SOFTWARE MANTENIBLE

Diseño de software con pruebas unitarias

Con ejemplos en VisualBasic.net

Instructor: Oscar Centeno

Octubre – Noviembre, 2017

Primera Parte

Si cada caso de uso de un sistema de software es diseñado de una manera fácil de probar, este podrá modificarse, corregirse y mejorarse en el tiempo con menos riesgos de incidentes y retrasos. Lograrlo requiere que el equipo de desarrollo esté entrenado en especificación de requerimientos, el diseño de los algoritmos y la automatización de las pruebas.

En esta primera parte, presentamos la técnica de especificación con ejemplos y veremos cómo implementarlos con pruebas unitarias.

En la segunda parte, miraremos la especificación de casos de uso y el diseño de sus algoritmos, y allí integraremos las técnicas de esta primera parte.

Nota: La mayor parte de las recomendaciones en este curso se dan con base en la experiencia del autor, y siempre pueden mejorarse pues estas están enfocadas en facilitar la aplicación de las técnicas en un ambiente de desarrollo real.

Contenidos

1 Pruebas unitarias

Las pruebas unitarias son la clave para tener un software fácil de probar. Presentamos aquí una definición y sus dos características.

1.1 ¿Por qué queremos usarlas?

Cuando hacemos un cambio en una parte de un Sistema, tenemos que asegurarnos de que no estemos afectando negativamente otra parte, produciendo defectos. Tradicionalmente, asignamos un grupo de ingenieros que usan el Sistema manualmente a través de la interfaz gráfica y buscan algún defecto.

Esto tiene sus inconvenientes. El *testing* manual requiere más personas, más tiempo (por lo que es costoso y lento) y además no es repetible, porque para volver a probar el sistema hay que hacerlo otra vez de manera manual con el mismo costo y riesgo. El tiempo ya fue invertido y no se recupera.

Hay otra manera de trabajar y es con el *testing* automatizado. Lo que queremos es que la mayor parte de nuestro software pueda verificarse automáticamente. Para lograrlo, creamos otra pieza de software que nos dice si la funcionalidad opera correctamente. Cuando hagamos un cambio, entonces podemos ejecutar esa pieza de software y en pocos segundos saber si todo sigue funcionando correctamente.

Hay distintos tipos de pruebas automatizadas, donde las más rápidas y confiables son las pruebas unitarias. Este es un ejemplo:

```
<TestMethod()> Public Sub SeCalculaLaMetrica()
    Dim esperado As String = "95%"

    Dim iteracion As New Iteracion
    iteracion.PuntosPlanificados = 100
    iteracion.PuntosTerminados = 95
    iteracion.HayInformacionDePuntosDisponible = True
    Dim obtenido As String = Calculos.CalculeLaMetricaDePuntos(iteracion)

    Assert.AreEqual(esperado, obtenido)
End Sub
```

1.2 Definición de una prueba unitaria

Una prueba unitaria es un método que puede invocar al código que queremos probar y determina si el resultado obtenido es igual al esperado. Si es igual, entonces decimos que la prueba es exitosa, si no, falla.

Si estos métodos los podemos ejecutar con un solo *clic*, entonces podemos ejecutar miles de ellos en muy poco tiempo.

1.3 Dos características

Las pruebas unitarias tienen las siguientes características:

1. Funcionan en memoria, por lo que son muy rápidas. Cada una tarda pocos milisegundos en ejecutarse. No acceden a recursos externos como archivos, bases de datos o *webservices*. No requieren configuraciones manuales, por lo que funcionan con un solo clic.
2. Son repetibles. Para que podamos confiar en ellas, las pruebas deben ejecutarse siempre de la misma manera. Queremos poder ejecutarlas durante el día con cada cambio que realicemos, o en el futuro cuando un programador (diferente a quien creó el Sistema) haga otro cambio. Para lograrlo, una prueba unitaria nunca debe depender de algo que cambie en el tiempo, como por ejemplo, la fecha actual, ni de alguna función aleatoria. En el ejemplo anterior, el resultado esperado es un número ya dado, así que la prueba siempre ejecutará de la misma manera.

