# TDUnit

TDUnit ist ein Unit Test Framework für den Team Developer von Unify / Gupta / Centura. Die Implementierung orientiert sich am Open Source Project NUnit (www.nunit.org), das ein Unit Test Framework für .NET ist.

## Was sind Unit Tests

- Unit Test ist eine von mehreren Komponenten der Software-Qualitätssicherung.
- Unit Tests testen einzelne Funktionen des Produktionscodes möglichst isoliert.
- Unit Tests können jederzeit und auch in ferner Zukunft aufgerufen werden.
- Unit Tests können automatisiert aufgerufen und ausgewertet werden.
- Unit Tests sind schnell.

## Was ist ein Unit Test Framework

- Es unterstützt den Entwickler so gut wie irgend möglich beim Schreiben und Ausführen von Unit Testfunktionen.

## Warum sollte man Unit Tests schreiben

- Um jetzt sicherzustellen, dass eine bestimmte Funktion tut was sie soll.
- Um in Zukunft sicherzustellen, dass eine bestimmte Funktion immer noch tut was sie soll.
- Damit man bei der Pflege von komplexer Software Zeit spart.

# TDUnit anwenden

## Installation

TDUnit besteht aus den beiden Komponenten

- TDUnit.apl
- CDK.apl (+abhängige DLL), das mit dem Team Developer installiert wird (optional)

## Aufsetzen eines Unit Test

### Erstellen einer Unit Test Quelldatei

- Ein Unit Test bezieht sich immer auf eine zu testende Produktions-Quellcode-Datei, z.B. SampleApp.apt.
- Für den zugehörigen Unit Test wird eine neue Quellcode-Datei mit dem Postfix "Tests" erstellt, im Beispiel SampleAppTests.apt.
- In SampleAppTests.apt werden zwei Libraries per "File Include" eingebunden:
    1. TDUnit.apl
    2. Die zu testende Produktionsdatei – hier SampleApp.apt

### Erstellen einer Unit Test Klasse

- Für jeden zu testenden, organisatorisch oder logisch zusammenhängenden Funktionsblock wird eine funktionelle Klasse erstellt. Es gelten dabei die folgenden drei Regeln:
    1. Die Klasse muss von _UnitTestBase abgeleitet werden

2. Der Name der Klasse ist je nach Fall wie folgt zu wählen
    - "InternalFunctionTests"
    - Name der funktionellen Klasse + "Tests"
    - Name des Dialogs oder Fensters + "Tests"
    - "ExternalFunction"+Name der DLL+"Tests"
3. Es muss eine Methode
    - mit dem Name "GetContext",
    - dem Rückgabetyp "String"
    - und dem Action Block "Return SalContextCurrent()" erstellt werden.
    - (Das ist leider eine kleine Schwäche von TDUnit, evt. findet jemand eine elegantere Lösung.)

### Schreiben der Unit Testfunktionen

- In der Testklasse sollten für jede Produktiv-Funktion einige Testfunktionen geschrieben werden.
- Der Name einer Testfunktion setzt sich zusammen aus
  [Name der Produktiv-Funktion] + "_" + [Testbedingungen] + "_" + [Erwartetes Verhalten]
  beispielsweise " Str2Date_InvalidDate_ReturnsFalse"
- Die Testfunktion besteht aus drei Teilen
    1. Arrange (präpariere z.B. die Funktionsparameter für die zu testende Funktion
    2. Act (führe die Testfunktion aus)
    3. Assert (bestätige, ob die Erwartung erfüllt wurde)
- Assert – also das Bestätigen des erwarteten Verhaltens – muss mit einer der von TDUnit zur Verfügung gestellten Assert-Funktionen durchgeführt werden.

### Ausführen des Unit Tests

Der Unit-Test lässt sich mit F7 bzw. Menü Debug / Go durchführen. Es werden automatisch alle Testfunktionen durchlaufen und protokolliert.

Das Ergebnis wird im TDUnit Testrunner-Fenster angezeigt. Erfolgreiche Tests sind grün gekennzeichnet, nicht erfolgreiche rot.

Ggf. kann in einer Testfunktion auch ein Breakpunkt gesetzt werden, so dass man die Fehlerursache analysieren kann.

Der Entwickler ist verpflichtet, dass alle Tests "grün werden".

# Referenz TDUnit

## Assert Funktionen

Die Assertfunktionen prüfen ein bestimmtes Verhalten und protokollieren es im Unit Test Log automatisch mit. Sieermitteln selbständig den Klassen- und Funktionsnamen der Testfunktionen und schreiben diesen ebenfalls in das Protokoll. Damit entfällt in den meisten Fällen die Notwendigkeit, dass man als Entwickler einen extra Fehlermeldungstext schreiben muss.

