L1 - TDD

Builds on A0 – *work with a partner if you can*

- Check due dates and partnering strategy on Vocareum!
- Try to finish by Thursday to get ready for Mockito exercises

Having trouble with A0 or L1?

- Check under Modules > Assignments > A0-L1 Guidance

This Thursday

- Mockito Prep Exercises
  - warmup exercises for A1
  - graded
  - prep/setup required on your IDE
  - finishing L1 will help

What's wrong with these test method names?

- SendAndAcceptFriendRequest
- twoPeopleCanJoinSocialNetworkAndSizeOfNetworkEqualsTwo
- friendshipTest2
- socialNetworkIsCreated
- noFriendRequests
- testMultipleFriendRequests
- afterAcceptingFriendRequestIncomingRequestsUpdated

What's wrong with these test method names?

- ~~SendAndAcceptFriendRequest~~ (Java method naming convention violation)
  - canSendAndAcceptFriendRequest
- ~~twoPeopleCanJoinSocialNetworkAndSizeOfNetworkEqualsTwo~~ (too specific)
  - sizeOfSNMatchesNoOfMultiplePeopleJoining
- ~~friendshipTest2~~ (way too ambiguous, and redundant too)
  - dontNeedToRepeatTheWordTestInATestMethod
- ~~socialNetworkIsCreated~~ (too ambiguous)
  - possibleToCreateSocialNetworkWithSingleMember
- ~~noFriendRequests~~ (too ambiguous)
  - noFriendRequestsMakesThisImpossible (replace "This" with the impossible condition)
- ~~testMultipleFriendRequests~~
  - multipleFriendRequestsCausesThis (replace "This" with desired effect)
- afterAcceptingFriendRequestIncomingRequestsUpdated (OK)

In general:

- doingThisCausesThisWhenThisHolds
- canDoThis
- canDoThisWithThisEffect
- canDoThisWithThisEffectWhenThisHolds
- thisShouldBePossible
- thisShouldNotBePossible

Replace "this", "do", "effect" to capture the test's purpose

Test Doubles: In-depth

Agenda

- What and why of test doubles
- How test doubles work
- Types of test doubles
  - Stubs (Dummies)
  - Fakes
  - Spies
  - Mocks
- Test double guidelines
- Mockito overview (skip to Exercise)
- Exercise
- More Mockito
- Assignment

We need test doubles to…

- Isolate the code under test
- Focus our tests on objects of interest
- Speed up test execution
- Make execution deterministic
- Simulate special conditions
- Gain access to hidden information

A test double is an *approximated* collaborator (dependency)!
(supports the Approximation principle)

- We have a unit (EUT) that we're really interested in testing
- It depends on many components (its *collaborators*) that need to be *approximated*

…because they are

- *unavailable*
- *expensive*
  - *slow*
  - *resource-intensive*
- *opaque*
- *non-deterministic*

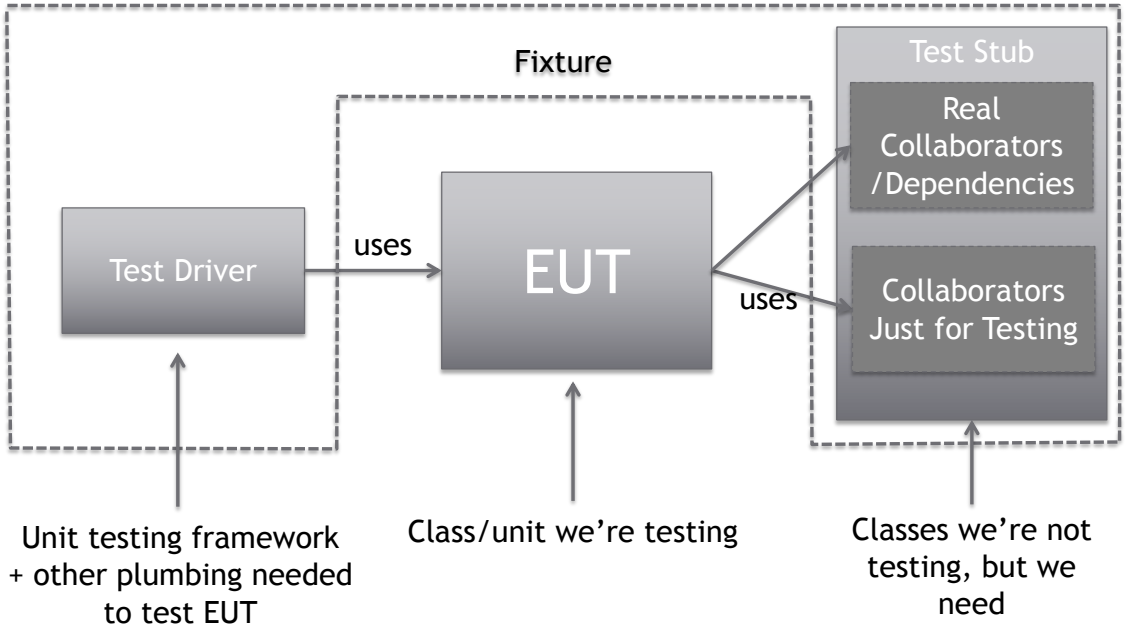
Double?