Otra manera de explicarlo es decir que una prueba unitaria es auto contenida y determinística. Es auto contenida puedes no tiene dependencias hacia recursos externos, y es determinística pues no tiene ningún elemento aleatorio.

1.4 Estructura de una prueba unitaria

Siendo que cada prueba unitaria es un método separado, miremos una estructura interna que vamos a recomendar durante este curso:

```
<TestMethod()> Public Sub SeCalculaLaMetrica()
    Dim esperado As String = "95%"

    Dim iteracion As New Iteracion
    iteracion.PuntosPlanificados = 100
    iteracion.PuntosTerminados = 95
    iteracion.HayInformacionDePuntosDisponible = True
    Dim obtenido As String = Calculos.CalculeLaMetricaDePuntos(iteracion)

    Assert.AreEqual(esperado, obtenido)
End Sub
```

La estructura la podemos denominar esperado-obtenido-comparación.

Esperado: El resultado del algoritmo que vamos a probar debe ser conocido de antemano, por lo que lo indicamos al inicio.
Obtenido: Luego, creamos los parámetros del algoritmo y lo invocamos para obtener un resultado.
Comparación: Realizamos una comparación para determinar si el comportamiento es el esperado. En esta comparación podríamos tratar con datos simples, con clases o con listas. Cada tipo de dato podría requerir una manera diferente para compararse. El uso usual para comparar tipos de datos sencillos es `Assert.AreEqual`.

Cada prueba unitaria es una función sin retorno (*Sub*), y debe tener el atributo *TestMethod* para que Visual Studio la pueda ejecutar.

1.5 Ejercicios

1. Código fuente: Descargue el código fuente de la dirección siguiente y abra la solución en Visual Studio. El proyecto contiene una serie de algoritmos y pruebas unitarias que estudiaremos durante el curso.

   https://goo.gl/kGVVKj

2. Práctica usando la herramienta:
   a. Mostrando la ventana del *Test Explorer* (*Test* + *Windows* + *Test Explorer*)
   b. Ejecutando las pruebas (*Run all*)
   c. Agrupando y filtrando (*Search filters*)
      i. Busque una prueba relacionada con "Garantia"
      ii. Ejecútela
      iii. ¿Cuánto tardó en ejecutar?
      iv. Modifique el código fuente de la fórmula en "CalculoDelAporteDeGarantia" y reejecute.
      v. Examine el mensaje de error
   d. Creando un proyecto de pruebas unitarias
   e. Creando una clase de pruebas
   f. Escribiendo una prueba unitaria (*TestMethod*)
   g. Inicialización (*TestInitialize*)
   h. Haciendo comparaciones (*Assert.AreEqual, ExpectedException*)

2 Especificación con ejemplos

Esta técnica se basa en la idea de que un requerimiento se entenderá mejor con un ejemplo concreto que con una descripción abstracta. Es decir, el primer objetivo es que los participantes de un proyecto comprendan el requerimiento y tengan claridad antes de implementarlo. El segundo objetivo es poder automatizar sus pruebas efectivamente.

2.1 *Gherkin*

Gherkin es el nombre de un lenguaje para explicar escenarios de datos que fue popularizado por la herramienta *Cucumber* y las ideas de *Behaviour Driven Development* (BDD) como técnica de desarrollo.

Sus palabras principales son tres:

- *Given*
- *When*
- *Then*

En español, utilizamos:

- Dado, Dada, Dados o Dadas
- Cuando
- Entonces

Con estas tres palabras clave podemos organizar la explicación de un algoritmo:

- Dado: Explica la información que existe antes de que la acción se dé.
- Cuando: Indica la acción que se ejecuta y la información que se da en ese momento.
- Entonces: Indica los resultados.

