ABL Unit Testing Part 1: Test Strategy

Mike Fechner
Director
Consultingwerk Software Services Ltd.

- Independent IT consulting organization
- Focusing on OpenEdge and related technology
- Located in Cologne, Germany, subsidiaries in UK and Romania
- Customers in Europe, North America, Australia and South Africa
- Vendor of developer tools and consulting services
- Specialized in GUI for .NET, Angular, OO, Software Architecture, Application Integration
- Experts in OpenEdge Application Modernization
Mike Fechner

- Director, Lead Modernization Architect and Product Manager of the SmartComponent Library and WinKit
- Specialized on object oriented design, software architecture, desktop user interfaces and web technologies
- 28 years of Progress experience (V5 … OE11)
- Active member of the OpenEdge community
- Frequent speaker at OpenEdge related conferences around the world
Agenda

- **Introduction**
- A simple ABL Unit Test
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
Introduction

- Developer of **SmartComponent Library** Framework for OpenEdge Developers
- Source code shipped to clients, 99% ABL code
- Used by over 25 customers
- Up to weekly releases (customers usually during development on a release not older than 3 month)
- Fully automated update of the framework DB at client
- Almost no regression bugs within last 10 years
- Can only keep up that pace due to automation
From a recent real world example

- Windows 10 Creators Upgrade (April 2017) breaks INPUT THROUGH statements from Progress 8.3 - OpenEdge 11.7
- Used in a method to verify email addresses (MX record lookup), manual test of that functionality not likely
- Jenkins Job alerted us around noon after the Windows update was applied to the build server
- Only two days later, discussions around the issue on StackOverflow, Progress Communities and later in PANS

Unit Tests saved the day! As we had a fix in place already!
Build #182 (18.04.2017 13:11:22)

Build-Artefakte

Summary Of Changes - View Detail

Gestartet durch vorgelagertes Projekt "SmartComponent Library 117-64", Build 229
ursprünglich ausgelöst durch:

- Build wurde durch eine SCM-Änderung ausgelöst.

Testergebnis (2 fehlschlagene Tests / +2)

- Consultingwerk.UtilTest.NetworkHelpEmailValidationTest_TestAddresNonValidMX
- Consultingwerk.OeraTests.ValidateTests, ValidateTests, TestValidateEmailInvalidMX
IF OSYS BEGINS "WIN";U THEN DO:

cFilename = SUBSTITUTE ("%1-nslookup.exe";U, SESSION;TEMP-DIRECTORY).


OUTPUT TO VALUE (cFilename).

PUT UNFORMATTED "set q-max":U SKIP .
PUT UNFORMATTED pcDomain SKIP .

INPUT THROUGH VALUE (SUBSTITUTE ("type 41 | nslookup":U, QUOTER (cFilename))

importLoop:
REPEAT ON ERROR UNDO, THROW:

IMPORT UNFORMATTED cOutput .


IF INDEX (cOutput, "***":U) > 0 THEN DO:

IF NUM-ENTRIES (cOutput, "":U) >= 2 THEN DO:

cErrorMessage = TRIM (ENTRY (2, cOutput, "":U)) + " (";U).

ELSE

   cErrorMessage = "Unknown Error occurred for Domain: ";U.

END.

END.
From a recent real world example

- A pretty simple API got broken; caused by a Windows update
- No matter if it’s Progress’ fault or Microsoft – it’s a 3rd party
- We execute our Unit Tests on OpenEdge 10.2B, 11.3, 11.6 and 11.7
- We execute our Unit Tests on Windows 10 and Linux (VMware)
- Considering to add additional Windows Versions in VM’s because of the Easter 2017 experience
Introduction

- “In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.”, Wikipedia
- A Unit should be considered the smallest testable component
- Unit Tests may be automated
- Automated Unit Tests simplify regression testing
- Write test once, execute for a life time
The test pyramid