Film industry: actors have *stunt doubles*

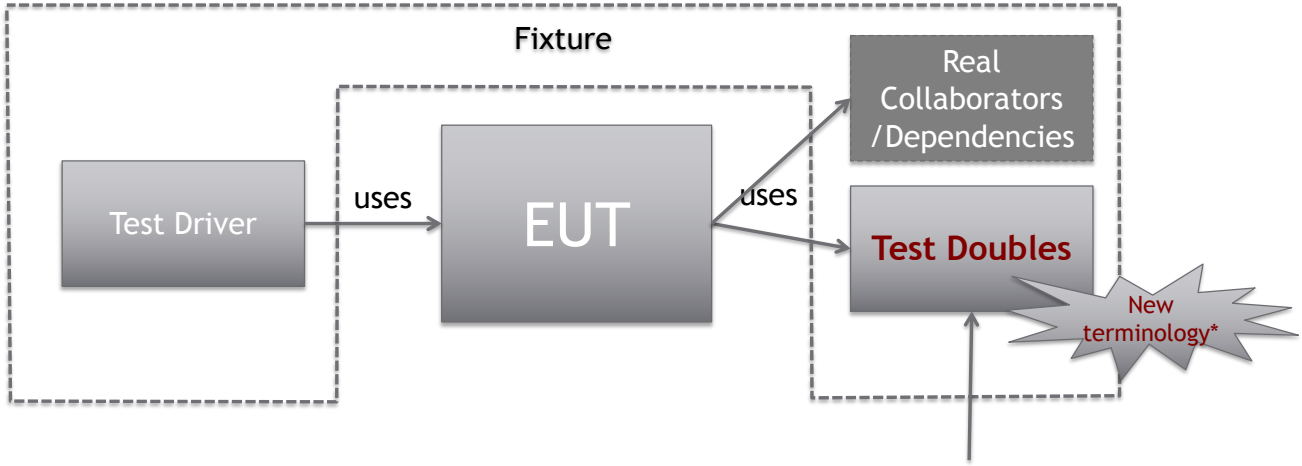
Testing: collaborators have *test doubles*

… when we are not testing the collaborator, but something that needs it!

In general-testing terms…



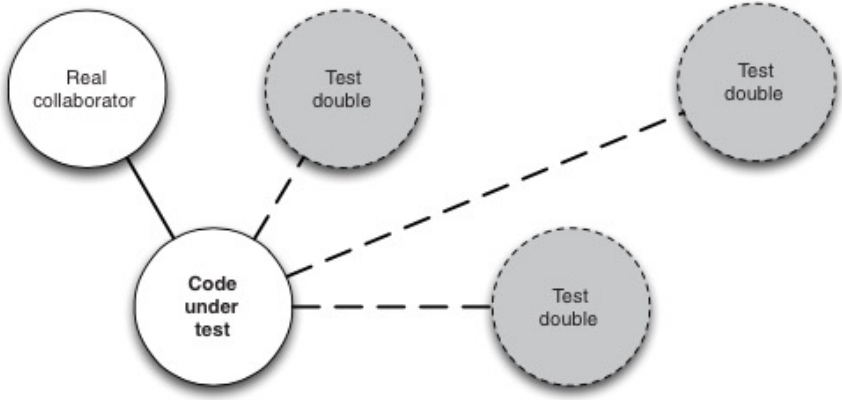
In general testing terms…



*Popularized by J.B. Rainsberger (JUnit Recipes) and G. Meszaros (xUnit Patterns)

Real collaborators can be mixed with doubles

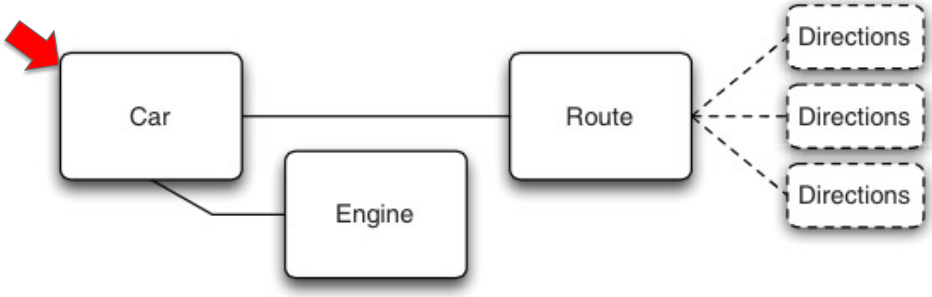
Not all collaborators need to become test doubles…



A lot of dependencies can be the real objects

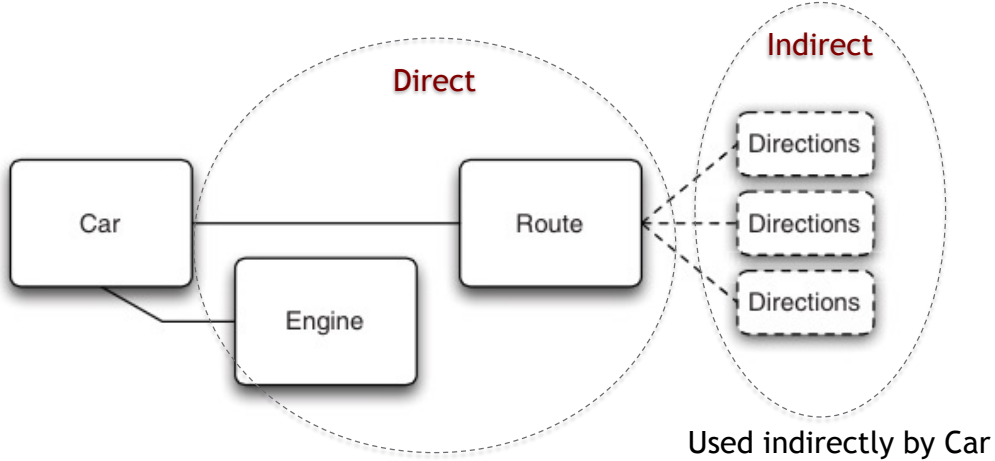
Indirect and direct collaborators

Which classes are good double candidates for testing Car?



