
MONITORIZACIÓN DE PROBAS

Título proxecto: Suite de Probas (VVS)

Ref. proxecto: Equipo 8

*Autores: Eloy Naveira Carro, Alejandro López Sánchez, Adrián
Leira Carro, Andrea Ardións Baña*

Validación e Verificación de Software

Data de aprobación	Control de versións	Observacións
16/12/2015	1.0	

1. Contexto

Neste documento recollemos unha suite de probas que aplicamos ao noso proxecto pra que gañase consistencia e estabilidade de cara a un futuro, xa que seguimos a filosofía de que "mais tempo en probas, menos tempo en mantemento".

Elaboramos un plan de probas, unha suite de probas, co obxectivo de que ofrezca a máxima confianza e así ter unha aplicación funcional, sen erros, e no caso de telos, ser conscientes e saber qué camiño tomar pra resolvelos.

2. Estado actual

O estado do proxecto antes de iniciar esta suite de probas era incerto, posto que só contábamos con probas de JUnit. Pra resolvelo, e mellorar a calidade das probas e do proxecto, decidimos tomar os seguintes camiños, e elaborar probas seguindo estas ferramentas de monitorización de probas:

- Ferramentas para automatizar a execución de probas dinámicas:

JUnit: antes de empezar a elaborar esta suite de probas, xa tiñamos test con JUnit. Permítenos elaborar probas unitarias e ver se os métodos se comportan da forma esperada.

- Ferramentas para automatizar a xeración de datos en execución de probas dinámicas:

QuickCheck: está ferramenta úsase, como di máis arriba, para a xeración aleatoria de casos de proba.

Fixéronse test de este tipo en tódolos métodos das clases test do sistema. Consiste basicamente en implementar un bucle en cada caso de proba que xere elementos aleatorios, xa sea usando xeradores proporcionados pola ferramenta, ou ben, creando uns personalizados. Fixéronse probas con estes dous tipos de xeradores.

A continuación móstrase un exemplo do funcionamento do caso de proba:

```

/**
 * ObtenerTitulo() test.
 */
@Test
public void obtenerTituloTest() {

    for (String anyString : Iterables.toIterable(PrimitiveGenerators.printableStrings())) {
        Cancion = new ImplementacionCancion(anyString,3);
        String otherString = PrimitiveGenerators.printableStrings().next();

        assertTrue(Cancion.obtenerTitulo().equals(anyString));
    }
}

```

Como pode observarse, neste caso úsase un xerador de strings proporcionado pola ferramenta.

A continuación móstrase un exemplo no que se usa un xerador personalizado:

```

@Test
public void eliminarTest() {

    List<Contenido> listaReproduccion = new ArrayList<Contenido>();

    for (ImplementacionCancion anyCancion : Iterables.toIterable(new CancionGenerator())) {
        emisoral.agregar(anyCancion, null);
        ImplementacionCancion otherCancion = new CancionGenerator().next();
        listaReproduccion.add(anyCancion);
        assertEquals(anyCancion,emisoral.buscar(anyCancion.obtenerTitulo()).get(0));

        emisoral.eliminar(anyCancion);
        assertNotEquals(listaReproduccion,emisoral.buscar(anyCancion.obtenerTitulo()));
        emisoral.eliminar(otherCancion);
        assertNotEquals(listaReproduccion,emisoral.buscar(anyCancion.obtenerTitulo()));
    }
}

