/**
* @Author Leonardo Sarmiento
* @Date Marzo 2020
*/

- **OBJETIVO DEL PLAN:**

Más allá de promover en este momento el desarrollo de pruebas unitarias/integración sobre funcionalidades anteriores que se han considerado, por otro medio, que funcionan bien, se busca una manera de testear, y dejar constancia, sobre nuevas funcionalidades. Considerando beneficios tales como:

. **Ahorro en tiempo**. Si su aprobación/testeo depende de otras actividades previas que conlleven mucho tiempo, como la carga de datos en la BBDD, tanto por tiempo requerido para la carga en sí como el tiempo que consume ejecutar la prueba si debe acceder a la BBDD.

. **No contaminar la BBDD**, con datos corruptos/inconsistentes que pudieran dejar ejecuciones incompletas, logs de procesos incompletos o de pruebas, entre otros.

. **Reducir errores y controlar modificaciones.**

- **SE IMPLANTÓ/PROBÓ SOBRE:**

. proyecto GIROS_WS_RF2017_4_EUS
. la rama develop-giroso-UF3-modUnitT

Se proveen 2 soluciones/desarrollos a continuación:
1. Cómo realizar las pruebas bajo el esquema propuesto.
2. Cómo ejecutar pruebas por categorías y/o en paralelo incluso.

---

1. Cómo realizar las pruebas bajo el esquema propuesto.

- PREÁMBULO:

Las pruebas unitarias nos sirven para testear que el código de nuestro proyecto funcione como lo esperado, pero más allá de esto, nos permitirá darnos cuenta si parte del código fue modificado, pudiendo afectar los resultados de una ejecución.

Preferiblemente se deberá ejecutar uno o varios bancos de pruebas antes de compartir el código con el resto del equipo de trabajo, esto es, antes de mergear los cambios de nuestra rama con la rama Dev y subir al repositorio remoto.

- PROBLEMAS:

1. Cuando desarrollamos un test unitario, este deberá probar la funcionalidad de un método particular, pero no abarcar todas las dependencias que este tiene con el resto del sistema, ya que eso sería un test de integración. Aun así, haremos tests de integración, pero no queremos evaluar partes del código ya testeadas en otra prueba o ciertas funcionalidades que consumen tiempo como accesos a la base de datos o solicitudes http. En el caso de los acceso a la bbdd, además de ser costosos en procesamiento, requieren una carga previa de datos sobre una bbdd real o de pruebas, cuando lo más seguro es que retorne una entidad o un objeto de negocio.

2. La manera en que se escribe el código, debe permitir, en mayor o menor medida, desarrollar pruebas de manera fácil. Esto se logra básicamente pasando dependencias de las clases a través del constructor y dependencias de los métodos como argumentos, nunca instanciar objetos dentro del método sino proveérselos desde otro llamado. En este caso, el problema es el uso de métodos estáticos y la instanciación dentro de los métodos.

3. Desconocimiento de las librerías de mockeo.

- SOLUCIÓN:

1. Se debe maquetar/mockear objetos para que se comporten de una manera predefinida cuando se ejecute una acción sobre estos. Para esto usamos librerías de mockeo, en este caso Mockito.

2. Usamos una librería que nos permite mockear métodos estáticos y trabaja en conjunto con Mockito, esto es PowerMock.

3. Se desarrolló una clase de pruebas en el proyecto, donde se explica detalladamente cómo utilizar la librería, pero haciendo llamados reales para una prueba de integración y otra prueba unitaria.

La clase es:
-src/test/physis.TestDocumentarProductoClienteUnitTests.java
Dentro sus métodos:

- templateForAnyTests() //plantilla con la estructura
- UT_documentarProductoCliente_documentarProductoCliente() //prueba unitaria para comenzar (mockeada)
- baja_temporal() //prueba exhaustiva de integración (mockeada), no afecta bbdd si se ejecuta, pero abarca todo el flujo de llamados para realizar la operación, excepto los accesos a bbdd, ya que estos se mockearon.