2.2 Un ejemplo sencillo

Iniciemos por un ejemplo sencillo, de un retiro de una cuenta de ahorros:

> Dada una cuenta con saldo 1000 colones
> Cuando se solicita un retiro de 998 colones
> Entonces el saldo es de 2 colones

Notemos que las palabras usadas son del negocio, y no hay detalles técnicos de almacenamiento de bases de datos o detalles de la interfaz de usuario. En este ejemplo, no explicamos de dónde se obtiene el saldo o cómo se almacena el nuevo saldo. Con esto, podemos centrar nuestra atención en los conceptos del negocio. Esto es importante pues hacerlo nos ayudará a conversar con un experto del negocio y verificar que estamos comprendiendo el requerimiento correctamente.

2.3 Escenarios

Cada combinación Dada/Cuando/Entonces se llama un "Escenario".
Así, el escenario anterior lo podemos expresar así:

Escenario: El saldo es afectado cuando se realiza un retiro de fondos.
El monto del retiro se debe restar al saldo para obtener el nuevo saldo de la cuenta.

Dada una cuenta con saldo 1000 colones
Cuando se solicita un retiro de 998 colones
Entonces el saldo es de 2 colones

Por facilidad, en este curso usaremos tablas *Excel* para comunicar los ejemplos *Gherkin*. Note que el título de un escenario puede ir acompañado de una explicación, sin embargo lo más importante no es documentar el ejemplo, sino conversar con un experto acerca del mismo para identificar áreas que no estemos comprendiendo correctamente.

2.4 Tablas de ejemplos

Tomando un ejemplo inicial, la conversación nos llevará a plantearnos otros casos de "¿qué pasaría si?". Entonces, dentro del mismo escenario, podemos explicar varios ejemplos de esta manera:

> Escenario: El saldo es afectado cuando se realiza un retiro de fondos.
>
> Dada una cuenta con saldo <saldo inicial>
> Cuando se solicita un retiro de <monto a retirar> colones
> Entonces el saldo es de <nuevo saldo> colones
>
> Ejemplos:

Ejemplo	Saldo inicial	Monto a retirar	Nuevo saldo
Se puede retirar fondos suficientes	1000	998	2
No se puede retirar más del saldo	1000	1001	1000
Se puede retirar el saldo completo	1000	1000	0
Se puede retirar decimales	1000	998.9	1.1

Cada nuevo ejemplo aclara situaciones límite (¿qué pasa si se trata de retirar más del saldo?) y tipos de datos (¿se puede retirar un monto con decimales?).

2.5 Recomendaciones

Al especificar un ejemplo, considere explicar un caso base inicial, luego, cree un ejemplo por cada situación especial, como por ejemplo "No se puede retirar más del saldo".

Además, siempre utilice las palabras del negocio, no palabras técnicas.

En los ejemplos en internet, usualmente encontrará que se especifica una interfaz gráfica con el lenguaje *Gherkin*. Sin embargo, en este curso, nos enfocaremos a ejemplos de lógica de negocio.

En los ejemplos, incluya solamente la información necesaria para el escenario. Esto ayudará a concentrarse en la esencial sin distraerse en detalles adicionales. En ejemplo anterior, la identificación del titular de la cuenta no es relevante, así que no lo incluimos.

2.6 Tablas de datos

En los ejemplos, podemos agregar tablas completas que implicarán que cada fila es un objeto, y toda la tabla es una lista. Por ejemplo:

Escenario: Cómo se genera la velocidad para una iteración
Se suman los puntos de las historias terminadas en dicha iteración

Dadas las historias siguientes

Historia terminada en iteración	Puntos
1	8
2	3
0	5
2	2
2	5
3	1
5	1

Cuando se genera la velocidad para la <iteración>
Entonces se obtiene la <velocidad>

Ejemplo	Iteración	Velocidad
No hay historias terminadas en la iteración	4	0
Hay una historia terminada en la iteración	1	8
Hay varias historias terminadas en la iteración	2	10

2.7 Especificando validaciones

Las especificaciones anteriores explican cómo se convierten o calculan datos, pero este formato es muy útil también para especificar validaciones, como por ejemplo en el siguiente:

Escenario: Valide las estadísticas de una iteración

Cuando se valida una iteración con <puntos planificados> y <puntos terminados>
Entonces se determina si <es válida o no>