```

E a implementación do xerador:

```

class CancionGenerator implements Generator<ImplementacionCancion> {
    Generator<Integer> iGen = PrimitiveGenerators.integers(-50,50);
    Generator<String> sGen = PrimitiveGenerators.printableStrings();

    public ImplementacionCancion next() {
        String anyTitulo = sGen.next();
        int anyDuracion = iGen.next();

        return new ImplementacionCancion(anyTitulo,anyDuracion);
    }
}

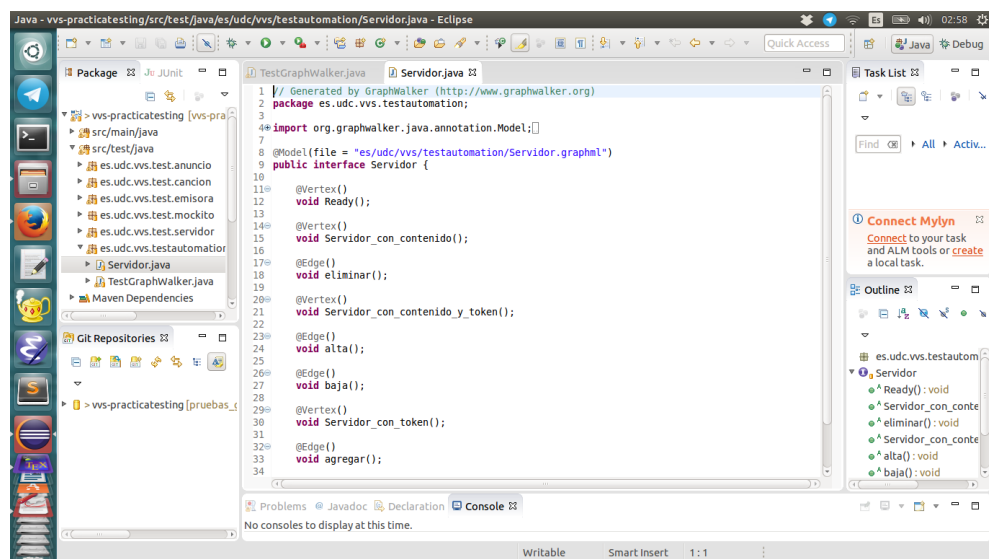
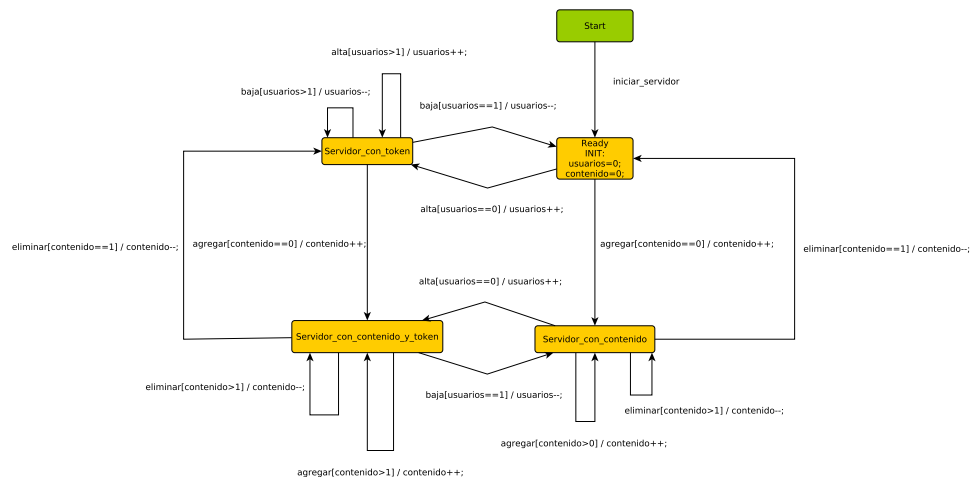
```

- Ferramentas para automatizar a xeración e execución de probas dinámicas:

GraphWalker é unha librería de Java para automatizar a xeración e execución de probas dinámicas. A idea xeneral consiste no deseño do

diagrama de estados do sistema para logo mediante unhas probas verificar o funcionamento contra este diagrama.

Unha vez realizado o diagrama de estados situase dentro do proxecto e a continuación xérase a interfaz que hai que implementar para as probas. Non chegamos a completar o uso desta ferramenta porque encontrámonos con numerosos erros que non soubemos arranxar.



- Ferramentas para validación da calidade das probas:

Cobertura: contar con Cobertura é crucial pra saber a calidade das probas, pra saber con exactitude a cantidade de código que estamos pro-

bando cos test.

Este foi o informe inicial tras a primeira proba de Cobertura:

Coverage Report - All Packages

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	11	78% 199/253	64% 91/142	2,197
es.udc.vvs.model.contenido	1	N/A	N/A	1
es.udc.vvs.model.contenido.anuncioimpl	1	83% 10/12	100% 2/2	1,286
es.udc.vvs.model.contenido.cancionimpl	1	78% 11/14	50% 1/2	1,286
es.udc.vvs.model.contenido.emisoraimpl	1	100% 26/26	87% 7/8	1,571
es.udc.vvs.model.servidor	2	100% 8/8	N/A	1
es.udc.vvs.model.servidor.servidorimpl	2	73% 129/175	62% 81/130	4,25
es.udc.vvs.model.util.exceptions	1	100% 2/2	N/A	1
es.udc.vvs.model.util.servidorutil	2	81% 13/16	N/A	1

Report generated by Cobertura 2.1.1 on 19/11/15 17:01.

