**Índice**

[Enunciado 2](#_Toc372818456)

[Comentarios del código del Grupo 7 6](#_Toc372818457)

[Diseño y Código 6](#_Toc372818458)

[Tests 6](#_Toc372818459)

[Herramientas utilizadas 7](#_Toc372818460)

[Hipótesis 8](#_Toc372818461)

[Aclaraciones 8](#_Toc372818462)

[Clases nuevas 9](#_Toc372818463)

[TestRunStoreAccess 9](#_Toc372818464)

[TestRunStore 9](#_Toc372818465)

[MemoryTestRunStore 9](#_Toc372818466)

[TxtFileTestRunStore 9](#_Toc372818467)

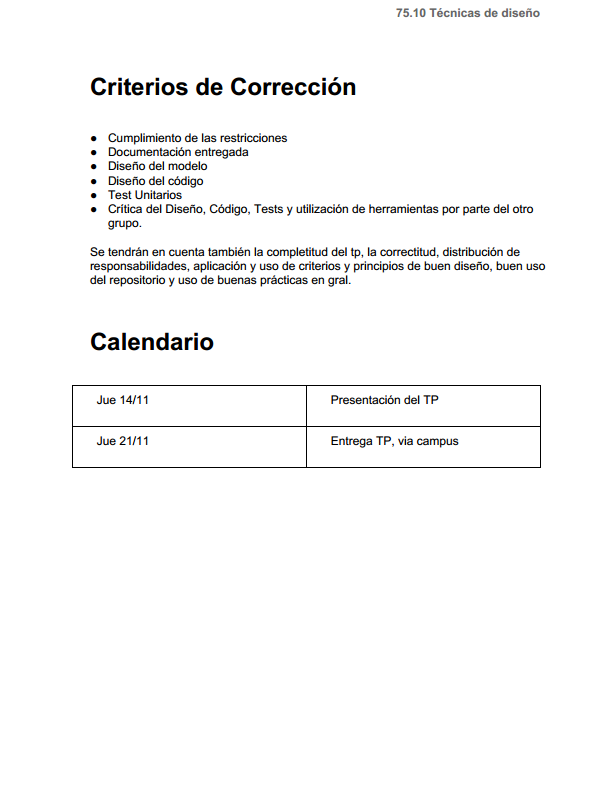
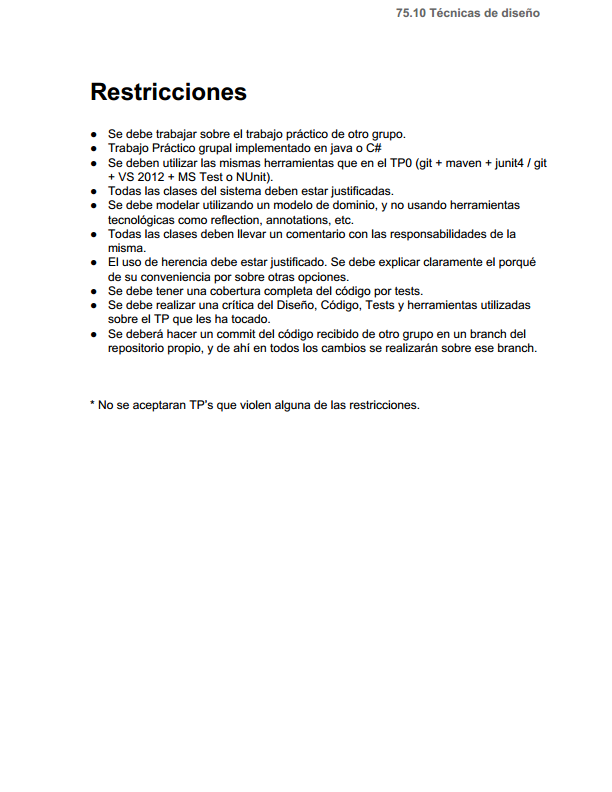
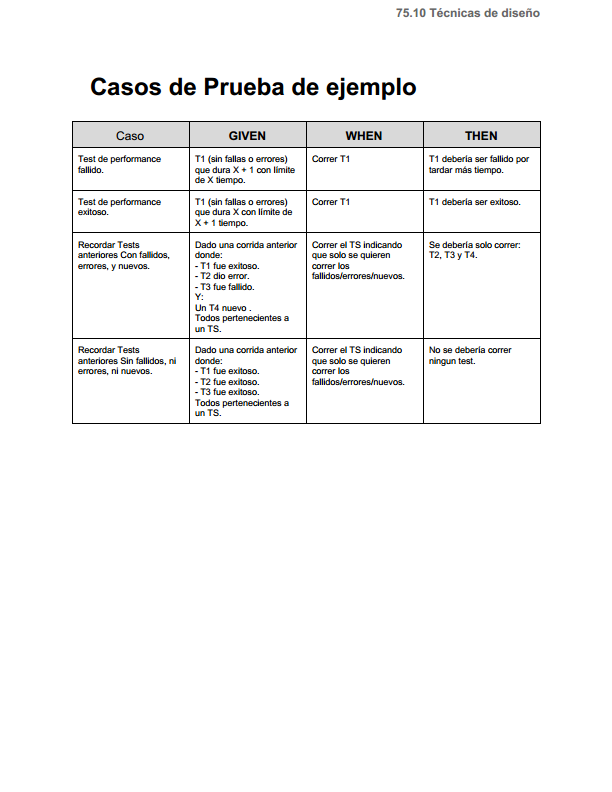
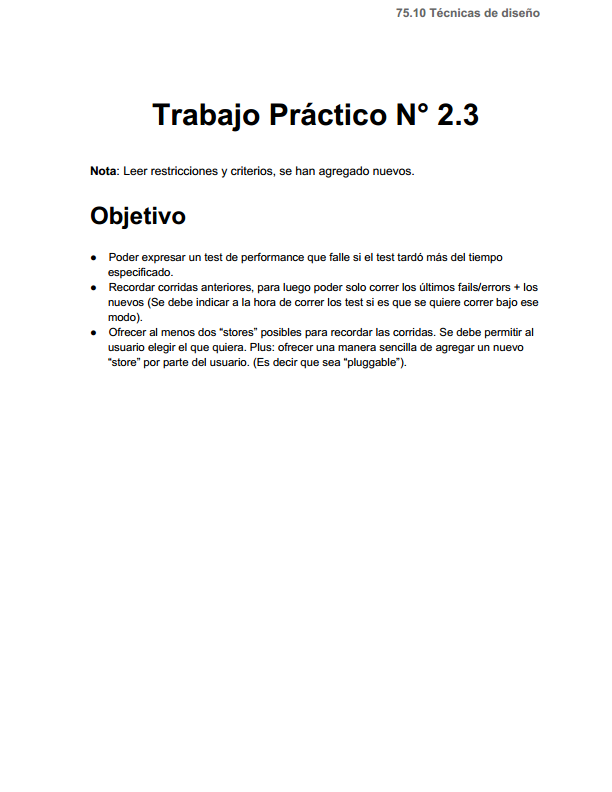
[Diagrama de clases 10](#_Toc372818468)

