FIUBA - 7510
Técnicas de diseño

Trabajo práctico 2: Unit test
2do cuatrimestre, 2013

Alumnos:

Federico Piechotka, padrón 92126
Mail: fpiechotka@gmail.com

Marco Lotto, padrón 91967
Mail: marcol91@gmail.com

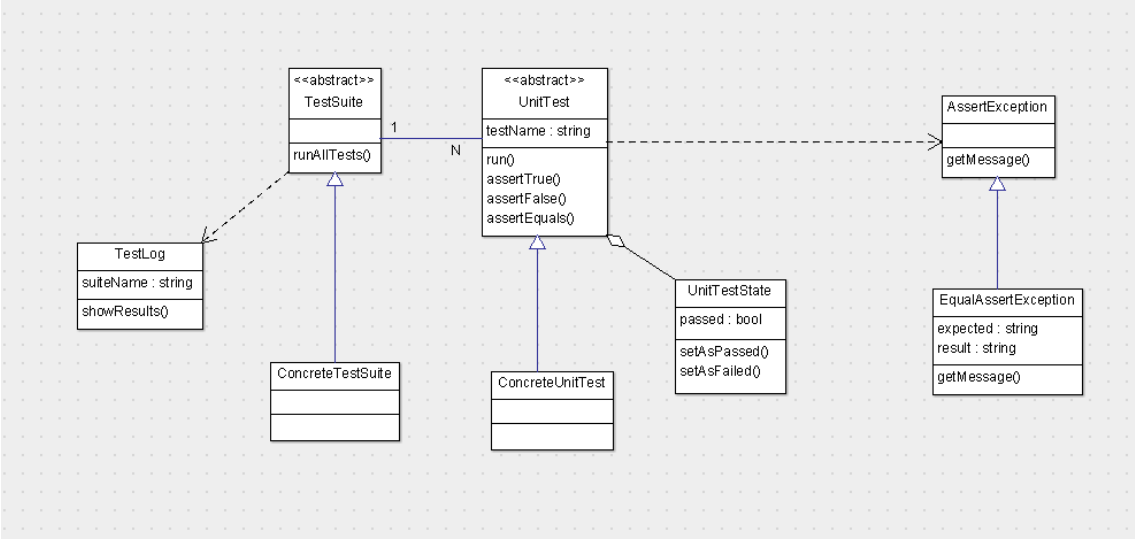
Tomás Reale, padrón 92255
Mail: tomasreale@gmail.com

Entrega 1

Introducción

La idea de este trabajo práctico consistió en el diseño e implementación de un framework pensado para correr test unitarios y a partir del resultado de los mismos mostrar un reporte.

Solución planteada



Hacemos una breve descripción de la solución planteada:
Los test unitarios estarán representados por la clase abstract UnitTest. En el método run de esta clase será donde se ejecute cada test, provocando que dicho test pase al estado failed o al estado passed (queriendo decir que el test funcionó correctamente).

Además, los UnitTest proveen distintos asserts para que aquellas clases que hereden de esta clase abstracta puedan utilizar, al igual que en los distintos frameworks conocidos de unit testing.

En caso de que algún assert falle, se lanzará una AssertException o una EqualAssertException (para assertEquals), para poder detectar los fallos en las corridas.

En el TestSuite, es donde tendremos la colección de todos los tests que necesitamos correr, con el objetivo de que cuando se desee ejecutar la corrida se llame al método runAllTests del TestSuite, que a su vez ejecutará cada test que tenga asignado. Luego de eso, mostrará los resultados al usuario por medio del TestLog.

Mediante el TestLog, llevamos a cabo la transmisión de los resultados hacia el usuario que está corriendo los tests. Sirve para que una vez que se hayan ejecutado todos y cada uno de los tests, se le pueda mostrar al usuario los

tests que fallaron, los que funcionaron correctamente, y la razón por la que fallaron aquellos que lo hicieron.

Las clases concretas que se ven en el diagrama son las que hereden de TestSuite y de UnitTest, que serán las que al fin y al cabo hagan uso efectivo de estas clases y permitan ejecutar tests concretos para luego conocer el resultado de su ejecución.

