



Micael Andersson

Senior
Architect & Developer

Code coverage

Introduction to developers

Micael Andersson
Senior Architect and Developer

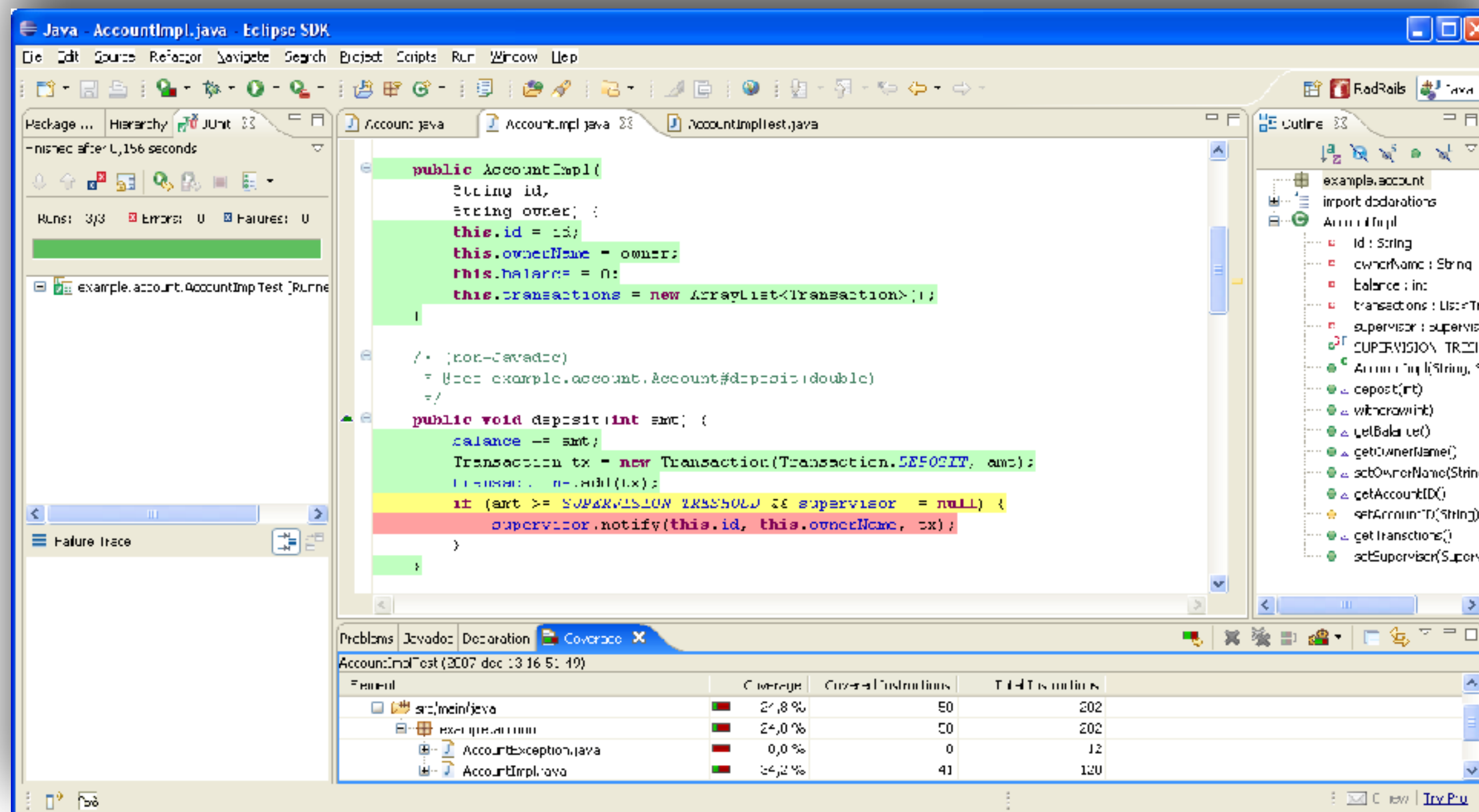


Code Coverage (Java - Demo)

Which statements of my application are being executed?

