Pruebas Unitarias:

Las pruebas unitarias presentan un enfoque el cual se basa en incrementar la correctitud y calidad del software, es utilizado en enfoques como por ejemplo "**Extreme Programming**".

Contario a métodos rigurosos como lo son el razonamiento formal y pruebas complejas, las pruebas unitarias, correctamente implementadas, ofrecen simplicidad y son menos costosas que otros enfoques de pruebas a productos de software. Este tipo de pruebas son más prácticas ya que en entornos donde las pruebas son automatizadas la retroalimentación es inmediata.

Las pruebas unitarias una estrategia de la evaluación del comportamiento de un programa. Los programas son descompuestos en unidades funcionales, las cuales son independientemente probadas. Cada "unit test" toma una sección del programa y con el uso de "test inputs" evalúa el comportamiento de la sección del programa a probar.

¿Consideraciones importantes de las pruebas unitarias?

Una prueba unitaria no debe de ser dependiente de:

- Configuraciones del entorno.
- Otros "Unit tests", incluyendo el orden de la ejecución.

Completo conocimiento de lo que se prueba: Una señal clara de que la prueba puede no ser efectiva es cuando se presentan dificultades a la hora de nombrar una prueba unitaria ya que significa un conocimiento pobre del componente a evaluar o más bien una prueba sumamente compleja en donde se realizan muchas pruebas las cuales pueden ser separadas.

Las pruebas deben de ejecutarse sin necesidad de una intervención manual, de ahí su importancia para estrategias de desarrollo de software como lo es extreme programming ya que permite delegar esas tareas a un sistema automático.

Las pruebas deben de ser ejecutadas cuantas veces se requiera y por ende debe de arrojar las mismas condiciones para las mismas condiciones.

Las pruebas unitarias deben de cubrir la casi totalidad del código, lo que implica que, si la cobertura es baja, gran parte del código está sin probar.

Beneficios de las pruebas unitarias:

- La aplicación correcta de las pruebas unitarias, permiten un ciclo de desarrollo de software más amigable, ya que la depuración es constante y por ende aporta en gran magnitud a que el código funcione de la forma esperada bajo las especiaciones definidas.
- Promueven la refactorización y mejora continua del producto de software.
- Facilita los procesos de integración entre sistemas. Lo anterior debido a que existen pruebas las cuales presentan dependencias externas modulares las cuales deben de ser

simuladas y por consiguiente permite al equipo de desarrollo preparase para la etapa de integración real.

- Aporta a la legibilidad del código, pues se puede comprender mejor qué hace una sección del código y que se espera de él.

Pruebas unitarias en un entorno C#

Se utilizó el entorno de desarrollo Visual Studio de Microsoft, el cual provee el lenguaje C#, y en el cual se puede compilar y ejecutar código. Además, este entorno permite utilizar frameworks de pruebas unitarias y adaptarlos. En versiones de pago, el entorno trae integrado un framework propio.

Para las pruebas unitarias existe un tipo de clase que se llama "assert class", esta clase se utiliza para afirmar cuales comportamientos de lo que se está probando son aceptables y cuáles no.

A continuación, algunos framework que se investigaron para realizar el ejemplo:

Visual Studio Unit Testing : El entorno nos permite desarrollar nuestras propias pruebas, las cuales se pueden: ejecutar, explorar y visualizar, este código es integrado al proyecto al proyecto y es de fácil funcionamiento. Las pruebas se pueden ejecutar de manera grupal o individual. En la figura se puede apreciar una ventana que muestra el resultado de ejecutar un conjunto de pruebas y como se expone cuales son correctas y en cuales surgió un error. Cabe resaltar que esta herramienta no genera código de pruebas, es necesario crear nuestro propio código. En la versión prime o pago de Visual Studio se ofrece un mejor framework para pruebas que cuenta con integración continua y otras características.

