**TECNICAS DE DISEÑO (75.10) – FIUBA**

**TRABAJO PRÁCTICO N° 2.2**

**INTEGRANTES**

|  |  |  |  |
| --- | --- | --- | --- |
| **PADRON** | **NOMBRE Y APELLIDO** |  | **MAIL** |
| **89630** | **Martín Andrés Lucero** |  | [**luceromartinandres89630@gmail.com**](mailto:l%75c%65rom%61%72%74%69%6e%61%6ed%72%65s%38%39%36%330@%67ma%69l%2e%63o%6d) |
| **85828** | **Natalia Merino** |  | **nati.merino@gmail.com** |
| **84441** | **Leandro Pellegrino** |  | **lrpellegrino@gmail.com** |

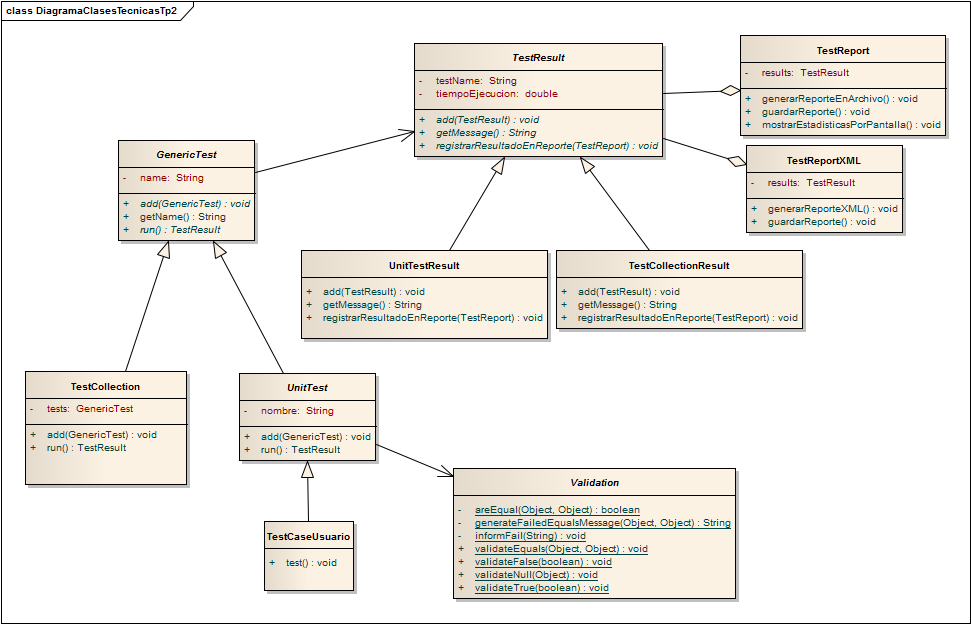
**Índice**

Diagrama de Clases 2

Diseño del Framework 2

Falencias del Diseño 3

# Diagrama de Clases



# \*La clase TestCaseUsuario no forma parte del framework. Su inclusión es a modo de ilustración

# Diseño del Framework

Partiendo de la base del diseño armado para la primer entrega se hizo un refactor de la relación contenedor – contenido entre TestCollection y UnitTest, utilizando el patrón Composite. Al utilizar este patrón los tests se ejecutan en forma recursiva, el usuario sólo ejecuta el run del objeto contenedor principal de tipo TestCollection, lo que desencadena la ejecución de todos los run de los TestCollection y UnitTest contenidos en él, donde cada padre sabe que Test ejecutar de acuerdo si alguno es skipeable o bien contenga los Tags, asi como tambien un determinad expresion regular. La necesidad de untilizar este patrón se nos presentó con el nuevo requerimiento de que se permita almacenar TestCollections dentro de TestCollections. El mismo patrón se utilizó para generar la estructura dónde se almacenan los resultados de los test.

Se mantuvo la clase utilitaria Validation, con todos métodos estáticos, de forma similar a la implementación de Assert de junit. La idea fue que sea una clase abstracta con todos métodos estáticos ya que no tiene estado (solo comportamiento -> clase utilitaria).

Al igual que para las entregas anteriores, para hacer un test, el usuario debe heredar de UnitTest y redefinir el método test(), en donde escribe su código de prueba. El framework se encarga de ejecutarlo e informar el resultado.

Para los diferentes tipos existentes se utilizo el patron Template Method, con el fin de poder modificar parte de su codificacion para lograr el objetivo deseado. Esto hace que cada test podamos definirle o setearle un run especifico a la hora de su corrida.

En el momento en que se ejecutan los tests se imprime por pantalla el resultado de las mismas, y se va generando una estructura de tipo TestResultCollection donde se almacenan objetos de tipo TestResult. Para generar el reporte, el usuario debe pasar por parámetro esa estructura al constructor de la clase TestReport e invocar al método generarReporteEnArchivo() de esta última.

Al igual que para las entregas anteriores, para hacer un test, el usuario debe heredar de UnitTest y redefinir el método test(), en donde escribe su código de prueba. El framework se encarga de ejecutarlo e informar el resultado.

# Falencias del Diseño

Al igual que para las entregas anteriores, el principal problema del diseño es que el cliente debe crear tantas clases que hereden de UnitTest como tests quiera hacer. Esto es así por el modelo de dominio pensado. El uso de herencia se justifica por lo dicho en el apartado anterior, pero esto no implica que sea la mejor solución para este trabajo práctico.

Con respecto al uso del patrón Composite, detectamos que la desventaja de usarlo es que tanto en la clase UnitTest como en UnitTestResult es que nos vimos obligados a implementar métodos que no se van a utilizar (como es el caso del método “add”). Sin embargo, consideramos que era mayor el beneficio obtenido con respecto a la ejecución de las pruebas, en forma polimórfica.

Algo similar sucede con el patron Template Method utilizado para el método run, donde la principal desventaja es que todos sus hijos necesitaban conocer todos los parámetros necesarios para su ejecución, eso no llevo a sobrecargar constructores evitando el pasaje de null.