Koskela 2013

Direct collaborators are best candidates for being doubles

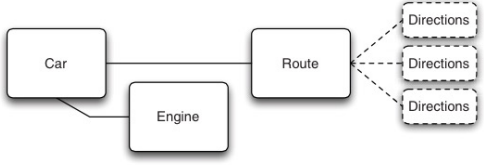


How Test Doubles Work

A double can speed up tests

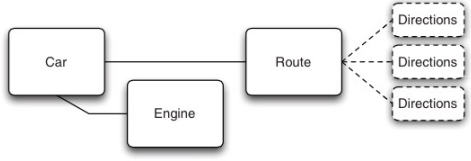
- Route uses a complex search algorithm to find shortest path between two GPS locations
- Algorithm is slow
- We're not interested in testing whether this algorithm is correct; we just need some directions to give to a Car object (we are testing the Car class)
- We can have a Route double return canned directions for testing Car

→ *Would speed up test execution*



A double can remove non-determinism

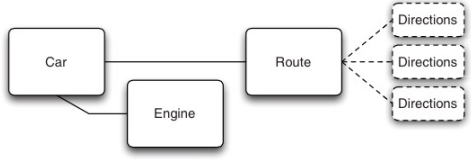
- Recall Route relies on real-time information to return directions (time of day, traffic conditions, road closures, new roads)
- This makes it non-deterministic for tester (dependent on specific environmental conditions that change)
- We need Route's behavior not to surprise Car tests
- We can have a Route double deterministically return the same directions under the same conditions that we can control



A double can simulate special conditions

- Route gets the directions from the internet, using a web service (say Google Maps)
- How should Car behave when internet connection is out?
- We want Car tests to reveal what should happen under such exceptional conditions
- We can have the Route double simulate lack of internet connection

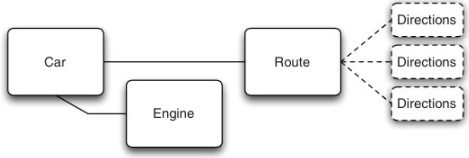
→ *Would help us simulate special conditions*



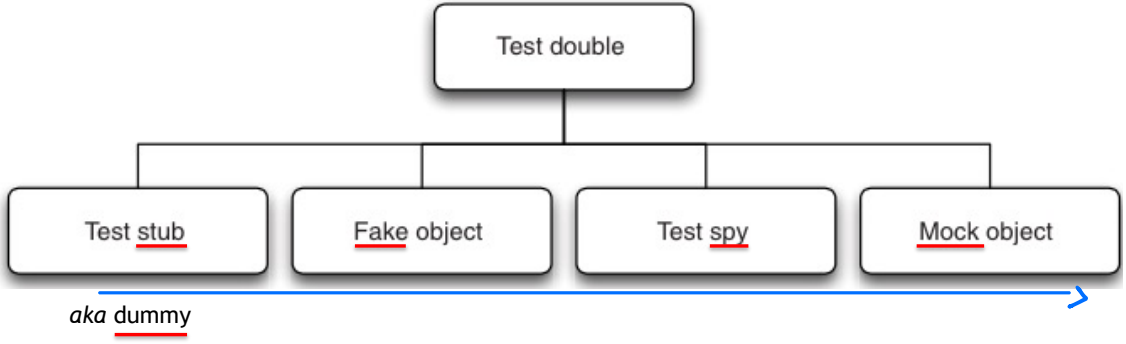
A double can make a collaborator less secretive

- Engine starts the Car's engine
- We want to test that engine is started when we ask the Car to start
- Engine's internal state is not accessible to tests
- An Engine double could reveal Engine's simulated internal state (started/idle) to the tests

→ *Would help expose hidden information*



So Many Kinds of Test Doubles, So Little Time



Differences among test double types are not *always clear:* fakes may act as stubs or spies; mocks often have spy-like behavior; a double may combine both stub and mock behavior; …

Koskela 2013

Stub (dummy) is just a… dummy

stub: (*noun*) a truncated or unusually short thing

A crude, static stand-in for the real collaborator for testing purposes

Simplest example: a code template, an object with all one-liner methods each of which returns a default/canned value (or does nothing)

- a *stubbed-out* class, as in:
  - "Let's stub this code out!" (when we are debugging)
  - "Let's create the stubs for this test" (during TDD)

Stub (dummy) example

Collaborator: a remote log server accessible through the Logger interface…

- The test isn't interested in what the code under test is logging
- We don't have a log server running, so it would fail anyway
- We don't want our test suite to generate lots of output that tests have no intention of checking

```
public class LoggerStub implements Logger {
   public void log(LogLevel level, String message) {
      // still a no-op
   }

   public LogLevel getLogLevel() {
      return LogLevel.WARN;  // hard-coded return value
   }
}
```

Easy, just inherit from real class, or implement the same interface!

We are not testing Logger, but a client of Logger

A Fake can almost be real

Sometimes stubs are not enough, and we need more realistic behavior, e.g.,

> "an optimized, thinned-down version of the real thing that replicates the behavior of the real thing, but without the persistent or expensive side effects and other [undesirable] consequences of using the real thing."

That's where a **fake** comes in handy…

A full fake can be substituted for the real object in every context during testing; a restricted fake only in some contexts!