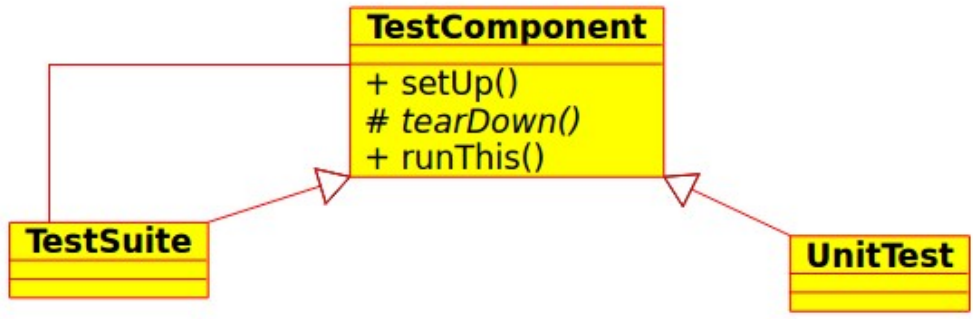
La razón por la que TestSuite es una clase abstracta de la cual extiende el usuario del framework, es porque seguramente se requiera agrupar diversos tests para correrlos todos juntos. Ahora muchas veces es conveniente no correr todos los tests juntos sino separarlos en diversas suites, es decir un usuario extiende TestSuite en tantas subclases como el crea conveniente, agrupando en cada uno los tests que el cree que deberían correr juntos.

Observar que cada UnitTest es un único test, a diferencia de por ejemplo jUnit que permite definir en cada TestCase mas de un test unitario.

Entrega 2

Solución planteada

Para afrontar los cambios en la nueva entrega del tp, planteamos una nueva estructura para los TestSuite y UnitTest. Esta puede observarse en el siguiente esquema:



En ella se aprecia que ahora tanto el TestSuite como el UnitTest extienden de una clase abstracta denominada TestComponent. Además esta contiene otros elementos TestComponent, por lo que respeta un patrón composite.

Con esta estructura estamos posibilitando que existan jerarquías enteras de TestSuites, unos dependientes de los otros, y que además cada uno tenga sus propios UnitTest.

Primero se implementó el setUp(Context) y el tearDown(Context). Estos se definieron en TestComponent y opcionalmente pueden ser redefinidos en la implementación particular del TestSuite y UnitTest. Estos dos métodos permiten ejecutar código y definir variables en el contexto antes y después del test (o test suite).
Todo el código que se declara en el contexto, sera independiente para cada test, es decir, aunque yo modifique dentro de un test una o mas variables del contexto, estos cambios solo serán efectivos dentro del alcance del mismo test, mientras que otros tests no verán estos cambios.

Tanto setUp(Context), tearDown(Context) y runThis(context) podrán ver el mismo contexto.
Dentro de cada uno se puede pedir contexto.getElement() y contexto.setElement(), para consueguir y setear el valor de una variable respectivamente. Al hacer setElement, si la variable no existe se crea en el contexto, sino se actualiza su valor.
Cada contexto hace un merge con el contexto de los TestSuite padres, es decir un TestSuite hijo de otro, o un UnitTest verán todo lo que ellos definan en el contexto y todos lo que sus padres hayan definido. Si hay una variable con el mismo nombre que se define en el padre y en el hijo, tendrá mas peso la declaración del hijo.

La forma en que se maneja el contexto es recordando para cada nombre de variable, quien fue el TestComponent que la definió. De esta forma cuando yo hago un contexto.getElement(“variable1”), si “variable1” fue definida en el padre, puede haber sido modificada por un test anterior, por lo que estará sucia.
Identificando esto, sabemos que fue el padre el que la definió por lo que antes de devolver valor de la misma, ejecutamos el setUp() del padre (o de la clase que la hubiese definido). La variable volverá a su estado original, y es así que logramos que cada variable sea independiente en cada test.
Todo esto es totalmente invisible para el usuario del framework, y tiene la ventaja que solo se recalculan las variables que sean necesarias para ese test.

Para los estados de los UnitTests , usamos el patrón State.
Nos dimos cuenta que había un code smell al ver ifs idénticos en la mayoría de los métodos. Se cuenta con un State Passed, Failed,NotTested. Estos encapsulan el comportamiento esperado del TestState para cada caso.