SOFTWARE MANTENIBLE

Diseño de software con pruebas unitarias

Con ejemplos en VisualBasic.net

2da parte

Instructor: Oscar Centeno

Octubre – Noviembre, 2017

Segunda Parte

En el estudio anterior visualizamos la generación de datos y los algoritmos de validación de una manera fácil de probar. La comparación se dio solamente a nivel del estado que los métodos retornan. En inglés, se le suele llamar "*State-based testing*". Completaremos el diseño y este estilo de pruebas, analizando la Composición de algoritmos.

El otro estilo de comparación de algoritmos fáciles de probar, en ingles se le suele llamar "*Interaction-based testing*", pues también vamos a verificar lo que es enviado por un algoritmo coordinador hacia fuentes de datos externas. Es decir, vamos a verificar las interacciones.

Todo este conocimiento lo sintetizaremos en el estudio de una técnica para visualizar el código legado y cómo reorganizarlo para que adopte una forma fácil de probar.

Recomendación: Más que escribir código fuente, esta sección requiere primero que lea el código fuente de referencia. Busque una comprensión clara del mismo y de cada mecanismo utilizado. Esa es la clave para después aplicar estas técnicas.

Contenidos

1 Composición de algoritmos

Descargue el código fuente que ilustra esta sección en https://goo.gl/kGVVKj, en la carpeta "2da parte\Composición de algoritmos".

1.1 Definición

Cualquier salida que produzca el sistema puede expresarse como una estructura de datos, es decir, una clase con propiedades. Cada propiedad será un escalar, una clase o algún tipo de lista (arreglos, listas, colecciones, diccionarios, etc.). Vamos a visualizar estas salidas como un árbol de algoritmos que en conjunto cooperan y cada uno tiene su estilo de especificación y de pruebas unitarias:

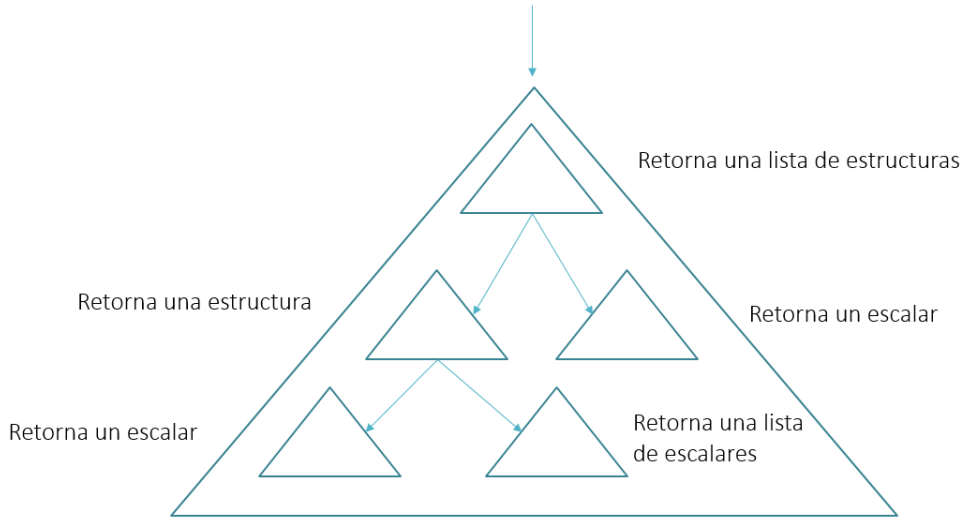


Figura 1 Teniendo un solo punto de entrada, una estructura compuesta es creada por varios algoritmos

Antes de programar cualquier funcionalidad, deberíamos tener muy claro cuál es el resultado esperado, y estos pasos nos ayudarán para ello:

1. Exprese un ejemplo de la salida del algoritmo.
2. Determine los algoritmos requeridos y su tipo.
3. Realice una especificación por ejemplos de cada uno.
4. Determine los parámetros de cada algoritmo, indicando tipos de datos.

A continuación, tenemos la explicación de cada paso:

1.2 Exprese un ejemplo de la salida del algoritmo.

JSON es un formato muy útil para expresar estructuras de datos, así que iniciaremos planteando un ejemplo. Le llamaremos "Reporte de métricas de un proyecto".


```
{
    "nombre": "Proyecto Asombroso",
    "iteraciones": [
        {
            "numero":"1",
            "fechaInicial":"2017-10-16",
            "fechaFinal":"2017-10-20",
            "metricaDePuntos":"80%",
            "metricaDeTiempoNoEfectivo":"5%"
        },
        {
            "numero":"2",
            "fechaInicial":"2017-10-23",
            "fechaFinal":"2017-10-27",
            "metricaDePuntos":"No hay datos",
            "metricaDeTiempoNoEfectivo":"5%"

        },
        {
            "numero":"3",
            "fechaInicial":"2017-10-30",
            "fechaFinal":"2017-11-03",
            "metricaDePuntos":"29%",
            "metricaDeTiempoNoEfectivo":"No hay datos"
        }
        ]
}
```


1.3 Determine los algoritmos requeridos y su tipo.

Inicie desde el paréntesis final, identificando el tipo de algoritmo: ¿es una lista, es una estructura o un escalar?