- Symbolizes different kind of tests that can be used to automate testing a (layered) application
- Unit Tests are relatively simple (cheap) to program, there should be lots of them
- API/Service Tests are more complex to write
- UI Tests are the most expensive to write and may require humans to execute them, may require frequent changes as the application evolves
- [https://martinfowler.com/bliki/TestPyramid.html](https://martinfowler.com/bliki/TestPyramid.html)
A customer’s testing stack for a web application

- Technology in use JavaScript, PASOE, Web Handlers for REST, OERA
- REST API’s
  - SOAP UI ([https://www.soapui.org/](https://www.soapui.org/)), including load scripts
  - NUnit (.NET Unit Testing) as the test manager knows this well, and C# allows more complex test logic or sequences
- Backend Unit Test: ABLUnit and SmartUnit
- JavaScript Unit Testing: Soon to be adding JSUnit to the mix
Agenda

- Introduction
- **A simple ABL Unit Test**
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
METHOD PUBLIC SalesPriceInfo CalculateSalesPrice (piItemNum AS INTEGER,
   piQty AS INTEGER,
   piCustNum AS INTEGER,
   pdtDate AS DATE):

DEFINE VARIABLE oReturn AS SalesPriceInfo NO-UNDO .

{&_proparse_ prrolint-nowarn(findnoerror)}
FIND Item WHERE Item.Itemnum = piItemNum NO-LOCK. // error on not available
{&_proparse_ prrolint-nowarn(findnoerror)}
FIND Customer WHERE Customer.CustNum = piCustNum NO-LOCK . // error on not available

IF piQty <= 0 THEN
    UNDO, THROW NEW InvalidParameterValueException ("piQty":U,
        STRING (piQty),
        THIS-OBJECT:GetClass():TypeName) .

IF pdtDate = ? THEN
    pdtDate = TODAY .

oReturn = NEW SalesPriceInfo (Item.Price,|  
   Item.Price * piQty,
   Item.Price * (100 - Customer.Discount) / 100,
   Item.Price * (100 - Customer.Discount) / 100 * piQty) .

RETURN oReturn .

END METHOD.
CLASS Demo.UnitTesting.Simple.PriceCalculationServiceTest:

    @Test.
    METHOD PUBLIC VOID TestValidPrice1 ():

        DEFINE VARIABLE oService AS PriceCalculationService NO-UNDO .
        DEFINE VARIABLE oPrice AS SalesPriceInfo NO-UNDO .

        oService = NEW PriceCalculationService() .

        oPrice = oService:CalculateSalesPrice (1 /* itemnum */,
               10 /* qty */,
               1 /* custnum */,
               12/24/2018) .

        Assert:Equals(24, oPrice:UnitPrice) .
        Assert:Equals(240, oPrice:TotalPrice) .

        Assert:Equals(15.6, oPrice:DiscountedUnitPrice) .
        Assert:Equals(156, oPrice:DiscountedTotalPrice) .

    END METHOD .
Test for a specific exception to be thrown

@Test (expected="Consultingwerk.Exceptions.InvalidParameterValueException").
METHOD PUBLIC VOID TestInvalidQty ():

DEFINE VARIABLE oService AS PriceCalculationService NO-UNDO .

oSservice = NEW PriceCalculationService() .

oSservice:CalculateSalesPrice (1 /* itemnum */ ,
   0 /* qty */ ,
   1 /* cust num */ ,
   12/24/2018) .

END METHOD.
Expect a very specific error from a method

```java
@Test.
METHOD PUBLIC VOID TestInvalidItem ():

    DEFINE VARIABLE oService AS PriceCalculationService NO-UNDO .
    oService = NEW PriceCalculationService() .
    oService:CalculateSalesPrice (4711, 10, 1, 12/24/2018) .
    Assert:RaiseError("No error thrown on invalid item") .

    CATCH err AS Progress.Lang.SysError:

        IF err:GetMessageNum (1) <> 138 OR NOT err:GetMessage (1) MATCHES "* Item *" THEN
            UNDO, THROW err . /* re-throw */

    END CATCH.

END METHOD.
```

** Item record not on file. (138)**
Demo

- Execute Unit Test in ABLUnit
- ABL Unit Launch Configuration in PDSOE
- ABLUnit View / Perspective
- Executing a single Unit Test Method
Agenda

