2do Cuatrimestre 2013

Facultad de Ingeniería
Universidad de Buenos Aires

TÉCNICAS DE DISEÑO (75.10)

TRABAJO PRÁCTICO 2.2

Integrantes:

Farina, Federico	90177	*federicojosefarina@gmail.com*
Bedecarats, Iván	87024	*igbedecarats@gmail.com*
Olmos Gil, Manuel	89887	*manuolmos88@gmail.com*

Indice

Enunciado

Objetivo:

El objetivo principal del TP es:

- Taggear tests (un string, ejemplo: FAST, SLOW, DB, INTERNET, etc)
- Poder agregar más de un tag a un test.
- Permitir desactivar un test con SKIP.
- Poder ejecutar test pertenecientes a un tag particular.
- Poder ejecutar test pertenecientes a varios tags.
- Poder ejecutar test pertenecientes a varios tags y/o que su nombre de test case coincida con una expresión regular y/o que su nombre de test suite coincida con una expresión regular.
- Tomar el tiempo a todos los test para poder agregarlos en reportes.
- Reporte textual por línea de comando. (Utilizar formato de la entrega anterior).
- Que el reporte textual por línea de comando anterior sea progresivo en real time (a medida que vaya teniendo resultados los muestra).
- Implementar un reporte xml siguiendo el siguiente XSD:

https://svn.jenkins-ci.org/trunk/hudson/dtkit/dtkit-format/dtkit-junit-model/src/main/resources/com/thalesgroup/dtkit/junit/model/xsd/junit-4.xsd

Restricciones:

- Trabajo Práctico grupal implementado en java o C#
- Se deben utilizar las mismas herramientas que en el TP0 (git + maven + junit4 / git + VS 2012 + MS Test o NUnit)
- Todas las clases del sistema deben estar justificadas.
- Se debe modelar utilizando un modelo de dominio, y no usando herramientas tecnológicas como reflection, annotations, etc.
- Todas las clases deben llevar un comentario con las responsabilidades de la misma.
- El uso de herencia debe estar justificado. Se debe explicar claramente el porqué de su conveniencia por sobre otras opciones.
- Se debe tener una cobertura completa del código por tests.
- Se pide ademas de tener los test unitarios junit/nunit, replicar tests utilizando el framework desarrollado en el TP.

Manual de usuario

Como empezar

La herramienta no requiere interacción con el humano para evaluar los resultados, y es facil de ejecutar muchas pruebas al mismo tiempo. Para testear algo esto es lo que se debe hacer:

- Crear una instancia de UnitTest
- Sobreescribir el método test.
- Cuando quieras evaluar un valor, usa los Asserts provistos por el framework.

Los Test Suites

Cada UnitTest puede estar contenido en un TestContainer de manera de agruparlos de acuerdo al requerimiento funcional que se esté probando.

El fixture o Contexto

Las clases que hereden de la clase UnitTest o de TestContainer podrán sobrescribir el método setUp() y el método tearDown(). Al hacerlo podrán tener acceso al fixture o contexto del test a ejecutarse. El contexto es un mapa cuya clave es un String que representa el nombre del campo a guardar en él, y el valor un Object para guardar cualquier objeto que se necesite usar en la ejecución del Test. De esta forma, en el método en el setUp() podemos agregar pares de <clave,objeto> para ser usados en la implementación del método test(). Luego en el método tearDown() podremos, por ejemplo, liberar los recursos que hayamos guardado en el contexto. El contexto es particularmente útil porque puede definirse en un TestContainer para luego ser compartido por todas las clases que implementen a Test y estén contenidas en él.

Tags y expresión regular para el nombre de los Tests

Toda clase que implemente a Test podrá tener asociado Tags, que son String que identifican y etiquetan a los Tests. Cuando el TestRunner corra los Tests que tiene asignado, se le puede decir que corra aquellos que posean los Tags que reciba por parametro. De esta forma, se puede filtrar qué Tests correrá el TestRunner.
A su vez, se le puede agregar una expresión regular regular al TestRunner para que solo corra aquellos Tests cuyo nombre cumpla con dicha expresión regular. Asi, se cuenta con una segunda manera de filtrar Tests.
Los tags y la expresión regular no son excluyentes, con lo cual se puede hacer que un TestRunner corra Tests por ambos métodos de filtro al mismo tiempo.

Ejecutar los tests

Lo primero que debemos hacer para utilizar la herramienta es agregar el jar como dependencia al proyecto sobre el que estemos trabajando.
Recomendamos crear un nuevo package **src/test/java** donde se deberá incluir las tests que se quieran ejecutar. Se puede ver abajo un ejemplo de ejecución.

