cphbusiness

COPENHAGEN BUSINESS ACADEMY

Unit Testing

Mocking Dependencies

Sample Exam Questions - Covering Today’s Topics

- Explain a few rules of what makes a good Unit test, and rules for what makes code testable (or untestable).
- Explain about the SOLID rules, plus a number of the additional rules given in Effective Unit Testing, focusing on WHY these rules makes code more testable
- Explain strategies/frameworks used to unit test single classes with dependencies
- Explain about Test Doubles and specific types like: Dummy Objects, Test Stubs, Mock Objects and their purpose for Unit testing objects with dependencies.
- Explain the difference between state based testing versus behavior based testing, and provide practical examples using JUnit and a relevant Mock-framework.
- Explain why testers love the Dependency Injection Pattern
- Explain using examples you have designed, how to unit-test:
    - A method with no return value
    - A dependency method with required inputs and no return value

(Topics in grey are based on previous recap slides- topics in green will be covered by the exercises)

Today’s Learning Goals - more specifically

- Introduce mocking
- Apply inversion of control and dependency injection

- Write tests for REAL LIFE projects from github

Unit Testing
One (more) definition

Unit tests focus on single classes. They exist to make sure that **your code** works.

They control all aspects of the context in which the class to be tested is executed, by replacing real collaborators with **test doubles**.

They know nothing about the users of the system they put to the test, and are unaware of layers, external systems and resources.

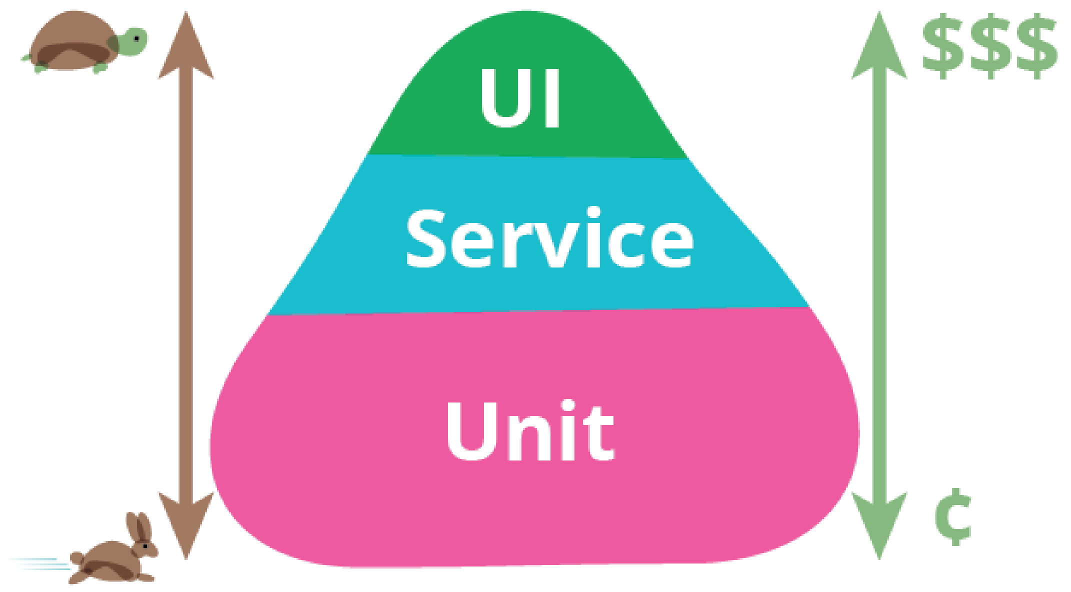
Ref: http://practicalunittesting.com/

Unit testing can’t stand alone ☺

Cliché Example of 2 unit tests, 0 integration tests

Ref.: https://blog.pragmatists.com/genuine-guide-to-testing-react-redux-applications-6f3265c11f63