*Adjunto un fichero .xlsx de guía utilizado para la prueba de integración.
('Flujo para mockeo proy GIROS-WS.xlsx')

2. Cómo ejecutar pruebas por categorías y/o en paralelo incluso.

- PREÁMBULO:

En un proyecto muy grande existirán muchas pruebas (unitarias/integración), estas deberían estar clasificadas por categorías, dentro de paquetes de clases, cada clase ejecuta métodos de prueba de alguna categoría o varias.

Las pruebas se pueden ejecutar con Maven, al momento de la compilación. Este tipo de ejecución viene bien cuando se va a hacer un despliegue con herramientas de CI/CD tipo Jenkins, así evitaremos hacer un despliegue en algún entorno si las pruebas fallan.

Maven permite configurar la ejecución de los tests, usando el plugin surefire, definir cantidad de threads si se hacen en paralelo, grupos de categorías, etc.

Aun así, los desarrolladores deben poder ejecutar sus pruebas antes de mergear el desarrollo en su rama con la rama Dev y compartir código que presente errores. Pero este tipo de ejecución se hace sobre el IDE, no usando Maven.

- PROBLEMAS:

1. Hubo dificultad para ejecutar pruebas desde Maven cuando se configuraba para ejecutar en paralelo y/o para el uso de categorías.

2. Queremos ejecutar desde el IDE Eclipse pruebas en paralelo y usar categorías, pero Eclipse no trae la opción de ejecutar pruebas en paralelo.

3. Por la forma en que se desarrolló el proyecto, se hace uso extensivo de objetos estáticos, pudiendo generar problemas a la hora de ejecutar pruebas en paralelo.

- SOLUCIÓN:

1. No se encontró. Es un problema de versiones posiblemente.

2. Se creó un custom JUnit Runner que haciendo uso de este nos permite definir las clases que se quieren ejecutar en paralelo y las categorías que se quieren ejecutar, así podemos definir todas las clases de pruebas en un único sitio y modificar las categorías a consideración de lo que se quiera probar en el momento.
Test Suite:

- src/test/test_tools.custom_runners.*
- src/test/test_tools.suites_executors.*
    - ✓ ParallelTestExecutor.java
    - ✓ ParallelTestExecutor2.java (recomendado)

3. Trabajar con diferentes ClassLoaders, uno por clase de test, para que cada método se ejecute aislado del otro, siendo así, las variables globales y estáticas no darán problemas.
(preparado pero no adjunto al proyecto)

- **Finalmente las dependencias Maven:**

Ya agregadas al pom.

Se está utilizando JUnit 4.1, se pudiera actualizar sin inconveniente, o incluso a la 5+ (jupiter) que es compatible con la 4 también y trae buenas mejoras.
No creo que haya problema ya que no encontré uso de la librería en ningún sitio buscando algún 'Assert*'

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.3.3</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-core</artifactId>
  <version>${powermock.version}</version><!--2.0.6-->
  <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-module-junit4</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-api-mockito2</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
</dependency>
```

En mi caso particular, no logré descargar dependencias ni en este ni en otro proyecto desde el ordenador asignado, parece que todas las solicitudes pasan por el proxy aunque intenté cambiar eso. Si es tu caso, descarga los siguientes y agrega al classpath en ese orden de carga.

- algunas de estas pudieran no hacer falta, pero las dejo por si acaso.

javassist.jar
byte-buddy-1.10.8.jar
objenesis-3.1.jar
powermock-classloading-objenesis-2.0.5.jar

powermock-reflect-2.0.5.jar
powermock-module-javaagent-2.0.2.jar
powermock-module-junit4-rule-agent-2.0.5.jar

- estas sí

mockito-core-3.3.3.jar
powermock-core-2.0.5.jar
powermock-api-support-2.0.5.jar
powermock-api-mockito2-2.0.5.jar
powermock-module-junit4-common-2.0.5.jar
powermock-module-junit4-2.0.5.jar