[Diagrama de clases del paquete nuevo: Store 10](#_Toc372818469)

[Diagrama de clases de Diseño 11](#_Toc372818470)

[Diagrama de clases de Dominio 12](#_Toc372818471)

Enunciado



Comentarios del código del Grupo 7

Diseño y Código

* Nos parece que confundieron *Error* con *Failure*. El *Failure* es cuando falla. El *Error* es un caso no esperado en la ejecución del test (por ejemplo cuando las variables para comparar no están seteadas). Esto no es un error, pero por convención se usa la otra “forma de nomenclatura”: *Error* cuando pincha, *Failure* cuando no se obtiene lo que se espera.
* El “Reporte por pantalla” es poco claro. Primero, cada informe en sí es complejo de entender. Después se mezclan los informes nuevos con los informes viejos. Esto hace que sea difícil ver el resultado.
* Nos da la impresión de que utilizaron muchas interfaces con la idea de generar “una especie de header” con todas las firmas de los métodos, y luego otro con los métodos implementados. Esto causó que haya una gran cantidad de clases, que en un primer momento, era difícil de leerlos (y daba la impresión de que era un proyecto grande). Pero tras leer y entender su código, vimos que estuvo muy bien la jerarquía y elección de paquetes para las clases.
* Esto es algo menor, pero cuando recibimos su TP, había muchos *warnings* en el proyecto. La gran mayoría se debían a *import* que no se usaban (esto tiene relación con lo dicho de de muchos paquetes y clases del párrafo anterior).
* Cuando recibimos su TP, también había un par de métodos comentados. Suponemos que esto fue porque llegaron muy justos de tiempo con la última entrega.

Tests

* Nos pareció poco claro crear un nuevo Test. Aunque una vez que se entiende, es fácil. Pero para eso lo mejor es ver un ejemplo, ya que con sólo ver el código se complica.
* Por algunas cosas dichas anteriormente (principalmente el orden de los paquetes y la jerarquía de las clases), vimos que a la hora de hacer muchos tests, hay poco código duplicado.