Fake example

A fake that emulates a persistence object…

`UserRepository` also implements `IUSerRepository`

```
public class FakeUserRepository implements IUserRepository {
   // an in-memory fake user repo
   private Collection<User> users = new ArrayList<User>();

   public void save(User user) {
      if (findById(user.getId()) == null) {
                  users.add(user);
       }
   }

   public User findById(long id) {
      for (User user : users) {
         if (user.getId() == id) return user;
      }
      return null;
   }
}
```

During testing, behaves like a real UserRepository that accesses a real DB, but much faster and doesn't corrupt the production DB!

Fake example

A fake that emulates a persistence object…

```
public class FakeUserRepository implements IUserRepository {
   // an in-memory fake user repo
   private Collection<User> users = new ArrayList<User>();

   public void save(User user) {
      if (findById(user.getId()) == null) {
                  users.add(user);
       }
   }

   public User findById(long id) {
      for (User user : users) {
         if (user.getId() == id) return user;
      }
      return null;
   }
}
```

Which class are we testing here?

We are testing another class that uses UserRepository

Fake example

A fake that emulates a persistence object…

```
public class FakeUserRepository implements IUserRepository {
   // an in-memory fake user repo
   private Collection<User> users = new ArrayList<User>();

   public void save(User user) {
      if (findById(user.getId()) == null) {
                users.add(user);
       }
   }

   public User findById(long id) {
      for (User user : users) {
         if (user.getId() == id) return user;
      }
      return null;
   }
}
```

Not UserRepository!

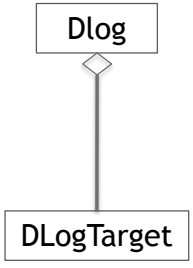
Another fake example (restricted or partial)

```
public class FakeRoute extends Route {
    private HashMap<RouteRequest, Route>
        myTown = new HashMap<RouteRequest, Route>;

    public FakeRoute(HashMap<RouteRequest, Route> theTown) {
      // initialize myTown using a fixed mapping of Source-Destination addresses to a fixed route
      this.myTown = theTown;
    }

    @Override // this is an expensive method
    public Directions findRoute(Address A, Address B) {
      // only works with known address pairs in myTown that tests can use
      return this.myTown.get(new RouteRequest(A, B));
    }

     // other behavior remains the same
}
```

A Spy deviously reveals secrets

Use a **spy** when the state of a collaborator is a secret, and you need to access that state to test an object

Spy example



Want to test whether DLog writes all messages to all targets

```
public class DLog {
   private final DLogTarget[] targets;

   public DLog(DLogTarget... targets) {
      this.targets = targets;
   }

   public void write(Level level, String message) {
      for (DLogTarget each : targets) {
         each.write(level, message);
      }
   }
}

public interface DLogTarget {
   void write(Level level, String message);
}
```

Collaborator DLogTarget has secret state

1. DLog is given a number of DLogTargets
2. Each target receives the same message
3. DLogTarget only defines the write() method

DLogTarget best renamed to IDLogTarget

Spies sneak in and report back

Note that we are not testing DLogTarget, but DLog!

```
public class DLogTest {
   @Test
   public void writesEachMessageToAllTargets() throws Exception {
      SpyTarget spy1 = new SpyTarget();
      SpyTarget spy2 = new SpyTarget();
      DLog log = new DLog(spy1, spy2);
      log.write(Level.INFO, "message");
      assertTrue(spy1.received(Level.INFO, "message"));
      assertTrue(spy2.received(Level.INFO, "message"));
   }

   private class SpyTarget implements DLogTarget {
      private List<String> log = new ArrayList<String>();

      public void write(Level level, String message) {
         log.add(concatenated(level, message));
      }

      boolean received(Level level, String message) {
         return log.contains(concatenated(level, message));
      }

      private String concatenated(Level level, String message) {
         return level.getName() + ": " + message;
      }
   }
}
```

1 Spies sneak in

2 Make write() do some bookkeeping

3 Spies report back

Let test ask for what it wants to know

DLogTarget best renamed to IDLogTarget

A Mock is complicated, but super flexible

an object configured at runtime to behave in a certain way under certain circumstances

A Mock is complicated, but super flexible

- can be precise by failing a test in a timely fashion when an unexpected event happens
- may behave like a spy or a (restricted) fake, or even a dummy depending on the test
- can verify object interactions, not just results
- is difficult to implement without a mocking framework
  - *because we should be able to specify (configure) behavior at run-time*
  - *because we need to track method calls inside the mock and introduce an API to check them*
- is almost always used with a mocking framework
  - Java mocking frameworks: EasyMock, JMock, Mockito, …

Mock example (JMock) -- complicated?

```
public class TestTranslator {
   protected Mockery context;

   @Before
   public void createMockery() throws Exception {
      context = new JUnit4Mockery();
   }

   @Test
   public void usesInternetForTranslation() throws Exception {
      final Internet internet = context.mock(Internet.class);
      context.checking(new Expectations() {{
         one(internet).get(with(containsString("langpair=en%7Cfi")));
         will(returnValue("{\"translatedText\":\"kukka\"}"));
      }});
      Translator t = new Translator(internet);
      String translation = t.translate("flower", ENGLISH, FINNISH);
      assertEquals("kukka", translation);
   }

   ...
}
```

1. a JMock interface

2. Create mock Internet object

3. Configure mock: what should it expect?

3.1. Internet#get should be called once with specified parameter

3.2. Should return specified value when called as above

4. Inject mock into Translator object and test

Mock example (Mockito) -- better?