Ejemplos:

Ejemplo	Puntos planificados	Puntos terminados	Es válida o no
Son validos	10	8	Es válida
Puntos planificados debe ser un número entero positivo	-1	8	Es inválida
Puntos planificados pueden ser cero	0	8	Es válida
Puntos planificados tiene un máximo de 100 puntos	101	8	Es inválida
Puntos planificados pueden ser el máximo permitido	10	8	Es válida
Puntos terminados debe ser un número entero positivo	10	-1	Es inválida
Puntos terminados pueden ser cero	10	0	Es válida
Puntos terminados tiene un máximo de 100 puntos	10	101	Es inválida
Puntos terminados pueden ser el máximo permitido	10	100	Es válida

Note que nos estamos enfocando en explicar los conceptos y no la implementación exacta de cómo se indica si los datos son válidos o no. La tabla funcionaría de la misma manera sea que decidamos comunicar los errores lanzando una excepción, retornando un mensaje de error, o un booleano.

Recomendaciones para especificar algoritmos de validación

Si el tipo de datos es un Texto, valide:

- Formatos
- Espacios en blanco
- Tamaño mínimo
- Tamaño máximo
- Relación con otros datos

Si el tipo de dato es un Número, valide:

- Decimales o enteros
- Mínimo
- Máximo
- Relación con otros datos

Si el tipo de dato es una Fecha, valide

- Formato
- Mínimo
- Máximo
- Relación con otros datos

2.8 Ejercicio: Interpretando tablas *Gherkin*

Para verificar que comprendemos la estructura de las tablas *Gherkin*, para cada ejemplo anterior, escriba la firma de una función que realice dicho algoritmo. Indique el nombre de cada función, sus parámetros y sus tipos, y su tipo de retorno.

2.9 Ejercicio: Escribiendo pruebas unitarias primero

Las pruebas unitarias nos ayudan en el diseño del código fuente, no sólo en las pruebas, y para sacar su máximo provecho aplicaremos la técnica llamada Desarrollo con Pruebas Primero (*"Test Driven Development"*).

Esta técnica nos dice que sigamos los siguientes pasos:

1. Falla: Escribimos la prueba unitaria antes del código del algoritmo. Ejecutamos la prueba y esta debe fallar.
2. Pasa: Escribimos el código fuente mínimo razonable para lograr que la prueba sea exitosa.
3. *Refactoring*: Eliminamos redundancia y mejoramos la legibilidad.

Para aprender dicha técnica, iniciaremos con una tabla *Gherkin* y la firma del algoritmo que vamos a programar. Así, podemos escribir la prueba unitaria aunque aún no hayamos programado el algoritmo. Usemos como base el escenario de los saldos:

Escenario: El saldo es afectado cuando se realiza un retiro de fondos.

Dada una cuenta con saldo <saldo inicial>
Cuando se solicita un retiro de <monto a retirar> colones
Entonces el saldo es de <nuevo saldo> colones

Ejemplos:

Ejemplo	Saldo inicial	Monto a retirar	Nuevo saldo
Se puede retirar fondos suficientes	1000	998	2
No se puede retirar más del saldo	1000	1001	1000
Se puede retirar el saldo completo	1000	1000	0
Se puede retirar decimales	1000	998.9	1.1

Escriba las cuatro pruebas unitarias con base en la especificación *Gherkin*.

Recomendaciones:

- Escriba las pruebas en papel para enfocarse en entender las tablas de *Gherkin*, y no en el Visual Studio.
- El nombre de la clase de pruebas unitarias: El nombre del método que probamos + "*Tests*"
- El nombre de cada prueba unitaria: El nombre del ejemplo.
- Recuerde usar la estructura esperado-obtenido-comparación.

2.10 Pasos para programar una especificación con ejemplos

1. Lea el ejemplo. Aclare conceptos y tipos de datos. Busque que los casos límite tengan un ejemplo.
2. Determine la firma del método por programar:

    - Nombre: un nombre imperativo que explique el algoritmo
    - Parámetros y sus tipos, que están dados por Dado y Cuando.
    - Salidas y sus tipos, que están dadas por Entonces.

