Unit Tests in Visual Studio

Was sind Unit Tests

Eine “Unit” ist eine Einheit von Code und umfasst meist eine einzelne Methode einer Klasse. Diese einzelnen Units werden in Tests mit Testdaten gefüttert und alle Abhängigkeiten werden mit sogenannten Mock Objekten (Objekte, welche statische Testdaten zurückliefern) ersetzt.
So wird sichergestellt, dass wenn ein Unit Test fehlschlägt, es nicht an einer Abhängigkeit der Unit hängt (wie z.B. eine Datenbank welche falsche Daten zurückliefert), sondern, dass der getestete Code einen Fehler hat.

Aufbau eines Unit Tests

Ein Unit Test ist in drei Phasen aufgeteilt:

1. **Arrange**: Alle Testdaten werden festgelegt und benötigte Mock Objekte werden erzeugt.
2. **Act**: Der zu testende Code wird aufgerufen und die Testdaten werden übergeben.
3. **Assert**: Das Ergebnis des zu testenden Code wird mit dem erwarteten Ergebnis verglichen.

```
namespace BankTests
{
    [TestClass]
    0 references
    public class BankAccountTests
    {
        [TestMethod]
        | 0 references
        public void Debit_WithValidAmount_UpdatesBalance()
        {
            // Arrange
            double beginningBalance = 11.99;
            double debitAmount = 4.55;
            double expected = 7.44;
            BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

            // Act
            account.Debit(debitAmount);

            // Assert
            double actual = account.Balance;
            Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
        }
    }
}
```

Test Driven Development (TDD)

Test Driven Development beschreibt eine Vorgehensweise der Entwicklung, bei der zuerst die Tests entstehen und den Rahmen einer Funktionalität vorgeben. Danach wird der Code geschrieben, welcher dann den Test bestehen muss.
Es gibt viele verschiedene Vorgehensweisen für Test Driven Development, welche sich in den Details der Umsetzung unterscheiden. Sie teilen sich jedoch fast alle eine allgemeine Grundstruktur, welche sich in 3 Schritte unterteilen lässt:

1. Zuerst überlegt man, wie die zu implementierende Methode sich verhalten soll bei gewissen Eingaben. So plant man hier zum Beispiel schon, was eine zu erwartende Ausgabe wäre für eine gewisse Eingabe.
   Dieses geplante Verhalten wird in einem oder mehrerer Unit Tests festgehalten
2. Wenn der genaue Rahmen der Funktionalität mit Unit Tests beschrieben wurde, geht man zur Implementierung über.
   Hierbei sollte man sich zuerst nur darauf konzentrieren, die Tests zu bestehen und dabei so wenig Code wie möglich zu schreiben und auch nicht unbedingt zu optimieren. Nur das Bestehen der Tests steht im Vordergrund.
3. In der letzten Phase, wenn also alle Tests wieder grün sind, wird optimiert und refactored.

Wenn all diese Phasen durchlaufen sind, ist das Endergebnis eine neue Funktionalität, welche bereits während der Entwicklung getestet wurde und somit robuster ist.

Aufgaben

Teil 1: Person-Klasse

1.1 Anforderungen:

Erstelle eine C#-Klasse namens `Person`, die grundlegende Informationen über eine Person speichert und verschiedene Methoden bereitstellt, um mit diesen Informationen zu arbeiten. Die Klasse soll zwei Felder enthalten: `name` (ein `string` für den Namen der Person) und `age` (ein `int` für das Alter der Person).

- **Felder der Person-Klasse**:
  - `name` (string): Der Name der Person.
  - `age` (int): Das Alter der Person.

1.2 Methoden der Person-Klasse:

1. **IsAdult()**:
   - Gibt `true` zurück, wenn die Person 18 Jahre oder älter ist, andernfalls `false`.
2. **IsChild()**:
   - Gibt `true` zurück, wenn die Person unter 18 Jahre alt ist, andernfalls `false`.
3. **CountChildren(Person[] people)**:
   - Zählt und gibt die Anzahl der Kinder (Personen unter 18 Jahren) in einem Array von `Person`-Objekten zurück.
4. **CountAdults(Person[] people)**:
   - Zählt und gibt die Anzahl der Erwachsenen (Personen ab 18 Jahren) in einem Array von `Person`-Objekten zurück.

1.3 Unit-Tests für die Person-Klasse:

- **Test-Szenarien**:
  1. **TestIsAdult_Adult**: Überprüfe, ob `IsAdult()` für eine Person, die 25 Jahre alt ist, `true` zurückgibt.
  2. **TestIsAdult_Child**: Überprüfe, ob `IsAdult()` für eine Person, die 16 Jahre alt ist, `false` zurückgibt.
  3. **TestIsChild_Child**: Überprüfe, ob `IsChild()` für eine Person, die 16 Jahre alt ist, `true` zurückgibt.
  4. **TestIsChild_Adult**: Überprüfe, ob `IsChild()` für eine Person, die 25 Jahre alt ist, `false` zurückgibt.
  5. **TestCountChildren**: Überprüfe, ob `CountChildren()` korrekt die Anzahl der minderjährigen Personen in einem Array zurückgibt.
  6. **TestCountAdults**: Überprüfe, ob `CountAdults()` korrekt die Anzahl der volljährigen Personen in einem Array zurückgibt.