Sucesivamente, determine los algoritmos en profundidad, hasta llegar a los datos escalares que deban generarse.

En el ejemplo, primero identificamos que el algoritmo retorna una estructura de datos, que contiene una lista de estructuras de datos. Luego, determinamos que "métrica de puntos" y "métrica de tiempo no efectivo" podrían ser cálculos que haya que realizar.

Así, la lista de algoritmos es:

1. Genere el reporte de métricas de un proyecto: retorna una estructura de datos.
2. Genere la lista de métricas de las iteraciones: retorna una lista de estructuras de datos.
3. Genere las métricas de una iteración: retorna una estructura de datos.

4. Genere la métrica de puntos de una iteración: retorna un escalar
5. Genere la métrica de tiempo no efectivo de una iteración: retorna un escalar.

1.3.1 Determine cómo se realizará las pruebas unitarias

Para cada tipo de algoritmo tenemos una manera de saber las pruebas unitarias por realizar:

Si es un dato escalar, con *Assert.AreEqual*. Se puede realizar las pruebas que sean necesarias de acuerdo a la especificación con ejemplos que realice.

Si es una estructura de datos, con *Assert.AreEqual*. Se puede realizar una sola prueba unitaria.

Si es una lista, con *CollectionAssert.AreEqual*. Se recomienda realizar tres pruebas. Una en donde se genere una lista vacía, otra en donde se genere una lista con un elemento y otra con tres.

1.3.2 El método *Equals*

Para poder probar las estructuras, será necesario implementar el método *Equals* de la clase donde se compare cada una de sus propiedades. Esto implica igualmente realizar sus pruebas unitarias. Por ejemplo, analicemos el caso de esta clase:

```
Public Class ProyectoCreado
    Public Property Id As Integer
    Public Property Nombre As String
    Public Property FechaDeInicio As Date
    Public Property FechaDeCreacion As Date

End Class
```

Para poder probar su método *Equals*, vamos a requerir cinco pruebas, pues una prueba unitaria demostrará el caso de igualdad y luego habrá una prueba unitaria por cada propiedad en donde no sea igual. Esta sería su especificación con ejemplos:

Escenario: Cómo comparar proyectos creados

Dado un proyecto con estos datos

Id	Nombre	Fecha de inicio	Fecha de creación
1	MiProyecto	2018-10-26	2018-11-01

Cuando se compara con otro proyecto con <id>, <nombre>, <fecha de inicio> y <fecha de creación>

Entonces, se determina si <son iguales>.

Ejemplos:

Ejemplo	Id	Nombre	Fecha de inicio	Fecha de creación	Son iguales
Son iguales	1	MiProyecto	2018-10-26	2018-11-01	Sí
Id no es igual	2	MiProyecto	2018-10-26	2018-11-01	No
Nombre no es igual	1	MiProyecto2	2018-10-26	2018-11-01	No
Fecha de inicio no es igual	1	MiProyecto	2018-10-11	2018-11-01	No
Fecha de creación no es igual	1	MiProyecto	2018-10-26	1999-12-31	No

Esta es una implementación sencilla del método *Equals*:

```
Public Class ProyectoCreado
    Public Property Id As Integer
    Public Property Nombre As String
    Public Property FechaDeInicio As Date
    Public Property FechaDeCreacion As Date

    Public Overloads Overrides Function Equals(otherObject As Object) As Boolean
        Dim otroProyecto As ProyectoCreado = CType(otherObject, ProyectoCreado)

        Return Id = otroProyecto.Id _
            And Nombre = otroProyecto.Nombre _
            And FechaDeInicio = otroProyecto.FechaDeInicio _
            And FechaDeCreacion = otroProyecto.FechaDeCreacion
    End Function

End Class
```

¿Se puede utilizar alguna comparación por medio de *Reflection* para no tener que comparar cada propiedad?

Sí, hay librerías y código en internet que demuestran cómo hacerlo. El inconveniente es que el rendimiento de las pruebas unitarias no será el mismo, pues *Reflection* las hará más lentas.

1.4 Realice una especificación por ejemplos de cada uno.

Inicie especificando los algoritmos más sencillos, primero las reglas de negocio que son los cálculos que determinan los escalares, y de allí hacia las estructuras más compuestas.