Este é o informe final:

Coverage Report - All Packages

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	11	94% 242/257	86% 111/128	2,23
es.udc.vvs.model.contenido	1	N/A	N/A	1
es.udc.vvs.model.contenido.anuncioimpl	1	83% 10/12	100% 2/2	1,286
es.udc.vvs.model.contenido.cancionimpl	1	85% 12/14	100% 2/2	1,286
es.udc.vvs.model.contenido.emisoraimpl	1	92% 26/28	70% 7/10	1,714
es.udc.vvs.model.servidor	2	100% 8/8	N/A	1
es.udc.vvs.model.servidor.servidorimpl	2	96% 173/178	87% 100/114	4,3
es.udc.vvs.model.util.exceptions	1	100% 2/2	N/A	1
es.udc.vvs.model.util.servidorutil	2	81% 13/16	N/A	1

Report generated by Cobertura 2.1.1 on 17/12/15 14:47.

Estos datos muestran que ao principio as probas non pasaban por todo o código, pero o mellorar a calidade das mesmas, agora atopámonos con mellores resultados. Tivemos que optar por test de mellor calidade, que abarcasen as máximas líneas de código posibles.

E isto non e todo, porque cobertura axudounos a darnos conta de que tiñamos métodos sen testear, e ademáis, de malas prácticas agora de deixar métodos dunha Interface sen implementación.

O ideal, e acadar mais dun 80 de porcentaxe en cobertura, e iso é o que conseguimos.

PIT: Mutation Testing axudaranos a mellor a calidade das probas atopadas, con pistas pra que as melloremos. E unha maneira de resolver erros que coas probas unitarias non atoparíamos nunca, posto que lle da a volta a situación pra analizar o código. O obtetivo, que todas as mutacions

rematen en KILLED.

A primeira vez que o pasamos (depois de amañar as carencias de Cober-
tura) atopámonos cos seguintes resultados:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
7	85% <div><div></div><div>202/239</div></div>	68% <div><div></div><div>101/148</div></div>

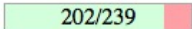
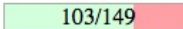
Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
es.udc.vvs.model.contenido.anuncioimpl	1	83% <div><div></div><div>10/12</div></div>	100% <div><div></div><div>6/6</div></div>
es.udc.vvs.model.contenido.cancionimpl	1	79% <div><div></div><div>11/14</div></div>	83% <div><div></div><div>5/6</div></div>
es.udc.vvs.model.contenido.emisoraimpl	1	100% <div><div></div><div>26/26</div></div>	93% <div><div></div><div>14/15</div></div>
es.udc.vvs.model.servidor	1	100% <div><div></div><div>8/8</div></div>	100% <div><div></div><div>2/2</div></div>
es.udc.vvs.model.servidor.servidorimpl	2	81% <div><div></div><div>134/165</div></div>	66% <div><div></div><div>72/109</div></div>
es.udc.vvs.model.util.servidorutil	1	93% <div><div></div><div>13/14</div></div>	20% <div><div></div><div>2/10</div></div>

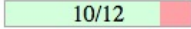
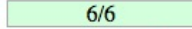
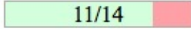
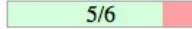
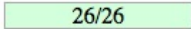
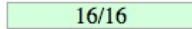
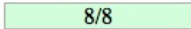
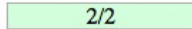
Centrándonos na clase Emisora, vemosos que seguindo unha das recomen-
dacions de PIT a mutación ada un 16/16.

```
-      if (predecesor != null) {  
-  
-          int pos = listaReproduccion.indexOf(predecesor);  
+      if (predecesor == null) {  
  
-          listaReproduccion.add(pos+1, contenido);  
+          listaReproduccion.add(contenido);  
  
+      }  
+  
+      if (predecesor != null) {  
  
-      }else{  
+          int pos = listaReproduccion.indexOf(predecesor);  
  
-          listaReproduccion.add(contenido);  
+          listaReproduccion.add(pos+1, contenido);
```

