

VOLVO

TDD

Breaking Dependencies

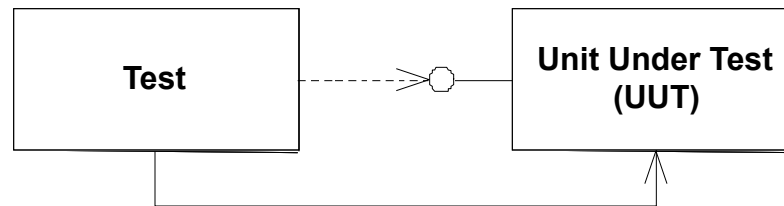
Design properties and Design goals

For Units:

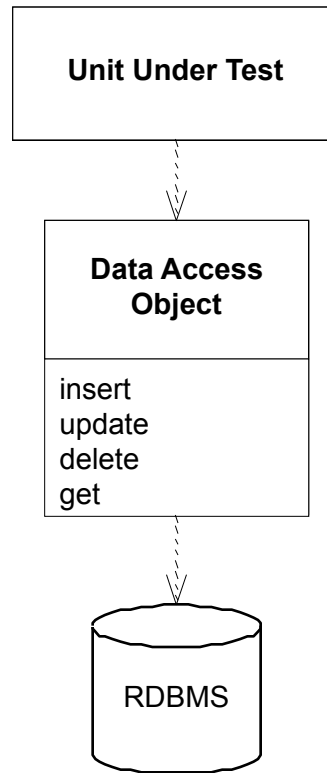
- Modularity
- High cohesion
- Low coupling

For Tests:

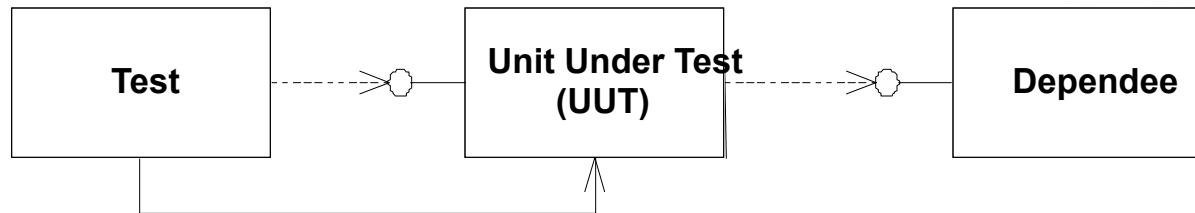
- Modularity
- Locality



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



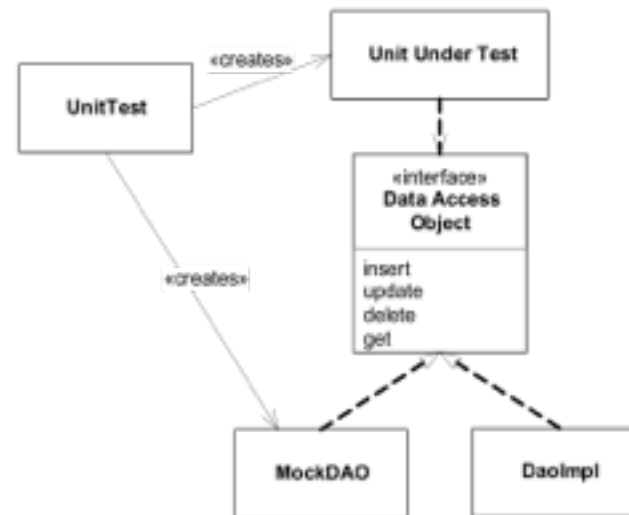
Strategies for testing Units that depend on other units



- 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 EE or .NET)
- Let the Test create the real context

