

Universidad de Buenos Aires
FACULTAD DE INGENIERÍA
Departamento de Computación
75.10 – Técnicas de diseño
TRABAJO PRÁCTICO 2.2

ANÁLISIS DEL REQUERIMIENTO – DISEÑO DE UNA FRAMEWORK

Curso: 2013 – 2do Cuatrimestre

Turno: Jueves

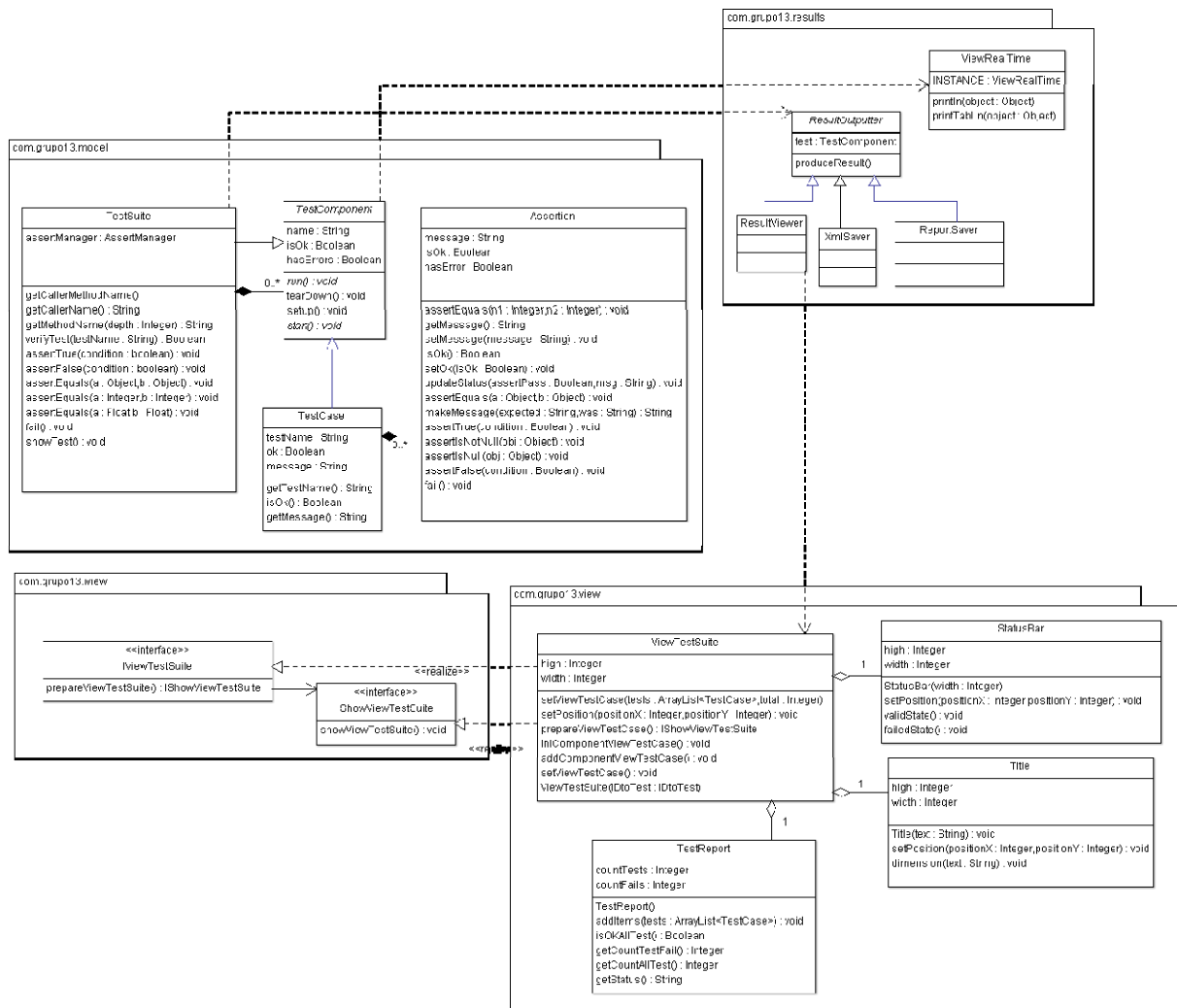
GRUPO Nº 13		
APELLIDO, Nombre	Número de Padrón	e-mail
Barrea, Ignacio	86225	ignacio@tictaps.com
Chavar, Hugo	90541	hechavar@gmail.com
Schmoll, Edward Erik	90135	erikschmoll@gmail.com
Fecha de inicio: 14-11-2013		
Fecha de Aprobación:		
Firma Aprobación:		