Herramientas utilizadas

* El Grupo 7 utilizaron las herramientas del lenguaje Java especificadas en el enunciado (git + maven + junit4), lo mismo que en nuestro TP.
* La diferencia es que ellos trabajaron en el sistema operativo Linux, mientras que nosotros en Windows. Esto no trajo problemas a la hora de intercambiar TP, ya que Java es multiplataforma.

Hipótesis

* Al momento de evaluar los tests, no se van a ejecutar 2 tests con el mismo nombre.
* Si se le cambian los datos (los valores a comparar por el “assert”) a un test que ya dio “OK”, eso no hace que el test se corra de vuelta (en la próxima ejecución).
* Un TestSuite cuyos resultados fueron todos "OK" no va a volver a correrse. Si quiere volver a correrse, debe crearse un nuevo TestSuite, o cambiarle el nombre.
* La clase **TestRunStoreAcccess** controla el acceso al **TestRunStore**. Es una mezcla entre *Singleton* y *Proxy*. Por defecto si se la quiere usar sin inicializar toma un “.txt”.
* El objetivo de recordar las corridas es para luego correr solamente lo que dio erróneo. Por esa razón, lo único que se graba en un archivo son los tests que dieron “OK”, que es lo que se debe ignorarse en la próxima corrida.

Aclaraciones

A la hora de correr los tests del JUnit (paquete “ar.fiuba.tecnicas.tetesteo.impl” en “src/test/java”), se debe correr cada clase individualmente. Esto se debe a la cualidad *Singleton* de **TestRunStoreAcccess**, que necesita que se corra los .java por separado, y no todos juntos (por ejemplo, parándose en un carpeta superior, y correr JUnit).

Clases nuevas

TestRunStoreAccess

Clase que es una combinación entre *Singleton* y *Proxy* para controlar el acceso a la utilización del **TestRunStore**. Si no se especifica un **TestRunStore**, por defecto se utilizará **TxtFileTestRunStore**.

TestRunStore

Interfaz que deben cumplir todos los tipos de “Store”.

Por cómo **TestRunStoreAccess** utiliza a su “Store”, con esta jerarquía de “Store” se aceptarían más tipos de “Store”. Simplemente se crea una nueva clase de “Store” deseado, y se le indica a **TestRunStoreAccess** que use ese “Store”.

MemoryTestRunStore

Clase que implementa **TestRunStore**. Este “Store” guarda la información de las ejecuciones de test en memoria.

Cabe destacar que en una nueva corrida del programa, se pierden los datos de las ejecuciones anteriores.