The Test Pyramid

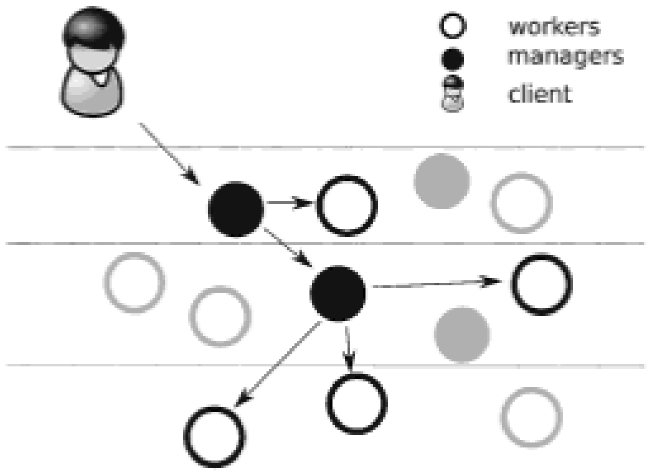


- Testing through the UI like this is slow, and tests are brittle
- API layer tests can provide many of the advantages of end-to-end tests but avoid many of the complexities of dealing with UI frameworks
- But: If my high level tests are fast, reliable, and cheap to modify - then lower-level tests aren't needed.

Ref: https://martinfowler.com/bliki/TestPyramid.html

Unit Testing - Challenges

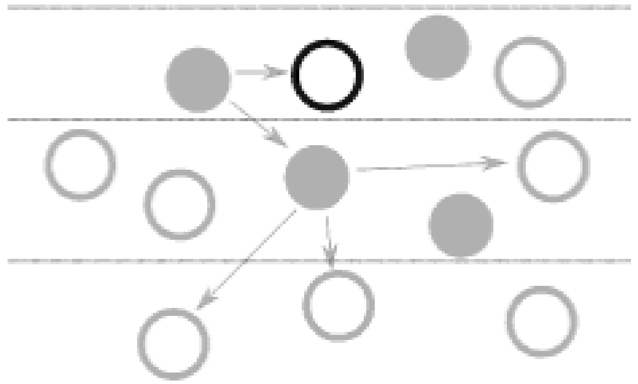
A typical OO System Abstraction



Typically in an OO-design a single request is solved by delegating subtasks to a number of objects. In the figure, circles represent objects, arrows are messages being passed between them and lines represent layers (View, Services, DAO etc.)

Ref: http://practicalunittesting.com/

Scope of a Unit Test



Unit test representation of the system, with only one element (the SUT) visible.
The greyed out elements symbolize those parts of the system not touched fully by the test or replaced by test doubles.
A unit test is always located inside a single layer.

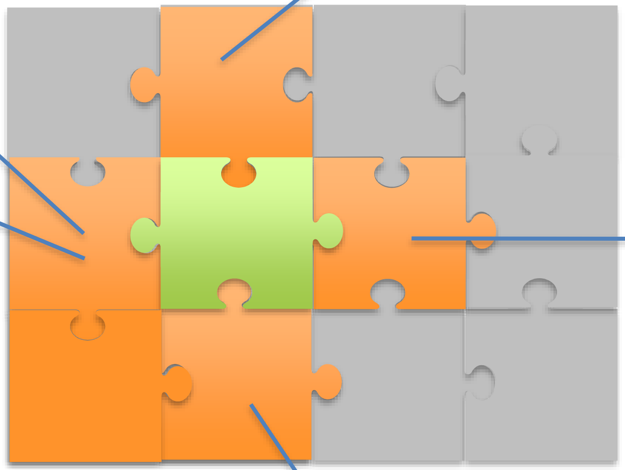
Unit Testing - Challenges

When testing a class
What if this class depends on 🟩
classes/systems that...

Supplies non-deterministic results (current time/date, temperature etc.)?

Is not yet created?

Provide behaviour not acceptable for a unit test (prints, send a mail, controls external hardware etc.)



Is complex and itself relies on external resources (web-service calls, file-I/O, external hardware etc.)?

Relies on a Database (takes a long time to start, could be on a remote server, must be kept clean etc.)?