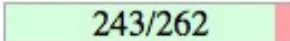
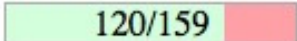
Tras amañar a implementación de Emisora, o resultado tras isto mellora:

Number of Classes	Line Coverage	Mutation Coverage
7	85% 	69% 

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
es.udc.vvs.model.contenido.anuncioimpl	1	83% 	100% 
es.udc.vvs.model.contenido.cancionimpl	1	79% 	83% 
es.udc.vvs.model.contenido.emisoraimpl	1	100% 	100% 
es.udc.vvs.model.servidor	1	100% 	100% 

E por último, chegamos a conclusión de que aplicando outras ferramentas de proba, podemos conseguir acadar mellores resultados pasando de novo PIT-Mutation, pasando dun 85 de porcentaxe e 69 respectivamente, a 93 e 75, que é mais aceptable tengo en conta como é PIT.

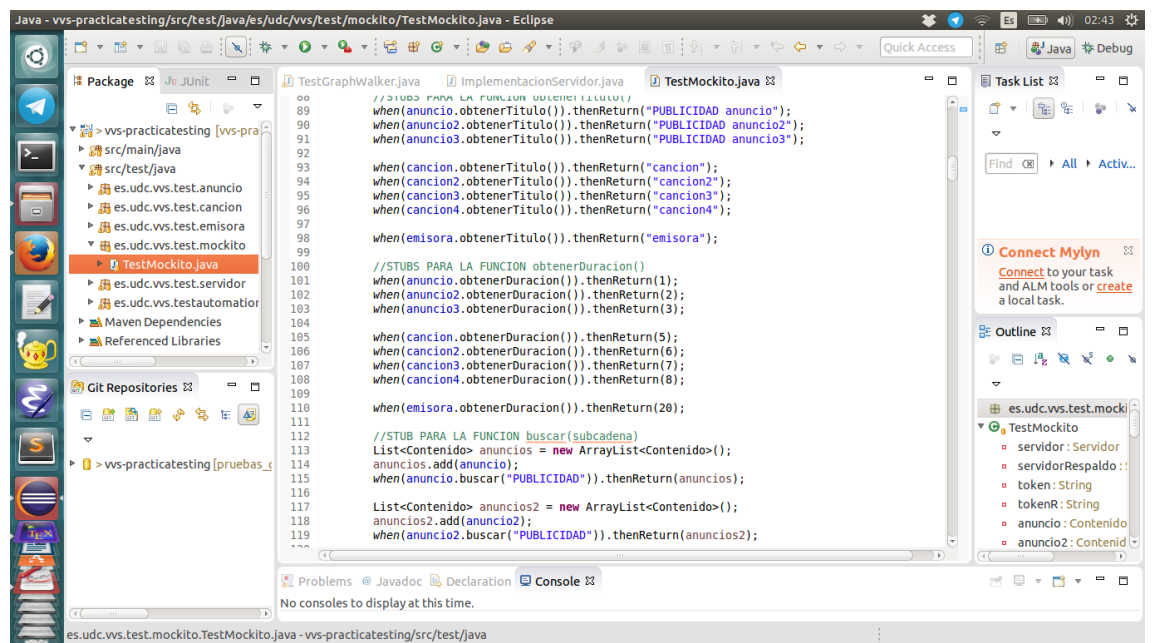
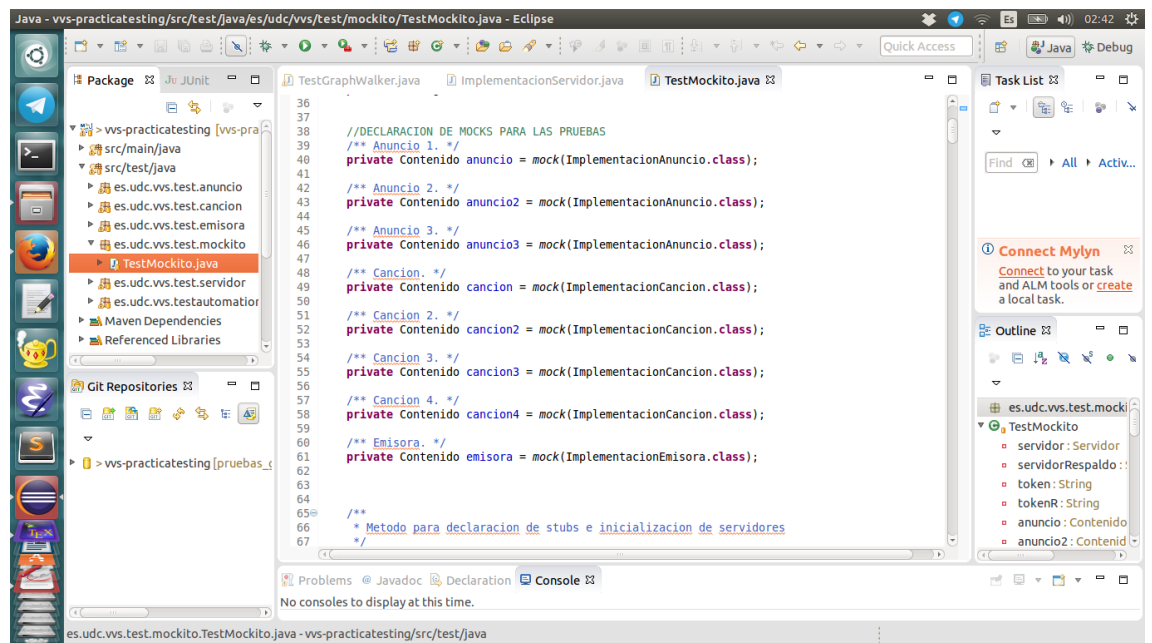
Number of Classes	Line Coverage	Mutation Coverage
7	93% 	75% 