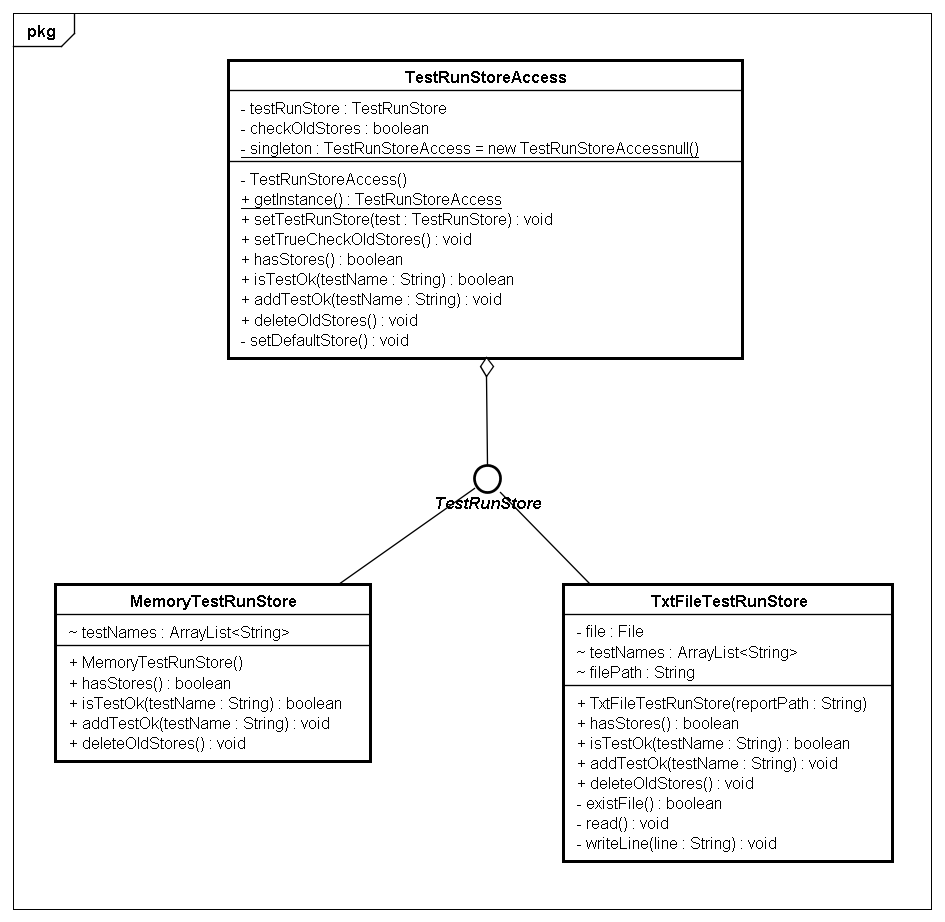
TxtFileTestRunStore

Clase que implementa **TestRunStore**. Este “Store” guarda la información de las corridas en memoria. Este “Store” guarda en un archivo de texto plano con extensión ‘.txt’ la información de las ejecuciones de test.

En una nueva corrida del programa, se puede recuperar información de ejecuciones pasadas.

Diagrama de clases

Diagrama de clases del paquete nuevo: Store



En el resto de los paquetes y clases, aunque se haya modificado algunas líneas de código para agregar las funcionalidades nuevas, se mantuvieron las mismas relaciones, jerarquías y usos. Por lo tanto los diagramas quedan igual a la versión anterior del TP.

Diagrama de clases de Diseño

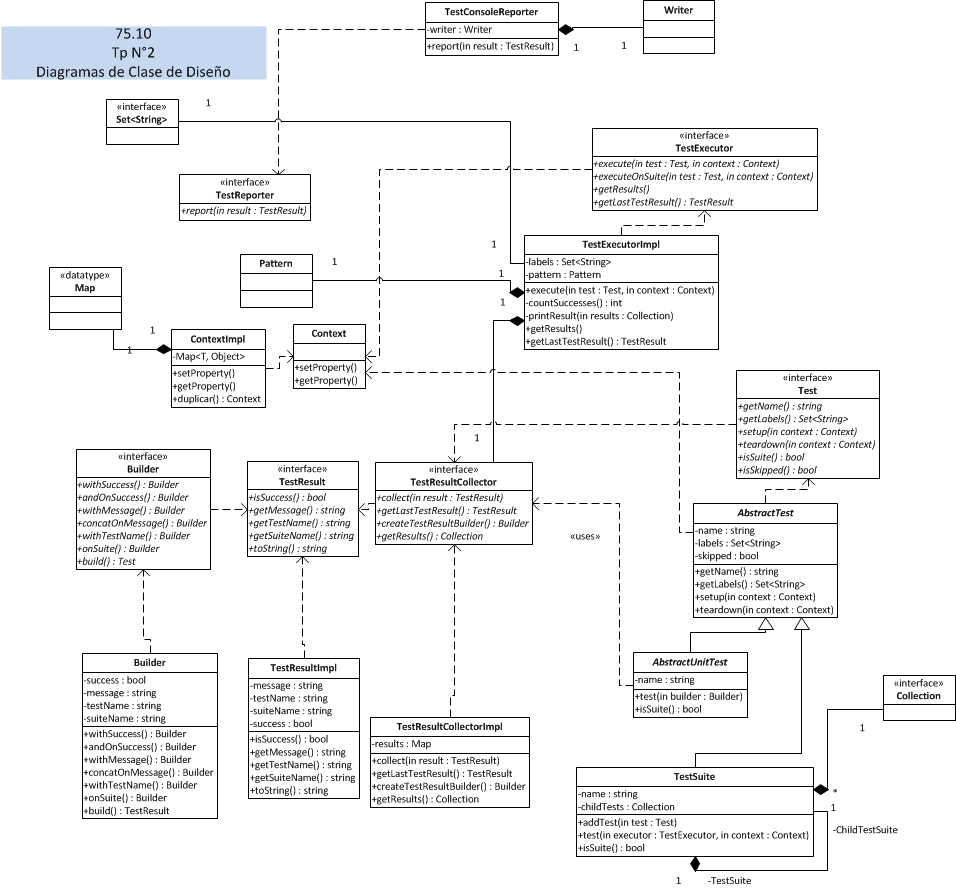


Diagrama de clases de Dominio