Observaciones:

Objetivo

Producir código basados en los principios de la programación orientada a objetos, que éste sea descriptivo sin necesidad de comentar cada línea, haciendo un desarrollo basándonos en las buenas prácticas con la ayuda de herramientas como GIT, GITHUB, MAVEN, entre otras.

Diagrama de Clases



Casos de Prueba

Contamos con tres casos:

1. Prueba Unitaria
 - a. Usando FrameWork JUNIT
2. Prueba Productiva
 - a. Un proyecto "Calculator" esta dispuesto a utilizar nuestra FrameWork TestCase

2.a.

1. Creación de los casos de prueba

CP 1

PRERREQUISITOS
Crear un Calculator y hacer la suma de 2 con 2
RESULTADO ESPERADO
La prueba debe finalizar correctamente

CP 2

PRERREQUISITOS
Crear un Calculator y hacer la multiplicación de 2 con 10
RESULTADO ESPERADO
La prueba debe finalizar correctamente

CP 3

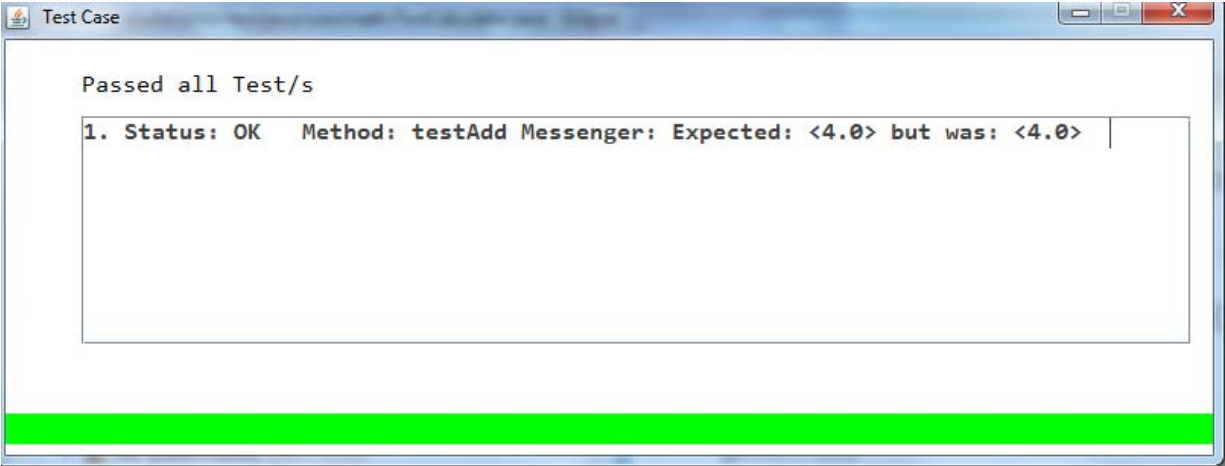
PRERREQUISITOS
Crear un Calculator y hacer la resta entre 5 y el resultado de hacer la multiplicación de 4 con 2
RESULTADO ESPERADO
La prueba debe finalizar correctamente

CP 4

PRERREQUISITOS
Crear un Calculator y hacer la división de 10 con 5 esperando como resultado un 0
RESULTADO ESPERADO
La prueba debe finalizar con una falla