3. Cree la clase de pruebas unitarias, en una estructura de *folders* igual al código por probar
4. Cree el primer método de pruebas unitarias, con el primer ejemplo, indicando:
    - El resultado esperado
    - La inicialización de parámetros y la asignación del resultado obtenido
    - La comparación por realizar.

5. Falla: Ejecute mirando que el código compile pero que la prueba falle.
6. Pasa: Escriba el código que razonablemente haga que la prueba pase.
7. *Refactoring*: mejore el diseño para simplificar, eliminar redundancia y mejorar legibilidad.
8. Repita sucesivamente para cada ejemplo, mirando que todas las pruebas anteriores sigan funcionando.

Recomendaciones

- Inicie a programar solamente cuando entienda muy bien el ejemplo y sus conceptos.
- Programe un ejemplo a la vez.
- Siempre vea que la prueba falle antes de intentar hacerla exitosa.

3 Un algoritmo fácil de probar

Cada vez que tengamos la tarea de crear una funcionalidad completa de software debemos planificar cómo organizar los distintos algoritmos involucrados. Un buen diseño hace que la mayoría de los algoritmos puedan verificarse con pruebas unitarias. Es decir, hay una relación directa entre un buen diseño y un sistema fácil de probar.

Un algoritmo fácil de probar tiene dos características:

1. Es observable.
2. Está aislado.

3.1 Observable

Un algoritmo es observable cuando nos retorna su resultado y a este lo podemos analizar. La siguiente función es observable:

```
Public Shared Function GenereLaValoracion(fechaActual As Date,
                                          informacionOficial As InformacionOficial,
                                          emision As Emision) As Valoracion
    Dim valoracion As New Valoracion

    valoracion.ISIN = emision.ISIN

    Dim diasParaElVencimiento As Integer
    diasParaElVencimiento = (informacionOficial.FechaDeVencimiento - fechaActual).Days
    Dim porcentajeDeCoberturaRevisado As Decimal
    If diasParaElVencimiento >= informacionOficial.DiasMinimos Then
        porcentajeDeCoberturaRevisado = informacionOficial.PorcentajeDeCobertura
    Else
        porcentajeDeCoberturaRevisado = 0
    End If
    valoracion.PorcentajeDeCobertura = porcentajeDeCoberturaRevisado

    Dim valorDeMercado As Decimal
    valorDeMercado = emision.MontoNominalDelSaldo * (informacionOficial.PrecioLimpio / 100)
    valoracion.ValorDeMercado = valorDeMercado

    valoracion.AporteDeGarantia = valorDeMercado * porcentajeDeCoberturaRevisado

    Return valoracion
End Function
```

Si es observable, entonces podemos invocar al algoritmo y comparar si el resultado obtenido es el que esperamos con los parámetros dados.

La siguiente función no es observable:

```
Public Shared Sub GenereLaValoracion(fechaActual As Date,
                                     informacionOficial As InformacionOficial,
                                     emision As Emision)
      Dim valoracion As New Valoracion

      valoracion.ISIN = emision.ISIN

      Dim diasParaElVencimiento As Integer
      diasParaElVencimiento = (informacionOficial.FechaDeVencimiento - fechaActual).Days
      Dim porcentajeDeCoberturaRevisado As Decimal
      If diasParaElVencimiento >= informacionOficial.DiasMinimos Then
          porcentajeDeCoberturaRevisado = informacionOficial.PorcentajeDeCobertura
      Else
          porcentajeDeCoberturaRevisado = 0
      End If
      valoracion.PorcentajeDeCobertura = porcentajeDeCoberturaRevisado

      Dim valorDeMercado As Decimal
      valorDeMercado = emision.MontoNominalDelSaldo * (informacionOficial.PrecioLimpio / 100)
      valoracion.ValorDeMercado = valorDeMercado

      valoracion.AporteDeGarantia = valorDeMercado * porcentajeDeCoberturaRevisado

      Repositorio.Guarde(valoracion)
End Sub
```

Si la función realiza sus operaciones, pero guarda su resultado en una base de datos o en un archivo, será más difícil y lento de probar. Si la prueba se automatiza ya no la podremos considerar una prueba unitaria, pues ya no ejecuta totalmente en memoria. Es decir, ya no es fácil de probar.