Simpler, but less expressive

```
@Test
public void usesInternetForTranslation() throws Exception {
    final Internet internet = mock(Internet.class);
    when(internet.get(argThat(containsString("langpair=en%7Cfi"))))
                .thenReturn("{\"translatedText\":\"kukka\"}");
    Translator translator = new Translator(internet);
    String result = translator.translate("flower", ENGLISH, FINNISH);
    assertEquals("kukka", result);
}
```

인터넷

1. Create mock Internet object

2. Configure mock: when Internet#get is called with specified parameter, it will return specified value

3. Inject mock and perform test

Which class are we testing here? (What's EUT?)

Translator

Error in Koskela section 3.3.4: a Mockito *verify* statement is not necessary in this case!

Mocks are controversial

(Arguably valid) objections to mocks

- they are too much work
- they are difficult to maintain
- they can be brittle
- they can be cumbersome
- they can be difficult to understand
- they are often unnecessary (we could often just use fakes/spies created from scratch)

They can be very useful when used appropriately!

Test Double Guidelines

Pick the right double for the test

- Stub (dummy) if you just want collaborators to be there and feed canned responses to tests

Caution: "stub" in testing terminology is any dependency/collaborator, whereas in test double terminology, it is the simplest double you can imagine

Pick the right double for the test

- Stub (dummy) if you just want collaborators to be there and feed canned responses to tests
- Fake if your test needs realistic behavior from a collaborator that's unavailable or infeasible for your test's purpose, and simple stubbing does not do the job

Pick the right double for the test

- Stub (dummy) if you just want collaborators to be there and feed canned responses to tests
- Fake if your test needs realistic behavior from a collaborator that's unavailable or infeasible for your test's purpose, and stubbing does not do the job
- Spy (using a hand-made spy) if you need to add extra behavior to track and reveal internal state

Pick the right double for the test

- Stub (dummy) if you just want collaborators to be there and feed canned responses to tests
- Fake if your test needs realistic behavior from a collaborator that's unavailable or infeasible for your test's purpose, and stubbing does not do the job
- Spy (using a hand-made spy) if you need to add extra behavior to reveal internal state
- Mock if you care about certain interactions between two objects, you want to configure/change behavior at run-time, you need complex spy-like behavior in addition to fake-like behavior, or you want to mix the above behaviors in a single test double

Stub queries, mock actions*

- Stub queries
  - your test needs to ask a collaborator a question, but you don't care much about the actual answer, just that it returns a valid, well-formed object the test can use
  - your tests need to ask a small set of questions whose answers they know
- Mock actions
  - your tests need to know which methods are called on collaborator objects (interactions)
  - your tests need to know how many times certain methods are called, how they are called, and under what circumstances

*J.B. Rainsberger, JUnit Recipes

We still apply: Arrange-Act-Assert

Just apply the time-honored testing pattern…

```
@Test
public void usesInternetForTranslation() throws Exception {
   final Internet internet = context.mock(Internet.class);
   context.checking(new Expectations() {{
      one(internet).get(with(containsString("langpair=en%7Cfi")));
      will(returnValue("{\"translatedText\":\"kukka\"}"));
   }});
   Translator t = new Translator(internet);

   String translation = t.translate("flower", ENGLISH, FINNISH);

   assertEquals("kukka", translation);
}
```