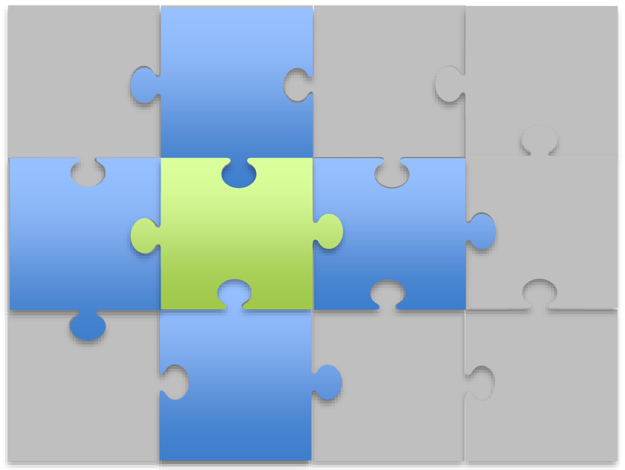
🟩	Class in focus
🟧	Dependendencies
⬜	Unrelated classes

Testing systems with dependencies

Clean code is code that does one thing well
Bjarne Stroustrup

Clean tests are focused tests that fail for only one reason
Petri Kainulainen

How to get Clean Code and Clean Tests →
Surround Objects Under Test with predictable **test doubles**



🟩 Class in focus (SUT)

🟦 Depending On Component (DOC)

⬜ Unrelated classes

Test Doubles

In general, test doubles are used to replace DOC's allowing us to:

- Gain full control over the environment in which the SUT is running
- Verify interactions between the SUT and its DOC's

Vocabulary Recap

SUT	System Under Test. The part of the system being tested. Depending on the type of test the granularity can range from a single class to the complete system
DOC	Depend On Component. Any entity that is required by the SUT to fulfill its duties
Test Double	Test Doubles are used to replace DOC's allowing us to: • Gain full control over the environment in which the SUT is running • Verify interactions between the SUT and its DOC's

Kinds of Test Doubles
Mocks, Fakes, Stubs and Dummies …

It is very easy to get confused about the terminology related to **Test Doubles**, since many articles use different terms to mean the same thing, and sometimes even mean different things for the same term ;-)

These are some of the terms you will meet "out there".

Dummy Objects, Mocks, Stubs, Fakes, Spy Objects etc..

For more details: http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html

Kinds of Test Doubles
Mocks, Fakes, Stubs and Dummies …

Fake objects *have working implementations, but usually take some shortcut which makes them not suitable for production (for example using an in-memory database)*

Stubs *provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.*

Spies *are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.*

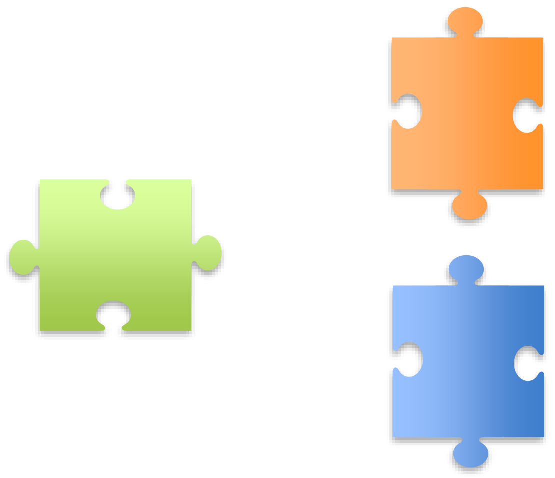
Mock Objects *are objects pre-programmed with expectations which form a specification of the calls they are expected to receive*

