Unit Testing

Contents

Unit Testing

(Naik & Tripathy, 2008) (Vance, 2014)

Unit testing refers to testing program units in isolation. It verifies the value added by the code being constructed. The goal is to find defects during implementation. It is performed by the developer of the unit.

Commonly Understood Program Units

- Procedures
- Functions
- Classes
- Methods

Properties of a Good Test

- Independent and isolated
- Test a single behavior
- Readable
- Reliable and repeatable
- Equal quality to production code
- Valuable

Reasons for Testing in Isolation

- Errors are associated with a specific unit so can be easily fixed
- Easier to execute many distinct and desirable paths

Scope of Testing

- Each line of code
- Each predicate in the unit
- Performs intended purpose
- Has no known errors

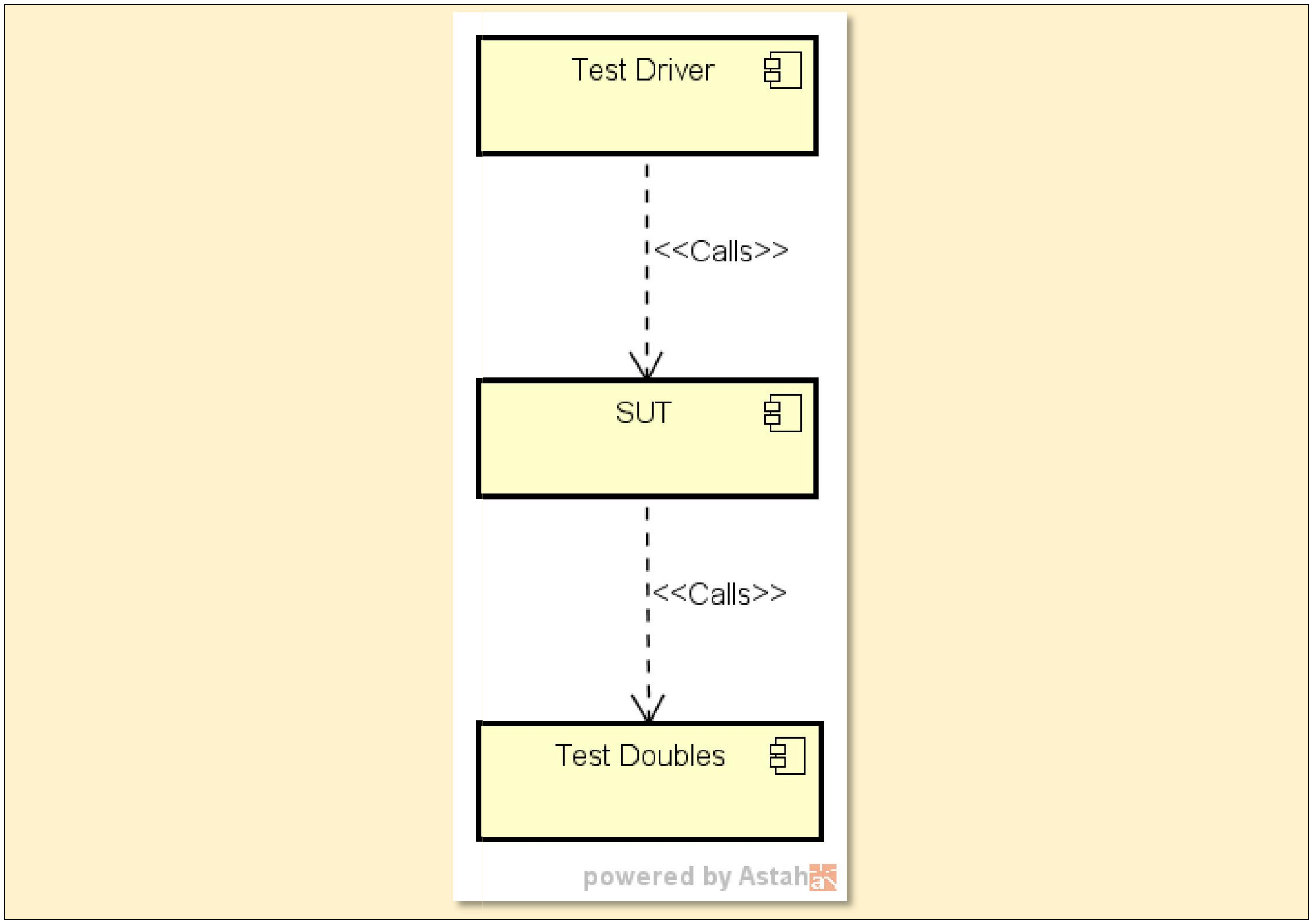
Phases

- Dynamic Unit Testing – Execution based testing, the unit is executed on a computer and its outputs and effects observed and verified
- Static Unit Testing – Non-execution based testing, the code is reviewed by someone

Dynamic Unit Testing

1. The SUT is taken out of its actual execution environment.
2. The actual execution environment is emulated and the SUT executed with a variety of selected inputs (test cases).

3. The outputs of the SUT is compared to precomputed values. If there is a difference between the actual and expected output then there is an error and there is a defect in the SUT.



Test Driver

The test driver invokes the SUT using the test cases. It compares the actual output with the expected output and reports the test result. I.e. whether or not the test passed. In other words, the test driver ***asserts*** that the actual output meets the expected output criterion.

Test Doubles (Fowler, 2016)

Test doubles replace dependent modules or systems of the SUT. Theymay provide evidence that theywere called and they may return precomputed values to the caller so that the SUT can continue its execution.

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).
- **Stubs**provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about

calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

- **Mocks** are objects pre-programmed with the expectation the calls they are expected to receive.

Scaffolding

The combination of the test drivers and the test doubles.

Dijkstra's Doctrine (Desikan & Ramesh, 2006)

Testing can only prove the presence of defects, never their absence. Exhaustive testing of a program is never possible; therefore, we need techniques to select effective test data.