Los algoritmos de estructuras y listas deberían requerir un solo ejemplo. Si encuentra algún caso donde un ejemplo no sea suficiente, considere si sería más sencillo separar en dos algoritmos. Esto sucede en ocasiones donde el resultado de salida tiene diferentes propiedades dependiendo de algún tipo que se recibe por parámetro.

1.5 Determine los parámetros de cada algoritmo, indicando tipos de datos.

Iniciando por el algoritmo de la estructura completa, indique los parámetros.
Por ejemplo:

1. Genere el reporte de métricas de un proyecto: recibe un proyecto, que contiene nombre y una lista de los datos de sus iteraciones.
2. Genere la lista de métricas de las iteraciones: recibe la lista de los datos de las iteraciones.
3. Genere las métricas de una iteración: recibe los datos de una iteración. Cada iteración consta de número, fecha de inicio, fecha final, hay información de puntos disponible, puntos planificados, puntos terminados, hay información de días disponible, capacidad del equipo, días no efectivos.
4. Genere la métrica de puntos de una iteración: usa algunos datos de una iteración, por lo que podemos decir que recibe una iteración.
5. Genere la métrica de tiempo no efectivo de una iteración: usa algunos datos de una iteración, por lo que podemos decir que recibe una iteración, igualmente.

2 Casos de uso

2.1 Definición

Es una secuencia de pasos entre un actor y el Sistema que logra un resultado de valor observable. Por ejemplo:

Caso de uso: Agregar Proyecto

Flujo básico
1. En la lista de proyectos, el actor indica que desea agregar un Proyecto.
2. El Sistema le muestra un formulario donde se le solicita Nombre y Fecha de inicio.
3. El actor indica los datos válidos.
4. El Sistema procesa la solicitud y crea el Nuevo Proyecto con Nombre, Fecha de inicio y Fecha de creación.
5. El Sistema muestra la lista de proyectos actualizada.

Flujos alternos
1. Se debe manejar errores de validación.

Al ser una explicación de alto nivel, muchos detalles de funcionalidad quedan por fuera y se deben analizar posteriormente. Además, un caso de uso se puede componer de múltiples interfaces gráficas, en este caso, por ejemplo:

- Mostrar la lista de proyectos
- Mostrar el formulario
- Confirmar el proyecto creado.

En estas se hacen validaciones, conversiones y envíos de estructuras de datos al servidor, como:

- Consultar la lista de proyectos
- Agregar un proyecto

En el servidor se valida y se genera otras estructuras. Además, se obtiene y envía estructuras hacia fuentes externas de datos y herramientas.

Por esto, una sola explicación textual, nunca es suficiente.

El análisis de un caso de uso debería considerar aspectos como:
- Bosquejos de la interfaz gráfica
- Especificación de las validaciones en la interfaz gráfica
- Para cada comunicación con el servidor:

- Las estructuras de datos enviadas y recibidas del servidor.
- Especificaciones con ejemplos de la lógica
- Las estructuras enviadas a fuentes externas
- Consideraciones no funcionales

Todas estas perspectivas son las que nos pueden dar una comprensión clara de la tarea de desarrollarlo y cómo hacerlo eficientemente.

En la siguiente sección miraremos algunas guías para organizar el código fuente, con base en los casos de uso. Luego, trabajaremos con los Algoritmos que coordinan. Cada uno de estos algoritmos corresponderá a un llamado hacia el servidor.

2.2 Organización del código fuente

Estas son algunas recomendaciones para mantener la separación entre casos de uso y algoritmos.

Separación vertical. Aún más importante que separar capas de componentes como "Webservice", "Acceso a datos" y "Logica de negocio", estas recomendaciones buscan una separación vertical, donde cada funcionalidad pueda entenderse y desarrollarse con menos dependencias en otras.

Una carpeta por Caso de uso. Separar cada caso de uso en su propia carpeta y *namespace* ayuda a entender el sistema de acuerdo a su uso. Además, ayuda a determinar más claramente el impacto de los cambios.