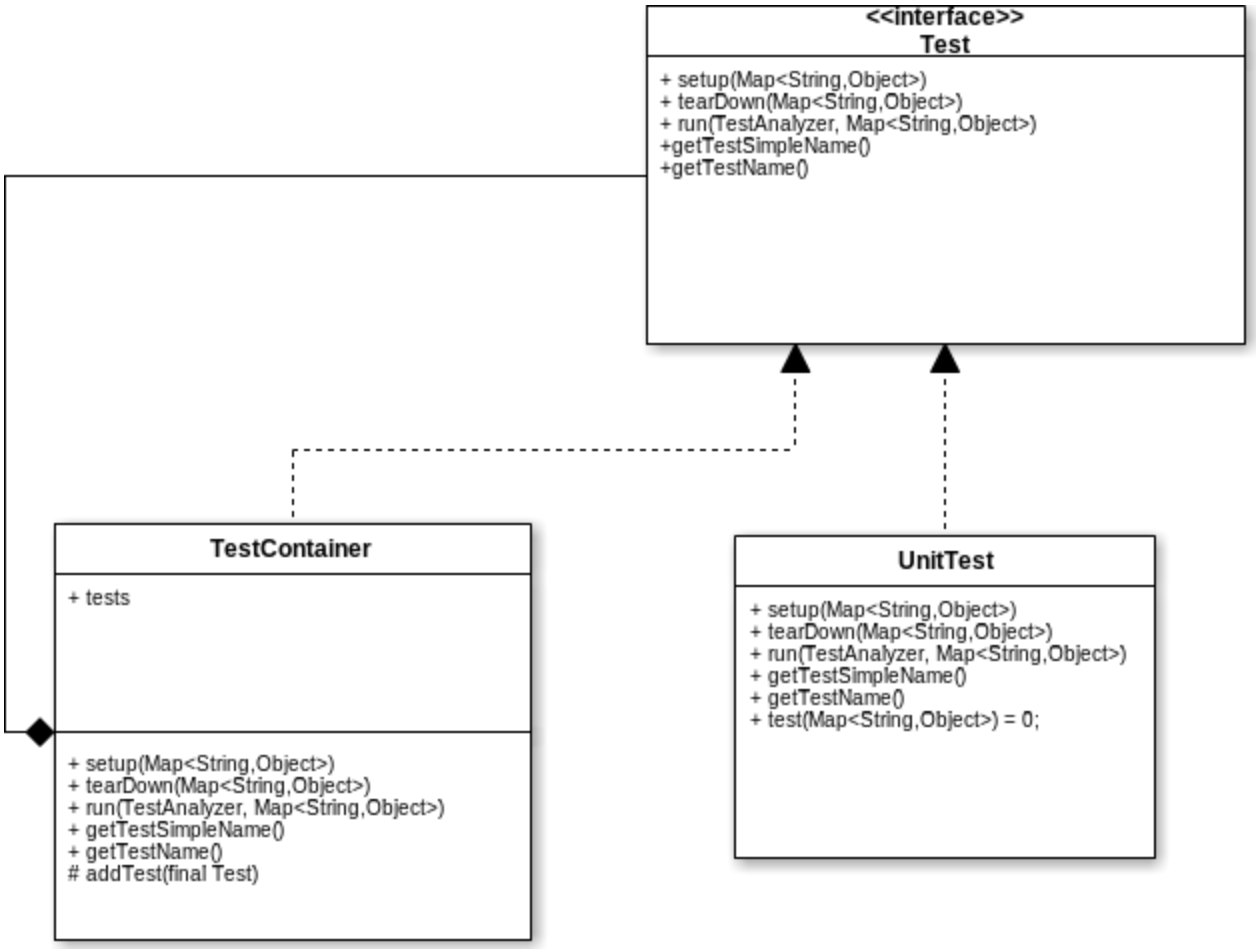
```
import fi.uba.ar.unitester.framework.TestRunner;

public class Tester {
	public static void main (String[] args) {
		TestRunner testRunner=new TestRunner();
		testRunner.addTest(new OneTest());
		testRunner.addTest(new AnotherTest());
		testRunner.execute();
	}
}
```

Lo cual generará un reporte de cada prueba que se haya corrido y sus resultados.

Patrones utilizados

1. **Container Pattern:** La clase TestContainer es una clase contenedora de implementaciones de la interfaz Test. Como a su vez la clase TestContainer implementa dicha interfaz, no solo podrá almacenar instancias de la clase UnitTest sino que también podrá contener otros TestContainers. De esta forma se puede tener un árbol o grafo de Tests a ejecutar a partir de un TestContainer.