Control Flow Testing

i. Draw a control flow graph from a unit
ii. Select a few control flow testing criteria
iii. Identify paths in the control flow graph to satisfy the selection criteria
iv. Derive path predicate expressions from the selected paths
v. Solve the path predicate expressions to generate values of the test input data

Data Flow Testing

i. Draw a data flow graph from a unit
ii. Select a few data flow testing criteria
iii. Identify paths in the data flow graph to satisfy the selection criteria
iv. Derive path predicate expressions from the selected paths
v. Solve the path predicate expression to generate values of the test input data

Domain Testing

Computation errors and domain errors are defined and then test data are selected to catch those defects. Computation errors are caused by a defect in an assignment. Domain errors are caused from a defect in the execution path. Input domains may be characterized by the following properties: closed boundary, open boundary, closed domain, open domain, extreme point, and adjacent domain. Three kinds of boundary errors are: closure error, shifted-boundary error, and tilted boundary error.

Functional Program Testing

i. Identify the input and output domains of a program
ii. For a given input domain, select some *special* values and compute the expected outcome
iii. For a given output domain, select some special values and compute the input values that will cause the unit to produce those output values
iv. Consider various combinations of the values chosen

Testing Techniques

- **Pairwise testing** – for a given number of input parameters each possible combination of values for any pair of parameters are covered by at least one test case
- **Equivalence class (EC) partitioning** – divide the input domain of the SUT into groups of test cases that have a similar effect on the system
- **Boundary value analysis**–the boundary conditions of each EC are analyzed in order to generate test cases.
- **Decision tables** – There exist a rule for each combination of conditions. Each rule of the decision table represents a test case.

Unit Testing using NUnit

Case Study: Tony Gaddis, Starting out with Java, 4th Edition, Chapter 9, Programming Challenge 9

Design an abstract class named ***`BankAccount`*** *to hold the following data for a bank account:*

- *Balance*
- *Number of deposits this month*
- *Number of withdrawals*
- *Annual interest rate*
- *Monthly service charges*

The class should have the following methods:

- *`Constructor`*
  - *The constructor should accept arguments for the balance and annual interest rate*
- *`Deposit`*
  - *A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits.*
- *`Withdraw`*
  - *A method that accepts an argument for the amount of the withdrawal. The method should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.*
- *`CalculateInterest`*
  - *A method that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:*
    - *Monthly Interest Rate = Annual Interest Rate / 12*
    - *Monthly Interest = Balance * Monthly Interest Rate*
    - *Balance = Balance + Monthly Interest*

- *`MonthlyProcess`*
    - *A method that subtracts the monthly service charges from the balance, calls the CalculateInterest method, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.*

*Next, design a **`SavingsAccount`** class that extends the BankAccount class. The **`SavingsAccount`** class should have a status field to represent an active or inactive account. If the balance of a savings account falls below $25, it becomes inactive. No more withdrawals can be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following methods:*

- *`Withdraw`*
    - *A method that determines whether the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the superclass version of the method.*
- *`Deposit`*
    - *A method that determines whether the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. The deposit is then made by calling the superclass version of the method.*
- *`MonthlyProcess`*
    - *Before the superclass method is called, this method checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the superclass field that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.)*

Structure of an NUnit Test

- The testing class for the SUT → Decorated with attribute `[TestFixture]`
- Methods to test the behavior of unit → Decorated with attribute `[Test]`
- E.g.

```
[TestFixture]
public class ABankAccount
{
      [Test]
      public void ShouldSetBalanceAndAnnualInterestRateWhenConstructed()
      {
      }
}
```

- `[Category(string)]` – Allows grouping tests
- `[ExpectedException( typeof( ArgumentException ), ExpectedMessage="expected message" )]`
- `[Ignore("Ignore a test")]`
- `[Setup]` - This attribute is used inside a `TestFixture` to provide a common set of functions that are performed just before each test method is called.

- `[SetUpFixture]` - This is the attribute that marks a class that contains the one-time setup or teardown methods for all the test fixtures under a given namespace.
- `[TearDown]` - This attribute is used inside a `TestFixture` to provide a common set of functions that are performed after each test method.
- `[TestCase(parameters)]` - serves the dual purpose of marking a method with parameters as a test method and providing inline data to be used when invoking that method. E.g. `[TestCase(12,3,4)]`

Naming Conventions

- The combination of the class and method name should form a sentence to explain the test. The methods may begin with 'Should.'

  E.g. `ABankAccount ShouldSetBalanceAndAnnualInterestRateWhenConstructed()`

- The unit under test is instantiated as ***sut*** (system under test)
- E.g.

```
[TestFixture]
public class ABankAccount
{
      [Test]
      public void ShouldSetBalanceAndAnnualInterestRateWhenConstructed()
      {
            var sut = new BankAccount(100.0m, 0.01);
      }
}
```

Assertions

- Are used to validate the behavior
- Use NUnit's**`Assert`** class
- E.g.

```
[TestFixture]
public class ABankAccount
{
      [Test]
      public void ShouldSetBalanceAndAnnualInterestRateWhenConstructed()
      {
            var sut = new BankAccount(100.0m, .01);
            Assert.That(sut.Balance, Is.EqualTo(100.0m));
            Assert.That(sut.AnnualInterestRate, Is.EqualTo(0.01));
      }
}
```

Constraint-Based Assert Model (Constraint-Based Assert Model (NUnit 2.4), 2015)

The constraint-based Assert model uses a single method of the `Assert` class for all assertions. The logic necessary to carry out each assertion is embedded in the constraint object passed as the second parameter to that method.

E.g. `Assert.That(sut.Balance, Is.EqualTo(100.0m));`

Equality Constraint

- tests that two objects are equal
- `Is.Equal.To(object)`
- `Is.Not.EqualTo(object)`
- `Is.EqualTo(object).Within(tolerance)`
- `Is.EqualTo(object).IgnoreCase`
- `Is.EqualTo(object).AsCollection`

Same as Constraint

- tests that two object references refer to the same object
- `Is.SameAs(reference)`
- `Is.Not.SameAs(reference)`

Condition Constraints

- `Is.Null`
- `Is.True`
- `Is.False`
- `Is.NaN` - tests for floating NaN
- `Is.Empty` - tests for empty string or collection
- `Is.Unique` - tests that collection contains unique items

