

1. Pruebas estructurales de caja blanca con la herramienta Checkstyle.

Añadimos el siguiente plugin a nuestro pom del proyecto:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>3.0.0</version>
    </plugin>
  </plugins>
</reporting>
```

Si existe un fallo que dice que no existe una clase principal definida, lo arreglamos añadiendo el siguiente código a nuestro pom:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.3</version>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-project-info-reports-plugin</artifactId>
  <version>2.7</version>
</plugin>
```

Ejecutamos en terminal en la raíz de nuestro proyecto:

```
mvn site
```

Tras completarse el análisis, genera una carpeta /target/site con dos partes diferenciadas. Una parte contempla todo tipo de información sobre el proyecto: gestión de plugins, información de las dependencias, del source repository, sobre desarrolladores, etc. La otra parte es sobre los análisis llevados a cabo por checklist que, como en este caso no hemos especificado un checklist.xml con un análisis customizado por nosotros, hace el suyo por defecto.

Genera un archivo checklist.html con mucha información sobre el código.

Archivos analizados, las reglas que ha aplicado para el análisis y luego por cada archivo todos los errores encontrados especificando la línea concreta.

2. PIT. Pruebas de mutación

Las pruebas de mutación son muy interesantes ya que pueden medir la calidad de nuestros tests a partir de las mutaciones eliminadas.

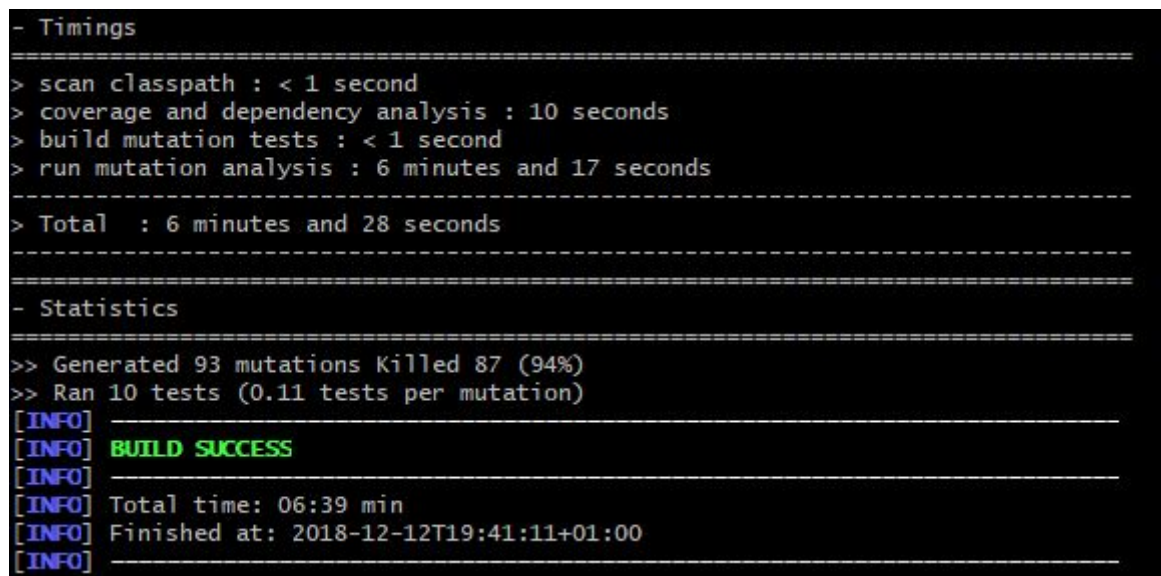
PIT funciona de la siguiente manera: utilizando nuestro código corre nuestros test de unidad modificando nuestro código. Al cambiar el código, esto resultaría en que los tests fallasen, si los tests sobreviven es que existe algún tipo de fallo en el código que no se contempla.

Añadimos el plugin de pit a nuestro pom:

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.4.3</version>
</plugin>
```

Hacemos un mvn clean y mvn install para instalar el .jar en nuestro repositorio maven. Tras esto, ejecutamos lo siguiente:

mvn org.pitest:pitest-maven:mutationCoverage, “the mutation coverage goal analyses all classes in the codebase that match the target tests and target classes filters.”



