TUTORIAL ON UNIT TESTING
INF4290

Erik Arisholm
Daglig leder, Testify AS
Professor in software engineering, UiO
What we will cover

- A practical introduction to (class-level) unit testing of OO systems
- Refactoring for testability
- Isolation frameworks
- State-based versus behavior-based unit testing
- Test case specification and implementation
- Guided by an example
  - Relatively simple ATM machine written in C#
  - Black box modeling: CTE-XL (similar to “category-partition”)
  - Unit testing tools: Visual Studio, NUnit, Ncover, Rhino Mocks
  - All of the techniques presented are directly applicable to Java and most other OO languages as well, except for (minor) syntactical differences
- Complete working code will be posted on the course website, all required tools are available as open source or trial/free versions
Properties of “good” unit tests

• **Isolation** (a somewhat controversial opinion)
  • The tests of class X are not dependent on having implemented collaborating classes, or tests for collaborating classes.
  • The tests of class X should not fail due to faults in collaborating classes.
  => Enables Test-Driven Development, “need-driven” testing, or “top-down” testing

• **Completeness**
  • A unit test should test all possible services within a class, also those that are not currently in use by other classes.

• **Independence**
  • Each unit test should be self-contained and should “work” independently of whatever other unit tests are executed.

• **Simplicity**
  • One test = one scenario
  • Strive for simple test fixtures
  • Tests should be fast, e.g., by avoiding calls to databases if possible
Example unit testing framework

- CTE-XL
- Black-box test case specifications
- Test class w/ Test Fixture
- Isolation: Mocks/Stubs
- Test driver
- Class
- White-box Coverage
- NUnit
- Rhino Mocks
- NCover
The ATM case study

Welcome!
Please enter your account number: 12345
Enter your PIN: 54321
Main menu:
1 - View my balance
2 - Withdraw cash
3 - Deposit funds
4 - Exit
Enter a choice: 1
Balance Information:
- Available balance: $1,000.00
- Total balance: $1,200.00
Main menu:
1 - View my balance
2 - Withdraw cash
3 - Deposit funds
4 - Exit
Enter a choice: 3
Please input a deposit amount in CENT$ (or 0 to cancel): 1000
Please insert a deposit envelope containing $10.00 in the deposit slot.
Your envelope has been received.
The money just deposited will not be available until we
verify the amount of any enclosed cash, and any enclosed checks clear.
Main menu:
1 - View my balance
2 - Withdraw cash
3 - Deposit funds
4 - Exit
Enter a choice: 4
Exiting the system...
Thank you! Goodbye!
Press any key to continue . . .
Dependencies in the first version of the ATM

What a mess!! 😞
ATM case study - first unit tests…

• Task 1
  • *ATMRunner* is a “top-level” class of the application
  • It first authenticates a user, and then lets the user perform one or more transactions (account balance, withdraw, deposit) before exiting
  • Handling actual “transactions” is delegated to other control classes
  • We want to unit test the behaviour of *ATMRunner*

• But how? It is simply not testable (a typical situation)!
  • We need to *control* the class in order to execute the tests
  • currently this is not possible, as *ATMRunner* uses a *Keypad* class to get input from a user, which in turn reads from standard input. Thus, *ATMRunner* is controlled by *indirect input* from another class (Keypad)
  • We need to *observe* results of exercising the class methods, either in terms of changes in state or observable outputs, and compare with *expected* results
  • Currently we cannot easily observe results directly, as the results are presented indirectly via the *Screen* class, which outputs text directly to standard output (the console).
  • No state variables in *ATMRunner* available to observe state changes.
Simple classification of fake objects

• Test Dummy
  • Just an object needed for execution of the class, but no control or observation needed

• Test Stub
  • Enables us to control what values are returned when the class under test calls methods on a collaborating object

• Test Mock
  • Enables us to observe what method calls (including any parameters) that are made to a collaborating object from the class under test
    • This is also known as behaviour verification (as opposed to state verification)

• Other, more elaborate classifications exist
  • See for example the book *xUnit Test Patterns* (reference at the end of the slides)
Refactoring the ATM to be more testable

• Extract an interface of Keypad (IKeypad) and Screen (IScreen) to allow replacing underlying implementation with stub/mock implementations that are used for testing
  • In our case, we need a stub for Keypad (to control) and a mock for Screen (to observe)