7. **TestCountChildren_NoChildren**: Überprüfe, ob `CountChildren()` korrekt `0` zurückgibt, wenn keine minderjährigen Personen vorhanden sind.
8. **TestCountAdults_NoAdults**: Überprüfe, ob `CountAdults()` korrekt `0` zurückgibt, wenn keine volljährigen Personen vorhanden sind.

1.4 Erweiterung - Berechnung des BMI (Body-Mass-Index):

- Füge der Person-Klasse zwei Methoden hinzu:
  1. **CalculateBMI()**:
     - Berechnet den Body-Mass-Index der Person basierend auf Gewicht und Körpergröße.
     - **Formel**: Gewicht / (Größe)²
  2. **GetBMIStatus()**:
     - Gibt den BMI-Status der Person zurück:
       - Übergewicht: BMI > 25
       - Normalgewicht: 18.5 ≤ BMI ≤ 25
       - Untergewicht: BMI < 18.5

1.5 Unit-Tests für BMI-Berechnung:

- **Test-Szenarien**:
  1. **TestCalculateBMI_Overweight**: Berechnung des BMI für eine übergewichtige Person.
  2. **TestGetBMIStatus_Overweight**: Überprüfung, ob der Status für Übergewicht korrekt zurückgegeben wird.
  3. **TestGetBMIStatus_NormalWeight**: Überprüfung des Status für eine Person mit Normalgewicht.
  4. **TestGetBMIStatus_Underweight**: Überprüfung des Status für eine Person mit Untergewicht.
  5. **TestGetBMIStatus_ExactNormalWeight**: Test für eine Person mit einem BMI von genau 25 (Grenzwert für Normalgewicht).
  6. **TestGetBMIStatus_ExactLowerNormalWeight**: Test für eine Person mit einem BMI von genau 18.5 (Grenzwert für Normalgewicht).

Wichtiger Hinweis:
Stelle sicher, dass du die Unit-Tests für die Methoden `IsAdult()`, `IsChild()`, `CountChildren()` und `CountAdults()` zuerst implementierst und überprüfst. Sobald du die Erweiterung für die BMI-Berechnung hinzufügst, müssen diese Tests eventuell aktualisiert werden, um sicherzustellen, dass alle Methoden in der Person-Klasse korrekt getestet werden.

Teil 2: Geometrische Formen

2.1 Anforderungen:

Erstelle ein neues C#-Projekt, das eine Hierarchie von geometrischen Formen abbildet. Jede Form muss ihre eigene Fläche und ihren Umfang berechnen können. Die folgenden geometrischen Formen werden implementiert:

- **Kreis**
- **Rechteck**
- **Dreieck**

2.2 Abstrakte Klasse `Form`:

- **Eigenschaften**:
    1. **Farbe** (string): Die Farbe der Form.
- **Konstruktor**:
    1. Übergibt die Farbe der Form.
- **Abstrakte Methoden**:
    1. **BerechneFlaeche()**: Berechnet die Fläche der Form.
    2. **BerechneUmfang()**: Berechnet den Umfang der Form.

2.3 Kreis-Klasse:

- **Eigenschaften**:
    1. **radius** (double): Der Radius des Kreises.
- **Konstruktor**:
    1. Übergibt den **Radius** und die **Farbe** des Kreises.
- **Methoden**:
    1. **BerechneFlaeche()**:
        - **Formel**: Fläche = π * r²
    2. **BerechneUmfang()**:
        - **Formel**: Umfang = 2 * π * r

2.4 Rechteck-Klasse:

- **Eigenschaften**:
    1. **laenge** (double), **breite** (double): Die Länge und Breite des Rechtecks.
- **Konstruktor**:
    1. Übergibt die **Länge**, die **Breite** und die **Farbe** des Rechtecks.
- **Methoden**:
    1. **BerechneFlaeche()**:
        - **Formel**: Fläche = Länge * Breite
    2. **BerechneUmfang()**:
        - **Formel**: Umfang = 2 * (Länge + Breite)

2.5 Dreieck-Klasse:

- **Eigenschaften**:

    1. **basis** (double), **hoehe** (double): Die Basis und Höhe des Dreiecks.
- **Konstruktor**:
    1. Übergibt die **Basis**, die **Höhe** und die **Farbe** des Dreiecks.
- **Methoden**:
    1. **BerechneFlaeche()**:
        - **Formel**: Fläche = 0.5 * Basis * Höhe
    2. **BerechneUmfang()**:
        - **Formel**: Umfang = Basis + 2 * Höhe (Diese Annahme gilt für ein rechtwinkliges Dreieck, bei dem die beiden anderen Seiten gleich der Höhe sind.)

2.6 Unit-Tests für Geometrische Formen:

Erstelle Unit-Tests, um sicherzustellen, dass die Berechnungen für jede Form korrekt sind. Die Tests sollen folgende Punkte abdecken:

1. **Kreis:**
    - Teste die Berechnung der Fläche: π × r².
    - Teste die Berechnung des Umfangs: 2 × π × r.
    - Teste, ob die Farbe korrekt gesetzt wird.
2. **Rechteck:**
    - Teste die Berechnung der Fläche: Länge × Breite.
    - Teste die Berechnung des Umfangs: 2 × (Länge + Breite).
    - Teste, ob die Farbe korrekt gesetzt wird.
3. **Dreieck:**
    - Teste die Berechnung der Fläche: 0.5 × Basis × Höhe.
    - Teste die Berechnung des Umfangs: Basis + 2 × Höhe.
    - Teste, ob die Farbe korrekt gesetzt wird.