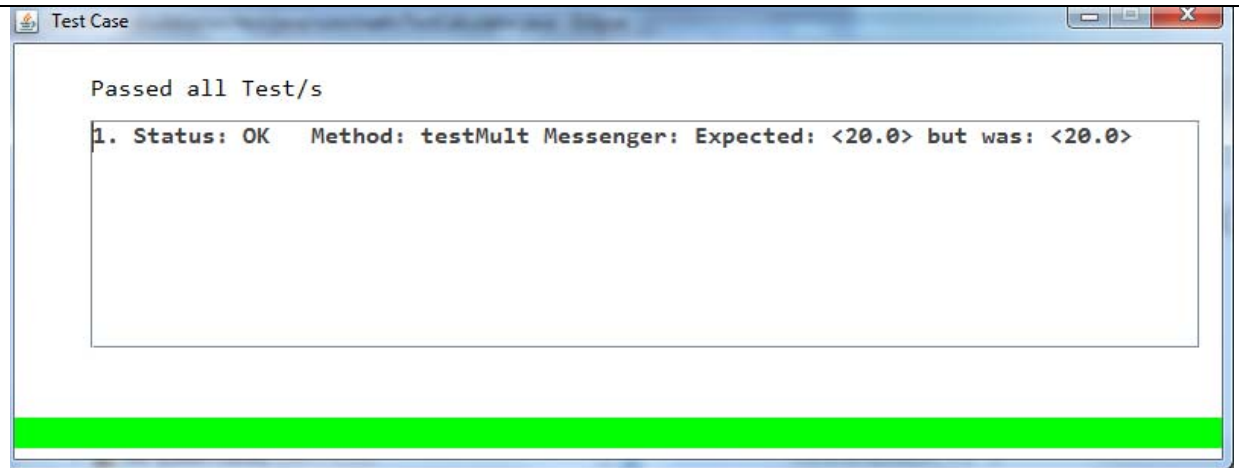
3. Implementación de los casos de prueba

CP 1

PRERREQUISITOS
Crear un Calculator y hacer la suma de 2 con 2
RESULTADO ESPERADO
<pre>... @Override public void run() { testAdd(); } public void testAdd() { Calculator calculator = new Calculator(); assertEquals(4.0, calculator.addAwithB(2, 2)); } ...</pre> 
Según lo especificado: La prueba debe finalizar correctamente
Prueba superada

CP 2

PRERREQUISITOS
Crear un Calculator y hacer la multiplicación de 2 con 10
RESULTADO ESPERADO
<pre>@Override public void run() { testMult(); } public void testMult(){ Calculator calculator = new Calculator(); assertEquals(20.0, calculator.mulAwithB(2, 10)); }</pre>



Según lo especificado: La prueba debe finalizar correctamente

Prueba sueprada

CP 3

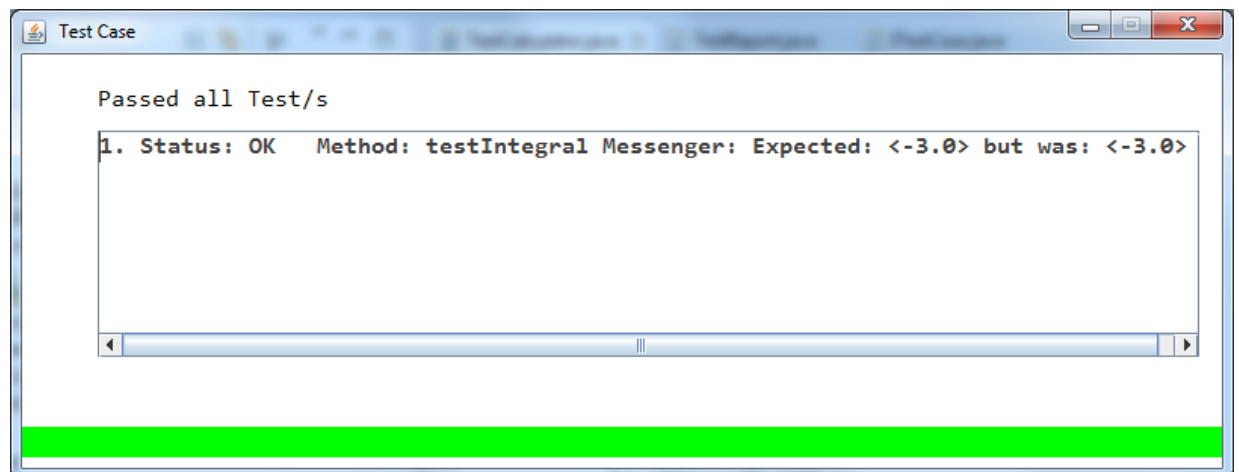
PRERREQUISITOS

Crear un Calculator y hacer la resta entre 5 y el resultado de hacer la multiplicación de 4 con 2