- Introduction
- A simple ABL Unit Test
- **Structure of a Unit Test**
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
Structure of a Unit Test

- (ABL) Unit Tests may be developed in procedures and in classes
- A Unit Test is a method or internal procedure which executes a piece of code and asserts the result of that piece of code
- Unit Tests may be included in the compilation unit which is tested
- Unit Tests may be placed in separate class or procedure files to keep them separated from deployed code (my preference)
- Unit Test classes and methods or procedures may not have parameters
- Unit Test methods or procedures are annotated with @Test.
<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Test</td>
<td>Identifies that a method or a procedure is a test method or procedure.</td>
</tr>
<tr>
<td>@Setup</td>
<td>Executes the method or procedure before each test. This annotation prepares the test environment such as reading input data or initializing the class.</td>
</tr>
<tr>
<td>@TearDown</td>
<td>Executes the method or procedure after each test. This annotation cleans up the test environment such as deleting temporary data or restoring defaults.</td>
</tr>
<tr>
<td>@Before</td>
<td>Executes the method or procedure once per class, before the start of all tests. This annotation can be used to perform time-sensitive activities such as connecting to a database.</td>
</tr>
<tr>
<td>@After</td>
<td>Executes the method or procedure once, after all the tests are executed. This annotation is used to perform clean-up activities such as disconnecting from a database.</td>
</tr>
<tr>
<td>@Ignore</td>
<td>Ignores the test. You can use this annotation when you are still working on a code, the test case is not ready to run, or if the execution time of test is too long to be included.</td>
</tr>
<tr>
<td>@Test (expected=&quot;ExceptionType&quot;)</td>
<td>Fails the method if the method does not throw the exception mentioned in the expected attribute.</td>
</tr>
</tbody>
</table>
Initialization/cleanup annotations

- **@Before** and **@After** methods can be used to initialize and shut down framework components (or mocks of those) required to execute all unit test methods/procedures in test class/procedure.

- **@Setup** and **@TearDown** methods can be used to initialize and cleanup for every test method in a test class.
  - Ensure that every test has the same starting point, e.g. loading of data into temp-tables etc.
 Assert-Classes and methods

- Simple way to test a value received by the tested method
- STATIC methods
- A single method call that
  - Tests a value
  - THROWS’s an error when the value does not match the expected value
  - Fire and forget
Assert-Classes and Methods

- OpenEdge.Core.Assert
- Consultingwerk.Assertions.*
- Roll your own

Consultingwerk.Assertion.Assert:EqualsCaseSensitive (cReturnValue, “This is the expected value”).
CLASS Assert:

Member of OpenEdge.Core

Inherits Progress.Lang.Object

Summary:

ABL Syntax:
- USING
- USING OpenEdge.Core.Assert.
- DEFINE VARIABLE class1 AS CLASS Assert.
  class1 = NEW Assert().

...
/**
 * Purpose: Validates that two character values are equal based on a raw compare
 * Notes:
 * Throws: Consultingwerk.Assertion.AssertException
 * @param pcValue1 The first value to compare
 * @param pcValue2 The second value to compare
 */

METHOD PUBLIC STATIC VOID EqualsCaseSensitive (pcValue1 AS CHARACTER,
                                               pcValue2 AS CHARACTER):

    IF COMPARE (pcValue1, "NE":U, pcValue2, "CASE-SENSITIVE":U) THEN
        UNDO, THROW NEW AssertionError (SUBSTITUTE ("Value &1 does not equal &2",
                                                   QUOTER (pcValue1),
                                                   QUOTER (pcValue2)), 0).