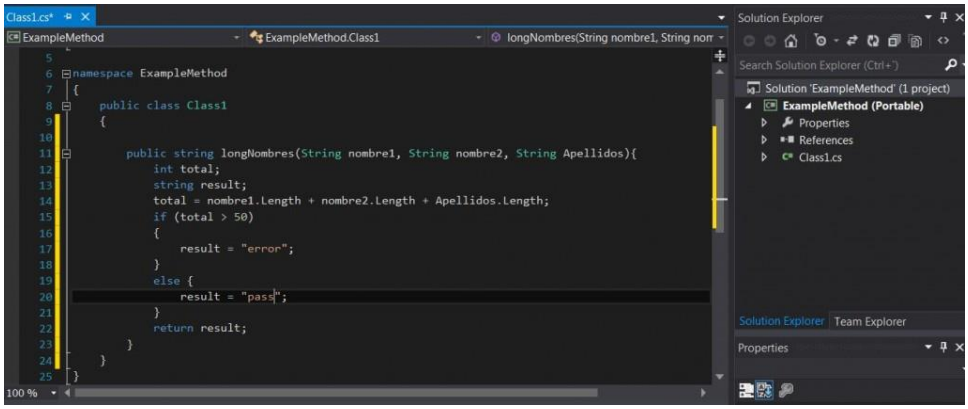
XUnit.net: Es una herrramienta de pruebas enfocada a la comunidad, de código abierto para los frameworks de C#, F#, VB.NET y otros lenguajes .NET. XUnit trabaja con ReSharper, CodeRush, TestDriven y Xamarin. Es parte de la fundación .NET, y opera sobre su conducta de código. Tiene su licencia sobre Apache 2. Para poder utilizar xUnit se necesita tener instalado Visual Studio, que será el IDE utilizado para desarrollar los proyectos y las pruebas unitarias.

Beneficios:

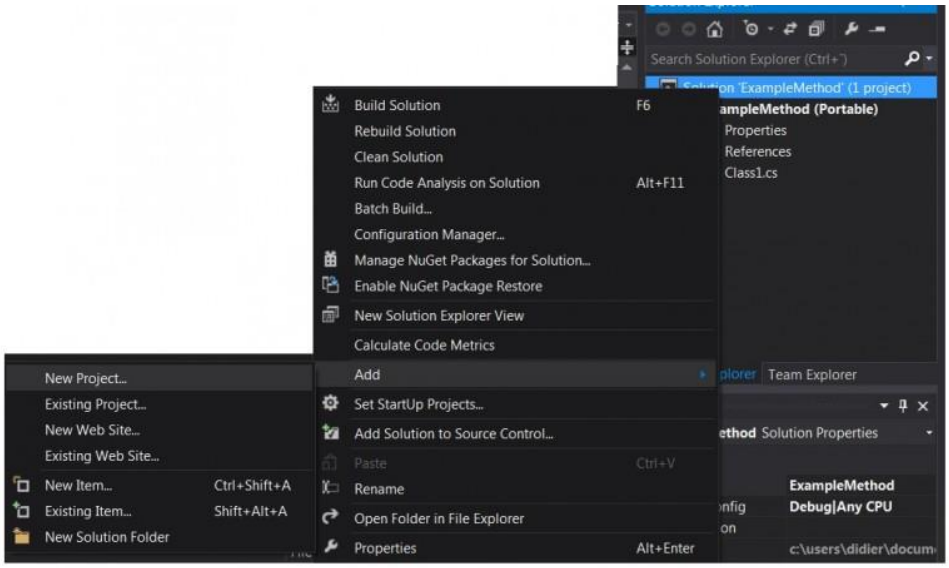
- Limpieza en el código, al separar la parte importante del código, que en lo que se puede la parte de preparación y limpieza.
- Utilizando métodos como after se ejecuta incluso cuando salta una excepción en el test.