2. **Template Pattern:** La clase abstracta UnitTest le da una implementación al método run() de la interfaz Test. Esta clase deja sin implementar almétodoo test() de esa interfaz, para que que las clases que hereden de ella lo implementen a test() como lo encuentren necesario. De esta forma, el método test() es una plantilla para ser definida por el cliente que será utilizada de un forma establecida por el framework.

```
public final void run(final TestAnalyzer analyzer,  final Map<String, Object> context) {
  analyzer.start(this);
  try {
    setup(context);
    test(context);
    analyzer.addPassed(this);
  } catch (AssertionFailError e) {
    analyzer.addError(this, e);
  } catch (Throwable e) {
    analyzer.addFailure(this, e);
  }
  tearDown(context);
}
```

3. **Strategy Pattern:** Hay varios strategy pattern implementados en el TP, los más importantes son los que definen la estrategia sobre como cerrar el handler responsable de generar el reporte y la estrategia utilizada para generarlo.

El reporter està hecho de esta forma:

```
public abstract class Reporter {

        /**
         * Writer to handle output
         */
        protected PrintWriter writer;

        /**
         * Reporter strategy
         */
        protected ReporterStrategy reporter;

        /**
         * Report closer strategy
         */
        protected ReporterCloserStrategy closer;


        /**
         * Close the writer
         */
        public void close() {
            closer.close(writer);
        }

        /**
        *  Create the output
        */
        public void generateTestReport(final TestAnalyzer analyzer) {
            reporter.createReport(analyzer, writer);
        }

      …..
}
```

Y luego se tienen las siguientes clases para definir las distintas maneras de hacer lo mismo, en este caso generar y cerrar un reporte:

- **ConsoleAndTextFileReporterStrategy**
  Salida para los reportes hechos por consola y archivos de texto.

- **XmlReporterStrategy**
  Salida para los reportes hechos en xml.

- **EmptyCloserStrategy**
  En los casos en el que la salida del reporte es por consola, el handler está asociado con la salida estándar por lo que no se debe cerrar.

- **FileCloserStrategy**
  En los casos que se usa un archivo como salida este es el Closer adecuado.

4. **Builder Pattern:**

Usamos este patròn para generar las diferentes partes que se asocian con un reporte. Las partes son :

1. Un writer que maneja la salida
2. Un Reporter que define la manera en que se imprime la salida
3. Un closer que define la manera de cerrar el handler asociado al writer..

De esta forma todos los reportes deben definir los 3 métodos siguientes:

```
public abstract class Reporter {

        public final void createReporter(){
            buildWriter();
            buildReporter();
            buildCloser();
            Validate.notNull(writer,"Writer can not be null");
            Validate.notNull(reporter,"Reporter can not be null");
            Validate.notNull(closer,"Closer can not be null");
         }

         protected abstract void buildWriter();
         protected abstract void buildReporter();
         protected abstract void buildCloser();
         ….
}
```

De esta forma la clase Reporter trabaja solo contra interfaces y no contra clases concretas por lo que tengo un buen uso del principio de inversión de dependencias.

Luego para generar un reporte defino los pasos de la siguiente forma:

```
public abstract class ReportGeneratorAbsractFactory {

   /**
    * Factory method
    *
    * @return a concrete Reporter
    */
   public abstract Reporter createReportGenerator();
```

```
    /**
     * Report generator building steps.
     *
     */
    public void generateReport(TestAnalyzer analyzer) {
        Reporter reportGenerator = createReportGenerator();
        reportGenerator.createReporter();
        reportGenerator.generateTestReport(analyzer);
        reportGenerator.close();
    }
}
```

Por lo que necesito tener una clase concreta que se encargue de generar el Reporte dependiendo del formato que necesite. De esto se encarga la clase ReportFactory que encapsula la creacion de los reportes.

```
public class ReportFactory extends ReportGeneratorAbsractFactory {

    public static final String CONSOLE_REPORT = "CONSOLE_REPORT";
    public static final String TEXT_REPORT = "TEXT_REPORT";
    public static final String XML_REPORT = "XML_REPORT";
    private String format;

    /**
     * Defines the format of the report, it can be one of:
     * {@link CONSOLE_REPORT}
     * {@link TEXT_REPORT}
     * {@link XML_REPORT}
     *
     * @param aFormat
     */
    public ReportFactory(String aFormat) {
        format = aFormat;
    }


    /**
     * Create reporter type
     *
     * @return Reporter
     */
    public Reporter createReportGenerator() {
```

```
    if (format.equals(CONSOLE_REPORT)) {
        return new ConsoleReporter();
    } else if (format.equals(TEXT_REPORT)) {
        return new TextReporter();
    } else {
        return new XmlReporter();
    }
}
```

Diagrama de Clases