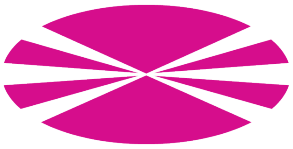


UNIVERSIDADE DA CORUÑA

Monitorización de pruebas
VVS

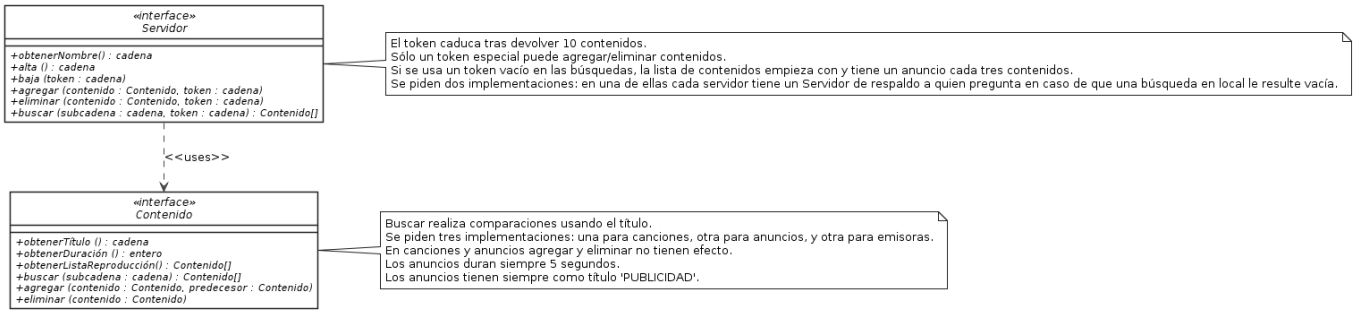
Historial de revisiones

Fecha	Versión	Descripción	Autores
16/12/2015	1.0	Ejercicio de refactorización	Xoán Andreu Barro Torres F. Javier Moure López Emma Oitavén Carracedo

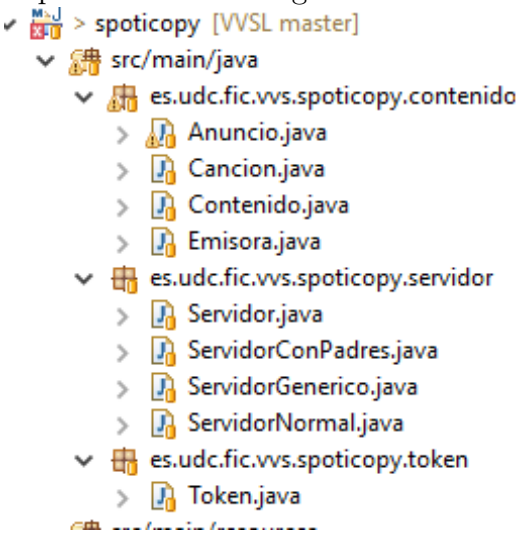
Índice

1. Contexto

Este documento hace referencia a las pruebas realizadas sobre el proyecto de la asignatura VVS llamado Spoticopy[1] encontrado en el repositorio de GitHub. Dicha aplicación simula el comportamiento de una aplicación de música. El enunciado de la funcionalidad es el siguiente:



A partir de dicho diagrama obtuvimos la siguiente relación de clases:



El paquete contenido tienen siempre un nombre, una duración y una lista de reproducción. Son las unidades funcionales con las que funcionará nuestro servidor. El paquete servidor es el que utiliza el paquete contenido para simular el funcionamiento de una aplicación que reproduce música. Distinguiremos ServidorGenerico que contiene el comportamiento genérico de un servidor. Las particularidades se implementan en distintas clases que heredaran de ServidorGenerico. El servidor normal simplemente implementa la búsqueda además de heredar del servidor genérico. La peculiaridad de un ServidorConPadres es que, de no poder devolver contenidos que cumplan el criterio de busqueda, solicita los contenidos de otro servidor, llamado padre, y devuelve lo que el le proporcione. Cualquier servidor puede ser el padre, no necesariamente otro Servidor ConPadres (aunque seria posible montarse arboles, listas o mallas de Servidores). OJO,la implementación no contempla el caso de un anillo de servidores, intentarlo creara un bucle infinito.