Ejemplo de pruebas unitarias en un entorno C# con xUnit

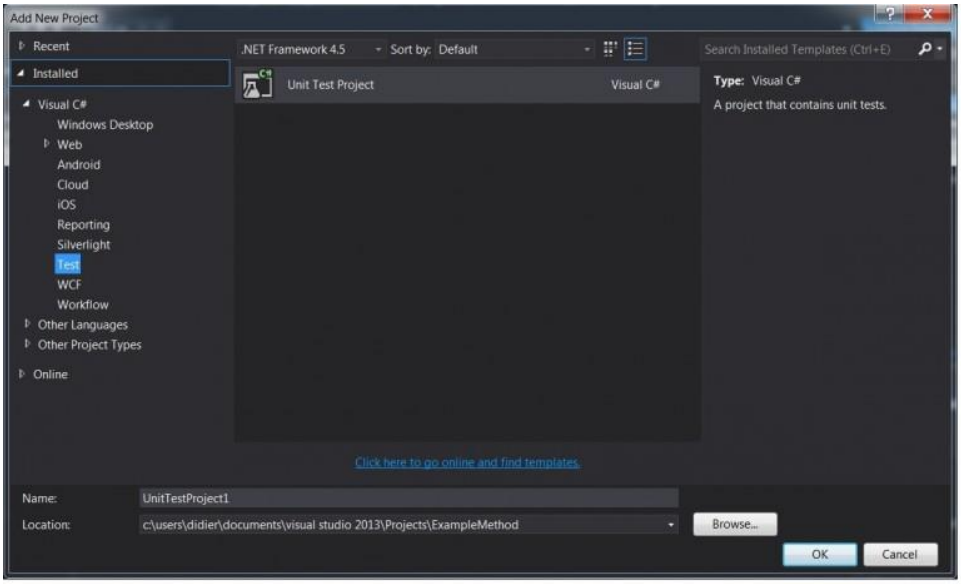
Para esta prueba se verificará el comportamiento de una única clase con un solo método en un proyecto de Visual Studio que recibe un nombre de usario y evalua que sea menor a 50.



- Primero se debe de crear un nuevo proyecto para la prueba unitaria en Visual Studio.
- Luego a la solución del proyecto original se le debe agregar el proyecto de la prueba unitaria:



- Se debe de agregar una referencia al proyecto que se quiere probar. Esto se hace al lado izquierdo del menú.



- Luego en el proyecto de unit test se debe de evaluar el comportamiento de la función longnombres del proyecto inicial.

```
[TestClass]
public class UnitTest1
{
    Class1 ayu = new Class1();
    [TestMethod]
    public void edadCorrecta(){

        //Arrange
        string nombre1 = "test";
        string nombre2 = "test";
        string apellidos = "test";
        string result;
        string resultadoEsperado = "pass";
        //Act
        result = ayu.longNombres(nombre1, nombre2, apellidos);
        //Assert
        Assert.AreEqual(resultadoEsperado, result);
```

Se utiliza el objeto assert que es el que compara los resultados a ver si el comportamiento es el mismo. Lo cúal da el siguiente resultado:

Pruebas unitarias en un entorno WEB y JavaScript

KarmaJS es una herramienta que proporciona un web server que permite someter en tiempo real las pruebas unitarias de código de **javascript** para que el usuario se entere de inmediato ante una falla. Este **framework** es desarrollado por el equipo que desarrolló Angular. El objetivo de Karma es proporcionar, un ambiente productivo de pruebas para los desarrolladores, además de enfocarse de proveer retroalimentación inmediata.

KarmaJS permite realizar pruebas en dispositivos reales, así como buscadores web, inclusive permite la integración de **Brwosers** sin interfaz gráfica, como lo es **PhantomJS** para realizar pruebas y que los resultados sean mostrados mediante alguna Terminal destinada a la aplicación.

Junto con herramientas dedicadas como **JasmineJS**, se permite describir las pruebas a desarrollar. **JasmineJS** Es un **framework** dedicado a realizar pruebas enfocadas al comportamiento de código escrito en **javascript** y junto con **KarmaJS** se extiende a aplicaciones desarrolladas en **AngularJS**.

KarmaJS, al integrarlo a un proyecto, observa cada uno de los archivos definidos para realizar pruebas mediante un archivo de configuración, de esta manera cuando se guarden cambios, **KarmaJS** realizará las pruebas necesarias en los buscadores establecidos de la misma forma almacenados en el archivo de configuración.

KarmaJS se enfoca en pruebas unitarias, es decir de bajo nivel, pues trabaja sobre pequeñas unidades de código, en caso de realizar pruebas más complejas, se recomienda **ProtactorJS**.

Ejemplo de pruebas unitarias en un WEB y JavaScript

Configuración del proyecto web para realizar las pruebas.

Las tecnologías a utilizar:

- **NodeJs,** JavaScript del lado del servidor y junto con **NPM** y **Bower** permitirán agregar dependencias de código de forma más amigable.
- **AngularJS V1.X, javascript** del lado del cliente el cual será sometido apruebas definidas mediante el **framework JasmineJS** y ejecutadas por KarmaJS.