END METHOD.
Agenda

- Introduction
- A simple ABL Unit Test
- Structure of a Unit Test
- **Unit Testing Tooling**
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
Unit Testing Tooling

- #1 tool supporting Unit Testing: Structured Error Handling
  - Unit Tests rely heavily on solid error handling
  - Unit Testing tool can’t trace errors not thrown far enough
- ABLUnit OpenEdge’s Unit Testing tool integrated into PDSOE
- Proprietary ABL Unit Testing tools
  - ProUnit
  - OEUnit
  - *SmartUnit (component of the SmartComponent Library)*
- All very similar but different in detail
JUnit legacy

- NUnit, JSUnit, ABLUnit, SmartUnit, …
- Most unit tests follow the JUnit conventions
- Usage of @Test. annotations (or similar)
- JUnit output file de facto standard
  - xml file capturing the result (success, error, messages, stack trace) of a single test or a test suite
  - Understood by a bunch of tools, including Jenkins CI
  - No formal definition though
JUnit output file

- results.xml produced by ABLUnit and similar tools
- Visualized by ABLUnit View
- Visualized / trended by Jenkins CI
- Visualized by ANT’s JUnit task (produces html output) or similar tools
ANT

- Apache Build Scripting Language
- XML based batch file, OS-independent
- ANT-File may contain multiple targets (sub routines)
- Sub routines may have dependencies to each other
- Macros
- Error-Handling & Conditional execution
- Properties/Variables/Parameters
ANT

- Originally a Java-Build System
- Compiles Java-Code, executes JUnit Tests, etc.
- Other built in features (among many others):
  - File manipulations, copy, delete, …
  - ZIP, UNZIP
  - SCM Interaction
- Extensible via plug-ins (offering further ANT Tasks)
ANT

- ANT supports Unit Test execution
- ABLUnit Task delivered by PSC
- ABLUnit Task in PCT
- PCTRun to execute your own unit tests
- ANT scripts may be executed as part of a build pipeline, nightly builds, after every source code commit
<target name="run_tests">
  <ABLUnit destDir="Demo/UnitTesting/Simple" dlcHome="${progress.DLC}"/>
  <fileset dir="Demo/UnitTesting/Simple" includes="**/*.cls"/>
  <propath>
    <pathelement path="."/>
    <pathelement path="../ABL"/>
  </propath>
  <DBConnection dbName="sports2000" dbDir="c:/Work/SmartComponents4NET/117_64/DB/sports2000" singleUser="true">
    <PCTAlias name="dictdb"/>
  </DBConnection>
</ABLUnit>

<exec executable="c:\Users\${env.USERNAME}\AppData\Roaming\npm\junit-viewer.cmd" dir="Demo/UnitTesting/Simple">
  <arg value="--results=."/>
  <arg value="--save=results.html"/>
</exec>

<exec executable="c:\Windows\System32\cmd.exe" dir="Demo/UnitTesting/Simple">
  <arg value="/c"/>
  <arg value="start"/>
  <arg value="results.html"/>
</exec>
</target>
PCT

- [https://github.com/Riverside-Software/pct](https://github.com/Riverside-Software/pct)

- ANT tasks for OpenEdge
- Progress Compiler Tools
- open-source
- „Support“ via Github Issue-Tracking
ABLUnit

Gilles QUERRET edited this page on 29 Jul 2016 · 5 revisions

Description

Run an ABLUnit tests sequence. For further information, refer to the progress documentation.

XML namespace

```xml
<pct:ABLUnit />
```

Parameters

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>destDir</td>
<td>Directory where to put result file. Don't use destDir under Linux, as a bug prevents results.xml from being generated</td>
<td>Base directory</td>
</tr>
<tr>
<td>writeLog</td>
<td>Creates <code>ablunit.log</code> in temp directory in case of error</td>
<td>False</td>
</tr>
<tr>
<td>haltOnFailure</td>
<td>Stop the build process if a test fails (errors are considered failures as well)</td>
<td>False</td>
</tr>
</tbody>
</table>

† Only one of those attributes is mandatory ‡ Mandatory attribute

ABLUnit inherits attributes from PCT and PCTRun.
Jenkins CI Server

- Continuous Integration – permanent merging of various changes
- Forked from Hudson CI
- Standard tool for centralized execution of build and test jobs
- Controlled environment for the execution of (Build or Test) „Jobs“
- Visualization of success or failure of jobs, visualization of Unit Test results
- Emails on failure or other events
Jenkins CI Server