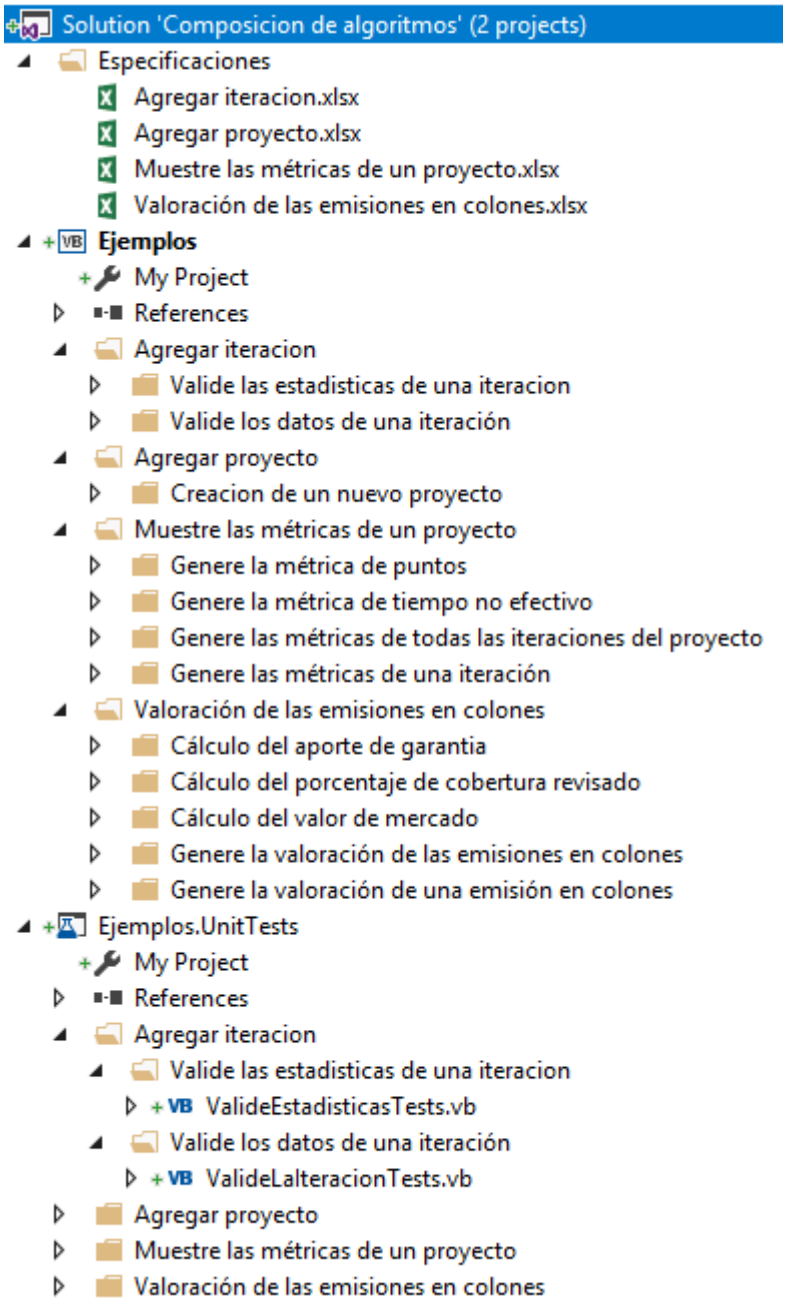
Cuando las clases y demás archivos se organizan por su tipo, como por ejemplo, "entidades", "repositorios", "servicios" y "controladores", se tiende a tener muchos archivos por carpeta y a que estos sean grandes. Los conflictos por cambios son comunes y todo ello contribuye también a que el sistema sea difícil de entender. La organización por tipo funciona bien en aplicaciones muy pequeñas, pero no cuando se requiere trabajar en equipo.

Un beneficio inmediato de la organización por uso es que se tiene menos archivos por carpeta y cada uno es más pequeño. Esto ayuda mucho a facilitar la comprensión.

Una carpeta por flujo de datos. Por ejemplo, si en Agregar Proyecto, la interfaz gráfica requiere consultar ciertos datos para mostrar un formulario, entonces ese es un flujo de datos que debería tener su propia carpeta. Otro sería el propio flujo de datos donde se envía la información para agregar un proyecto nuevo.

Una carpeta por algoritmo. Por claridad, es de ayuda agrupar cada algoritmo en una carpeta con un nombre claro. Las clases que realicen dicho algoritmo se usan y cambien siempre juntas, y así se ayudará a la eficiencia del programador.

Organización de las especificaciones. Igualmente, se puede mantener las especificaciones por caso de uso, cercanas al código fuente. Al ser ejemplos útiles para comprender el código fuente se puede trazar fácilmente al codigo fuente.



3 Algoritmos que coordinan

3.1 Definición

Un caso de uso puede tener varias interacciones entre un actor y el sistema. Cada una de ellas puede ser un flujo de datos totalmente diferente. A partir de ahora asumiremos que estamos trabajando sobre solamente un flujo.