Instalación de la herramienta, bajo un entorno de Linux Mint 18 de 64 bits.

Instalacción de NodeJS: En una terminal escribir los siguientes comandos.

```
$curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
$sudo apt-get install -y nodejs
```

Iniciar unproyecto en NodeJS: En una terminal ejecutar el siguiente comando.

```
$ npm init
#dejar los valores por defecto que la terminal #hace referencia.
```

Instalación de Bower: En una terminal ejecutar el siguiente comando.

```
$ npm install bower
```

Instalación de las Herramientas de Karma En una terminal ejecutar el siguiente comando.

```
$ npm install karma --save-dev
# Install plugins that your project needs:
$ npm install karma-jasmine karma-chrome-launcher jasmine-core –save-dev
$ npm install -g karma-cli
```

Creación de la configuración del proyecto KarmaJS.

Mediante la ejecución del siguiente comando: Se establece la configuración del proyecto.

```
$ karma
```

Estructura del proyecto:

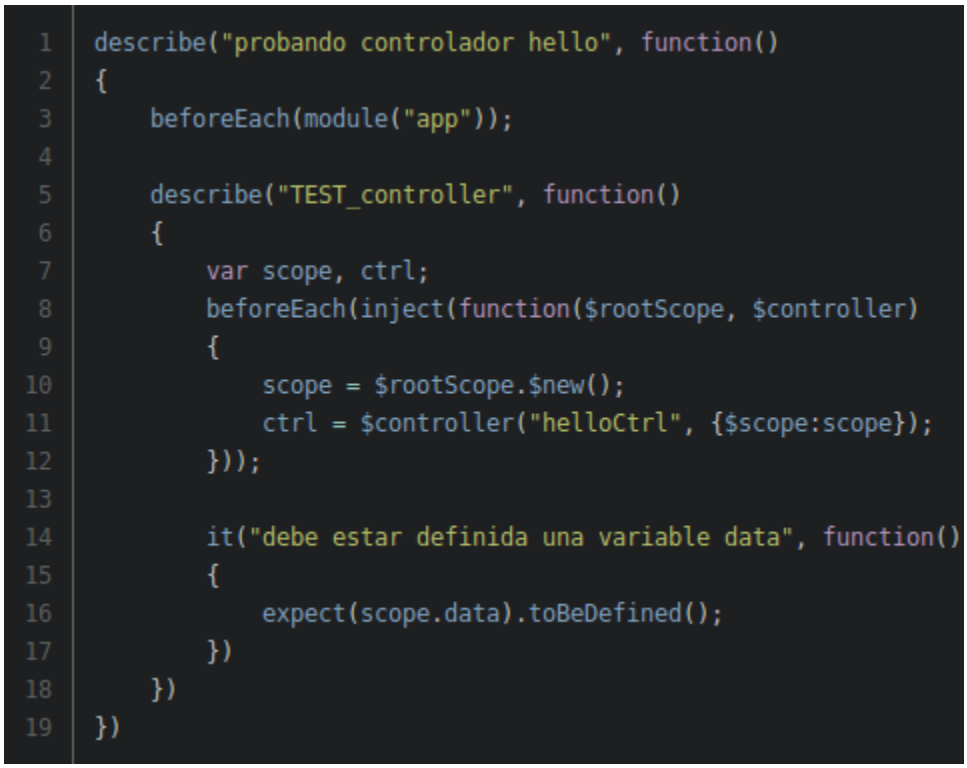
Name	Last commit
app/controllers	first add
bower_components	first add
node_modules	first add
tests/controllers	first add
karma.conf.js	first add
package.json	first add

Imagen 1: Proyecto de Git Lab utilizado.