• How to inject stub/mock implementations into a class under test
  • Alt. 1: Dependency injection at the constructor level
    • E.g., ATMRunner(IKeypad myKeypadImpl, IScreen myScreenImpl)
  • Alt. 2: Dependency injection as a setter property/method.
    • E.g., ATMRunner::setKeypad(IKeypad myKeypadImpl), ATMRunner::setScreen(IScreen myScreenImpl)
  • Alt. 3: Dependency injection using an object factory
    • E.g., myKeypad = ObjectFactory.CreateKeypad();
ATM v2 classes

Transaction
Abstract Class

Withdrawal
Class
- Transaction

Deposit
Class
- Transaction

BalanceInquiry
Class
- Transaction

Screen
Class

Methods
- AccountNumber
- AskDepositAmount
- CancelingTransaction
- ConfirmDeposit
- DisplayBalance
- DisplayDollarAmount
- DisplayMessage
- DisplayMessageLine
- GoodBye
- InsufficientFundsAccount
- InsufficientFundsATM
- InvalidAccountOrPin
- InvalidSelection
- MainMenu
- NoEnvelopeReceived
- PIN
- RequestDeposit
- TakeCash
- Welcome
- WithdrawalOptions

Keypad
Class

Methods
- GetInput

ATMObjectFactory
Class

Methods
- CreateKeypad
- CreateScreen
- SetKeypad
- SetScreen

DepositSlot
Class

Methods
- IsDepositEnvelopeReceived

ATMRunner
Class

Fields
- currentAccountNumber
- userAuthenticated

Methods
- ATMRunner
- AuthenticateUser
- CreateTransaction
- DisplayMainMenu
- PerformTransactions
- RunOnce

Nested Types

cashDispenser

BankDatabase
Class

Methods
- AuthenticateUser
- BankDatabase
- Credit
- Debit
- GetAccount
- GetAvailableBalance
- GetTotalBalance

Account
Class

Fields
- Account
- Credit
- Debit
- ValidatePIN

Screen
Class

Methods
- AccountNumber
- AskDepositAmount
- CancelingTransaction
- ConfirmDeposit
- DisplayBalance
- DisplayDollarAmount
- DisplayMessage
- DisplayMessageLine
- GoodBye
- InsufficientFundsAccount
- InsufficientFundsATM
- InvalidAccountOrPin
- InvalidSelection
- MainMenu
- NoEnvelopeReceived
- PIN
- RequestDeposit
- TakeCash
- Welcome
- WithdrawalOptions
Injecting stub and mock in *ATMRunner*

```java
public class ATMRunner {
    // enumeration that represents main menu options
    private enum MenuOption {
        BALANCE_INQUIRY = 1,
        WITHDRAWAL = 2,
        DEPOSIT = 3,
        EXIT_ATM = 4
    }

    private bool userAuthenticated; // true if user is authenticated
    private int currentAccountNumber; // user's account number
    private IScreen screen; // reference to ATM's screen
    private IKeypad keypad; // reference to ATM's keypad
    private CashDispenser cashDispenser; // ref to ATM's cash dispenser
    private DepositSlot depositSlot; // reference to ATM's deposit slot
    private BankDatabase bankDatabase; // ref to account info database

    // parameterless constructor initializes instance variables
    public ATMRunner() {
        userAuthenticated = false; // user is not authenticated to start
        currentAccountNumber = 0; // no current account number to start
        screen = ATMObjectFactory.CreateScreen(); // create screen
        keypad = ATMObjectFactory.CreateKeypad(); // create keypad
        cashDispenser = new CashDispenser(); // create cash dispenser
        depositSlot = new DepositSlot(); // create deposit slot
        bankDatabase = new BankDatabase(); // create account info database
    }
}
```

- Interfaces instead of class instance
- Returns “normal” or “fake” implementation
The object factory

```java
public class ATMObjectFactory {
    static IKeypad customIKeypad = null;
    static IScreen customIScreen = null;

    static public IKeypad CreateKeypad() {
        if (customIKeypad != null)
            return customIKeypad;
        return new Keypad();
    }
    static public void SetKeypad(IKeypad kp) {
        customIKeypad = kp;
    }

    static public IScreen CreateScreen() {
        if (customIScreen != null)
            return customIScreen;
        return new Screen();
    }
    static public void SetScreen(IScreen scr) {
        customIScreen = scr;
    }
}
```

- Will return fake or real object
- Here we can inject a Keypad test stub
- Will return fake or real object
- Here we can inject a Screen test mock
Definition of the Keypad Test Stub