Useful to identify incomplete testing (ECLemma plug-In)

(Install from Eclipse Marketplace Client)



But ...

Focusing only on coverage is **not sufficient**, you may miss:

- Missing code
- Incorrect handling of boundary conditions
- Timing problems
- Memory Leaks

Use coverage sensibly

- Objective, but incomplete
- Too often distorts sensible action



Code coverage

Exercise / Demo



Exercise - Code Coverage

Run your previous tests from Exercise Stack (or FizzBuzz or Account Demo), to make sure you have adequate Code Coverage!

Use ECLemma()





Micael Andersson
Senior
Architect & Developer

Design for Testability

Introduction to developers

Micael Andersson
Senior Architect and Developer



Pitch



Design for test



Goals of this presentation

“Design for test” Theory

- Introduce aspects on Coupling
- Explain Law Of Demeter (LoD)
- Explain Dependency injection
- Explain Synthetic Mock

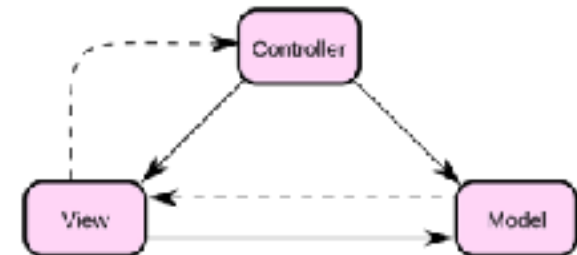


Test UI, use MVC = Model-View-Control

User Interfaces are **notoriously difficult** to test, but...

Use MVC-Pattern to Split a complex application into separate, **cohesive** parts which separates presentation from application logic **enables testing of the application logic in isolation.**

N.B. User Interfaces could be tested with UI automation framework like **Sikuli** and UI test frameworks like **Jubula**, but it's not covered here.



Law of Demeter* (a.k.a LoD or principle of least knowledge)

Any method should have **limited knowledge about its surrounding** object structure.

Hence, this is worse...

```
public class SomeUnit {  
    private Dependee dependee;  
    public SomeUnit() {  
        this.dependee = new SomeDependee();  
    } ...  
}
```

*Demeter, Greek goddess of agriculture, “distribution-mother”



Law of Demeter (Contd.)

This is better, because SomeUnit is agnostic of SomeDependee class

```
public class SomeUnit {  
    private IDependee dependee;  
    public SomeUnit() {  
    }  
  
    public setDependee (IDependee dependee) {  
        this.dependee = dependee;  
    } ...  
}
```

Use an interface instead!!

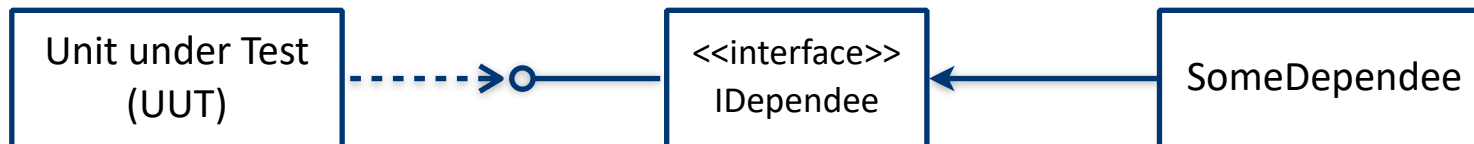


LoD - Don't Talk To Strangers

If there are no strong reasons why two classes should talk to each other directly, *they shouldn't!*