El proyecto está compuesto por:

- Un directorio en la ruta **app/controllers** en donde se define el módulo de la aplicación de angular y el controlador a evaluar.
- Una carpeta llamada **bower_components y otra llamada node_modules,** que permite el manejo de dependencias al proyecto web.
- La carpeta llamada **test/controllers**, se encuentra definida la prueba a realizarse.
- El archivo de configuración de Karma, **karma.conf.js**, donde se define el navegador donde se ejecutarán las pruebas. También permite establecer banderas de control, como lo es un observador que realiza las pruebas en el momento en que se modifica y se guardan cambios a archivos. Este archivo de configuración permite definir sobre cuáles archivos realizar las pruebas y sobré cuáles archivos no.
- El archivo **package.json** se encuentra todas las dependencias instaladas en el proyecto para asegurar el correcto funcionamiento del proyecto web.

Configuración de la prueba:



```
describe("probando controlador hello", function()
{
    beforeEach(module("app"));

    describe("TEST_controller", function()
    {
        var scope, ctrl;
        beforeEach(inject(function($rootScope, $controller)
        {
            scope = $rootScope.$new();
            ctrl = $controller("helloCtrl", {$scope:scope});
        }));

        it("debe estar definida una variable data", function()
        {
            expect(scope.data).toBeDefined();
        })
    })
})
```

Imagen 2: Código de la prueba.

La prueba se basa en verificar si un objeto está definido en el controlador, para ello se define en la carpeta de **TEST** en donde se indica lo siguiente:

Utilizando **JasmineJS**, definimos el TEST, en la línea número 5, para ello se utiliza el controlador denominada, "**helloCtrl**". Se pretende evaluar la existencia de objeto "data", que pertenece al objeto "**scope**" que es proporcionado mediante inyección de código.

Se utiliza la función "**toBedefined()**" para asegurarse que el objeto es evaluado en su definición.

En el controlador, el objeto se comentó en el propósito de detectar el error.

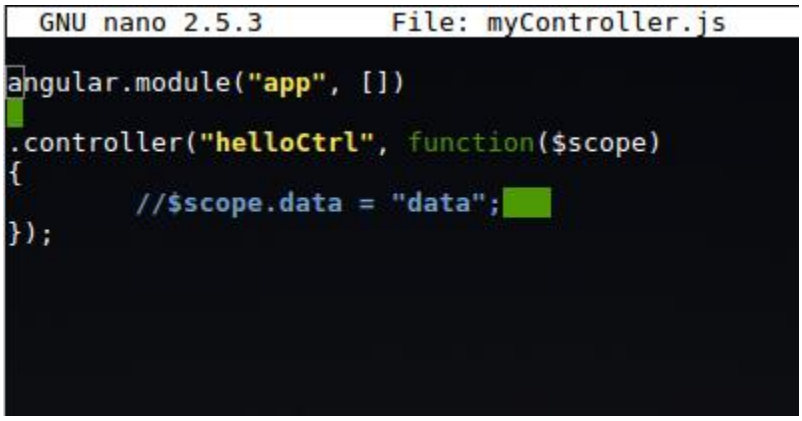


Imagen 3: Controlador MyController.js

Ejecución de la prueba:

Mediante el comando: **$ karma start,** se habilitarán las pruebas definidas y se mostrará el browser en cual se harán las aplicaciones, lo resultados se mostrarán en terminal.

Resultados de la prueba:

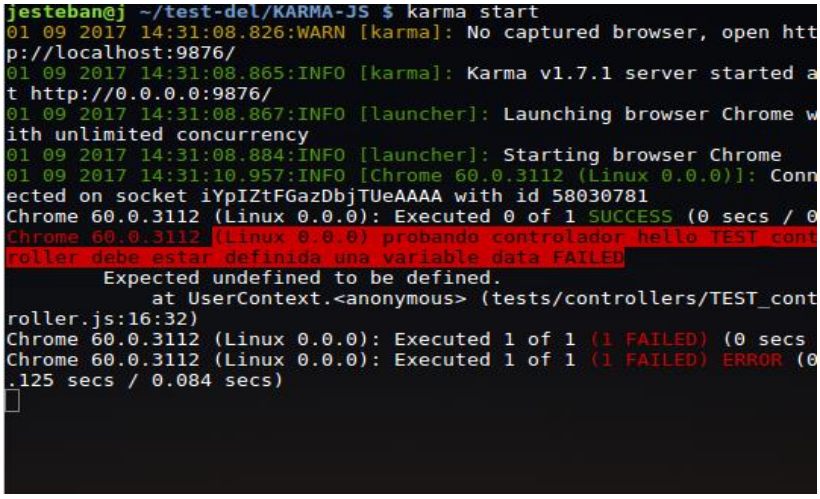


Imagen 4: Resultados de la prueba con la variable "scope.data" sin definir.