RESULTADO ESPERADO

```
@Override
public void run() {
    testIntegral();
}

public void testIntegral() {
    Calculator calculator = new Calculator();
    calculator.rememberResul(calculator.mulAwithB(4, 2));
    assertEquals(-3.0, calculator.minusAwithAccumulator(5));
}
```



Según lo especificado: La prueba debe finalizar correctamente

Prueba sueprada

CP 4

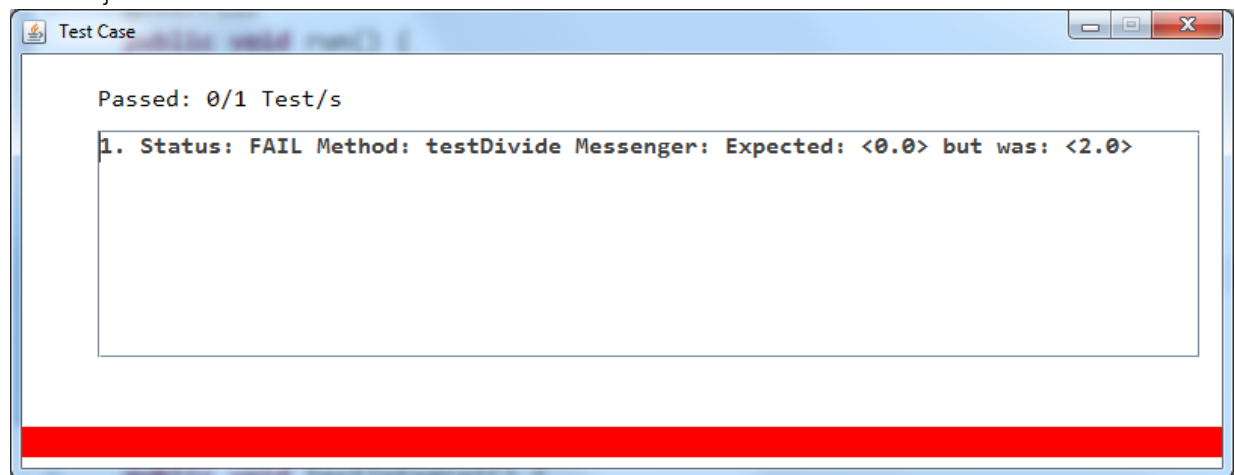
PRERREQUISITOS

Crear un Calculator y hacer la división de 10 con 5 esperando como resultado un 0

RESULTADO ESPERADO

```
@Override
public void run() {
    testDivide();
}

public void testDivide(){
    Calculator calculator = new Calculator();
    assertEquals(0.0, calculator.divideAwithBparts(10, 5));
}
```



Según lo especificado: La prueba debe finalizar con una falla

Prueba sueprada

Manual de usuario

Para utilizar el servicio que brinda este Framework usted debe crear una clase donde escribirá todos los métodos que desea probar y heredar de TestCase, la misma le obligará a usted a crear el método run, en este tiene que hacer los llamados de todos sus métodos que desea probar en la misma ejecución.

Usted va a poder aprovechar los siguientes métodos que le proporciona este Framework para validar los métodos de su clases a probar.

Metodos	Parámetros	Valor de retorno	Descripción
Start	N/A	void	Inicia el testeo
getCallerName	N/A	String	Obtiene el nombre del método que lo llamo
verifyTest	testName	boolean	Verifica si el método existe
assertTrue	Condition	Void	Valida si una expresión booleana es verdadera
assertFalse	Condition	Void	Valida si una expresión booleana es falsa
assertEquals	ObjA, ObjB	Void	Compara dos objetos del tipo OBJECT
assertEquals	EnteroA, EnteroB	Void	Compara dos enteros
assertEquals	DecimalA, DecimalB	Void	Compara dos decimales
Fail	N/A	Void	Genera una excepción
Setup	N/A	Void	Setea una única configuración para un set de test a probar. Opcional
tearDown	N/A	void	Se ejecuta al finalizar el TestCase para finalizar y limpiar las instancias necesarias. Opcional

Método RUN, en este método se van a llamar todos los metodosTest que querramos probar en el mismo conjunto de ejecución:

Por ejemplo

```
public class TestCalculator extends TestCase{

    @Override
    public void run() {
        testAdd();
    }

    public void testAdd() {
        Calculator calculator = new Calculator();
        assertEquals(4.0, calculator.addAwithB(2, 2));
    }

}
```

Métodos Setup y TearDown pueden ser utilizados para inicializar instancias antes de ejecutar los tests, y limpiar posteriormente en caso de ser necesario.