becomes

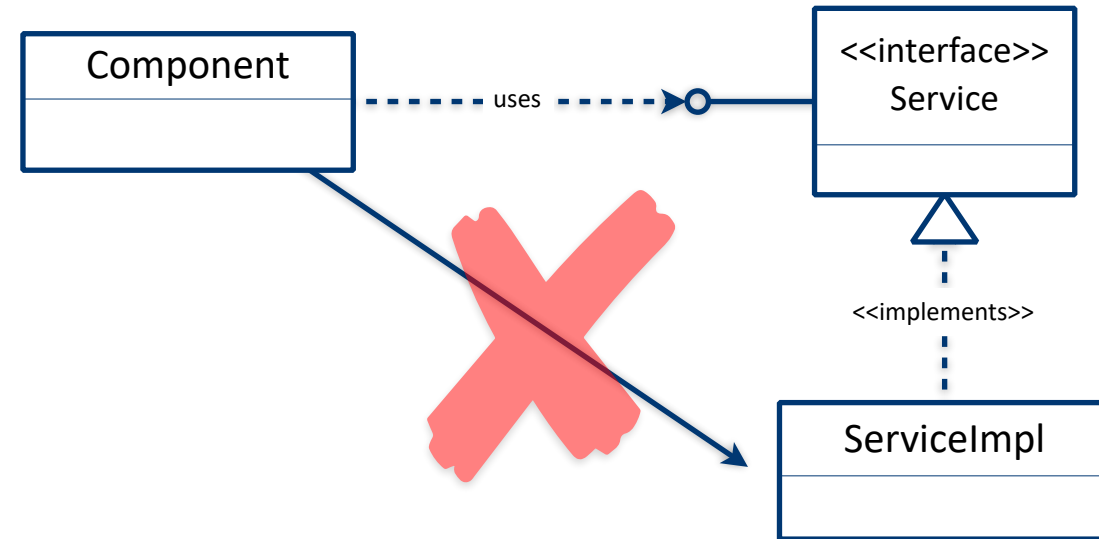


Dependency Injection - pattern

Dependency Injection provides a technique for **managing dependencies between** components in a decoupled way.

Dependency Management where **coupling is attached/detached at runtime**.

Makes it easier to “Unit test” components in isolation: Out of container and with mocked dependencies