La franja roja indica que la variable debe estar definida, de ahí el error en donde de ejecuta la operación.

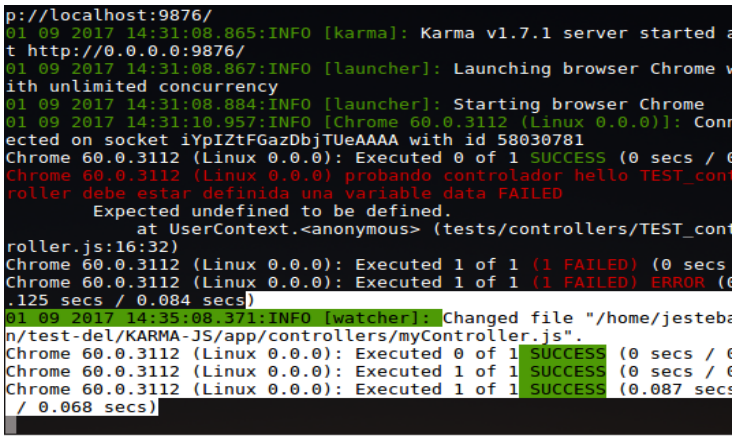


Imagen 5: Detección de cambios y confirmación de éxito en las pruebas.

Para el siguiente resultado se define en el código el valor de "scope.data" a un valor concreto, al ser guardado el archivo Karma detecta los cambios y realiza las pruebas.

Conclusiones

- La calidad del código mejorará, se reducirán los tiempos de depuración y la corrección de incidencias y por tanto el cliente estará mucho más contento porque la aplicación hace lo que él quiere que haga.
- Se reduce drásticamente el tiempo dedicado para detección de errores.
- Se pueden mezclar de manera uniforme casos de prueba de bajo y alto nivel, esta mezcla se puede dar en diversas combinaciones, o de otras diversas maneras, como empezar con los de alto nivel hasta llegar a bajo nivel.
- No hay un enfoque correcto o mejor que otro, la mejor manera de afrontar un problema es hacer una combinación de diferentes enfoques y depende totalmente de la situación que se desea resolver.
-

Bibliografía

Are Unit Tests Part of Your Team's Performance Reviews?. (2017). *InfoQ*. Retrieved 2 September 2017, from https://www.infoq.com/articles/unit-tests-performance-review

Bhushan, S. (2017). *Comparing High- and Low-Level Test Cases*. *Optimus Information Inc*. Retrieved 2 September 2017, from http://www.optimusinfo.com/high-low-level-test-cases/

Fowler, M. (2017). *bliki: UnitTest. martinfowler.com*. Retrieved 2 September 2017, from https://martinfowler.com/bliki/UnitTest.html

Getting Started. (2017). *Jasmine.github.io*. Retrieved 2 September 2017, from https://jasmine.github.io/pages/getting_started.html

introducción a karma js | The Bit Jazz Band. (2017). *Mmfilesi.com*. Retrieved 2 September 2017, from https://www.mmfilesi.com/blog/introduccion-a-karma-js/

Lund, U. (2017). *ECOOP 2002 — Object-Oriented Programming | SpringerLink*. *Link.springer.com*. Retrieved 2 September 2017, from https://link.springer.com/content/pdf/10.1007/3-540-47993-7.pdf#page=243

Sen, K., Marinov, D., & Agha, G. (2017). *CUTE*. Retrieved 2 September 2017, from

Tutorial: Crear y ejecutar pruebas unitarias en código administrado. (2017). *Msdn.microsoft.com*. Retrieved 2 September 2017, from https://msdn.microsoft.com/es-es/library/ms182532.aspx

What We Talk About When We Talk About Unit Testing. (2017). *InfoQ*. Retrieved 2 September 2017, from https://www.infoq.com/presentations/unit-testing-tips-tricks

Writing Good Unit Tests. (2017). *InfoQ*. Retrieved 2 September 2017, from https://www.infoq.com/news/2017/01/writing-good-unit-tests