Este tipo de algoritmo es un pasador de información entre otros algoritmos, por lo que confía en que los demás funcionan correctamente. Este es el sitio donde realizamos el manejo de errores. Clasificaremos Los algoritmos que coordina en cuatro tipos:

1. Los que generan nuevas estructuras de datos: En la primera parte estudiamos este tipo y los diferenciamos de acuerdo al tipo de dato que retornan:
    a. Escalares (datos simples)
    b. Listas de escalares
    c. Estructuras de datos (clases con propiedades)
    d. Listas de estructuras de datos
2. Los que validan: Igualmente los estudiamos en la primera parte y estos son la línea defensiva de una funcionalidad de software, al no permitir que se realice operaciones en el sistema si los datos de entrada no cumplen las validaciones.
3. Los que consultan datos de fuentes externas: Por ejemplo, obtienen información de una base de datos o consultan información de *webservices*.
4. Los que envían datos a fuentes externas: Por ejemplo, envían a guardar a la base de datos o invocan a un *webservice* de otro sistema para confirmar una transacción.

Ilustraremos los cinco tipos de algoritmos con estos íconos:

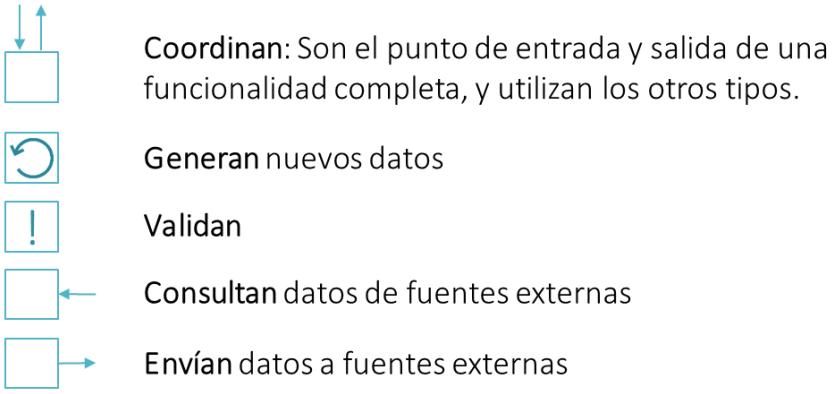


Figura 2 Tipos de algoritmos

3.2 El diseño de un coordinador

3.2.1 Escriba los pasos requeridos para lograr la salida esperada

Usando los cuatro tipos de algoritmos, piense en los pasos que requiere para lograr la salida esperada. Por ejemplo, esta es la explicación de la secuencia de pasos de un coordinador del caso de uso "Agregar proyecto":

1. Se recibe los datos del nuevo proyecto: nombre y fecha de inicio
2. Se obtiene los datos de la base de datos que se necesitan para validar el nuevo proyecto.
3. Se valida el nuevo proyecto por crear.
4. Se genera el nuevo proyecto por crear: nombre, fecha de inicio, fecha de creación
5. Se envía a guardar el nuevo proyecto a la base de datos
6. Se retorna el nuevo proyecto creado: id, nombre, fecha de inicio, fecha de creación.

Note que es una explicación técnica, donde pensamos en las estructuras de datos y en los tipos de algoritmos. La audiencia somos quienes desarrollamos el software. Determine los tipos de datos de cada estructura y propiedad.

3.2.2 Plantee un ejemplo completo del flujo básico

En ocasiones podemos explicarlo de manera sencilla como a continuación, pero no debemos limitarnos a un documento. Quizá sea más claro usar un diagrama en una pizarra o en otro formato. Lo importante es que tengamos muy claro cómo la información fluye y se transforma.

1. Nuevo proyecto:
    a. Nombre: `Proyecto Asombroso`
    b. Fecha de inicio: `2018-10-26`
2. Información obtenida:
    a. Proyectos existentes: ninguno.
    b. Fecha actual: `2018-09-01`
3. El proyecto es válido.
4. El nuevo proyecto por crear es:
    a. Nombre: `Proyecto Asombroso`
    b. Fecha de inicio: `2018-10-26`
    c. Fecha de creación: `2018-09-01`
5. El nuevo proyecto creado es:
    a. Id: 1
    b. Nombre: `Proyecto Asombroso`
    c. Fecha de inicio: `2018-10-26`
    d. Fecha de creación: `2018-09-01`
6. Se retorna el nuevo Proyecto creado.

3.2.3 Considere los flujos alternos

Determine los errores de validación y que debe manejar errores no esperados. Por ejemplo, en el caso de un *webservice*:

- Si se da un error de validación, se debe indicar con una excepción que contiene el mensaje de error.
- Si se da un error esperado, se debe indicar con una excepción.

En caso de estar diseñando un API HTTP, se va a determinar los códigos de status HTTP de cada flujo alterno.