3.2 Aislado

Un algoritmo es aislado si todo lo que necesita lo recibe a través de sus parámetros:

```
Public Function GenereLaValoracion(
                      fechaActual As Date,
                      informacionOficial As InformacionOficial,
                      emision As Emision) As Valoracion
```

¿De qué maneras podríamos tener un algoritmo que no reciba lo que necesita por parámetros?

- Si accede a variables o constantes globales.
- Si obtiene datos de una base de datos, archivo o *webservice*.
- Si depende de condiciones externas como, por ejemplo, si accede directamente a funcionalidades aleatorias (*Random*) o usa la fecha actual (*Date.Now*).

Entonces, un algoritmo no está aislado si accede a recursos externos dentro de su ejecución.

Note por qué el siguiente algoritmo no está aislado:

```
Public Shared Function GenereLaValoracion(emision As Emision) As Valoracion
    Dim valoracion As New Valoracion

    valoracion.ISIN = emision.ISIN

    Dim informacionOficial As InformacionOficial
    informacionOficial = Repositorio.Consulte(emision.ISIN)

    Dim diasParaElVencimiento As Integer
    diasParaElVencimiento = (informacionOficial.FechaDeVencimiento - Date.Now).Days
    Dim porcentajeDeCoberturaRevisado As Decimal
    If diasParaElVencimiento >= informacionOficial.DiasMinimos Then
          porcentajeDeCoberturaRevisado = informacionOficial.PorcentajeDeCobertura
    Else
          porcentajeDeCoberturaRevisado = 0
    End If
    valoracion.PorcentajeDeCobertura = porcentajeDeCoberturaRevisado

    Dim valorDeMercado As Decimal
    valorDeMercado = emision.MontoNominalDelSaldo * (informacionOficial.PrecioLimpio / 100)
    valoracion.ValorDeMercado = valorDeMercado

    valoracion.AporteDeGarantia = valorDeMercado * porcentajeDeCoberturaRevisado

    Return valoracion
End Function
```

En conclusión, para diseñar un algoritmo fácil de probar debemos asegurarnos de que reciba todo por sus parámetros y que retorne su resultado.

4 Pruebas unitarias confiables

Cuando una prueba unitaria falla debe ser una indicación clara de que alguien cambió algo en el código fuente y produjo un defecto en el funcionamiento de nuestra aplicación. Es decir, no queremos nunca que una prueba nos dé un "falso" positivo, donde una prueba en rojo no signifique nada. Eso hace que el equipo de desarrollo deje de ponerles atención y perdemos cualquier beneficio de la técnica.

En este artículo veremos cómo lograr pruebas confiables.

Recordemos que una prueba automatizada es unitaria si cumple estas características:

1. Funciona en memoria: No accede a archivos, a bases de datos, a webservices u otra infraestructura.
2. Es repetible: Funciona con un clic pues no requiere configuraciones manuales, no depende de la fecha actual o de alguna función aleatoria. Son determinísticas.

Además de lo anterior, una prueba unitaria debería ser confiable para que nos ayude en nuestros desarrollos. Cuando no lo es, escuchamos frases como las siguientes:

- "La prueba falló porque tiene un defecto en su lógica. No es un defecto del software que prueba."
- "Corra las pruebas en este otro orden para que corran bien."
- "Esa prueba puede fallar por muchos motivos, tendremos que estudiar qué sucedió."
- "Las pruebas pasan pero realmente no prueban nada"

Las siguientes son tres indicaciones de que una prueba unitaria es confiable:

1. No contienen lógica
2. Son independientes
3. Realizan una comparación (pero solo una)

4.1 No contienen lógica

Las pruebas unitarias no realizan ningún cálculo matemático, operación ni tienen ninguna estructura de control. Es decir, no encontraremos ningún *if*, *switch*, *while*, *for* o *try*. De esta manera, reducimos la probabilidad de que las pruebas contengan defectos y son más sencillas de entender. Una prueba unitaria debería tener una estructura sencilla como esta:

```
<TestMethod()> Public Sub SeCalculaLaMetrica()
    Dim esperado As String = "95%"

    Dim iteracion As New Iteracion
    iteracion.PuntosPlanificados = 100
    iteracion.PuntosTerminados = 95
    iteracion.HayInformacionDePuntosDisponible = True
    Dim obtenido As String = Calculos .CalculeLaMetricaDePuntos(iteracion)

    Assert.AreEqual(esperado, obtenido)
End Sub
```

4.2 Son independientes

Las pruebas deben poderse ejecutar en cualquier orden y no dependen de resultados de otras pruebas. Para lograrlo, no utilizamos variables globales, ni ningún pase de datos del resultado de una prueba hacia otra. Por esto, es bueno utilizar el patrón de esperado, obtenido y una comparación.

4.3 Realizan una comparación (pero solo una)

Una buena prueba verifica un resultado esperado contra uno obtenido (*actual* vs *expected*). Recordemos que esto requiere que el algoritmo sea observable.

Partiendo de esto, una prueba unitaria debe tener un solo motivo para fallar. Es decir, realiza una sola comparación. De esta manera, cuando falle sabremos rápidamente qué sucede y podremos corregirlo con confianza. Si la prueba tiene múltiples *Asserts*, cuando uno de ellos falle, no sabremos qué sucedió con los siguientes, y seremos más lentos para mantener todo el software.

La comparación usualmente es entre dos datos o estructuras, pero en este caso, la comparación es realmente la expectativa de una excepción:

```
<TestClass()>
Public Class ValideEstadisticasTests
    Private iteracion As Iteracion

    <TestInitialize>
    Public Sub Inicialice()
        ' Inicialice el escenario valido aquí
        iteracion = New Iteracion
        iteracion.HayInformacionDePuntosDisponible = True
        iteracion.PuntosPlanificados = 10
        iteracion.PuntosTerminados = 8
        iteracion.HayInformacionDeDiasDisponible = True
        iteracion.CapacidadDelEquipo = 100
        iteracion.DiasNoEfectivos = 20
    End Sub

    <TestMethod(), ExpectedException(GetType(ArgumentException))>
    Public Sub PuntosPlanificadosNoDebenSerNegativos()
        iteracion.PuntosPlanificados = -1

        ValidacionDeEstadisticas.ValideEstadisticas(iteracion)
    End Sub
```

Ahora, vamos a mencionar que hay tipos de algoritmos de acuerdo a lo que retornan... Lo importante de saber que hay diferentes tipos es porque cada uno de ellos se prueba con una estrategia de comparación específica.

- Un algoritmo que retorna un dato escalar (número, texto, booleano, fecha)
- Un algoritmo que retorna una lista de escalares
- Un algoritmo que retorna una estructura de datos (una clase con propiedades)
- Un algoritmo que retorna una lista de estructuras de datos

Por ahora, la noción principal que deseo que tengamos es que una prueba unitaria finaliza con una comparación (y solo una).

En resumen, una buena prueba unitaria es aquella que no tiene lógica alguna, es independiente y realiza una sola comparación. Esto lo podemos lograr si el algoritmo que probamos es observable y aislado.

4.4 Videos

Como acompañamiento a este manual, tenemos los videos siguientes donde se explican más casos de pruebas unitarias confiables y las consecuencias contrarias:

Video	Dirección	Duración
Video: Pruebas unitarias confiables	https://goo.gl/z97Fo9	10 min.
Video: Pruebas unitarias no confiables (ejemplos y riesgos)	https://goo.gl/ycXa1D	16 min.

5 Referencias adicionales

Los siguientes son algunos recursos recomendados relacionados a los temas estudiados:

- Libro "Specification by example"
- Libro “The art of unit testing”
- Gherkin Language
- www.softwaremantenible.com