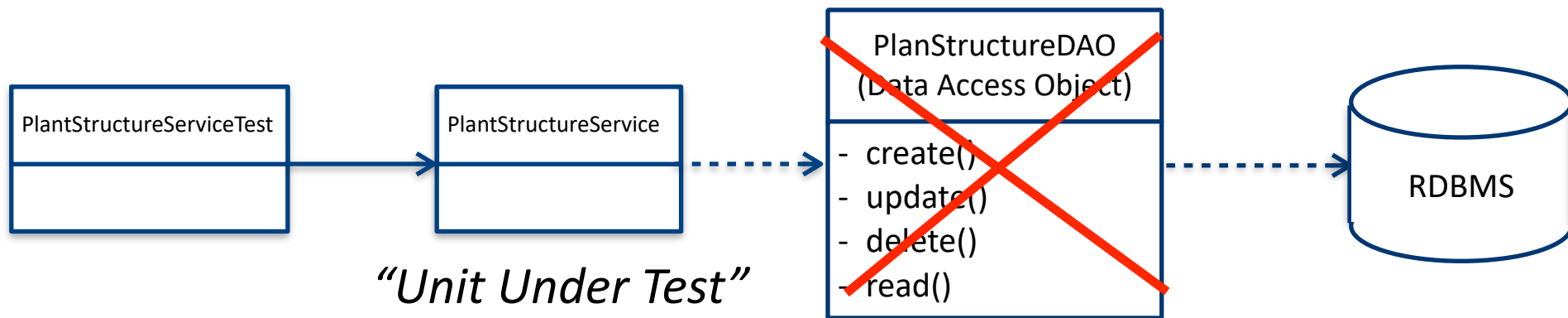


Side effects

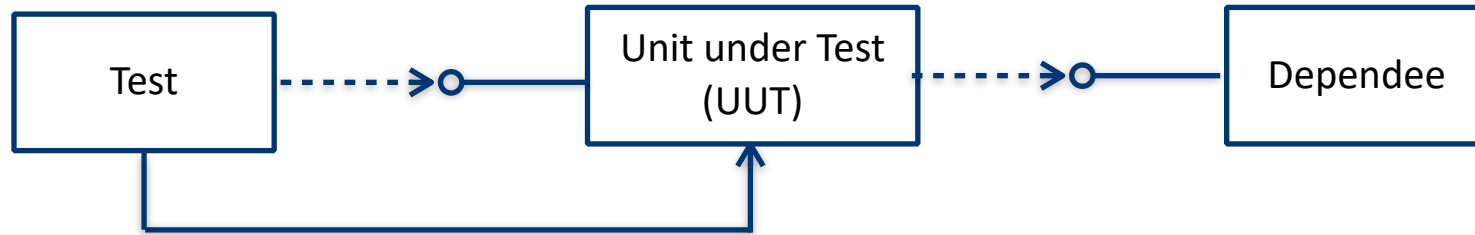
But what about units that depend on other units (with potential **side effects**)?

We can't of course ignore the side effects!

We need to Mock!



Strategies for testing Units that depend on other units



Break the dependency: Let the Test create a **synthetic 'Mock' context**

Run and test the Unit within it's natural context (In **Container** in the case of Java or .NET)

Let the Test create the context needed.