- Executes ANT scripts (and other scripts)
- Imports JUnit result files
- Provides trending on stability of software project
- Can propagate build artefacts based on test results
Pipeline SCL2090

Vollständiger Projektname: 11.7/SmartComponent Lib

Stage View

Average stage times:
(Average full run time: ~37min 56s)
## Überblick

Average stage times:
(Average full run time: ~37min 56s)

<table>
<thead>
<tr>
<th>#25</th>
<th>Feb 20</th>
<th>1 commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#24</th>
<th>Feb 20</th>
<th>1 commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:07</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#23</th>
<th>Feb 20</th>
<th>1 commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#22</th>
<th>Feb 20</th>
<th>No Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>06:49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declarative: Checkout SCM</th>
<th>Info</th>
<th>Standard build</th>
<th>Unit Tests</th>
<th>:U Test</th>
<th>Parameter Comments Test</th>
<th>Localizable Test</th>
<th>Declarative: Post Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1min 19s</td>
<td>837ms</td>
<td>9min 5s</td>
<td>20min 50s</td>
<td>11s</td>
<td>3min 24s</td>
<td>4s</td>
</tr>
<tr>
<td></td>
<td>1min 32s</td>
<td>850ms</td>
<td>8min 38s</td>
<td>21min 27s</td>
<td>14s</td>
<td>4min 2s</td>
<td>6s</td>
</tr>
<tr>
<td></td>
<td>1min 33s</td>
<td>801ms</td>
<td>10min 6s</td>
<td>22min 8s</td>
<td>15s</td>
<td>5min 19s</td>
<td>7s</td>
</tr>
<tr>
<td></td>
<td>1min 1s</td>
<td>874ms</td>
<td>8min 26s</td>
<td>19min 20s</td>
<td>102ms</td>
<td>52ms</td>
<td>56ms</td>
</tr>
<tr>
<td></td>
<td>1min 10s</td>
<td>826ms</td>
<td>9min 12s</td>
<td>20min 25s</td>
<td>14s</td>
<td>4min 17s</td>
<td>5s</td>
</tr>
</tbody>
</table>

© 2018 Consultingwerk Ltd. All rights reserved.
Build #23 (20.02.2018 07:25:46)

Summary Of Changes - View Detail

45315 by Mike Fechner (Consultingwerk42_Stream) on 20.02.2018 07:23:28

Executing a single unit test

Branch indexing

Testergebnis (4 fehlgeschlagene Tests / +4)

Consultingwerk.SmartFrameworkTests.Zalmoxis.SmartTableTest.FetchSmartTable
Consultingwerk.SmartFrameworkTests.Zalmoxis.SmartTableTest.UpdateSmartTable
Consultingwerk.SmartFrameworkTests.Zalmoxis.SmartTableTest.UpdateSmartTable2
Regression


Fehlermeldung


Stacktrace

Consultingwerk/SmartFramework/Zalimoxis/getSmartKeyFieldAssignmentType.p at line 667 (E:\Jenkins\workspace\0-XICqWNFQ5DKUCA3NBRINCR5TPFNPWQFCDIKA4US3FPQ4LISU42XQ\ABL\Consultingwerk\SmartFramework\Zalimoxis\getSmartKeyFieldAssignmentType.r)
TestFetch Consultingwerk.SmartFrameworkTests.Zalimoxis.KeyFieldAssignmentTest at line 119 (E:\Jenkins\workspace\0-XICqWNFQ5DKUCA3NBRINCR5TPFNPWQFCDIKA4US3FPQ4LISU42XQ\UnitTests\Consultingwerk\SmartFrameworkTests\Zalimoxis\KeyFieldAssignmentTest.cls)
ExecuteTest Consultingwerk.SmartUnit.TestRunner.TestRunner at line 1124 (E:\Jenkins\workspace\0-XICqWNFQ5DKUCA3NBRINCR5TPFNPWQFCDIKA4US3FPQ4LISU42XQ\ABL\Consultingwerk\SmartUnit\TestRunner\TestRunner.r)
Execute Consultingwerk.SmartUnit.TestRunner.TestRunner at line 947 (E:\Jenkins\workspace\0-XICqWNFQ5DKUCA3NBRINCR5TPFNPWQFCDIKA4US3FPQ4LISU42XQ\AEl Consultingwerk\SmartUnit\runtest.p at line 520 (E:\Jenkins\workspace\0-XICqWNFQ5DKUCA3NBRINCR5TPFNPWQFCDIKA4US3FPQ4LISU42XQ\ABL\Consultingwerk\SmartUnit\runtest.p at line 71 (C:\Users\build\AppData\Local\Temp\pctinit1758.p at line 71 (C:\Users\build\AppData\Local\Temp\pctinit1758.p)
Measuring your Unit Test Coverage