```
- Timings
=====
> scan classpath : < 1 second
> coverage and dependency analysis : 10 seconds
> build mutation tests : < 1 second
> run mutation analysis : 6 minutes and 17 seconds
=====
> Total   : 6 minutes and 28 seconds
=====

- Statistics
=====
>> Generated 93 mutations Killed 87 (94%)
>> Ran 10 tests (0.11 tests per mutation)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 06:39 min
[INFO] Finished at: 2018-12-12T19:41:11+01:00
[INFO] -----
```

En la imagen podemos ver que para nuestras clases se generaron en total 93 mutaciones de las cuales 87 fueron eliminadas. Las 6 restantes indican que existe algún tipo de fallo en el código.

Si en la salida existen timeouts, pueden indicar dos cosas, o que una mutación ha causado un bucle infinito o que PIT cree que un bucle infinito es incorrecto o no tiene sentido.

Stackoverflow: “Pitest detects these warnings by comparing the execution time of each test with the time it took when no mutation is present. If the test takes significantly more time to run then the process is killed and the mutation marked as timed out.”

Los mutators que se usan por defecto son:

Conditional boundary: que reemplazan operadores relacionales por sus mutaciones.

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

Y otros como Incrementos de variables locales, operadores condicionales, etc. La lista completa de los mutadores por defecto está disponible en: <http://pitest.org/quickstart/mutators/>

Este comando analiza todas las clases que pueden ser testeadas y reporta el line y mutation coverage. En `/target/pit-reports/$current-date+hour$/index.html` aparece toda la información del análisis.

3. Spotbugs

Añadimos al pom lo siguiente:

```
<plugin>
  <groupId>com.github.spotbugs</groupId>
  <artifactId>spotbugs-maven-plugin</artifactId>
  <version>3.1.8</version>
  <dependencies>
    <dependency>
      <groupId>com.github.spotbugs</groupId>
      <artifactId>spotbugs</artifactId>
      <version>3.1.9</version>
    </dependency>
  </dependencies>
</plugin>
```

A partir de aquí tenemos varias opciones con `mvn spotbugs:XXXX`

- `spotbugs`: analiza el proyecto en busca de bugs y genera un xml
- `check goal`: corre el plugin igual que `spotbugs` y falla el build si detecta bugs
- `gui`: lanza SpotBugs GUI para analizar el xml

- help: info de ayuda

```
$ mvn spotbugs:check
[INFO] Scanning for projects...
[INFO]
[INFO] -----< net.gjerrull.etherpad:etherpad_lite_client >-----
[INFO] Building Etherpad Lite Client 1.2.14-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spotbugs-maven-plugin:3.1.9:check (default-cli) > :spotbugs @ etherpad_lite_client >>>
[INFO]
[INFO] --- spotbugs-maven-plugin:3.1.9:spotbugs (spotbugs) @ etherpad_lite_client ---
[INFO] Fork Value is true
[java] SLF4J: No SLF4J providers were found.
[java] SLF4J: Defaulting to no-operation (NOP) logger implementation
[java] SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
[java] Warnings generated: 3
[INFO] Done SpotBugs Analysis....
[INFO]
[INFO] <<< spotbugs-maven-plugin:3.1.9:check (default-cli) < :spotbugs @ etherpad_lite_client <<<
[INFO]
[INFO] --- spotbugs-maven-plugin:3.1.9:check (default-cli) @ etherpad_lite_client ---
[INFO] BugInstance size is 3
[INFO] Error size is 0
[INFO] Total bugs: 3
[ERROR] Found reliance on default encoding in net.gjerrull.etherpad.client.GETRequest.send(): new java.io
GETRequest.java:[line 42] DM_DEFAULT_ENCODING
[ERROR] Found reliance on default encoding in net.gjerrull.etherpad.client.POSTRequest.send(): new java.t
t POSTRequest.java:[line 51] DM_DEFAULT_ENCODING
[ERROR] Found reliance on default encoding in net.gjerrull.etherpad.client.POSTRequest.send(): new java.t
At POSTRequest.java:[line 48] DM_DEFAULT_ENCODING
[INFO]

To see bug detail using the Spotbugs GUI, use the following command "mvn spotbugs:gui"
```

La herramienta nos encuentra 3 bugs relacionados con el encoding, ya que no especificamos ninguno en el código.

4. GraphWalker

Graphwalker es una herramienta basada en testeo de modelos, genera tests. Lee modelos en forma de grafos dirigidos, aquellos en los que las aristas tienen un sentido definido, y genera un grafo con nodos y aristas. Los nodos representan un estado de verificación y las aristas una acción.

How-to: añadimos la dependencia al pom y el plugin

```
<dependency>
  <groupId>org.graphwalker</groupId>
  <artifactId>graphwalker-core</artifactId>
  <version>3.4.2</version>
</dependency>
<dependency>
  <groupId>org.graphwalker</groupId>
  <artifactId>graphwalker-java</artifactId>
  <version>3.4.2</version>
</dependency>
```

```

<plugin>
  <groupId>org.graphwalker</groupId>
  <artifactId>graphwalker-maven-plugin</artifactId>
  <version>3.4.2</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate-sources</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Uso de graphwalker:



4.1 Model design

En esta parte diseñaremos el test que queremos correr. En un Model-based test, el propósito es ver cómo se comporta nuestro sistema bajo unas condiciones concretas que nosotros diseñamos en el test. El resultado es un modelo con nodos que representan un estado y aristas que representan una transición o acción.