2. Estado actual

Listaxe de funcionalidades actuais, as súas especificacións, as persoas responsables do seu desenvolvemento, e as persoas responsables do proceso de proba. Para cada funcionali-

[1] https://github.com/andreu-barro/VVS

dade: número de probas obxectivo, número de probas preparadas, porcentaxe executada e porcentaxe superada. Se esta información é profusa e se almacena noutra fonte, referencia á fonte. Se é cambiante, referencia a unha *shapshot* ou resumo do mais destacado.

Las funciones que se ocupan de la funcionalidad de nuestra aplicación son las siguientes son las que serán evaluadas:

- Clase Anuncio

  String obtenerTitulo():

  int obtenerDuracion()

  List:Contenido obtenerListaReproduccion()

  List:Contenido buscar(final String subcadena)

  void agregar(final Contenido contenido, final Contenido predecesor)

  eliminar(final Contenido contenido)

- Clase Cancion

  String obtenerTitulo()

  int obtenerDuracion()

  List:Contenido obtenerListaReproduccion()

  List:Contenido buscar(final String subcadena)

  void agregar(final Contenido contenido, final Contenido predecesor)

  eliminar(final Contenido contenido)

- Clase Emisora

  String obtenerTitulo()

  int obtenerDuracion()

  List:Contenido obtenerListaReproduccion()

  List:Contenido buscar(final String subcadena)

  agregar(final Contenido contenido, final Contenido predecesor)

  eliminar(final Contenido contenido)

- Clase Token

  String alta()

  baja(final String token)

  boolean isAdminToken(final String token)

  long obtenerUsos(final String token)

  usarToken(final String token)

- Clase ServidorGenerico

  String obtenerNombre()

  List:Contenido getContenidos()

Token getToken()

String alta()

baja(final String tok)

agregar(final Contenido contenido, final String tok)

eliminar(final Contenido contenido, final String tok)

- Clase ServidorNormal

  List:Contenido buscar(final String subcadena, final String tok)

- Clase ServidorConPadres

  List:Contenido buscar(final String subcadena, final String tok)

El objetivo es probar que todas estas funciones funcionan correctamente, para ello aplicaremos pruebas de unidad, pruebas dinámicas, pruebas de rendimiento y chequearemos el estilo de programación. Después de aplicarle las herramientas de pruebas y utilizar las herramientas de validación como cobertura y PIT creemos que nuestras funciones son estables y que puede que tengamos una bastantes pruebas realizadas y el código testeado para fiarnos de él.

3. Registro de pruebas

3.1. Pruebas unidad: JUnit

JUnit se utiliza para realizar pruebas unitarias sobre nuestra aplicación, nos sirven para encontrar errores y solventar los problemas en la programación de forma manual. Se realizan pruebas de todas las funciones implementadas en la aplicación. Los casos de prueba se implementan manualmente como parte de funciones de prueba, utilizamos la directiva assertEqual(Expected, Expr).

3.2. Pruebas basadas en propiedades: Quickcheck

QuickCheck es una herramienta para generar automáticamente y ejecutar casos de prueba aleatorios, basados en especificaciones de propiedades. Es decir, ejecutar pruebas con esta herramienta, significa instanciar las propiedades n veces. Las pruebas se detienen al encontrar un caso concreto en el que la propiedad no se cumple (contraejemplo).La ejecución con éxito significa que ninguno de los casos generados incumplió la propiedad Se crean generadores de:

- GeneradorContenido
- GeneradorCancion
- GeneradorServidor
- GeneradorServidorVacio

La programación de generadores para aplicar Quickcheck se realiza en la siguiente clase: es.udc.fic.vvs.spoticopy.generadorTest.servidorTest.