Comparison Constraints

- Comparison constraints work on numeric values, as well as other objects that implement `IComparable`.
- `Is.GreaterThan(value)`
- `Is.GreaterThanOrEqualTo(value)`
- `Is.AtLeast(value)`
- `Is.LessThan(value)`
- `Is.LessThanOrEqualTo(value)`
- `Is.AtMost(value)`

Type Constraints

- `Is.TypeOf(type)`
- `Is.InstanceOfType(type)`
- `Is.AssignableFrom(type)`

String Constraints

- `Text.Contains(string)`
- `Text.DoesNotContain(string)`
- `Text.StartsWith(string)`
- `Text.DoesNotStartWith(string)`
- `Text.EndsWith(string)`
- `Text.DoesNotEndWith(string)`
- `Text.Matches(pattern)` - tests that a regex pattern is matched

- `Text.DoesNotMatch(pattern)` - tests that a regex pattern is matched

Collection Constraints

- `Is.All…`
- `Has.All…`
- `Has.Some…`
- `Has.None…`
- `Is.Unique`
- `Has.Member(object)`
- `Is.EquivalentTo(collection)`
- `Is.SubsetOf(collection)`

`E.g.`

```
int[] iarray = new int[] { 1, 2, 3 };
string[] sarray = new string[] { "a", "b", "c" };

Assert.That(iarray, Is.All.Not.Null);
Assert.That(iarray, Is.All.GreaterThan(0));

Assert.That(sarray, Is.Unique);

Assert.That(new string[] { "c", "a", "b" }, Is.EquivalentTo(sarray));
Assert.That(new int[] { 1, 2, 2 }, Is.Not.EquivalentTo(iarray));
Assert.That(new int[] { 1, 3 }, Is.SubsetOf(iarray));
```

Case Study: Creating the Project

1. Create a `BankAccount.Logic` C# class library project in Visual Studio and name the solution `BankAccountApp`
2. Create a `BankAccount.UnitTests` C# class library project in the `BankAccountApp` solution
3. Add a reference from `BankAccount.UnitTests` to `BankAccount.Logic`
4. Use NuGet to install ***NUnit*** (use version 2.6.4 because this is what the test adapter supports) and ***NUnit Test Adapter*** to `BankAccount.UnitTests`
5. In `BankAccount.Logic`, create the `BankAccount` class:

```
public abstract class BankAccount
{
   public decimal Balance { get; set; }
   public double AnnualInterestRate { get; set; }

   public BankAccount(decimal balance, double annualInterestRate)
   {
      this.Balance = balance;
      this.AnnualInterestRate = annualInterestRate;
   }
}
```

6. In `BankAccount.UnitTests`, create the testing class. The BankAccount class is abstract so it needs to be faked for testing.

```
[TestFixture]
public class ABankAccount
{
    public class FakeBankAccount : BankAccount
    {
        public FakeBankAccount(decimal balance, double annualInterestRate)
           : base(balance, annualInterestRate)
        {
        }
    }

    [Test]
    public void ShouldSetBalanceAndAnnualInterestRateWhenConstructed()
    {
        var sut = new FakeBankAccount(100.0m, .01);
        Assert.That(sut.Balance, Is.EqualTo(100.0m));
        Assert.That(sut.AnnualInterestRate, Is.EqualTo(0.01));
    }
}
```

7. Select Test > Windows > Test Explorer from the menu bar
8. Building the solution will discover the tests
9. Run the test from the test explorer

Post-conditions

A post-condition is a condition or predicate that must always be true just after the execution of some section of code or after an operation in a formal specification. We use post-conditions to formulate the assertions.

Recall the Deposit method:

Deposit

A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits.

Post-conditions of Deposit

```
Balance == Balance@Pre + depositAmount
NumberOfDeposits == NumberOfDeposits@Pre + 1
```

`Balance@Pre` refers to the balance at the precondition, in other words, the old balance. So `Balance == Balance@Pre + depositAmount` means the new balance is the old balance plus the deposit amount.