- Unit Test Coverage: % of lines of code which are executed during unit tests
- There are only two kinds of people that know there Unit Test Coverage:
  - Those that don’t use Unit Tests at all
  - Those that measure Unit Test Coverage using SonarSource
SonarSource: Code Quality measuring
SonarQube by SonarSource

- Commonly used Lint tool
- Support for various programming languages via plug-ins
- Java, JavaScript, C#, HTML, XML, CSS, …
- OpenEdge Plugin developed by Riverside Software (Gilles Querret)
  - engine open source
  - rules commercial
- Available since 2016, permanently new features added
SonarQube by SonarSource

- Locates problems or potential bugs
- Violation of coding-standards
- Code duplication
- **Unit-Test coverage**

- Web-Dashboard
- CLI Utility (HTML or XML Reports)
- Eclipse Integration
Demo

- Sonar Lint Plugin into Progress Developer Studio
oRenderer:RenderInstances (oFields,
        phAttributes:DATASET,
        hInstanceBuffer::ContainerObjectMasterGuid,
        phAttributes::_ObjectInstanceGuid,
        oDescriptor,
        hDataset,
        cTables).

ELSE

oRenderer:RenderInstances (oFields,
        phAttributes:DATASET,
        hInstanceBuffer::ContainerObjectMasterGuid,
        phAttributes::_ObjectInstanceGuid,
        oDescriptor,
        phDataset,
        pcTables).

RETURN oGroupBox.

FINALLY:

GarbageCollectorHelper:DeleteObject(hInstanceBuffer).

END FINALLY.

END METHOD.
Agenda

- Introduction
- A simple ABL Unit Test
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
Object oriented or procedural?

- Procedures can be unit tested
- In fact, ABLUnit supports the execution of test-procedures as well
- OO-thinking however simplifies writing testable code
- Procedural code has tendency to be monolithic
- “Mocking” of dependencies requires patterns such as factories or dependency injection
  - In theory possible with procedures
  - More natural in object oriented programming
Writing testable code

- A huge financial report or invoice generation is barely testable in whole
- Large
- May call sub routines
- If it fails, what has been causing this?
  - A bug in code
  - False assumptions
  - Wrong data in DB?
- Output: A PDF file, how to assert this?
Writing testable code

- Break up financial report into a bunch of smaller components
- Test individual components
- Test report as a whole
- This allows to narrow down source of reported errors
- Separate report logic from output logic
  - Financial report should return temp-tables first
    - This can be tested
  - A separate module produces PDF output based on temp-table data
    - Testing difficult
Errors must be THROWN

- BLOCK-LEVEL ON ERROR UNDO, THROW almost mandatory
- Alternative Form of solid error handling
- Unit Testing tools don’t capture ** Customer record not on file (138) when written to stdout or a message box
Testing PROTECTED members

- When unit test is in a separate class, it only has access to PUBLIC methods of the class to be tested
- Making internal methods PUBLIC for the purpose of testing is the wrong approach!
- Solution:
  - Unit Test class can inherit from class to be tested to access PROTECTED
  - (some) Unit Test methods may be placed inside the class to be tested to access PRIVATE members
  - A combination
Agenda

- Introduction
- A simple ABL Unit Test
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- Advanced Unit Testing Features
Mocking Dependencies

- Writing Unit Tests (for complex code) is a permanent fight against dependencies (and the bugs in them)
- If the PriceInfoService relies on the CustomerBusinessEntity, the ItemBusinessEntity, an InventoryService and the framework’s AuthorizationManager you’re always testing the integration of 5 components
- Who’s fault is it, when the test fails?
- How do we test extreme situations? Caused by unexpected data returned from one of the dependencies?
Mocking Dependencies - Wikipedia

- “In object-oriented programming, **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.”