3.3. Validación de calidad de las pruebas: Cobertura

Cobertura(Plugin cobertura) es una herramienta libre (GPL) escrita en Java, que nos permite comprobar el porcentaje de código al que accedemos desde los test. Es decir, Cobertura nos permite saber cuanto código estamos realmente probando con nuestros test. De esta forma Cobertura se convierte en una potente herramienta de trabajo, ya que lo podemos usar como medida de calidad (mientras más código tengamos probado, más garantías tenemos de que podemos hacer refactorizaciones sin peligro). Nuestro objetivo es llegar a cobertura cercana a 100.

3.4. Pruebas dinámicas de unidad: Mockito

Para poder crear un buen conjunto de pruebas unitarias, es necasario que nos centremos exclusivamente en la clase a testear, simulando el funcionamiento de las capas inferiores (pensad por ejemplo en olvidarnos de la capa de acceso a datos, DAO). De esta manera estaremos creando test unitarios potentes que os permitiría detectar y solucionar los errores que tengáis o que se cometan durante el futuro del desarrollo de vuestra aplicación. Para esta tarea nos apoyaremos en el uso de mock objects, que no son más que objetos que simulan parte del comportamiento de una clase, y más especificamente vamos a ver una herramienta que permite generar mock objects dinámicos, mockito.

Pruebas que se realizarán en el paquete es.udc.fic.vvs.spoticopy.mockito, sólo se realizarán las pruebas del servidor, debido a que las pruebas de componentes ya son pruebas de unidad.

3.5. Pruebas no funcionales: JETM

JETM permite realizar pruebas de rendimiento y comprobar la velocidad de ejecuciones al implementar unas pruebas con n iteraciones. Se pueden generar test con múltiples iteraciones para detectar problemas de rendimiento.

3.6. Validación de calidad de las pruebas (mutación testing): PIT

El método que utilizaban en el sistema para realizar estas mediciones lo denominaba "Programa mutado". Básicamente, el Mutation testing consiste en introducir pequeñas modificaciones en el código fuente de la aplicación, a las que denominaremos mutaciones o mutantes.

Si las pruebas pasan al ejecutarse sobre el mutante, el mutante sobrevive. Si las pruebas no pasan al ejecutarse sobre el mutante, el mutante muere El objetivo es que todos los mutantes mueran, así podremos decir que el test responde a la definición concreta del código y que lo prueba correctamente. Este concepto se basa en dos hipótesis:

- Hipótesis del programador competente: La mayoría de los errores introducidos por programadores Senior consisten en pequeños errores sintácticos.
- Hipótesis del efecto de acoplamiento: Pequeños fallos acoplados pueden dar lugar a otros problemas mayores.

Problemas de mayor orden serán revelados por mutantes de mayor orden, que se crean mediante la unión de multiples mutaciones.

3.7. Pruebas estructurales: CheckStyle

Checkstyle es una herramienta de desarrollo que ayudar a los programadores a escribir código Java para que se adhiera a un estándar de codificación. Automatiza el proceso de comprobación de código Java. Esto lo hace ideal para los proyectos a los que se desea aplicar un estándar de codificación.

Checkstyle es altamente configurable y se puede hacer para apoyar casi cualquier estándar de codificación. De tal manera que se puedan suministrar diferentes estándares de código para su posterior comprobación mediante la herramienta.

Reglas del CheckStyle El conjunto de reglas disponible es muy completo y está clasificado en los siguientes grupos:

- Comentarios Javadoc: facilitar el mantenimiento pasa por comentar el código, pero luego los comentarios también hay que mantenerlos... CheckStyle tiene muchas reglas para los javadoc y es muy flexible. Te permite, por ejemplo, obligar a comentar los nombres de clases, todos los métodos menos los get/set y los atributos públicos.
- Convenciones de nombres: puedes definir una expresión regular para el nombre de todo.
- Cabeceras: expresiones regulares para la cabecera de los ficheros.
- Imports: reglas para los import, como no usar *, imports sin usar, etc.
- Violaciones de tamaño: define un máximo para el tamaño de tus clases, métodos, líneas y número de parámetros de un método. Espacios en blanco: un montón de reglas para definir donde se ponen espacios en blanco y tabuladores en el código.
- Modificadores: establece un orden para los modificadores y evita modificadores innecesarios.
- Bloques: reglas para los bloques de código y sus llaves.
- Problemas en la codificación: Acá hay de todo, desde malas prácticas tipo asignaciones internas y posibles fuentes de bugs como definir un método equals que no es el equals(Object), a cosas más estéticas o poco prolijas, como que el default sea el último elemento en un switch o paréntesis innecesarios.
- Diseño de clases: varias reglas sobre el diseño de interfaces y clases, con especial atención en las excepciones.
- Duplicados: te permite definir un mínimo de líneas para buscar código duplicado en tus clases.
- Métricas: define máximos para métricas como complejidad ciclomática, complejidad de expresiones lógicas, npath, líneas de código seguidas sin comentar y dependencia de clases.
- Misceláneo: variables final, indentación, un buscador de expresiones regulares y varias cosas más.
- J2EE: reglas para EJBs.
- Otros: internos a CheckStyle y activados por defecto.

- Filtros: para eventos de auditoria del propio CheckStyle, no hace falta mirarlos.

Checkstyle es una herramienta de desarrollo para ayudar a los programadores escribir código Java que se adhiere a un estándar de codificación. Para comprobar el estilo, pasamos la herramienta CheckStyle a nuestra aplicación y comprobamos el resultado:

4. Registro de errores

4.1. Pruebas unidad: JUnit

En la primera iteración se encuentra que faltan test de funciones, se añaden los test que faltan y se comprueba q no fallan.

Todos los errores localizados, modificaciones necesarias, etc. pueden encontrarse referenciados en el documento de CHANGELOG.txt, en el cual se encuentran las correcciones hasta el momento.

4.2. Validación de la calidad de las pruebas: Cobertura

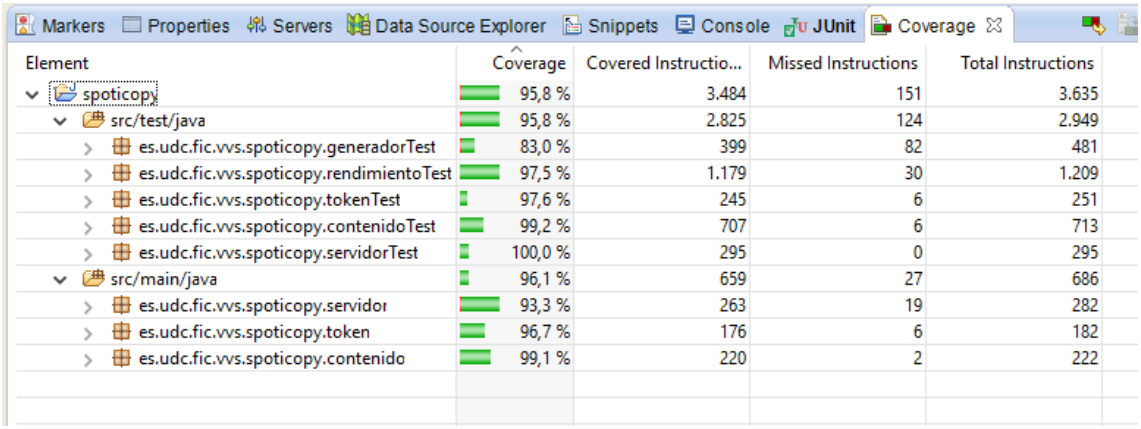
Realizada la prueba de cobertura de pruebas, se observa que faltan muchos test por implementar, por lo que se procede a implementar los test que faltan, según la información que nos ofrece el plugin de ecobertura.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
spoticopy	61,7 %	512	318	830
src/main/java	50,2 %	317	315	632
es.udc.fic.vvs.spoticopy.token	25,2 %	31	92	123
Token.java	25,2 %	31	92	123
es.udc.fic.vvs.spoticopy.contenido	41,7 %	98	137	235
Emisora.java	27,8 %	37	96	133
Anuncio.java	51,7 %	31	29	60
Cancion.java	71,4 %	30	12	42
es.udc.fic.vvs.spoticopy.servidor	68,6 %	188	86	274
ServidorGenerico.java	41,4 %	24	34	58
ServidorConPadres.java	59,6 %	68	46	114
ServidorNormal.java	94,1 %	96	6	102
src/test/java	98,5 %	195	3	198