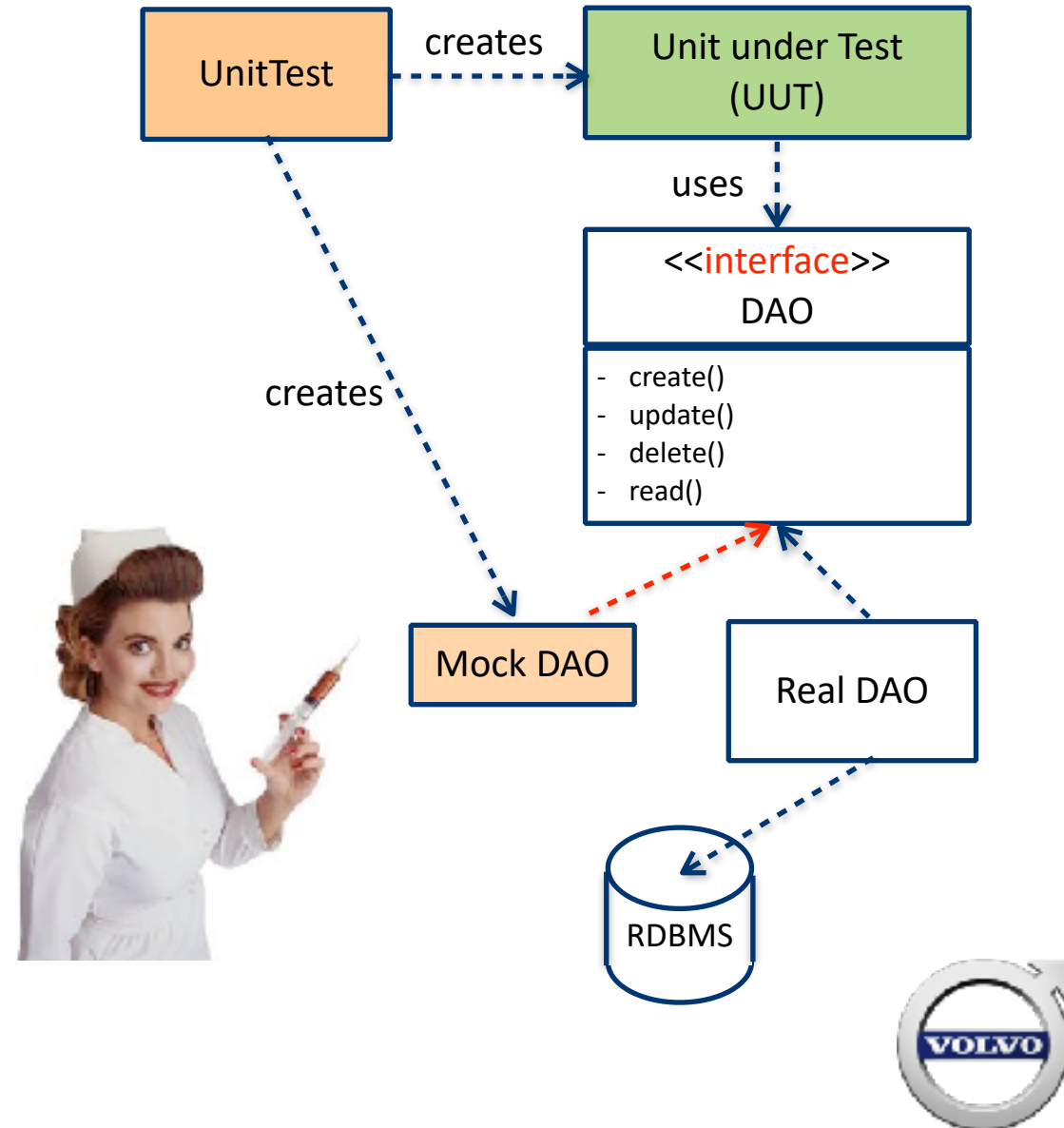


Synthetic context → Mocking

Implements the same interface as the resource that it represents

Enables configuration of its behaviour from outside (i.e. from the test class, in order to achieve locality) by **injecting** a mock at runtime

Enables registering and verifying *expectations* on how the resource is used



Thank you for listening!

Exercises





Micael Andersson

Senior

Architect & Developer

Mocking Java

Introduction to developers

Micael Andersson
Senior Architect and Developer



Mocking

(Using plain Java)

System in Production



- Component Under Test
- Depended on Components
- Additional Components

System in Unit Test



- Component Under Test
- Mocks for Components

Why Mock?

Main principle of TDD:

Unit tests should act as a safety net and **provide quick feedback**.

A **test sometime take time** to execute. Mainly due to the following reasons:

- A test acquires a connection from the database that fetches/updates data
- It connects to the Internet and downloads files
- It interacts with an SMTP server to send/receive e-mails
- It performs I/O operations



Do we need to test the real things?

Do we really need to acquire a database connection or download files to test code?