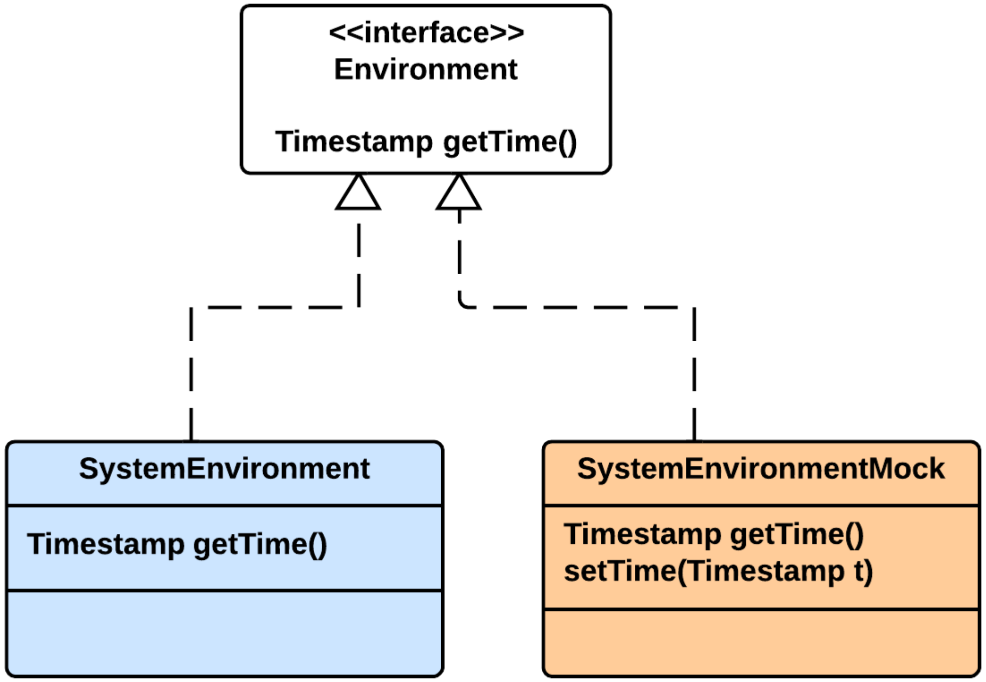
Dummy Objects *are* objects passed around but never used, i.e., its methods are never called (for example be used to fill the parameter list of a method)

DO NOT spend more than a few minutes on the definitions above, since different literature, tools etc. will provide their own definitions ;-(

For more details: http://xunitpatterns.com/

Unit Tests and Mock Objects





Returns the real system time

Returns a known time set via the setTime(..) method.

- 🟩 Class in focus
- 🟧 Mocks for the Unittest
- 🟦 Real dependency class

Mock example

```
public interface Environment {
   Timestamp getTime();
}
public class SystemEnvironment implements Environment {
  @Override public Timestamp
  getTime() {
    return new Timestamp(new Date().getTime());
  }
}
public class SystemEnvironmentMock implements Environment {
  private Timestamp time;
  @Override public Timestamp getTime() {
    return time;
  }
  public void setTime(Timestamp time) {
   this.time = time;
  }
}
```

Stubs versus Mocks - According to Fowler

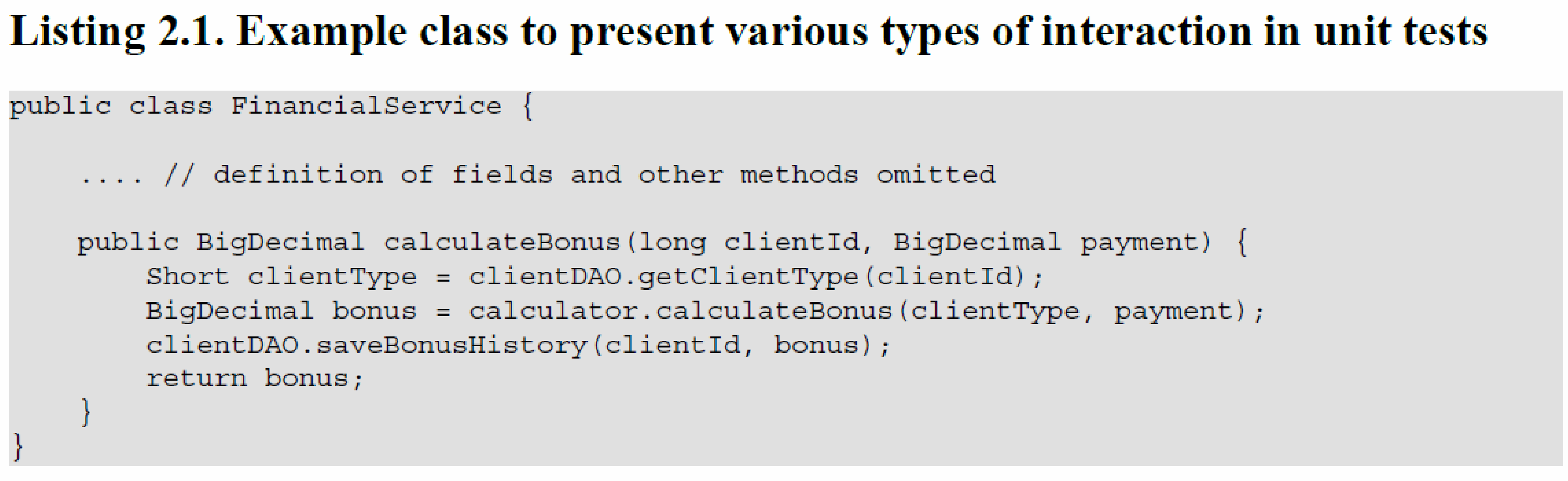
This difference is actually two separate differences