- Ferramentas para facilitar as probas dinámicas de unidade:

Mockito: é unha librería de Java (baseado en EayMock) para a creación de Mocks ou obxetos simulados moi empregados en probas unitarias en TDD (Test Driven Development), que é unha práctica de programación baseada na creación de probas primeiro. Mockito creouse co fin de correxir algúns dos problemas de EasyMock.

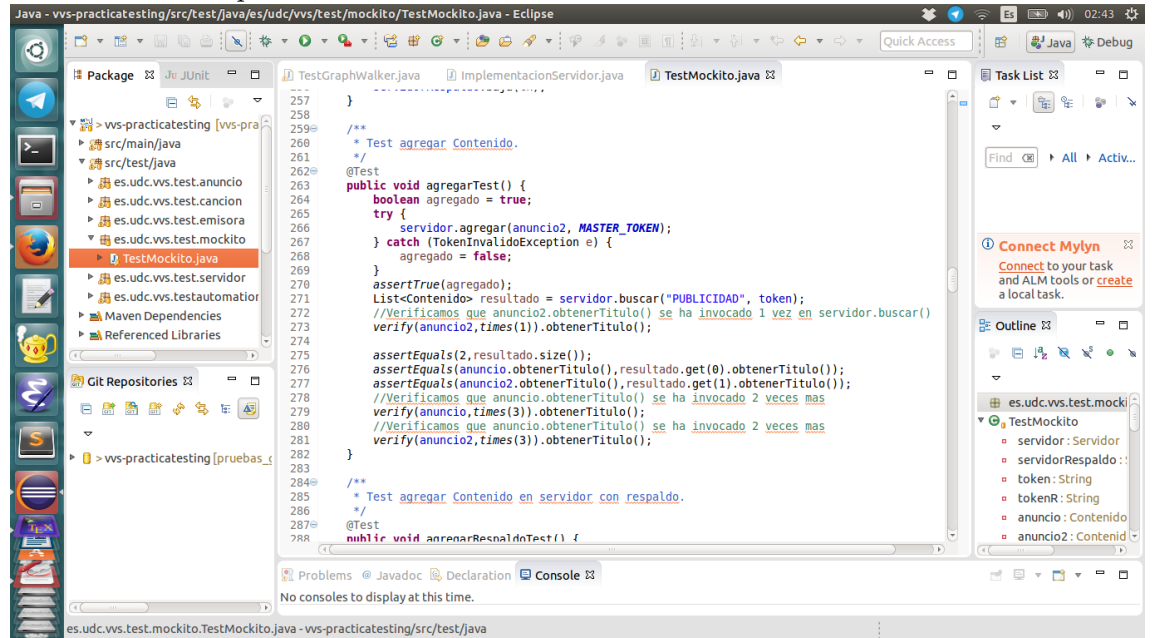
Na nosa práctica simulamos o comportamento das implementacións da interfaz Contenido para realizar as probas do Servidor.