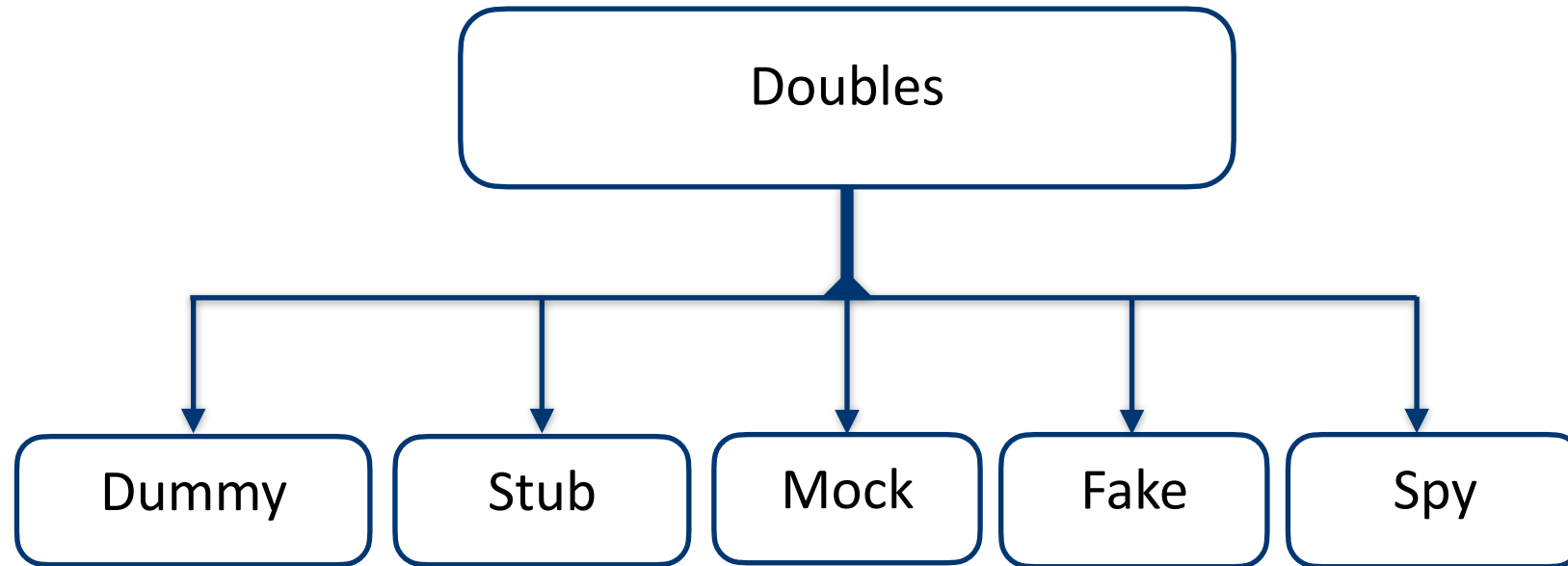
The answer is **YES!**

If it doesn't connect to a database or download the latest stock price, some parts of the system remain untested. So, DB interaction or network connection is mandatory for some parts of the system, and these needs **integration tests**.

But! To **unit test** these parts, the external dependencies can be mocked out.



Test Doubles



Dummy

A dummy would be like a movie scene where the **double doesn't perform anything but is only present** on the screen.

They are used when the actual actor is not present, but their presence is needed for a scene.



Dummy example

Similarly, dummy objects are passed to avoid `NullPointerException` for mandatory parameter objects as follows:

```
@Test
```

```
public void testIssuingABook() {  
    Book javaBook = new Book("Java 101", "123456");  
    IMember dummyMember = new DummyMember(); //<- our dummy  
    javaBook.issueTo(dummyMember);  
    assertEquals("Fail...", javaBook.numberOfTimesIssued(), 1);  
}
```

N.B. The need for dummies is decreasing since introduction of the concept of `Optional<T>` in Java 8 since Optionals can cope with null values.

```
public final class Optional<T>() extends Object {...}
```



Stub

A stub **delivers indirect inputs** to the caller when the stub's methods are called.

Stubs are programmed **only for the test scope**, hence it is in the test branch of the source tree.

Stubs **may record other information** such as the number of times the methods were invoked and so on.



Stub example

Account transactions should be rolled back if the ATM's money dispenser fails to dispense money. How can we test this when we don't have the ATM machine, or how can we simulate a scenario where the dispenser fails? We can do this using the following code:

```
public interface IDispenser {  
    void dispense(BigDecimal amount) throws DispenserFailed;  
}  
  
public class AlwaysFailingDispenserStub implements IDispenser {  
    public void dispense(BigDecimal amount) throws DispenserFailed  
    {  
        throw new DispenserFailed (ErrorType.HARDWARE,  
            "Failing dispense");  
    }  
}
```



Stub example (cont.)

@Test

```
public void test_transaction_is_rolledback_when_hardware_fails() {  
    // Arrange: set up a failing dispenser  
    Account myAccount = new Account("John", 2000.00);  
    TransactionManager txMgr = TransactionManager.forAccount(myAccount);  
    txMgr.registerMoneyDispenser(new AlwaysFailingDispenserStub());  
  
    // Act:  
    WithdrawalResponse response = txMgr.withdraw(500.00);  
  
    // Assert: no success and that no change on the account  
    assertEquals("Fail...", false, response.wasSuccess());  
    assertEquals("Fail...", 2000.00, myAccount.balance());  
}
```



Fake

Fake objects are **mostly working implementations**, the fake class **extends the original class**, but it usually hacks the performance, which makes it unsuitable for production.