- A difference in how test results are verified: a distinction between state verification and behavior verification.
- A whole different philosophy to the way testing and design play together, which he terms as the **classical** and **mockist** styles of Test Driven Development

https://martinfowler.com/articles/mocksArentStubs.html

Don’t forget to also read/skim this short article:
https://www.thoughtworks.com/insights/blog/mockists-are-dead-long-live-classicists

Unit testing different types of interactions


Listing 2.1. Example class to present various types of interaction in unit tests

```
public class FinancialService {

    .... // definition of fields and other methods omitted

    public BigDecimal calculateBonus(long clientId, BigDecimal payment) {
        Short clientType = clientDAO.getClientType(clientId);
        BigDecimal bonus = calculator.calculateBonus(clientType, payment);
        clientDAO.saveBonusHistory(clientId, bonus);
        return bonus;
    }
}
```

State testing

Direct inputs & outputs

Behavior testing

Indirect inputs & outputs

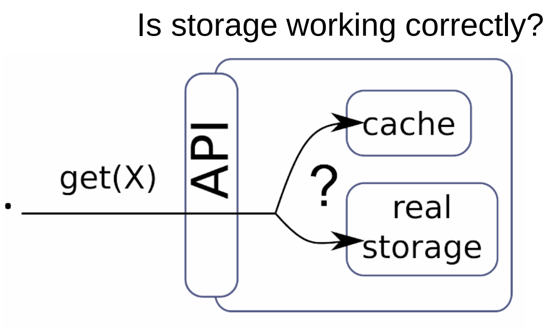
Why Test Indirect Interactions?

Doesn't it violate encapsulation and information hiding?

- In TDD, mocking is part of the design activity

  In OOD, each object has constrained functionality and interact with other objects to get the job done

- In some code cases, state testing won't do the job.



Example: Retrieving objects from a cache

When asked for an object with key X, our system with its cache should act according to the following simple rules:

1. if the object with key X is not in any storage location, the system will return null,

2. if the object with key X exists in any storage location, it will be returned,

a. if it exists in the cache storage, it will be returned from this storage location,

b. the main storage location will be searched only if the object with key X does not exist in the cache storage