3.2.4 Programe los algoritmos de validación y generación

Con base en las especificaciones por ejemplos, programe los algoritmos de validación y generación.

3.3 Organización del código fuente

Este es un ejemplo de la organización propuesta para un caso de uso, en donde en la raíz de la carpeta "Agregar proyecto" se encuentra el algoritmo que coordina. Usualmente es llamado un Coordinador, "Controller" o Servicio.

Note que se indica claramente cuáles estructuras de datos son entradas o salidas del caso de uso.

Además todas las dependencias a fuentes de datos externas se ubican en "Acceso a datos".

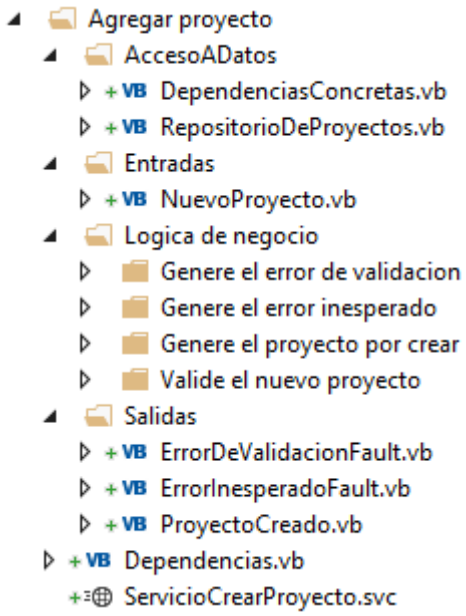


Figura 3 Organización de un caso de uso

3.3.1 Diseñe el coordinador fácil de probar

Ya que el coordinador obtiene y envía datos de fuentes externas necesitamos una manera de aislarlo para poder verificarlo con pruebas unitarias. Para esto, crearemos una variable de instancia del tipo de una clase abstracta llamada "Dependencias", que también será inicializada en el constructor. Cada paso de obtener y enviar datos a fuentes externas será un método de la clase "Dependencias".

Leamos el siguiente coordinador de un servicio web WCF que utiliza todos los tipos de algoritmo:

```
<ServiceContract()>
Public Class ServicioCrearProyecto
    Private dependencias As Dependencias

    Public Sub New()
        dependencias = New DependenciasConcretas
    End Sub

    Public Sub New(dependencias As Dependencias)
        Me.dependencias = dependencias
    End Sub

    <FaultContract(GetType(ErrorDeValidacionFault))>
    <FaultContract(GetType(ErrorInesperadoFault))>
    <OperationContract()>
    Public Function AgregueUnProyecto(nuevoProyecto As NuevoProyecto) As ProyectoCreado
        ' Algoritmo que obtiene datos de fuentes externas
        Dim fechaActual = dependencias.ObtengaLaFechaActual()

        Try
            ' Algoritmo que obtiene datos de fuentes externas
            Dim proyectosExistentes As List(Of String)
            proyectosExistentes = dependencias.ObtengaLosNombresDeLosProyectosExistentes()

            Validacion.ValideElNuevoProyecto(nuevoProyecto, proyectosExistentes)
            Dim proyectoCreado As ProyectoCreado
            proyectoCreado = GeneracionDeProyecto.GenereElProyectoPorCrear(nuevoProyecto, fechaActual)

            ' Algoritmo que envia datos a fuentes externas
            Return dependencias.Guarde(proyectoCreado)
        Catch exception As ArgumentException
            Throw GeneracionDeErrorDeValidacion.GenereElError(exception, fechaActual)
        Catch exception As Exception
            Throw GeneracionDeErrorInesperado.GenereElError(exception, fechaActual)
        End Try
    End Function
End Class
```

Esta es la clase "Dependencias":

```
Public MustInherit Class Dependencias
    Public MustOverride Function ObtengaLosNombresDeLosProyectosExistentes() As List(Of String)
    Public MustOverride Function Guarde(proyectoCreado As ProyectoCreado) As ProyectoCreado
    Public MustOverride Function ObtengaLaFechaActual() As Date
End Class
```

Para implementar los llamados concretos a la base de datos y a cualquier otra herramienta, contamos con la clase "DependenciasConcretas":

```
Public Class DependenciasConcretas
    Inherits Dependencias

    Public Overrides Function ObtengaLosNombresDeLosProyectosExistentes() As List(Of String)
        Return RepositorioDeProyectos.ObtengaLosNombresDeLosProyectosExistentes()
    End Function

    Public Overrides Function Guarde(proyectoCreado As ProyectoCreado) As ProyectoCreado
        Return RepositorioDeProyectos.Guarde(proyectoCreado)
    End Function

    Public Overrides Function ObtengaLaFechaActual() As Date
        Return Date.Now
    End Function
End Class
```

Esta clase no tendrá pruebas unitarias, pero no contendrá lógica de negocio.