Installing a fake pothole



Fake example

```
public class AddressDAO extends SimpleJdbcDaoSupport {

    public void batchInsertOrUpdate(List<AddressDTO> addressList, User user) {
        List<AddressDTO> insertList = buildListWhereLastChangeTimeMissing(addressList);
        List<AddressDTO> updateList = buildListWhereLastChangeTimeValued(addressList);
        int rowCount = 0;

        if (!insertList.isEmpty()) {
            rowCount = getJdbcCollaborator().batchUpdate(INSERT_SQL, "...");
        }
        if (!updateList.isEmpty()) {
            rowCount += getJdbcCollaborator().batchUpdate(UPDATE_SQL, "...");
        }
        if (addressList.size() != rowCount) {
            raiseErrorForDataInconsistency("...");
        }
    }

    public JdbcCollaborator getJdbcCollaborator() { // <----- Fake this method !!!
        // JdbcCollaborator has complicated implementation, interacts with DB..
    }
}
```



Fake example (cont.)

```
public class FakeAddressDAO extends AddressDAO {
    private JdbcCollaborator jdbcCollaboratorStub = new JdbcCollaboratorStub();
    // JdbcCollaboratorStub inherit from JdbcCollaborator

    @Override // <-- To get rid of the complex collaborator and side effects
    public JdbcCollaborator getJdbcCollaborator() {
        return jdbcCollaboratorStub;
    }
}

@Test
public void testAddressDAO() {
    //Arrange
    AddressDAO addressDAO = new FakeAddressDAO();
    User user = new User("Apple Inc.");
    List<AddressDTO> addressList = new ArrayList<AddressDTO>();
    AddressDTO addressDTO = new AddressDTO("One Infinite Loop", "Cupertino", "CA");
    addressList.add(addressDTO);
    // Act
    addressDAO.batchInsertOrUpdate(addressList, user);
    // Assert ....
}
```

The Fake subclasses the real object
and overrides the get-method
that returns a stubbed
JdbcCollaborator.
Use the fake in the test.



Mock

Mock objects have expectations.

A test expects a value from a mock object, and during execution, a mock object **returns the expected result**.

Also, mock objects can keep track of the number of times a method on a mock object has been invoked.



Mock example

The following example is a continuation of the ATM example with a mock version. In the previous example, we stubbed the dispense method of the Dispenser interface to throw an **DispenserFailed** exception; here, we'll use a mock object to replicate the same behaviour.

```
public class MockDispenser implements IDispenser {  
    private int notes100;  
    private int notes500;  
  
    public MockDispenser(int notes100, int notes500) {  
        this.notes100 = notes100;  
        this.notes500 = notes500;  
    }  
  
    @Override  
    public int dispense(int amount) throws DispenserFailed {  
        int amountLeft = dispenseNotes(amount, 500, notes500);  
        return dispenseNotes(amountLeft, 100, notes100);  
    }  
  
    private int dispenseNotes(int amount, int noteValue, int noOfNotes) {  
        if (amount % noteValue == 0) {  
            int noOfBills = amount / noteValue;  
            if (0 < noOfBills) {  
                if (noOfBills <= noOfNotes) {  
                    amount = amount - noOfBills * noteValue;  
                    System.out.println("Dispense " + noOfBills + " x " + noteValue + "-notes.");  
                } else {  
                    throw new DispenserFailed("Not enough " + noteValue + "-notes!");  
                }  
            }  
        }  
        return amount;  
    }  
}
```



Mock example

The test main focus is to verify “no money is withdrawn” in the TransactionManager.withDraw(...) method.

```
IDispenser mockedDispenser;
```

```
@Test
public void
testTransactionIsRolledBackWhenDispenserIsShortOnNotes() {
    Account myAccount = new Account(2000.00, "John");
    int noOf100notes = 2;
    int noOf500notes = 1;
    mockedDispenser = new MockDispenser(noOf100notes,
noOf500notes);
```

```
    TransactionManager txMgr =
TransactionManager.forAccount(myAccount);
    txMgr.registerMoneyDispenser(mockedDispenser);

    txMgr.withdraw(1500);
    assertEquals(2000.00, myAccount.getBalance(), 0.01f);

    txMgr.withdraw(300);
    assertEquals(2000.00, myAccount.getBalance(), 0.01f);
}
```

```
public class TransactionManager {
```

```
    private Account account;
    private IDispenser dispenser;
```

```
    public TransactionManager(Account account) {
        this.account = account;
    }
```

```
    public static TransactionManager forAccount(Account myAccount) {
        return new TransactionManager(myAccount);
    }
```

```
    public void registerMoneyDispenser(IDispenser dispenser) {
        this.dispenser = dispenser;
    }
```

```
    public void withdraw(int amount) {
        System.out.println("Trying to withdraw " + amount + " kr");
        try {
            dispenser.dispense(amount);
            account.withDraw(amount);
        } catch (DispenserFailed e) {
            String msg = e.getMessage();
            System.out.println("Can not dispense money, " + msg);
        }
    }
```

```
}
```



Spy

Spy is a variation of a mock/stub, but instead of only setting expectations, **it records** the calls made to the collaborator.