• The test stub maintains a list of inputs
  • array of int in this case
• The test class populates this list with values before a test, to control the return values
• Each time the GetInput() method of the stub is called, it returns the next number from the list, instead of actual user input

```java
public class MyKeyPadStub : IKeypad
{
    public int[] numbers = new int[100];
    int idx = 0;

    public int GetInput()
    {
        int retval = numbers[idx];
        idx++;
        return retval;
    }
} // end method GetInput
```
Definition of the Screen Test Mock

The mock maintains a list of calls made to its public methods (array of string in this case)

Whenever a method is called, the call is added to the list

After a test, the test class can query the list of calls stored in the mock, to determine if expected calls were made by the class under test

```java
public class MyScreenMock : IScreen
{
    public string[] outstrings = new string[100];
    int idx = 0;

    public void Welcome()
    {
        outstrings[idx] = "Welcome()"; idx++;
    }

    public void AccountNumber()
    {
        outstrings[idx] = "AccountNumber()"; idx++;
    }

    public void PIN()
    {
        outstrings[idx] = "PIN()"; idx++;
    }

    public void InvalidAccountOrPin()
    {
        outstrings[idx] = "InvalidAccountOrPin()"; idx++;
    }

    public void MainMenu()
    {
        outstrings[idx] = "MainMenu()"; idx++;
    }

    public void InvalidSelection()
    {
        outstrings[idx] = "InvalidSelection()"; idx++;
    }

    public void GoodBye()
    {
        outstrings[idx] = "GoodBye()"; idx++;
    }
}
Nunit guidelines

- One test class for each application class under test
- One test package for each application package under test
- At least one test method for each public class method

Naming conventions:
- Test package: `<Project>.Test`
  - Example: “ATM.Test”
- Test class: `<Class>Test`
  - For example “ATMRunnerTest”, “DepositTest”, “WithdrawTest”
- Test method: `<Class method name>_Scenario`
  - Example: “Execute_positive_amount”

- Use [SetUp] and [TearDown] to reuse code across tests.
Example NUnit test of **ATMRunner**

```csharp
[TestFixture]
public class ATMRunnerTest
{

    MyKeyPadStub myKeypadStub;
    MyScreenMock myScreenMock;
    ATMRunner theATM;

    [SetUp]
    public void Setup()
    {
        myKeypadStub = new MyKeyPadStub();
        myScreenMock = new MyScreenMock();

        ATMOBJECTFACTORY.SetKeypad(myKeypadStub);
        ATMOBJECTFACTORY.SetScreen(myScreenMock);
        theATM = new ATMRunner();
    }

    [Test]
    public void RunOnce_ValidAccountInvalidPinValidAccountValidPinExit()
    {
        myKeypadStub.numbers[0] = 12345;
        myKeypadStub.numbers[1] = 1;
        myKeypadStub.numbers[2] = 12345;
        myKeypadStub.numbers[3] = 54321;
        myKeypadStub.numbers[4] = 4;

        theATM.RunOnce();
        Assert.AreEqual("Welcome", myScreenMock.outstrings[0]);
        Assert.AreEqual("AccountNumber", myScreenMock.outstrings[1]);
        Assert.AreEqual("PIN", myScreenMock.outstrings[2]);
        Assert.AreEqual("InvalidAccountOrPin", myScreenMock.outstrings[3]);
        Assert.AreEqual("Welcome", myScreenMock.outstrings[4]);
        Assert.AreEqual("AccountNumber", myScreenMock.outstrings[5]);
        Assert.AreEqual("PIN", myScreenMock.outstrings[6]);
        Assert.AreEqual("MainMenu", myScreenMock.outstrings[7]);
        Assert.AreEqual("GoodBye", myScreenMock.outstrings[8]);
    }
}
```
NUnit test results

• All tests pass, but it is not a complete test suite. It just checks that user authentication works, then it exits.
• We also need to test that the class behaves as expected if a user chooses to perform one or more transactions (withdrawal, deposit, balance inquiry).
Using NCover to assess test coverage

```csharp
private void PerformTransactions()
{
    Transaction currentTransaction; // transaction being processed
    bool userExited = false; // user has not chosen to exit

    // loop while user has not chosen exit option
    while (!userExited)
    {
        // show main menu and get user selection
        int mainMenuSelection = DisplayMainMenu();