Since there are two post-conditions, two tests are needed:

```
[Test]
public void ShouldIncreaseTheBalanceAfterADeposit()
{
   var sut = new FakeBankAccount(0.0m, .01);
   var balanceAtPre = sut.Balance;
   sut.Deposit(100.0m);
   Assert.That(sut.Balance, Is.EqualTo(balanceAtPre + 100.0m));
}

[Test]
public void ShouldIncrementTheNumberOfDepositsByOneAfterADeposit()
{
   var sut = new FakeBankAccount(0.0m, .01);
   var numberOfDepositsAtPre = sut.NumberOfDeposits;
   sut.Deposit(100.0m);
   Assert.That(sut.NumberOfDeposits, Is.EqualTo(numberOfDepositsAtPre + 1));
}
```

Post-conditions of Withdraw

```
Balance == Balance@Pre -withdrawAmount
NumberOfWithdrawals == NumberOfWithdrawals@Pre + 1
```

```
[Test]
public void ShouldDecreaseTheBalanceAfterAWithdrawal()
{
   var sut = new FakeBankAccount(100.0m, .01);
   var balanceAtPre = sut.Balance;
   sut.Withdraw(10.0m);
   Assert.That(sut.Balance, Is.EqualTo(balanceAtPre - 10.0m));
}

[Test]
public void ShouldIncrementTheNumberOfWithdrawalsByOneAfterAWithdrawal()
{
   var sut = new FakeBankAccount(100.0m, .01);
   var numberOfWithdrawalsAtPre = sut.NumberOfWithdrawals;
   sut.Withdraw(10.0m);
   Assert.That(sut.NumberOfWithdrawals, Is.EqualTo(numberOfWithdrawalsAtPre + 1));
}
```

Post-conditions of CalculateInterest

`Balance == Balance@Pre +MonthlyInterest`

  Where: `MonthlyInterest = Balance@Pre * MonthlyInterestRate`

  Where: `MonthlyInterestRate = AnnualInterestRate / 12`

```
[Test]
public void ShouldIncreaseBalanceAfterCalculatingInterest()
{
   var sut = new FakeBankAccount(100.0m, .01);
   var balanceAtPre = sut.Balance;
   var monthlyInterest = balanceAtPre * (decimal)(.01 / 12);
   sut.CalculateInterest();
   Assert.That(sut.Balance, Is.EqualTo(balanceAtPre + monthlyInterest));
}
```

Post-conditions of MonthlyProcess

```
Balance == Balance@Pre – MonthlyServiceCharge@Pre +MonthlyInterest
      Where: MonthlyInterest = Balance@Pre * MonthlyInterestRate
      Where: MonthlyInterestRate = AnnualInterestRate / 12
NumberOfWithdrawals == 0
NumberOfDeposits == 0
MonthlyServiceCharge == 0
```

```
[Test]
public void ShouldUpdateBalanceAfterTheMonthlyProcess()
{
   var sut = new FakeBankAccount(100.0m, .01);
   sut.Deposit(50.0m);
   sut.Withdraw(50.0m);
   var balanceAtPre = sut.Balance;
   var monthlyServiceChargeAtPre = sut.MonthlyServiceCharge;
   var monthlyInterest = balanceAtPre * (decimal)(.01 / 12);
   sut.MonthlyProcess();
   Assert.That(sut.Balance, Is.EqualTo(balanceAtPre - monthlyServiceChargeAtPre + 
monthlyInterest));
}

[Test]
public void ShouldResetNumberOfWithdrawalsAfterTheMonthlyProcess()
{
   var sut = new FakeBankAccount(100.0m, .01);
   sut.Deposit(50.0m);
   sut.Withdraw(50.0m);
   sut.MonthlyProcess();
   Assert.That(sut.NumberOfWithdrawals, Is.EqualTo(0));
}

[Test]
public void ShouldResetNumberOfDepositsAfterTheMonthlyProcess()
{
   var sut = new FakeBankAccount(100.0m, .01);
   sut.Deposit(50.0m);
   sut.Withdraw(50.0m);
   sut.MonthlyProcess();
   Assert.That(sut.NumberOfDeposits, Is.EqualTo(0));
}

[Test]
public void ShouldResetMonthlyServiceChargeAfterTheMonthlyProcess()
{
   var sut = new FakeBankAccount(100.0m, .01);
   sut.Deposit(50.0m);
   sut.Withdraw(50.0m);
   sut.MonthlyProcess();
   Assert.That(sut.MonthlyServiceCharge, Is.EqualTo(0m));
}
```