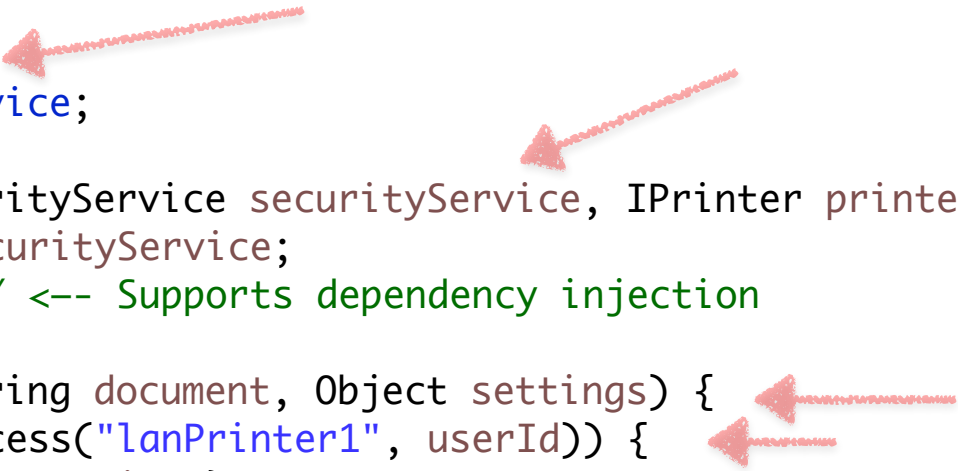


Spying, Listening, Eavesdropping



Spy example

```
public class ResourceAdapter {  
    IPrinter printer;  
    ISecurityService securityService;  
  
    public ResourceAdapter(ISecurityService securityService, IPrinter printer) {  
        this.securityService = securityService;  
        this.printer = printer; // <-- Supports dependency injection  
    }  
    void print(String userId, String document, Object settings) {  
        if (securityService.canAccess("lanPrinter1", userId)) {  
            printer.print(document, settings);  
        }  
    }  
}  
  
public interface IPrinter {  
    void print(String document, Object settings);  
}  
  
public interface ISecurityService {  
    public boolean canAccess(String name, String id);  
}
```



Spy Example cont.

```
public class FakeSecurityService implements ISecurityService { // <-- Fake a service
    public boolean canAccess(String name, String id) {
        return true;
    }
}

public class SpyPrinter implements IPrinter { // <-- Implement a Spy
    private int noOfTimesCalled = 0;
    @Override
    public void print(String document, Object settings) {
        noOfTimesCalled++;
    }
    public int getInvocationCount() {
        return noOfTimesCalled;
    }
}

@Test
public void testVerifyCanPrint() throws Exception {
    SpyPrinter spyPrinter = new SpyPrinter(); // <-- Arrange
    adapter = new ResourceAdapter(new FakeSecurityService(), spyPrinter);
    adapter.print("john", "helloWorld.txt", "all pages");
    assertEquals(1, spyPrinter.getInvocationCount());
}
```



When to use mocking (and when not to)

Mocking is great for

- **Breaking dependencies** between well-architected layers or tiers
- Testing corner cases and **exceptional** behaviour

Mocking is less ideal for

- Replacing awkward 3rd party APIs
- Responsibilities which involves large amounts of **state or data**, which could be more conveniently expressed in a "native" format

This is clearly a judgement call: If breaking a dependency using mock objects **cost more** effort than **living with the dependency**, then the mock strategy is probably not a good idea



Thank you for listening!

Exercises