- “In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.”
Mocking

- Requires abstraction of object construction
- PriceInfoService should not NEW CustomerBusinessEntity as this would disallow to mock this
- Rather rely on Dependency Injection or CCS Service Manager component (or similar) to provide CustomerBusinessEntity or a mock based on configuration
- Same technique applies to any other sort of dependent components
CCS Business Entity getData instead of FIND in DB

DEFINE VARIABLE oItemBusinessEntity AS ItemBusinessEntity NO-UNDO .

(GET-CLASS(IBusinessEntity),
ItemBusinessEntity) .

oItemBusinessEntity:getData (NEW GetDataRequest ("eItem",
    SUBSTITUTE ("ItemNum = &1", QUOTER (piItemNum)),
    OUTPUT DATASET dsItem) .

{&_proparse_prolint-nowarn(findnoerror))
FIND FIRST eItem NO-LOCK.
Agenda

- Introduction
- A simple ABL Unit Test
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- **Dealing with Data**
- Advanced Unit Testing Features
Dealing with Data

- We’re using ABL to develop database applications
- Application functionality highly dependent on data in a database
- That’s a resource that’s difficult to deal with …
Don’t use a shared database for Unit Tests

- Your tests may rely on stock data or price data in the database
- A different developer may modify those records for his tests
- This can break your test
Don’t reuse your own database

- Your test sequence will include tests that modify data
- Maybe there is even a test to remove the item record that some other test depends on
  - Suddenly after adding this new test, a different test fails as the database contents are no longer the same
Solutions to the database dependency

- Always restore a known database state from a backup
- Or rebuild a database for each test run from .d and .df
  - This may be easier when the database schema may change during a test sequence
- You may need to rebuild a database multiple times during a test sequence
- Produces lots of Disk I/O
- Disk I/O on one of the SSD’s of the build server if the bottleneck in our test environment (CPU and memory barely busy)
Transactions

- When used carefully database transactions can be a solution to test modifying or deleting records
  - Execute deletion of a record
  - Test that it’s really gone (CAN-FIND)
  - UNDO transaction in test-class
- May cause side-effects if the code to be tested relies on a specific transaction behavior influenced by the fact that there’s an outer transaction now
Mock the code that accesses the DB

- May follow OERA or CCS principles
- Data Access class should be the only code that ever access the database
- Not even the business entity should be able to know that the data access class is using data from an XML file instead
Agenda

- Introduction
- A simple ABL Unit Test
- Structure of a Unit Test
- Unit Testing Tooling
- Writing testable code
- Mocking dependencies
- Dealing with Data
- **Advanced Unit Testing Features**
Scenario driven Unit Tests

- Many Unit Tests are alike
- Testing read functionality of Business Entity a very repeating tasks
- Should test for runtime performance characteristics
  - Runtime (subject to system performance fluctuations)
  - Records accessed in database
- Should test for values (e.g. calculated values)
- Tests can be expressed as scenario instead of code
SmartUnit Feature

- Unit Test tool of the SmartComponent Library
- https://documentation.consultingwerkcloud.com/display/SCL/Scenario-based+Unit+Tests+for+Business+Entity+FetchData+%28read%29+operations
Markup Driven Assertions

- Read Operations
  - NumResults
  - CanFind (allows to find for Unique Key + Calculated Field value)
  - CanNotFind
  - MaxRuntime (may fail, when test server is busy)
  - MaxReads (in the database)

- Update Operations
  - Expected validation messages or similar output
Questions