Por Ejemplo

```
@Override
public void setup(){
    Calculator calculator = new Calculator();
}

@Override
public void run() {
    testAdd();
}

public void testAddOnePlusTwo() {
    assertEquals(3.0, calculator.addAwithB(1, 2));
}

public void testAddOnePlusOne() {
    assertEquals(2.0, calculator.addAwithB(1, 1));
}

@Override
public void tearDown(){
    calculator = null;
}
```

Responsabilidades de Clases:

- **TestSuite:** Clase de la cual debe heredar el cliente para poder usar el framework de tests. Permite definir metodos `setup()` y `tearDown()`. Se debe redefinir el metodo `run()` con la lista de tests a correr.
 - Puede incluir `TestCases` u otros `TestSuites`, para esto se usa el patron Composite.
- **TestComponent:** Representa el componente generico del patron Composite, de esta clase heredan `TestSuite` y `TestCase`
- **Assertion:** Determina si una evaluación es verdadera, guarda el resultado, y, en caso de que haya fallado la evaluación almacena un mensaje explicativo.
- **TestCase:** Almacena información de los tests individuales que definió el cliente y que son ejecutados dentro del método `run()` de `TestSuite`. Almacena una lista de `Assertion` que son las ejecuciones dentro del test.
- **ViewTestSuite:** Es la vista encargada de recibir el array de resultados y los muestra en una interfaz grafica.

Export de resultados en archivo TXT:

El Cliente ademas de ver los resultados en la Interfaz Gráfica, tiene la opción de guardarlos en un archivo TXT. Para esto solo tiene q llamar al metodo `saveTestResults()` luego de llamar al `start()`.

El archivo de salida tiene el Log con todos los resultados, y además la cantidad de Tests ejecutados, la cantidad de Failures y de Errors.

Automaticamente creara una carpeta llamada "testLogs" donde se creara un archivo con los resultados de cada ejecución.

El nombre del archivo es `testResult` + un time stamp para que sea único. Además facilita la identificación del log correcto.

Ejemplo de uso de Regex:

Al utilizar la clase heredada de `TestSuite`, se puede settear un String conteniendo una Expresion Regular, que se usara para validar el nombre del test a ser ejecutado.

Si el nombre concuerda con la Expresion Regular el test será agregado al conjunto de pruebas, de lo contrario será ignorado.

Se puede settear la `Regex` a usar mediante el metodo "`setRegex(String)`", y debe ser configurado previamente a llamar al `start()`.

Por ejemplo se podría utilizar para correr solo los tests que incluyan la palabra "Null" de la siguiente manera

```
public static void main(String[] args) {
```