Also defines double's behavior

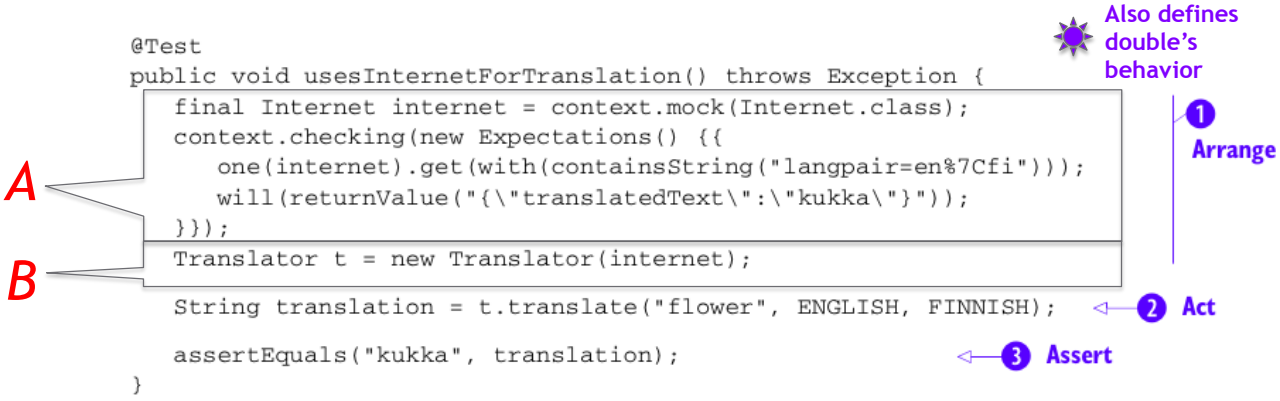
1 Arrange

2 Act

3 Assert

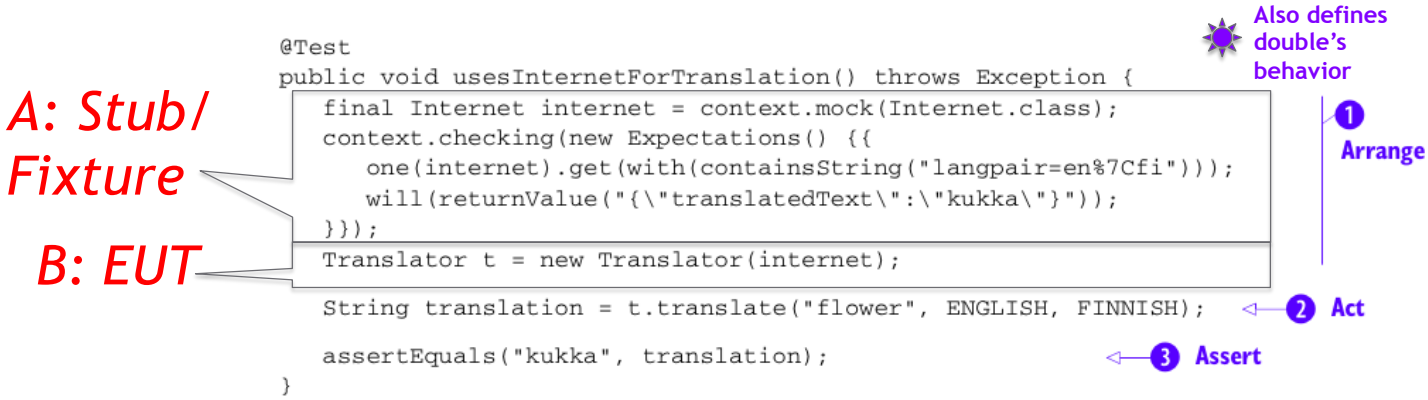
JMock example

Arrange-Act-Assert



What are sections A and B called in testing terminology?

Arrange-Act-Assert



Use weakest double that will do the job

(−*intrusive*) Stub (Dummy) < Fake < Spy < Mock (+*intrusive*)

Restricted/Partial Fake < Full Fake

In production code: Inject your dependencies

Dependency: collaborator that you'd like to replace with a double for testing

*Collaborators should **not** be instantiated where they are used*

- acquire them via [context-aware] factory methods
- **pass them into object under test using**
  - constructor injection
  - property (setter) injection

… so that they can be substituted by doubles

Dependency injection

..is giving an object its instance variables (James Shore)

```
public class Example {
  private DatabaseThingie myDatabase;

  public Example() {
    myDatabase = new DatabaseThingie();
  }                              defined

  public void doStuff() {
    ...
    myDatabase.getData();
    ...    used
  }
}
```

Dependency: class Example depends on an instance of DatabaseThingy

Unfortunately, it is defined (instantiated) in the same class it's used

Can't double myDatabase when testing Example class!

How can you fix this?

http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html

Dependency injection

```
interface IDatabaseThingie;

public class Example {
  private IDatabaseThingie myDatabase;

  public Example() {
    myDatabase = new RealDatabaseThingie();
  }

  public Example(IDatabaseThingie useThisDatabaseInstead) {
    myDatabase = useThisDatabaseInstead;
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Default database

Constructor injection

Now my test can ***inject*** a `DatabaseThingie` ***double*** into `Example`:

```
IDataBaseThingie fakeDB = new FakeDatabaseThingie();
Example example = new Example(fakeDB)
```

Dependency injection


```
interface IDatabaseThingie;

public class Example {
  private IDatabaseThingie myDatabase;

  public Example() {
    myDatabase = new RealDatabaseThingie();
  }

  public Example(IDatabaseThingie useThisDatabaseInstead) {
    myDatabase = useThisDatabaseInstead;
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Which class are we testing here?
(Which one is the EUT?)

A. IDatabaseThingy
B. RealDatabaseThingy
C. FakeDatabaseThingy
D. Example
E. Something else

Dependency injection

```
interface IDatabaseThingie;

public class Example {
  private IDatabaseThingie myDatabase;

  public Example() {
    myDatabase = new RealDatabaseThingie();
  }

  public Example(IDatabaseThingie useThisDatabaseInstead) {
    myDatabase = useThisDatabaseInstead;
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Which class are we testing here?
(What's EUT?)

A. IDatabaseThingy
B. RealDatabaseThingy
C. FakeDatabaseThingy
D. Example
E. Something else

Choose your tools

A variety of mocking frameworks/libraries with different APIs…
Which one to choose?

- How easy to use?
- How compact is the API?
- How fast?
- How expressive? (Do I need expressive?)
- How strict or flexible are they? (Do I need strict or flexible?)
- What kind of code am I testing?
  - Legacy (JMock?)
  - Greenfield (Mockito?)

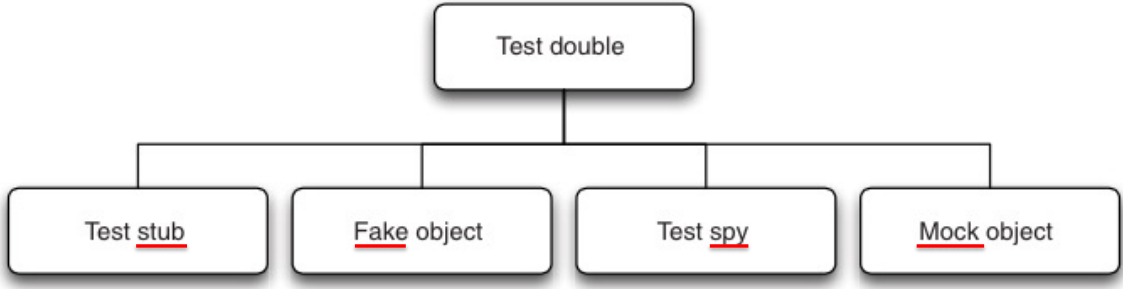
Mockito

A doubling/mocking framework for Java

www.mockito.org

See Canvas > Assignments > Participation >
Mockito Exercises and Prep

Mockito can create all of and any combination of these, but it calls all of them either “mocks” or “spies”



Mockito

Allows you to easily create:

- Stubs
- Restricted fakes
- Mocks with easy interaction verification
- Simple mocks with stub-like behavior
- Complex mocks with intricate expectations, configurable runtime behavior
- Spies that instrument real-object interactions (keep the real collaborator, but spy on its method calls)
- Partial mocks mixing real-object behavior with mock behavior

Creating a mock object and verifying its interactions are easy