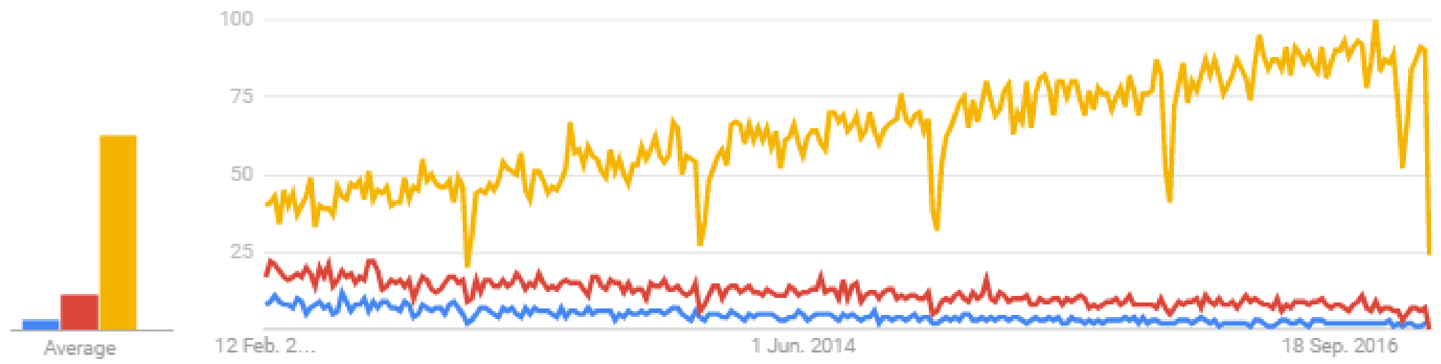
Mocking with Java

One way to make absolutely sure, we would never Unit Test/Mock was if we had to implement all mocking code manually
But (obviously) there are great tools out there to help us ☺

If you were a Java Based Company, trying to decide for a relevant framework, these are probably the ones that would pop up:

● JMock | ● EasyMock | ● Mockito

Let's see what others are using (searching for)



Live: https://www.google.com/trends/explore?q=JMock,EasyMock,Mockito

Mocking with mockito

We will go for mockito ☺

Massive StackOverflow Community voted Mockito the best mocking framework for java.

Top 10 Java library across all libraries, not only the testing tools.

Ref: http://mockito.org (front page)

Awesome recourses

The “awesome” series of listings of (good?) resources for a given category has made its way into testing.

If you want alternatives to the tools presented during this Test course, take a look here (especially if you are not that much into Java): https://github.com/atinfo/awesome-test-automation

Mocking – Using Mockito 1

Mockito has essentially two phases, one or both are executed as part of unit tests: stubbing and verification

Verification

Verification is the process of **verifying** interactions with our mocks. It lets us determine how our mocks were called, and how many times. It lets us look at the arguments of our mocks to make sure they are as expected.
It lets us ensure that exactly the values we expect are passed to our collaborators, and that nothing unexpected happens. Verification lets us determine exactly what happened to the mock.

Example

Mocking – Using Mockito 2

Mockito has essentially two phases, one or both are executed as part of unit tests: stubbing and verification

Stubbing

Stubbing is the process of **specifying** the behavior of our mocks. It is how we tell Mockito what we want to happen when we interact with our mocks. Stubbing makes it simple to create all the possible conditions for our tests. It let's us control the responses of our mocks, including forcing them to return any value we want, or throw any exception we want. It allows us to code different behaviors under different conditions. Stubbing lets us control exactly what the mock will do.

Example

Mocking – Using Mockito 3

Of all the many mock-terms introduced earlier, Mockito only uses **Mock** and **Spy**

Mock

With Mock we are creating a **complete** mock or fake object mock, where the default behavior (which you can change) of the methods is **do nothing**

With a Mock instance we can test both STATE and BEHAVIOR

Spy

With Spy we are using a real object and we are only spying/mocking specific methods of it

> *Spies should be used carefully and occasionally, usually when dealing with legacy code. Best practice is not to use Spy to partially mock the class under test, but instead to partially mock dependencies.*
> ***The class under test should always be a real object***

From Mockito.org:

Don't mock everything

Mocking – Using Mockito
Quick getting started

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoTest
{
    @Mock
    ArrayList<String> listMock;

    @Test
    public void testAppend() {
        //Alternative way of creating the Mock
        // ArrayList<String> listMock = mock(ArrayList.class);
        when(listMock.get(0)).thenReturn("Hello");
        when(listMock.get(1)).thenReturn("World");

        String res = listMock.get(0);
        assertEquals("Hello", res);
        res = listMock.get(1);
        assertEquals("World", res);

        verify(listMock,times(2)).get(anyInt());
    }
}
```

This ensures that mocked values will be populated

Mock the ArrayList instance

Configure which values are returned by the mock

Verify that `get(..)` was called 2 times

Mocking – Using Mockito

We will do a Quick Demo together based on this project:
https://github.com/Lars-m/sillygame.git

- After that you should go through this tutorial:
  https://www.javacodegeeks.com/2015/11/testing-with-mockito.html
- And complete the study point exercises (Midterm Assignment)