| Funktion | Parameter | Beschreibung |
|---|---|---|
| AssertIsEmptyString | String: sIn1<br>String: sMsg | Prüft, ob sIn1 ein leerer String ist. Die Angabe von sMsg ist optional. Dort kann für den Fehlerfall eine Mitteilung mitgegeben werden. Ist nichts angegeben, wird ein Standardtext erstellt. |
| AssertAreEqualStrings | String: sIn1<br>String: sIn2<br>String: sMsg | Prüft Strings auf Gleichheit. sIn1 ist typischerweise das Resultat eines Funktionsaufrufs, sIn2 ist der Erwartungswert. Beispiel:<br>Call AssertAreEqualStrings( HelloWorldFkt(), 'Hello World', '' ) |
| AssertAreNotEqualStrings | String: sIn1<br>String: sIn2<br>String: sMsg | Prüft, ob Strings ungleich sind. |
| AssertIsEmptyDate | Date/Time: dtIn1<br>String: sMsg | Prüft dtIn2 auf DATETIME_Null |
| AssertAreEqualDates | Date/Time: dtIn1<br>Date/Time: dtIn2<br>String: sMsg | s.o. |
| AssertAreNotEqualDates | Date/Time: dtIn1<br>Date/Time: dtIn2<br>String: sMsg | s.o. |
| AssertIsNumberNull | Number: nIn1<br>String: sMsg | Prüft, ob nIn1 = NUMBER_Null ist |
| AssertAreEqualNumbers | Number: nIn1<br>Number: nIn2<br>String: sMsg | s.o. |
| AssertAreNotEqualNumbers | Number: nIn1<br>Number: nIn2<br>String: sMsg | s.o. |
| AssertIsHandleNull | Window Handle: Handle1<br>String: sMsg | Beim Parameter Handle1 muss es sich nicht zwangsläufig um ein Window Handle handeln. Auch File- oder Sql-Handles können übergeben werden. |
| AssertIsTrue | Boolean: bIsTrue<br>String: sMsg | s.o. |
| AssertIsFalse | Boolean: bIsFalse<br>String: sMsg | s.o. |
| AssertFail | String: sMsg | Logt in jedem Fall einen Fehler. Kann verwendet werden, wenn man zwar die Testfunktionen schon mal geplant hat, sie aber noch nicht ausprogrammiert sind. Oder aber in einem Testfunktionsteil der garantiert nicht aufgerufen werden sollte. Hier ist die Angabe von Nachricht in |

| | | |
|---|---|---|
| | | sMsg sehr sinnvoll und sollte auch wahrgenommen werden. |

## Referenz Funktionelle Klasse _UnitTestBase

Die Klasse _UnitTestBase bietet zwei überschreibbare Methoden an: SetUp() und TearDown()

### *SetUp*

Wird vor jedem einzelnen Testfunktions-Aufruf innerhalb einer Testklasse ausgeführt. Dort sollten Parameter initialisiert werden (arrange), die für alle Funktionen relevant sind.

### *TearDown*

Wird nach jedem einzelnen Testfunktions-Aufruf innerhalb der Testklasse ausgeführt. Ist dafür zuständig, die in SetUp() initialisierten Werte und Objekte wieder in den ursprünglichen Zustand zurückversetzt werden.

### *Klassenkommentierung__Ignore*

Wird die Testklasse in der Namenszeile mit "__Ignore" kommentiert, werden deren Testfunktionen vom Testrunner nicht ausgeführt. Stattdessen wird die Klasse im Log in lachsroter Farbe gekennzeichnet. Sie kann also nicht "vergessen" werden. "Ignore" darf nur ein temporärer Zustand sein. Irgendwann (spätestens zum nächsten Release) muss die Sache in Ordnung gebracht werden.

### *Klassenkommentierung __SlowTest*

Wird die Testklasse in der Namenszeile mit "__SlowTest" gekennzeichnet, wird sie vom Testrunner zunächst nicht ausgeführt und – wie bei __Ignore – lachsrot im Log gekennzeichnet. Zu verwenden ist dieses Attribut, wenn der ein oder andere Test etwas langsamer läuft, als der Entwickler Geduld hat und diese Tests gerade nicht interessieren. Im Testrunner können per CheckBox die langsameren Funktionen wieder zum Test freigegeben werden und per Button "Run Tests" insgesamt durchgeführt werden. Dieses Attribut kann dauerhaft gesetzt bleiben. Bei zukünftigen Nightly Tests wird das __SlowTest-Attribut ignoriert.

## TDUnit Testrunner

### *Klassenliste*

Die Klassenliste rechts oben zeigt alle Testklassen distinkt an. Der Klassenname ist grün, wenn alle zugehörigen Tests bestanden wurden. Er ist in rot gekennzeichnet, wenn auch nur ein Test fehlschlägt. Wurde er ausgelassen ( z.B. per __Ignore oder __SlowTest) wird er lachsrot gekennzeichnet.

Wird ein Klassenname per Doppelklick ausgewählt, wird das angezeigte Log auf genau diese Klasse eingeschränkt. Mithilfe der Checkbox "Classfiler On" kann der Filter wieder ebenfalls an- aber auch wieder ausgeschaltet werden.

### *Logliste*

Die Logliste zeigt für jede Testfunktion (bzw. für jeden ausgeführten Assert-Aufruf) einen Logeintrag an. Die Farbgebung ist identisch zu der in der Klassenliste.

Wählt man eine Zeile aus, sieht man in den Felder darunter den jeweiligen Logeintrag in kompletter Länge. Insbesondere bei den Messages sieht man dann die ggf. mehrzeiligen Kommentare.

### *Checkbox "Run Slow Tests"*

Wird diese Checkbox aktiviert, werden bei einem erneuten Testrun (Button "Run Tests") auch die mit __SlowTest gekennzeichneten Klassen ausgeführt.

### *Checkbox "Run Ignored Tests"*

Wird diese Checkbox aktiviert, werden bei einem erneuten Testrun (Button "Run Tests") auch die mit __Ignore gekennzeichneten Klassen ausgeführt.