```
// Let's import Mockito statically so that the code looks clearer
import static org.mockito.Mockito.*;

// mock creation — you can mock interfaces (List is an interface)
List mockedList = mock(List.class);

// using mock object
mockedList.add("one");
mockedList.clear();

// verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

these are called by the client (EUT) class that uses the mock object

these happen in tests that test the client class

Once created, mock will remember all interactions (calls made to its visible/public operations)

Adapted from www.mockito.org

Giving the mock a behavior is more complicated

```
// You can mock concrete classes, not just interfaces
LinkedList mockedList = mock(LinkedList.class);

// give behavior to mock (called stubbing in mockito)
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

// following prints "first"
System.out.println(mockedList.get(0));

// following throws runtime exception
System.out.println(mockedList.get(1));

// following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

Adapted from www.mockito.org

Void methods and exceptions can be handled

```
doThrow(new RuntimeException()).when(mockedList).clear();

// following throws RuntimeException:
mockedList.clear();
```

- You can use above form with both `doThrow(), doReturn(), doNothing()`
- Use with void methods and when stubbing spies (discussed later)*

*https://stackoverflow.com/questions/36343149/mockito-difference-between-dothrow-and-thenthrow

Adapted from www.mockito.org

Argument matching allows for flexible behavior definition and checking

Mockito normally uses the `equals()` method to match object arguments.

```
// behavior using built-in anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("element");

//behavior using hamcrest
//(let's say isValid() returns your own hamcrest matcher):
when(mockedList.contains(argThat(isValid()))).thenReturn("element");

// following prints "element"
System.out.println(mockedList.get(999));

// you can also verify using an argument matcher
verify(mockedList).get(anyInt());
```

matches any element that is valid

Some built-in matchers (similar to hamcrest): `anyInt()`, `anyString()`, …, `anyObject()`, `anyList()`, `contains(String)`, `endsWith(String)`, `startsWith(String)`, `isA(Class<T>)`, `isNull()`, `isNotNull()`, `same(T)`, …

Adapted from www.mockito.org

Verifying any # of invocations is possible

```
mockedList.add("once");
mockedList.add("twice");
mockedList.add("twice");

verify(mockedList).add("once"); // like an assertion
verify(mockedList, times(1)).add("once"); // same as above line

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeast(2)).add("five times");
```

Occurence argument

`times(1)` is assumed when second argument is omitted

Adapted from www.mockito.org

Can create new matchers using and, or, not, matchers

```
import static org.mockito.AdditionalMatchers.*;
import static org.mockito.Matchers.*;
```

Dependency to import to use and() or() operators

// Verify that doSomething is called with a string arg that starts with “prefix1” or “prefix2” and ends with “suffix”

```
verify(mockClass).doSomething(
   and(or(startsWith("prefix1"), startsWith("prefix2")),
      endsWith("suffix")
   )
);
```

Exercise: “Mockito Prep and Excercises” on Canvas

Make your stack implementation persistable

```
public class MyStack {
   private int maxSize = 10;
   private int[] stackArray;
   private int top;

  public MyStack() {
     stackArray = new int[maxSize];
    top = -1;
  }

  public void push(int j) {
      stackArray[++top] = j;
  }
```

```
    public int pop() {
        return stackArray[top--];
    }

    public int peek() {
        return stackArray[top];
    }

    public int size() {
        return top + 1;
    }

    public boolean isEmpty() {
        return (top == -1);
    }

}
```

Exercise: MyPStack (2) (see Canvas)

New behavior:

- All stack operations that change the stack's state should persist the value of the top element in a DB
- DB stores only top element for each stack object
- DB is shared among all stack objects
- If stack is empty, the DB entry for stack is deleted
- `#MyPStack.reset()`: new operation loads the value of stored element from DB, resets the stack to be a stack of one element having the loaded value
  - if stack is empty, does nothing

Exercise: Inject your dependency (3)

Make your stack implementation persistable

- `#MyPStack(IDataBase)`: new constructor instantiates the stack with a database

Exercise: IDataBase (4)

- DB has one table with two columns: **key**, **value**
- Each row represents a different stack object
- **key** is unique

- `void create(String key, int value);`
- `void update(String key, int newValue);`
- `int read(String key);`
- `void delete(String key);`

stack's unique id

top element

key	value
123	5
234	4
294	18
~~294~~	~~0~~

Exercise: Tasks (5)

- DB implementation (an instance of `IDataBase`) does not exist yet – we don't need it
- Test `MyPStack`'s persistence behavior thoroughly (can you test-drive using TDD?)
  - Mock the DB implementation -- assume DB implementation is correct (we're not testing that)
  - Verify that usual state changing stack operations invoke the right DB operations or don't invoke wrong DB operations
    - when stack is empty, there is no entry for it in the DB
    - push creates DB and saves value in initial push
    - subsequent pushes update the DB with the element just pushed
    - …
  - Test `reset()` by giving behavior to the mock

IDataBase

