INFORME DE PRÁCTICAS

Repositorio de proxecto:
https://github.com/ivangonzalezcotelo/practicaVVS

Participantes no proxecto:Borja Álvarez García, Iván González Cotelo

Validación e Verificación de Software

1. Descrición do proxecto

El proyecto consta en la implementación de una aplicación de gestión de subastas online. Esta aplicación permite el registro de usuarios, categorías y productos sobre los que se realizan las pujas apoyándose sobre una interfaz web.
La aplicación ha sido desarrollada integramente en JAVA y construida mediante la arquitectura maven, empleando a su vez las herramientas más comunes de diseño tales como spring, hibernate o jetty entre otras muchas y alojandose todos los datos de la misma sobre una base de datos relacional MySQL.

2. Estado actual

Este aplicación se realizó como parte de una asignatura de Programación Avanzada en el curso 2014/15 por los alumnos Breixo Camiña Fernández, David Torres Requeijo e Iván González Cotelo.
Las funcionalidades básicas de la aplicación giran en torno a sus cuatro entidades principales (Usuarios, Productos, Categorías y pujas).

- Gestión de usuarios
  - Registro de usuarios
  - Modificación de datos
  - Baja de usuario
  - Login/Logout de usuario
- Gestión de categorías
  - Consulta de categorías disponibles
- Gestión de productos
  - Registro de un producto a subastar
  - Eliminación de un producto
  - Búsqueda de productos
  - Consulta de productos propios
- Gestión de pujas
  - Realizar una puja sobre un producto
  - Consultar el estado de una puja
  - Consultar historial de pujas

Posteriormente como parte de la asignatura de Verificación y Validación de Software se ha recuperado dicha aplicación y se ha sometido a un proceso de testing. En dicho proceso se han añadido los test unitarios y de integración para el checkeo de la implementación de la capa modelo y se le ha sometido a un múltiple número de herramientas para comprobar la calidad del código implementado.
El equipo encargado de la fase de testing de la aplicación han sido los alumnos Borja Álvarez García e Iván González Cotelo.

2.1. Compoñentes avaliados

Como parte de la segunda y tercera iteración del proyecto se han creado pruebas unitarias y de integración que permiten evaluar de manera individual y colectiva cada funcionalidad de la aplicación. Estas pruebas están númeradas en la cabecera y recogidas de manera ordenada en dos ficheros separados.

- Pruebas de unidad : https://github.com/ivangonzalezcotelo/practicaVVS/blob/master/doc/Registro-pruebas-unidad.txt
- Pruebas de integración : https://github.com/ivangonzalezcotelo/practicaVVS/blob/master/doc/Registro-pruebas-integracion.txt

Para cada test realizado se especifica:

- Servicio al que está probando
- Método al que está probando
- Motivo del test
- Valores de entrada
- Valores esperados de salida
- Datos necesarios antes de ejecutar el test

Se han realizado un total de 81 test de unidad e integración, más tres test adicionales cuyo unico objetivo ha sido el control de los getters y los setters de las entidades principales.

- **48 Test de Unidad**
  - 2 Test DAO de pujas
  - 2 Test DAO de usuarios
  - 4 Test DAO de productos
  - 1 Test DAO de categorías
  - 13 Test Gestión de usuarios
  - 9 Test Gestión de pujas
  - 3 Test Gestión de categorías
  - 14 Test Gestión de productos
- **33 Test de Integración**
  - 6 Test Gestión de Pujas
  - 10 Test Gestión de Productos
  - 4 Test Gestión de Categorías
  - 13 Test Gestión de Usuarios

3. Especificación de probas

En este apartado se especificarán las diferentes herramientas usadas como parte del proceso testing de la aplicación así como los resultados obtenidos con ellas o los posibles problemas que han surgido.

3.1. Mockito

Como parte del diseño de las pruebas de unidad del proyecto, nos hemos apoyado en la herramienta mockito, para poder comprobar las funcionalidades de una manera unitaria, evitando arrastrar errores propios de las pruebas de integración. Esta herramienta importada de **org.mockito.Mock** permite realizar pruebas unitarias, al aislar el servicio del resto de clases(DAOS y otros servicios), siendo el propio tester el que especifica que datos devolverán dichas llamadas sin que el test acceda a ellas, permitiendo encontrar errores únicamente en la clase sobre la que se están realizando las pruebas. Además se ha sustituido la clase spring que corría las pruebas por **@RunWith(MockitoJUnitRunner.class)**.

Esta herramienta se ha empleado en la totalidad de los test unitarios especificados en la documentación anteriormente, siendo el nombre de las clases dentro del proyecto del tipo **XXXServiceUnitTest**.

3.2. CheckStyle

Se ha empleado la herramienta CheckStyle como parte del análisis estático de caja blanca. Esta herramienta permite detectar errores en la sintaxis de la implementación, falta de javaDoc y errores con los parámetros de entrada y salida entre otras funcionalidades. Para poder incorporarlo al proyecto hemos instalado el plugin **Checkstyle Plug-in 7.2.0** y el **conector m2e maven-checkstyle**, además de crear un fichero de configuración propio que ha sido añadido posteriormente al archivo pom.xml del proyecto.

La gran mayoría de checks a validar se dividen en 3 grupos:

- Formato estandar de las clases, se soluciona con una simple combinación de teclas en cada clase (ctrl+shift+f) para dar formato.
- Documentación javaDoc.
- Desorden, redundancia o escasez de palabras clave en la declaración de variables y métodos.