        // decide how to proceed based on user's menu selection
        switch ((MenuOption)mainMenuSelection)
        {
            // user chooses to perform one of three transaction types
            case MenuOption.BALANCE_INQUIRY:
            case MenuOption.WITHDRAWAL:
            case MenuOption.DEPOSIT:
                // initialize as new object of chosen type
                currentTransaction = CreateTransaction(mainMenuSelection);
                currentTransaction.Execute(); // execute transaction
                break;

            case MenuOption.EXIT_ATM: // user chose to terminate session
                userExited = true; // this ATM session should end
                break;

            default: // user did not enter an integer from 1-4
                screen.InvalidSelection();
                break;
        }
    }

    // end switch
    } // end while
} // end method PerformTransactions
```
More isolation required…

• By creating “fake objects” for *Keyboard* (a Stub object) and *Screen* (a Mock object), we can control the class under test and observe the results of our tests.

• But the dependencies of *ATMRunner* to all other classes in the system results in our “unit tests” for handling transactions by *ATMRunner* would become integration or system function tests, not isolated unit tests!
  • Complicates test setup and test oracle implementations if we are to write complete tests for the given class under test.
  • Also, all other classes need to be implemented before we can complete the tests.
  • Prevents early testing, results in slow tests, and dependencies to “external” or hard to test units (hardware, databases, …)
More dependency injection
ATM class dependencies after refactoring
public class ATMObjectFactory
{
    static IKeypad customKeypad = null;
    static IScreen customScreen = null;
    static IBankDatabase customBankDatabase = null;
    static ITransaction customBalanceInquiry = null;
    static ITransaction customWithdrawal = null;
    static ITransaction customDeposit = null;
    static IDepositSlot customDepositSlot = null;
    static ICashDispenser customCashDispenser = null;

    static public IKeypad CreateKeypad()
    {
        if (customKeypad != null) return customKeypad;
        return new Keypad();
    }

    static public void SetKeypad(IKeypad kp) {customKeypad = kp;}

    static public IScreen CreateScreen()
    {
        if (customScreen != null) return customScreen;
        return new Screen();
    }

    static public void SetScreen(IScreen scr) {customScreen = scr;}

    static public IBankDatabase CreateBankDatabase()
    {
        if (customBankDatabase != null) return customBankDatabase;
        return new BankDatabase();
    }

    static public void SetBankDatabase(IBankDatabase bnk) {customBankDatabase = bnk;}

    static public ITransaction CreateBalanceInquiry(int userAccountNumber, IScreen atmScreen, IBankDatabase atmBankDatabase)
    {
        if (customBalanceInquiry != null) return customBalanceInquiry;
        return new BalanceInquiry(userAccountNumber, atmScreen, atmBankDatabase);
    }

    static public void SetBalanceInquiry(ITransaction trans) {customBalanceInquiry = trans;}
}
Automating mocking and stubbing

• In the examples so far, we have coded the Keypad test stub and the Screen test mock by hand
  • Nice exercise to understand the underlying principles, but this is too time consuming, limiting and error prone as a general approach
  • We need an isolation framework that can do the job for us!

• Many isolation frameworks exist
  • RhinoMocks, Nmock, TypeMocks, Jmocks, EasyMock, Mockito, …
  • They have in common that they can create various kinds of fake test objects (Mocks, Stubs, Dummy objects) based on existing class or interface definitions.
  • Many of these tools use a Record-and-Replay metaphor as a means to tell the test
    • what stubs should do when/if they are called
    • what mocks should expect to receive as method calls, and to verify that the expected methods were indeed called
Setup of test fixtures with Rhino Mocks

```java
public class ATMRunnerTest {
    ATMRunner theATM;
    MockRepository fMock;
    IKeypad myKeypad;
    IScreen myScreen;
    ICashDispenser myCashDispenser;
    IDepositSlot myDepositSlot;
    IBankDatabase myBankDatabase;
    ITransaction myWithdrawal;
    ITransaction myDeposit;
    ITransaction myBalanceInquiry;

    [SetUp]
    public void Setup() {
        fMock = new MockRepository();
        myKeypad = fMock.StrictMock<Keypad>(null);
        myBankDatabase = fMock.StrictMock<IBankDatabase>(null);
        myScreen = fMock.StrictMock<IScreen>(null);
        myBalanceInquiry = fMock.StrictMock<ITransaction>(null);
        myDeposit = fMock.StrictMock<ITransaction>(null);
        myWithdrawal = fMock.StrictMock<ITransaction>(null);
        myCashDispenser = fMock.DynamicMock<ICashDispenser>();
        myDepositSlot = fMock.DynamicMock<IDepositSlot>();