A aplicación básica desta ferramenta consiste, en primeiro lugar, na declaración das clases mock e as súas cláusulas de comportamento, ou stubs, que definen os valores que teñen que devolver as distintas funcións ou métodos desa clase.



Unha vez preparados os mocks e definidos os stubs de cada un, pasamos

a implementación das probas. Nalgúns casos das probas empregamos a cláusula verify que comproba a execución do método sinalado, e incluso o número de veces que se executou.



Durante a realización destas probas atopouse un erro nas dúas implementacións do Servidor. O erro consistía no caso de que se buscara os anuncios contidos nas emisoras non encontraba ningún. Este erro debeuse a que non se comprobou este caso nas probas feitas previamente.

- Ferramentas para probas non funcionais:

JETM: é unha librería que se utiliza para axudar a localizar problemas de rendemento en aplicacións Java.

Ten un uso relativamente sinxelo. Consiste en colocar puntos de monitorización nos métodos nos que se queira comprobar o rendemento, e na clase test chamar a un monitor que mostre os resultados desa monitorización.

Decidiuse comprobar só o rendemento do Servidor, xa que o de respaldo é similar, e as implementacións de Contenido son bastante triviais.

Móstrase a continuación os resultados do monitor no Servidor:

```
[INFO ] [EtmMonitor] JETM 1.2.3 started.
```

Measurement Point	#	Average	Min	Max	Total
ImplementacionServidor:agregar	417	0,002	0,001	0,067	0,643
ImplementacionServidor:alta	411	0,013	0,005	1,884	5,226
ImplementacionServidor:baja	201	0,005	0,004	0,035	1,035
ImplementacionServidor:buscar	614	0,037	0,002	3,284	22,890
ImplementacionServidor:eliminar	202	0,002	0,001	0,060	0,342
ImplementacionServidor:existeUsuario	1015	0,001	0,001	0,023	1,253

- Ferramentas para probas estruturais/estáticas:

PMD: esta ferramenta permítenos darnos conta das malas decisións que tomamos a hora de programar. Atopámonos con todo tipo de recomendacions pra mellorar o código, como pode ser na clase Emisora que é a que tomamos de exemplo, e coa que conseguimos acadar uns bos resultados tras a aplicación desta ferramenta.

Aplicamos decisións como converter clases a final, ou mellorar a maneira de facer as probas empregando `assertEquals` mellor que `assertTrue` e `assertFalse`.

Este é o resultado de PMD antes de amañar Emisora:

The screenshot shows the PMD Violations Overview window. On the left, a 'Violations Outline' pane lists violations by line number and rule. The main pane shows a table of violations across the project.

Element	# Violations	# Violations/KLOC	# Violations/Method	Project
es.udc.vvs.model.contenido	9	N/A	N/A	vvs-project
es.udc.vvs.model.contenido.anuncioimpl	26	N/A	N/A	vvs-project
es.udc.vvs.model.contenido.cancionimpl	27	N/A	N/A	vvs-project
es.udc.vvs.model.contenido.emisoraimpl	30	N/A	N/A	vvs-project
es.udc.vvs.model.servidor	18	N/A	N/A	vvs-project
es.udc.vvs.model.servidor.servidorimpl	277	N/A	N/A	vvs-project
es.udc.vvs.model.util.servidorutil	17	N/A	N/A	vvs-project
es.udc.vvs.test.anuncio	56	N/A	N/A	vvs-project
es.udc.vvs.test.cancion	74	N/A	N/A	vvs-project
es.udc.vvs.test.emisora	81	818.2	11.57	vvs-project
EmisoraTest.java	81	818.2	11.57	vvs-project
es.udc.vvs.test.mockito	271	N/A	N/A	vvs-project

At the bottom of the table, there are summary statistics: Writable, Smart Insert, and 32 : 12.

Este é o resultado de PMD despois de amañar Emisora:

The screenshot shows the PMD IDE interface. On the left, the 'Violations Outline' pane lists various rules and their locations. On the right, the 'Violations Overview' pane provides a summary of violations across different elements. Below this, a code snippet is shown with a JUnit test method.