```
public interface IDataBase {

    void create(String key, int value);
    void update(String key, int newValue);
    int read(String key);
    void delete(String key);

}
```

MyStack -> MyPStack

```
public class MyPStack {

    private int maxSize = 10;
    private int[] stackArray;
    private int top;
    private IDataBase db;
    private String id;
    private static int nextId = 1;

    public MyPStack(IDataBase stackDB) {
        db = stackDB;
        stackArray = new int[maxSize];
        top = -1;
        id = String.valueOf(nextId);
        nextId++;
    }
    …
```

```
public String getId() {
    return id;
}
```

MyStackTest -> MyPStackTest

```
import static org.mockito.Mockito.*;
import org.junit.Before;
import org.junit.Test;
import org.mockito.InOrder;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.fail;
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.*;

public class MyPStackTest {

    private IDataBase db;
    private MyPStack s;

    @Before
    public void setUp() throws Exception {
        db = mock(IDataBase.class);
        s = new MyPStack(db);
    }
    …
```

Make sure all existing tests pass!

MyPStack – a first test using a mock

```
@Test
public void canInstantiateWithMockIDataBase() {
    assertThat(s, is(notNullValue()));
}
```

`initiallyThereIsNoEntryInDB`

Let’s test-drive this!

```
@Test
public void initiallyThereIsNoEntryInDB()
       throws OverflowException, InvalidOperationException {
    verify(db, never()).create(anyString(), anyInt());
}

// Make this test pass!
```

`pushSavesTopInDBDuringFirstPush`

Let’s test-drive this!

```
@Test
public void firstPushSavesTopInDBD()
                  throws OverflowException, InvalidOperationException {
    s.push(100);
    verify(db).create(s.getId(), s.peek());
}

// Make this test pass!
```

Self-reminders:

- Which class are we testing?
- Which is the collaborator?
- Which class is mocked?
- Which class is not mocked?

`pushUpdatesTopInDBInConsecutivePush`

Let’s test-drive this!

pushUpdatesTopInDBInConsecutivePush: Variation

```
@Test
public void pushUpdatesTopInDBInOrder()
    throws OverflowException, InvalidOperationException {
        s.push(100);
        s.push(200);
        s.push(300);
        InOrder inOrder = inOrder(db);
        inOrder.verify(db).update(s.getId(), 200);
        inOrder.verify(db).update(s.getId(), 300);
}
// should pass!
```

Why not verify first `inOrder.verify(db).update(s.getId(), 100);` ??
Because, first call to db is not an update, but a create.
Like this: `inOrder.verify(db).create(s.getId(), 100);`

`popUpdatesTopInDB`

Let’s test-drive this!

`resetReadsRightValueFromDB`

Let’s test-drive this!

`afterResetStackHasOnlyLastTopElement`

Let’s test-drive this!

`whenStackBecomesEmptyDBEntryIsDeleted`

Let’s test-drive this!

`canSaveMultipleStacksInDb`

Let’s test-drive this!

Continuing with Mockito…

Spying on real objects is a piece of cake

```
List list = new LinkedList();
List spy = spy(list); // now we are spying on a real object, not just a double

// optionally, you can override behavior of some methods:
doReturn(100).when(spy).size();  // prefer this form to when-then form with spy

// using the spy calls *real* methods
spy.add("one");
spy.add("two");

// prints "one" - the first element of a list
System.out.println(spy.get(0));

//size() method was stubbed - 100 is printed
System.out.println(spy.size());

//optionally, you can verify interactions with real object
verify(spy).add("one");
verify(spy).add("two");
```

Mixed stubbed and real behavior!
(a partial mock)

Not an ordinary "spy" capability!

This is what makes it a "spy"

This is like instrumenting the real object to see its hidden behavior, and optionally we can change the behavior

Consecutive calls to the same method can have different behavior

```
when(mock.someMethod("some arg"))
  .thenThrow(new RuntimeException())
  .thenReturn("foo");

// First call: throws runtime exception:
mock.someMethod("some arg");

// Second call: prints "foo"
System.out.println(mock.someMethod("some arg"));

// Any consecutive call: prints "foo" as well (last stubbing wins)
System.out.println(mock.someMethod("some arg"));
```

Adapted from www.mockito.org

Verification can be order-sensitive *or* check absence of further interactions

```
singleMock.add("was added first");
singleMock.add("was added second");
singleMock.add("was added third");

// create an inOrder verifier for a single mock
InOrder inOrder = inOrder(singleMock);

// following will make sure that add is first called with
// "was added first", then with "was added second"
inOrder.verify(singleMock).add("was added first");
inOrder.verify(singleMock).add("was added second");

// verifying that no further interactions happen
// following verification will fail
verifyNoMoreInteractions(singleMock);
```

Wrap the mock in a mock object that allows ordered verification

You don't have to verify all interactions one-by-one, but only the ones you're interested in testing in a given order

Adapted from www.mockito.org

Forgetting interactions & behavior allows us to start over

```
List mock = mock(List.class);
when(mock.size()).thenReturn(10);
mock.add(1);

reset(mock);
//at this point the mock forgot any interactions & behavior
```

You can change behavior at run-time in this way!

Adapted from www.mockito.org

More Mockito documentation (full API)

http://mockito.github.io/mockito/docs/current/org/mockito/Mockito.html

References

- Lasse Koskela: Effective Unit Testing, 2013

Assignment A1

Available on Canvas tonight…