The BankAccount class after this round of implementation:

```
public abstract class BankAccount
{
   public decimal Balance { get; set; }
   public double AnnualInterestRate { get; set; }
   public int NumberOfDeposits { get; private set; }
   public int NumberOfWithdrawals { get; private set; }
   public decimal MonthlyServiceCharge { get; private set; }

   public BankAccount(decimal balance, double annualInterestRate)
   {
      this.Balance = balance;
      this.AnnualInterestRate = annualInterestRate;
   }

   public void Deposit(decimal depositAmount)
   {
      Balance += depositAmount;
      NumberOfDeposits++;
   }

   public void Withdraw(decimal withdrawAmount)
   {
      Balance -= withdrawAmount;
      NumberOfWithdrawals++;
   }

   private void CalculateInterest()
   {
      Balance = Balance + (Balance * (decimal)(AnnualInterestRate / 12));
   }

   public void MonthlyProcess()
   {
      Balance -= MonthlyServiceCharge;
      CalculateInterest();
      NumberOfWithdrawals = 0;
      NumberOfDeposits = 0;
      MonthlyServiceCharge = 0m;
   }
}
```

SavingsAccount Class

```
public class SavingsAccount : BankAccount
{
    public enum AccountStatus { Active, Inactive };

    public AccountStatus Status { get; private set; }

    public SavingsAccount(decimal balance, double annualInterestRate)
        :base(balance, annualInterestRate)
    {
        Status = AccountStatus.Active;
        if (Balance <= 25.0m)
        {
            Status = AccountStatus.Inactive;
        }
    }
}
```

`public SavingsAccount(decimal balance, double annualInterestRate)`

Post:

1. `Balance == balance`
2. `AnnualInterestRate == annualInterestRate`
3. `Balance > 25 ➔ Status == SavingsAccount.AccountStatus.Active`
4. `Balance <= 25 ➔ Status == SavingsAccount.AccountStatus.Inactive`

`public Withdraw(decimal withdrawAmount)`

Post:

1. `Status == SavingsAccount.AccountStatus.Inactive ➔ Balance == Balance@Pre`
2. `Status == SavingsAccount.AccountStatus.Active ➔ Balance == Balance@Pre - withdrawAmount`

`public Deposit(decimal depositAmount)`

Post:

1. `Balance == Balance@Pre + depositAmount`
2. `Status == SavingsAccount.AccountStatus.Inactive AND Balance > 25➔ Status == SavingsAccount.AccountStatus.Active`
3. `Status == SavingsAccount.AccountStatus.Inactive AND Balance <= 25 ➔ Status == SavingsAccount.AccountStatus.Inactive`

```
public MonthlyProcess()
```

Post:

```
   1. Balance == Balance@Pre – MonthlyServiceCharge@Pre + MonthlyInterest
      Where: MonthlyInterest = Balance@Pre * MonthlyInterestRate
      Where: MonthlyInterestRate = AnnualInterestRate / 12
   2. NumberOfWithdrawals == 0
   3. NumberOfDeposits == 0
   4. MonthlyServiceCharge == 0
   5. Balance <= 25 ➔ Status == SavingsAccount.AccountStatus.Inactive
```

References

Constraint-Based Assert Model (NUnit 2.4). (2015). Retrieved from Nunit: http://www.nunit.org/index.php?p=constraintModel&r=2.4.8

Desikan, S., & Ramesh, G. (2006). *Software Testing Principles and Practices.* Pearson Education.

Fowler, M. (2016, February 1). *Mocks Aren't Stubs*. Retrieved from Martin Fowler: http://martinfowler.com/articles/mocksArentStubs.html

Naik, K., & Tripathy, P. (2008). *Software Testing and Quality Assurance Theory and Practice*. Wiley.

Vance, S. (2014). *Quality Code Software Testing Principles, Practices, and Patterns.* Pearson.