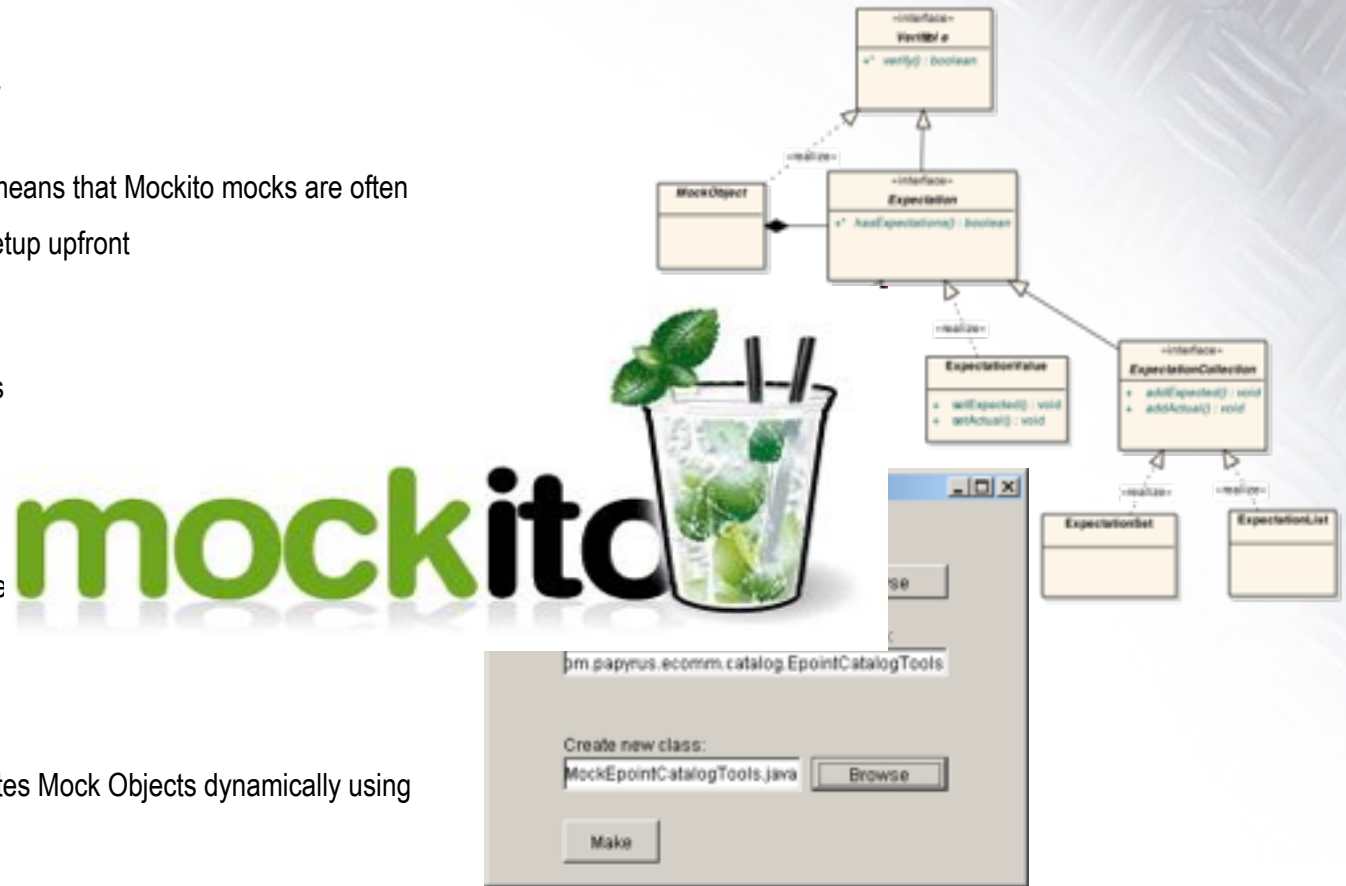
Synthetic context – MockObjects

- Implements the same interface as the resource that it represents
- Enables configuration of its behavior from outside (i.e. from the test class, in order to achieve locality)
- Enables registering and verifying *expectations* on how the resource is used



Frameworks and tools for creating

- code.google.com/p/mockito/
 - No expect-run-verify also means that Mockito mocks are often ready without expensive setup upfront
- www.mockobjects.org
 - Commonly used assertionsExpectation classes, which
- www.mockmaker.org
 - Tool which automatically generates Mock Objects or Interface
- www.easymock.org
 - Class library which generates Mock Objects dynamically using the Java Proxy class



EASYMOCK



- Mocks concrete classes as well as interfaces
- Little annotation syntax sugar - `@Mock`
- Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers (`anyObject()`, `anyString()` or `refEq()` for reflection-based equality matching)



Example usage

- Create MockObject
- Let the mock object know how to answer on an expected call
- Inject the MockObject in the class to be tested
- Run the test
- Verify that the mock object received the expected calls and parameters

```
@Test
public void testNotificationVetoShouldBeHonoured() {
    int amount = AccountImpl.SUPERVISION_TRESHOLD;

    Supervisor mockSupervisor = Mockito.mock(Supervisor.class);

    Mockito.when(mockSupervisor.notify(Mockito.anyString(),
        Mockito.anyString(), (Transaction) Mockito.anyObject()))
        .thenReturn(false);

    account.setSupervisor(mockSupervisor);

    try {
        account.deposit(amount);
        Assert.fail("SupervisorException expected");
    } catch (SupervisorException expected) {
        // expected
        System.err.println(expected);
    }

    Mockito.verify(mockSupervisor).notify(account.getAccountID(), account.getOwnerName(),
        new Transaction(Transaction.DEPOSIT, amount));
}
```


Typical usage scenario for Mock Objects in a TestCase

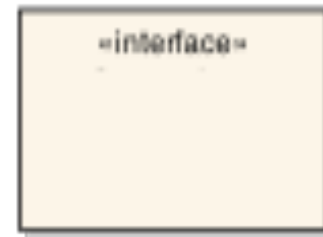
1. Instantiate mockobjects
2. Set up state in mockobjects, which govern their behavior
3. Set up expectations on mock objects
4. Execute the method(s) on the Unit Under Test, using the mockobjects as resources
5. Verify the results
6. Verify the expectations

Mockito Example in JVS Pos

- `com.volvo.jvs.pos.client.w.order.actions. AbstractMockStrutsTestCase`

Exercise 7

- Extend the tests for AccountImpl to use Mockito for validating correct usage of the Supervisor collaborator!



When to use Mock Objects (and when not to)

- Mock Objects are great for
 - Breaking dependencies between well-architected layers or tiers
 - Testing corner cases and exceptional behaviour
- Mock Objects are 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

Designing for Testability :

Law of Demeter (LoD or principle of least knowledge)

- Any method should have limited knowledge about its surrounding object structure.
- Named in honor of Demeter, “distribution-mother”, Greek goddess of agriculture
- Hence

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

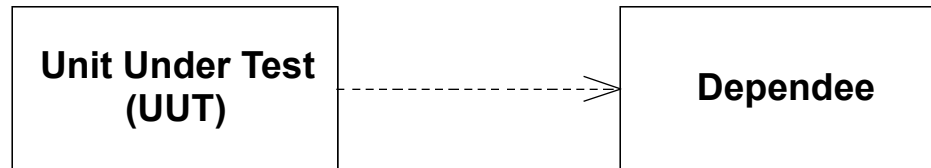
Law of Demeter (Contd.)

- becomes

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

Designing for Testability : 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



Designing for Testability : Dependency Injection

- What is it?
 - Dependency Management
 - Dependency Injection provides a mechanism for managing dependencies between components in a decoupled way
- Makes it easier to unit test components in isolation
 - Out of container and with mocked dependencies