Element	# Violations	# Violations/KLOC	# Violations/h
es.udc.vvs.model.contenido	18	N/A	
es.udc.vvs.model.contenido.anuncioimpl	52	N/A	
es.udc.vvs.model.contenido.cancionimpl	54	N/A	
es.udc.vvs.model.contenido.emisoraimpl	60	N/A	
es.udc.vvs.model.servidor	23	N/A	
es.udc.vvs.model.servidor.servidorimpl	277	N/A	
es.udc.vvs.model.util.servidorutil	32	N/A	
es.udc.vvs.test.anuncio	56	N/A	
es.udc.vvs.test.cancion	74	N/A	
es.udc.vvs.test.emisora	56	682.9	
EmisoraTest.java	56	682.9	
es.udc.vvs.test.mockito	542	N/A	

```

119 /**
120  * Test del metodo obtenerDuracion() para obtener la duración de una
121  * emisora.
122  */
123 @Test
124 public void obtenerDuracionTest() {
125
126     // ...

```

Como podemos ver, os resultados son moito mellores, demostrando que non só podemos deixarnos levar por probas como JUnit, senon tamen acadar unha boa suite de probas pra analizar cantos mais problemas mellor.

FindBugs: con esta ferramenta o que se quer probar é se hay algún tipo de bug potencial no sistema. Despois de executalo, o resultado é o seguinte:

The screenshot shows the FindBugs Bug Detector Report for the 'vvs-project'. It includes a summary table and a list of files with specific bugs.

Classes	Bugs	Errors	Missing Classes
11	1	0	0

Class	Bugs
es.udc.vvs.model.util.servidorutil.GenerarToken	1

Bug	Category	Details	Line	Priority
Found reliance on default encoding in es.udc.vvs.model.util.servidorutil.GenerarToken.generateRandomToken(): new String(byte[])	11BN	DM_DEFAULT_ENCODING	27	High

Con esto sácase en claro que existe un posible bug potencial na clase GenerarToken, pero despois de analizar a clase do erro, se chega á conclusión de que é un erro de encoding, polo que non se considera relevante.

3. Rexisto de probas

As causas dos atrasos, débéronse case na súa totalidade a falta de información a hora de preparar o entorno pra executar as probas. Sí e certo que contamos con moita documentación de cada tipo de proba na súa correspondente páxina oficial, pero moitas veces queda un pouco no aire cómo seguir os pasos. E por isto, polo que na súa maioría atopámonos con dúbidas sobre qué pasos seguir, erros tras a execución errónea de probas, pero que co paso do tempo fumos resolvendo sen problemas.

O mellor de todo isto sen dúbida, foi darnos conta como co paso do tempo e así que executábamos mais probas, acadábamos mellores resultados e descubríamos mellores formas de facer as cousas que no seu momento se nos pasaran por alto.

4. Rexistro de erros

Non atopamos probas que revelasen incumprimentos nas especificacions, pero sí faltaba algún método que non contempláramos no seu momento.

5. Estatísticas

- Número de erros encontrados diariamente e semanalmente: a medida que probamos as diferentes ferramentas de monitorización de probas, atopámonos con erros, pero non erros que foxen en contra da especificación, senon melloras.
- Nivel de progreso na execución das probas: o progreso foi progresivo. A execución de determinadas probas levounos a obter test mais precisos e completo, conseguindo unha evolución de probas moi positiva. Pasando por exemplo dun porcentaxe do 60 en cobertura ata mais do 80.
- Análise do perfil de detección de erros (lugares, compoñentes, tipoloxía): todos vinculados coa seguridade, visibilidade, algún de rendemento... malas prácticas de programación.
- Informe de erros abertos e pechados por nivel de criticidade: no seu momento tivemos issues abertas, que xiraron en torno a seguridade e

visibilidade de cada un dos componentes.

- Avaliación global do estado de calidade e estabilidade actuais: mellorable, pero estable, funcional e moito mellor que cando empezamos coa suite de probas.

6. Outros aspectos de interese

Cada proba foi crucial para acadar unha mellor calidade do proxecto. O análise do código con Cobertura axudounos a mellorar a calidade das probas, pero as demais probas, axudaron a mellor a calidade do proxecto na súa totalidade.

Trala execución de quickcheck tampouco se descubriron novos erros. Para o que serviu foi para incrementar en gran medida a confianza na robustez das nosas probas.