        ATMObjectFactory.SetKeypad(myKeypad);
        ATMObjectFactory.SetScreen(myScreen);
        ATMObjectFactory.SetBankDatabase(myBankDatabase);
        ATMObjectFactory.SetWithdrawal(myWithdrawal);
        ATMObjectFactory.SetDeposit(myDeposit);
        ATMObjectFactory.SetBalanceInquiry(myBalanceInquiry);
        ATMObjectFactory.SetCashDispenser(myCashDispenser);
        ATMObjectFactory.SetDepositSlot(myDepositSlot);

        theATM = new ATMRunner();
    }

    Initiate the tests
    Create fake objects (for ALL collaborators)
    Create dummy objects
    Tell Object Factory to use them
```
Black-box model of our tests with CTE-XL
A NUnit test with Rhino Mocks

```csharp
/* CTE-XL testcase design
   firstauthentication  secondauthentication  firstchoice  secondchoice  thirdchoice
Testcase 1  invalidaccount  validaccountvalidpin  balanceinquiry  invalidchoice  exit
Testcase 2  validaccountvalidpin  invalidchoice  deposit  exit
Testcase 3  invalidpin  validaccountvalidpin  withdraw  balanceinquiry  exit
Testcase 4  invalidaccount  validaccountvalidpin  deposit  withdraw  exit
Testcase 5  invalidaccount  validaccountvalidpin  exit  */

[Test]
public void RunOnce_TestCase_1_invalidaccount_validaccountvalidpin_balanceinquiry_invalidchoice_exit()
{
    using (fMock.Record())
    {
        myScreen.Welcome();
        myScreen.AccountNumber();
        myKeypad.GetInput(); LastCall.Return(777); //invalid account
        myScreen.PIN();
        myKeypad.GetInput(); LastCall.Return(111);
        myBankDatabase.AuthenticateUser(777, 111); LastCall.Return(false);
        myScreen.InvalidAccountOrPin();
        myScreen.Welcome();
        myScreen.AccountNumber();
        myKeypad.GetInput(); LastCall.Return(12345); //valid account
        myScreen.PIN();
        myKeypad.GetInput(); LastCall.Return(54321);
        myBankDatabase.AuthenticateUser(12345, 54321); LastCall.Return(true);
        myScreen.MainMenu(); //check that mainmenu is displayed
        myKeypad.GetInput(); LastCall.Return(1); //user chooses balance inquiry
        myBalanceInquiry.Execute(); //check that the balance inquiry transaction was executed
        myScreen.MainMenu(); //check that mainmenu is displayed
        myKeypad.GetInput(); LastCall.Return(5); //user chooses an invalid choice
        myScreen.InvalidSelection(); //check that user receives error msg
        myScreen.MainMenu(); //Check that main menu is displayed
        myKeypad.GetInput(); LastCall.Return(4); //user chooses to exit
        myScreen.Goodbye();
    }
    theATM.RunOnce();
    fMock.VerifyAll();
}
```
Ncover after test of ATMRunner
Black box model of Deposit class
Setup of test fixture for Deposit

[TestFixture]
class DepositTest
{
    IKeypad myKeypad;
    IScreen myScreen;
    IBankDatabase myBankDatabase;
    IDEpositSlot myDepositSlot;

    Deposit theDeposit;
    MockRepository fMock;

    [SetUp]
    public void Setup()
    {
        fMock = new MockRepository();
        myScreen = fMock.StrictMock<IScreen>(null);
        myKeypad = fMock.Stub<IKeypad>(null);
        myBankDatabase = fMock.StrictMock<IBankDatabase>(null);
        myDepositSlot = fMock.Stub<IDEpositSlot>(null);

        theDeposit = new Deposit(12345, myScreen, myBankDatabase, myKeypad, myDepositSlot);
    }
}
Example interaction based test for Deposit

```java
/*
   Amount      Envelope type
  Testcase 1  zero      Inserted
  Testcase 2  positive  Inserted
  Testcase 3  negative
  Testcase 4  positive  did not insert it
*/

[Test]
public void Execute_Testcase_1_zero()
{
    using (fMock.Record())
    {
        myScreen.AskDepositAmount();
        myKeypad.GetInput(); LastCall.Return(0);
        myScreen.CancelingTransaction();
    }
    theDeposit.Execute();
    fMock.VerifyAll();
}

