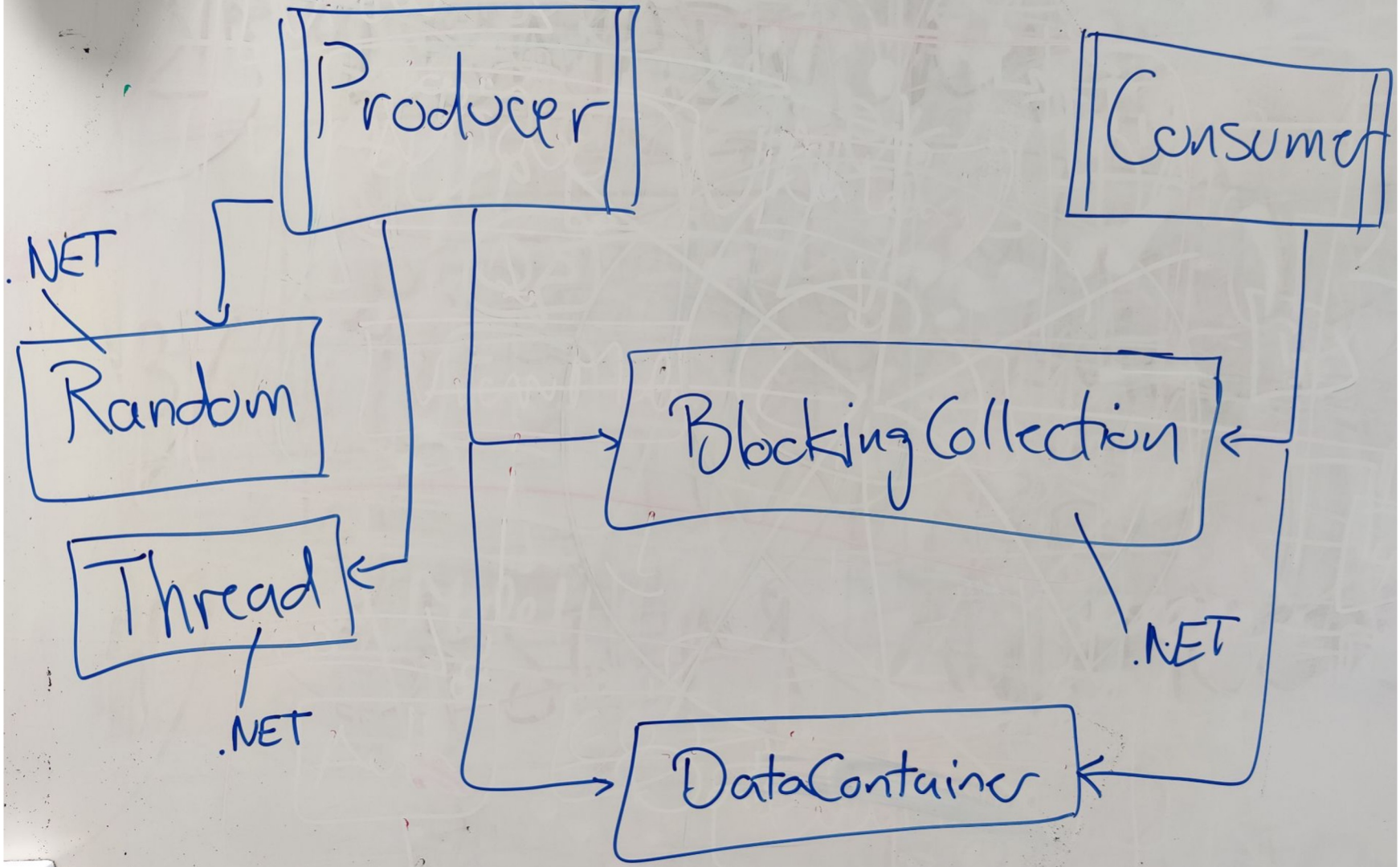
- The big differences between state- and interaction-based tests is *where* you make the assertion

- State-based test (stubs)
  - Test acts on UUT - UUT might interact with stub, but will change state
  - Test asserts that UUT assumed the expected state

- Interaction-based test (mocks)
  - Test acts on UUT - UUT interacts with mock and might change state
  - Test asserts on mock that UUT interacted as expected with the mock






Producer
Consumer
.NET
Random
Thread
.NET
BlockingCollection
.NET
DataContainer