3.3.2 Cómo probar el coordinador

Necesitamos una manera de verificar que los pasos de obtener y enviar a fuentes de datos externas son invocados correctamente. Para los pasos de obtener, necesitamos una manera de simular un retorno de datos cuando el coordinador lo necesite. Para los pasos de enviar, necesitamos verificar que sean invocados pues los consideramos una salida del algoritmo también.

A la simulación del funcionamiento de un obtener, se le llama un "*Stub*". A la verificación de la invocación de un enviar se le llama un "*Mock*".

Para realizar ambas funcionalidades, usaremos una librería llamada *NSubstitute* que puede ser instalada en el proyecto de Pruebas unitarias por medio de *NuGet*. En esta librería, un *Mock* o *Stub* es simplemente llamado un *Substitute*.

La siguiente es una prueba unitaria que simula un objeto *Dependencias* sustituto con el que podemos simular que se obtiene datos de la base de datos para verificar que el retorno del algoritmo es el esperado. Resalto el uso de la simulación de cada retorno de la base de datos y de la fecha actual:

```
<TestClass()> Public Class AgregueUnProyectoTests
    Dim obtenido As ProyectoCreado

    <TestMethod()> Public Sub RetornaElNuevoProyectoCreado()
        Dim esperado = ObtengaElProyectoCreadoConId()

        Dim dependencias As Dependencias = InicialiceLasDependencias()
        Dim servicio As New ServicioCrearProyecto(dependencias)
        Dim proyectoValido As NuevoProyecto = ObtengaUnProyectoValido()
        obtenido = servicio.AgregueUnProyecto(proyectoValido)

        Assert.AreEqual(esperado, obtenido)
    End Sub

    Private Shared Function InicialiceLasDependencias() As Dependencias
        Dim dependencias = Substitute.For(Of Dependencias)()

        dependencias.ObtengaLaFechaActual().Returns(New Date(2018, 9, 1))
        dependencias.ObtengaLosNombresDeLosProyectosExistentes().Returns(New List(Of String))

        Dim proyectoCreadoSinId As ProyectoCreado = ObtengaElProyectoPorCrear()
        Dim proyectoCreadoConId As ProyectoCreado = ObtengaElProyectoCreadoConId()
        dependencias.Guarde(proyectoCreadoSinId).Returns(proyectoCreadoConId)

        Return dependencias
    End Function

    Private Shared Function ObtengaUnProyectoValido() As NuevoProyecto
        Return New NuevoProyecto() With {
            .Nombre = "Proyecto Asombroso",
            .FechaDeInicio = New Date(2018, 10, 26)
        }
    End Function
```

Para verificar que se está llamando al enviar, de la siguiente manera indicamos que se debe haber recibido un llamado a "Guarde", una vez con el parámetro indicado. Esta verificación también compara el parámetro enviado a "Guarde", e internamente utiliza el método "*Equals*" de la clase "ProyectoCreado".

```
<TestMethod()> Public Sub EnviaAGuardarElNuevoProyecto()
    Dim dependencias As Dependencias = InicialiceLasDependencias()
    Dim servicio As New ServicioCrearProyecto(dependencias)
    Dim nuevo As NuevoProyecto = ObtengaUnProyectoValido()
    obtenido = servicio.AgregueUnProyecto(nuevo)

    ' Assert
    Dim proyectoPorCrear As ProyectoCreado = ObtengaElProyectoPorCrear()
    dependencias.Received(1).Guarde(proyectoPorCrear)
End Sub
```

3.4 Ejercicio: “Muestre las métricas de un proyecto”

Estudie el código fuente del caso de uso “Muestre las métricas de un proyecto” en la solución “El algoritmo coordinador”, con base en la siguiente secuencia:

3.4.1 Flujo básico

1. Se recibe el *id* del proyecto a consultar.
2. Se obtiene el proyecto de la base de datos: nombre y su lista de iteraciones.
3. Se genera el reporte de las métricas.
4. Se retorna el reporte.

3.4.2 Flujos alternos

Se debe manejar el caso si el proyecto indicado no existe.

3.5 Ejercicio: Programe coordinadores

Con base en las especificaciones por ejemplos ya programadas, especifique la secuencia de pasos del algoritmo coordinador (flujo básico y flujo alterno), sus ejemplos, su código y pruebas unitarias. Los casos de uso en la solución “El algoritmo coordinador”:

- Agregar iteración
- Valoración de las emisiones en colones

3.6 Ejercicio: Diseñe un caso de uso conocido

Plantéese el reto de diseñar un caso de uso que sea conocido por usted de manera que pueda analizar tanto la secuencia del algoritmo coordinador y sus distintos algoritmos, para luego programarlo con sus pruebas unitarias.