[Test]
public void Execute_Testcase_2_positive_inserted()
{
    using (fMock.Record())
    {
        myScreen.AskDepositAmount();
        myKeypad.GetInput(); LastCall.Return(50);
        myScreen.RequestDeposit(50);
        myDepositSlot.IsDepositEnvelopeReceived(); LastCall.Return(true);
        myScreen.ConfirmDeposit();
        myBankDatabase.Credit(12345, 50);
    }
    theDeposit.Execute();
    fMock.VerifyAll();
}
```
Ncover after test of ATMRunner and Deposit

Coverage Data  Data View Settings  Help

Explorer - Symbol Coverage

Trends & Statistics - ATM, ATMRunner

- Symbol Coverage: 100.00% (31 of 31)
- Branch Coverage: 100.00% (14 of 14)
- Method Coverage: 100.00% (6 of 6)

Max Cyclo. Comp.: 3
Num. Methods: 6

CashDispenser.cs  DepositTest.cs  ATMRunner.cs

1 // ATM.cs
2 // Represents an automated teller machine.
3 using System;
4 namespace ATM
5 {
6   public class ATMRunner
7   {

```csharp
```
Traditional unit testing: state verification

- So far we have seen examples of unit tests that compare expected behavior with actual behavior, as reflected by method calls to collaborating classes
  - This is known as behavior-based, or interaction-based unit testing
  - Has become very popular in the TDD/agile test community because you can fully isolate tests to only one class at the time by means of mocks and stubs
  - However, critics believe that the tests are too close to the implementation, and that as a result, the tests are too fragile
    - Small changes in the code may break the test
    - On the other hand, the oracle is very strong
- The traditionalist approach: state verification
  - Will often require that a test queries collaborating objects for changes in state as a result of running the class under test
  - Consequently, the tests become small integration tests
State-based test fixture for Deposit

[TestFixture]
class DepositTestState
{
  IKeypad myKeypad;
  IScreen myScreen;
  IBankDatabase myBankDatabase;
  IDEpositSlot myDepositSlot;

  Deposit theDeposit;
  MockRepository fMock;

  [SetUp]
  public void Setup()
  {
    fMock = new MockRepository();
    myScreen = fMock.DynamicMock<IScreen>(null);
    myKeypad = fMock.DynamicMock<IKeypad>(null);
    myDepositSlot = fMock.DynamicMock<IDEpositSlot>(null);
    // in this case we use the real object, which contains state that may change as
    // a result of performing a deposit transaction
    myBankDatabase = new BankDatabase();
    theDeposit = new Deposit(12345, myScreen, myBankDatabase, myKeypad, myDepositSlot);
  }
}

Now myScreen is a Dummy object

Uses the real database, which contains the state information
Example state-based test for Deposit

```csharp
/*
   Testcase | Amount   | Envelope type |
   -------  | -------- |-------------- |
   1        | zero     |              |
   2        | positive | inserted     |
   3        | negative |              |
   4        | positive | did not insert it |
*/

[Test]
public void Execute_Testcase_1_zero()
{
    using (fMock.Record())
    {
        myKeypad.GetInput(); LastCall.Return(0);
        // this method should NOT be called, but if it is, it would have returned true by default:
        myDepositSlot.IsDepositEnvelopeReceived(); LastCall.Return(true);
    }
    theDeposit.Execute();
    Assert.AreEqual(1000, myBankDatabase.GetAvailableBalance(12345));
    Assert.AreEqual(1200, myBankDatabase.GetTotalBalance(12345));
}

[Test]
public void Execute_Testcase_2_positive_inserted()
{
    using (fMock.Record())
    {
        myKeypad.GetInput(); LastCall.Return(50);
        myDepositSlot.IsDepositEnvelopeReceived(); LastCall.Return(true);
    }
    theDeposit.Execute();
    Assert.AreEqual(1000, myBankDatabase.GetAvailableBalance(12345));
    Assert.AreEqual(1250, myBankDatabase.GetTotalBalance(12345));
}
```

Verify state changes in Collaborating object, not behavior

Just controls the class, no Mocking
Coverage with state verification
And we found a fault in Deposit!

ATM.Test.DepositTest.Execute_TestCase_3_negative:
Rhino.Mocks.Expectations.ExpectationViolationException : IScreen.RequestDeposit(-100); Expected 0, Actual 1.

ATM.Test.DepositTestState.Execute_TestCase_3_negative:
   Expected: 1200m
   But was:  1100m
Recomended reading, reflecting the «state of practice»


• Gerard Meszaros: *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, ISBN 0131495054

• [http://martinfowler.com/articles/mocksArentStubs.html](http://martinfowler.com/articles/mocksArentStubs.html)
Thank you!