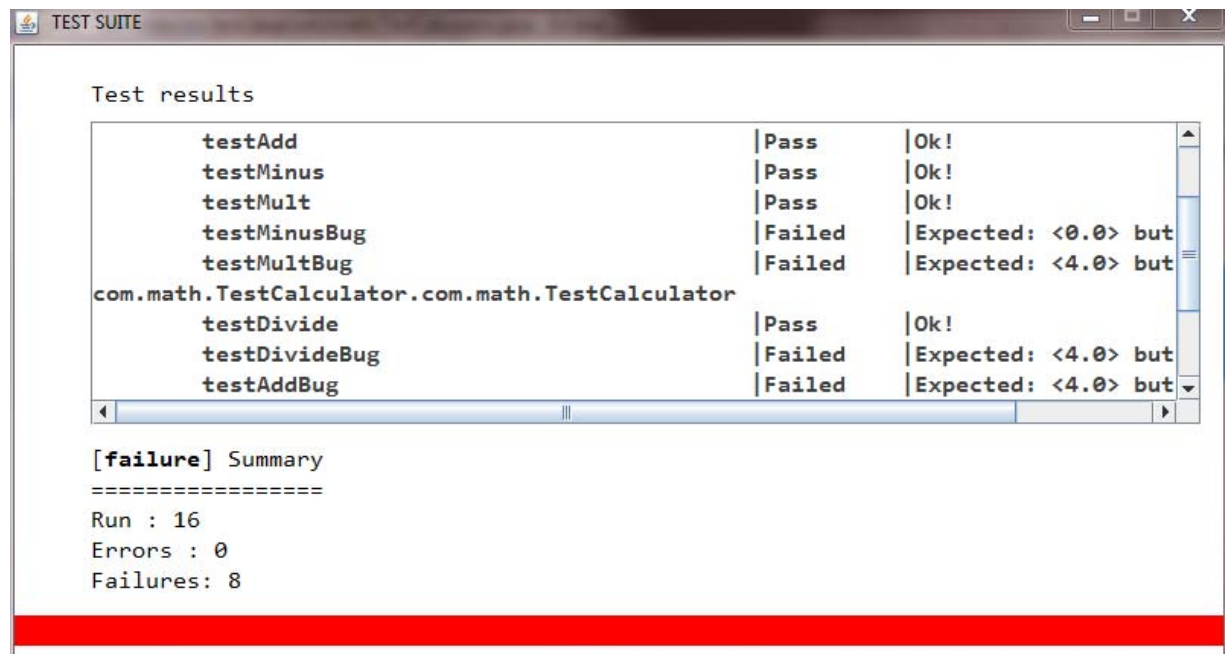
```
TestSuiteCliente someTest = new TestSuiteCliente ();  
  
someTest.setRegex("(.*)(Null(.*))");  
  
someTest.start();  
  
someTest.showTest();
```

Ejemplo de uso de TestSuites anidadas:

El cliente deberá generar las suites y anidarlas de la siguiente manera:

```
TestCalculator tc = new TestCalculator();  
TestCalculator tc2 = new TestCalculator();  
//Anidamiento de suites  
tc.addTestComponent(tc2);  
tc.start();
```

En la pantalla gráfica verá diferenciados ambas suites, con sus tests dentro:



Uso de Tags:

En la implementación de cada TestCase (método que representa un test del usuario), el usuario puede asignar uno o más tags para su futura identificación. Se pueden agregar individualmente inline utilizando el método:

```
tagTest("Nombre-Tag");
```

O a través de una la testSuite que lo contiene de la siguiente manera:

```
testSuite.tagTest("Nombre-test", "Nombre-tag");
```

A su vez, cada TestSuite puede elegir un sub set de tags a ejecutar dependiendo de sus Tags. Puede ejecutar todos los tests correspondientes a un Tag, o varios Tags a la vez. Para esto, el TestSuite tiene un metodo llamado addTagToExecute();

Ejemplo de Uso:

```
public void exampleMultipleTagTest() {  
    tagTest("INTERNET");  
    tagTest("SLOW");  
    assertTrue(true);  
}  
public static void main(String[] args) {  
    TestSuite oneTestSuite = new TestSuite1();  
    oneTestSuite.addTagToExecute("CONFIGURABLE");  
    oneTestSuite.tagTest("assertTrueTest", "CONFIGURABLE");  
    oneTestSuite.start();  
    someTest.showTest();  
    someTest.saveTestResults();  
}
```

Uso de Skip

Cada TestCase puede ser salteado sin necesidad de eliminar su llamada del main(). En cualquier momento dentro del TestCase se puede invocar al metodo skip() y entonces este método será ignorado en la ejecucion del TestSuite.

Ejemplo de Uso:

```
public void verifyTestThatExecuteFailReturnsFalse() {  
    skip()  
    assertFalse(test.verifyTest("exampleFailTest"));  
}
```

Análisis de patrones utilizados

Los patrones que utilizamos son Singleton, Strategy y Composite.

Composite para armar la estructura anidada de TestSuites y TestCases.

Strategy para la forma de mostrar los datos en bruto.

Singleton para mostrar los datos en real-time a medida que los tests son ejecutados.