4.2 Selection test

En esta parte se contempla las reglas para los tests, que serían que probar y cómo probarlo.



Ejemplo: `random(edge_coverage(100))`, recorrer el grafo de manera aleatoria cubriendo todos los estados.

4.3 Test paths generation

Pueden generarse de dos maneras, “offline” y “online”.

De manera offline especifica que el camino se genera una vez y se guarda en un formato intermedio antes de correr el test. El camino se genera desde línea de comandos y la salida se guarda en un file. El contenido del archivo es lo que se usaría para correr tests.

De manera online, los caminos se generan en tiempo de ejecución. Como ventajas evita generar archivos intermedios y que el código del test accede directamente al contexto de ejecución de los grafos, esto quiere decir que, cuando Graphwalker genera caminos, guarda la información relacionada con la generación de caminos en un contexto de ejecución.

4.4 Test execution

Se puede hacer de dos maneras, paso por paso o con un proyecto usando el arquetipo de maven.

Paso por pasa implica: crear los directorios, mover los grafos a éstos, creando un pom local, implementar los tests y correrlos. Se explica detalladamente con un ejemplo en http://graphwalker.github.io/Tests_execution/

La otra forma de correr los test es con un proyecto adicional usando el arquetipo de maven. Explicación detallada: http://graphwalker.github.io/create_a_test_using_maven/

How-to:

Primeramente creamos un grafo con la herramienta yEd, en este caso, hemos implementado el caso de grupos y pads. Movemos el grafo a /src/main/resources/net/gjerull/etherpad/client. Ejecutamos mvn graphwalker:generate-resources y esto nos genera una interfaz con todas las acciones (aristas del grafo) que tendremos que implementar. Seguidamente, mvn graphwalker:test.

5. JUnit-Quickcheck

JUnit-quickcheck es una librería que soporta tests basados en propiedades. Un property-based test es aquel en que se verifica que las salidas satisfacen unas propiedades concretas en base a múltiples entradas.

Ejemplo práctico: un programa que devuelva la lista de factores primos de un entero n mayor que uno. El test basado en propiedades debe probar que las salidas cumplen unos requisitos matemáticos, por ejemplo que todos los números de la lista de salida son primos y que su multiplicación equivale a n. Este tipo de pruebas aseguran a largo plazo que para cualquier entrada las propiedades se satisfacen.

How-to: añadimos las dependencias al pom.xml

```
<dependency>  
  <groupId>com.pholser</groupId>
```

```
<artifactId>junit-quickcheck-core</artifactId>
<version>0.8.1</version>
</dependency>
<dependency>
  <groupId>com.pholser</groupId>
  <artifactId>junit-quickcheck-generators</artifactId>
  <version>0.8.1</version>
</dependency>
```

Basics:

Creamos una clase que guarde nuestras propiedades que queremos que el sistema verifique y la marcamos con `@RunWith(JUnitQuickcheck.class)`. Después añadimos métodos públicos que devuelvan void, así cada uno de ellos representan una propiedad individual. Los marcamos con `@Property`. Cuando ejecutemos los tests, las propiedades serán verificadas contra valores random generados para cada uno de los parámetros de los métodos.

Corremos la clase como JUnit Test.

6. JETM

Java Execution Time Measurement es una herramienta para monitorizar los tiempos de ejecución de código Java. La mayor ventaja de esta herramienta es ver tiempos reales de ejecución de partes de código concretas de manera rápida y eficiente.

How-to: añadimos la dependencia al pom

```
<dependency>
  <groupId>com.google.code.jetm</groupId>
  <artifactId>jetm-maven-plugin</artifactId>
  <version>1.0.2</version>
</dependency>
```

Basics:

JETM funciona de la siguiente manera, simplemente creando puntos específicos llamados `EtmPoint` en nuestro código conoceremos los tiempos de ejecución de esos fragmentos de código. Más tarde, los llamados `EtmMonitor` recolectan los datos enviandoselos a un `EtmManager`.

Como nuestra aplicación se basa en peticiones HTTP, creamos dos monitores en peticiones GET y POST y cada vez que enviemos información en `GETRequest.send()` y `POSTRequest.send()`, los trozos de código marcados con `EtmPoint` serán medidos.

En nuestros tests, al haber usado Mocking, cada vez que iniciemos y paremos el MockServer, iniciamos y paramos, respectivamente el EtmMonitor.

Quizas JETM sea una herramienta para otro tipo de aplicaciones, por ejemplo, sistemas de gestión dónde lo que importa es la rapidez de respuesta o otros que tengan carga matemática.