3.7 Resumen

3.7.1 Características de un buen coordinador

a. Usa un algoritmo de validación en sus datos de entrada (parámetros) revisando formatos, rangos, tipos y también su validez dentro del contexto. Por ejemplo, si vamos a agregar el nombre de un Proyecto Nuevo, se valida que ese nombre tenga caracteres válidos y que no esté nulo o vacío. Y además, se valida que sea único para que no se repita con los nombres de los proyectos existentes en la base de datos.
b. No genera o convierte datos sino que usa otros algoritmos para ello.
c. Maneja errores de validación y cualquier otro error inesperado.
d. Cada algoritmo de envío recibe una estructura ya generada por otro algoritmo de generación. Las responsabilidades no se mezclan.
e. Es Observable. El algoritmo coordinador tiene salidas por medio de su propio resultado (*return*), en las excepciones que genera y también lo tiene en cada envío a fuentes externas.
f. Aplica la inversión de dependencias para ser Aislado.

3.7.2 Las pruebas unitarias del coordinador

a. Simulan cada consulta a fuentes de datos con un *Stub* de sus dependencias.
b. Cada prueba unitaria tiene una sola verificación:
    a. Verifican el flujo básico con el *Assert.AreEqual* de su retorno.
    b. Verifican cada envío con un *Mock*.
    c. Verifican cada fluyo alterno (errores de validación, errores inesperados, decisiones de negocio).

4 Código legado

4.1 Definición

Llamaremos código legado a cualquier código al que tenemos que dar mantenimiento. Puede que haya sido escrito por nosotros o no, pero usualmente es complicado de trabajar porque mezcla los distintos de algoritmos.

Usualmente los algoritmos que coordinan se programan de manera difícil de probar ya que son quienes se comunican con dependencias de datos externas, como por ejemplo:

1. Al consultar un *webservice* o *API*
2. Al enviar un correo electrónico
3. Al leer o escribir del *File System*
4. Al escribir a bitácora
5. Al usar la fecha actual
6. Al requerir algún número aleatorio

Además, usualmente se mezcla la generación de estructuras de datos, con validaciones y con dicha comunicación hacia dependencias externas.

Por estos motivos es que el código legado es riesgoso de modificar y toma mucho tiempo probarlo. Una pregunta usual es ¿cómo puedo rediseñar algo ya existente? ¿Es posible realizarlo de manera segura?

4.2 ¿Cómo rediseñar un código legado que no tenga un diseño sencillo de probar?

Los siguientes pasos explican el rediseño de una funcionalidad existente. Esto usualmente es riesgoso pues no hay ningún tipo de respaldo de pruebas unitarias para hacerlo. Antes de aplicarlos verifique que tenga una documentación de los casos de prueba que aseguran que la funcionalidad actual funcione adecuadamente. Ejecute esas pruebas antes y después del rediseño para tener confianza de no haber introducido regresiones.

Los siguientes pasos son ilustrados en el código fuente de referencia en la carpeta “2da parte\Código legado”:

1. Analice
    a. Analice el código existente y asigne las responsabilidades según vimos en la Figura 2 Tipos de algoritmos.
    b. Reagrupe el código de manera que pueda extraerse en bloques independientes posteriormente.
2. Extraiga algoritmos de validación y generación
    a. Extraiga los algoritmos a clases aparte, asegurando que los de validación y de generación sean fáciles de probar.

   b. Realice la especificación por ejemplos de cada algoritmo de validación y generación.
   c. Luego, realice sus pruebas unitarias.
3. Extraiga las dependencias externas
   a. Verifique que el algoritmo coordinador tenga esa única labor, aunque aún invoque a dependencias externas. Es decir, verifique que no haya ningún tipo de creación de estructuras de datos o validaciones. El código debería contar solamente con invocación a otras clases y el flujo de datos debería ser sencillo de seguir.
   b. Extraiga los llamados a las dependencias externas en métodos de una clase "Dependencias" que el algoritmo coordinador reciba en su constructor. Esta clase es la clave para aislar dichas dependencias.
   c. Haga que la clase de Dependencias sea abstracta, igualmente que los métodos que contiene. Mueva los métodos originales a una clase que implemente la clase Dependencias, por ejemplo llamada "DependenciasConcretas". El algoritmo coordinador continuará recibiendo Dependencias en su constructor.
   d. Cree un constructor sin parámetros en el que se inicialice la clase "DependenciasConcretas" y se asigne a la variable de instancia "Dependencias".
   e. Realice las pruebas unitarias del algoritmo coordinador.