Después utilizar Junit y Quickcheck para:

- Aumentar cobertura de Anuncio (casi al 100
- Aumentar cobertura de ServidorNormal (al 100
- Aumentar cobertura de ServidorGenerico (casi al 100
- Aumentar cobertura de Token
- Aumentar cobertura de ServidorConPadres

Quedó:



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
spoticopy	95,8 %	3.484	151	3.635
src/test/java	95,8 %	2.825	124	2.949
es.udc.fic.vvs.spoticopy.generadorTest	83,0 %	399	82	481
es.udc.fic.vvs.spoticopy.rendimientoTest	97,5 %	1.179	30	1.209
es.udc.fic.vvs.spoticopy.tokenTest	97,6 %	245	6	251
es.udc.fic.vvs.spoticopy.contenidoTest	99,2 %	707	6	713
es.udc.fic.vvs.spoticopy.servidorTest	100,0 %	295	0	295
src/main/java	96,1 %	659	27	686
es.udc.fic.vvs.spoticopy.servidor	93,3 %	263	19	282
es.udc.fic.vvs.spoticopy.token	96,7 %	176	6	182
es.udc.fic.vvs.spoticopy.contenido	99,1 %	220	2	222

4.3. Pruebas no funcionales: JETM

Resultado del rendimiento de JETM: hrefInformes/SiteTestInicial/jetm-timing-report.htmlInforme JETM

4.4. Validación de calidad de las pruebas (mutación testing): PIT

Realizado el mutation testing, resultados:
Informe PIT

4.5. Pruebas estructurales: CheckStyle

Podemos comprobar los herrores encontrados en el siguiente informe: Informe CheckStyle
Al pasar la herramienta de CheckStyle descubrimos que nuestra aplicación tiene unos 236 errores de estilo, es decir, que no cumple el estandar de programación java.

Resumen de errores encontrados:

- Faltan comentarios: En la mayoría de clases faltan comentarios javadoc. Se añaden.
- Mala indexación código y espacios: Se reestructura el código para que solventar dichos errores.

Se revisa el informe con los 236 errores de estilo y se eliminan los 236 errores de estilo.

4.6. Pruebas estáticas/estructurales: FindBugs

FindBugs es un programa que utiliza el análisis estático para buscar errores en el código de Java.

En la versión inicial podemos comprobar que tenemos los siguientes errores: Informe Find bugs

Errores encontrados:

- ST WRITE TO STATIC FROM INSTANCE METHOD: En la clase token existía un método que escribía en una variable estática, esto es una mala práctica cuando está siendo manipulado por varias instancias.
- RI REDUNDANT INTERFACES: ServidorNormal y ServidorConPadres implementa la misma interfaz que la superclase.
- BC EQUALS METHOD SHOULD WORK FOR ALL OBJECTS: El método equals (Object o) no debe hacer ninguna suposición sobre el tipo de o. Simplemente debe devolver false si o no es del mismo tipo que esta. La clase Anuncio asume el argumento es de tipo anuncio.
- HE EQUALS USE HASHCODE: La clase anuncio Esta clase anula Equals (Object) , pero no anula hashCode () , y hereda la implementación de hashCode () de java.lang.Object (que devuelve el código hash de identidad, un valor arbitrario asignado al objeto por el VM) . Por lo tanto , es muy probable que violaría el invariante que los objetos iguales deben tener iguales hashcodes la clase.
- NP EQUALS SHOULD HANDLE NULL ARGUMENT: En la clase anuncio, el método no funciona para cuando el objeto es nulo.

Después de identificar los errores, resolvedos los problemas mencionados y comprobamos el resultado:

5. Otros aspectos de interés

Nada de momento. Se conserva el apartado para el futuro.