Hemos reducido los checks iniciales de más de 2200 a 146. En estos 146 se recogen los siguientes:

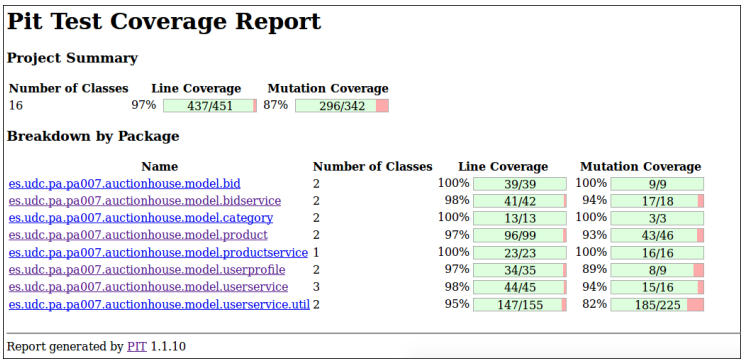
- El constructor de es.udc.pa.pa007.auctionhouse.model.product.Product tiene mas de 7 parametros, se deberia dividir en 2 objetos tal que Product que contiene un ProductDetails.
- Errores en el paquete es.udc.pa.pa007.auctionhouse.model.userservice.util proporcionado por el profesorado de la asignatura por lo que se ha eliminado de la fase testing.
- Todas las clases contenidas en el paquete es.ud.pa.pa007.auctionhouse.web.* ya que se corresponden a la capa de interfaz web que se ha eliminado de la fase testing por recomendación de la profesora.

3.3. PiTest

Como parte del control de calidad de los test implementados se ha utilizado la herramienta piTest de http://pitest.org/. Esta herramienta comprueba la calidad de la

implementación de los test del proyecto a través de un sistema propio de mutaciones. Esto consiste en realizar mutaciones tales como cambio de signos, valores null, o negación de condiciones y ejecutándolas dentro de nuestros tests. Podemos determinar la calidad de los mismos a través del porcentaje de mutaciones que han sobrevivido tras el paso por los test.

Hemos incorporado el plugin pitest en el archivo pom.xml y hemos realizado los test a través del comando **mvn org.pitest:pitest-maven:mutationCoverage**. Esto genera en la carpeta target un index.html que te permite consultar el número de lineas que ejecutan tus test y el número de mutaciones que han sobrevivido.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
16	97%	437/451	87%	296/342

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
es.udc.pa.pa007.auctionhouse.model.bid	2	100%	39/39	100%	9/9
es.udc.pa.pa007.auctionhouse.model.bidservice	2	98%	41/42	94%	17/18
es.udc.pa.pa007.auctionhouse.model.category	2	100%	13/13	100%	3/3
es.udc.pa.pa007.auctionhouse.model.product	2	97%	96/99	93%	43/46
es.udc.pa.pa007.auctionhouse.model.productservice	1	100%	23/23	100%	16/16
es.udc.pa.pa007.auctionhouse.model.userprofile	2	97%	34/35	89%	8/9
es.udc.pa.pa007.auctionhouse.model.userservice	3	98%	44/45	94%	15/16
es.udc.pa.pa007.auctionhouse.model.userservice.util	2	95%	147/155	82%	185/225

Report generated by PIT 1.1.10

Figura 1: *Captura del index.html generado*

Esta captura se corresponde al index.html generado en el último piTest realizado. Como tanto la capa de la interfaz web como la clase serviceutil (proporcionada por el profesorado) no entran dentro de la fase testing, tras el último test hemos obtenido unos resultados reales de:

- Line Coverage: 290/296 98 %
- Mutations Killed: 111/117 95 %

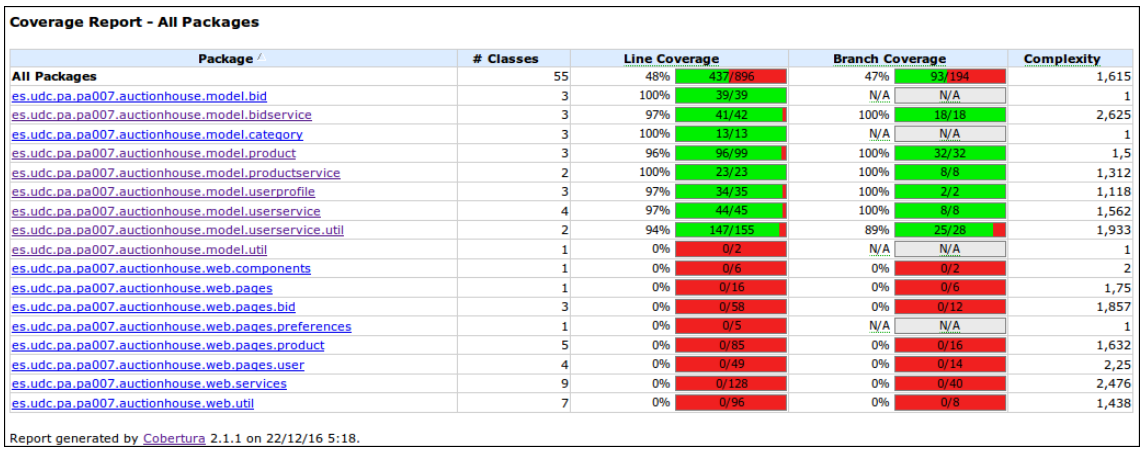
Las líneas sobre las que no se han realizado cobertura se corresponden a métodos de excepciones propias no testeadas y alguna línea de código implementada pero sin uso posterior. Las mutaciones que no han muerto corresponden únicamente a casos sin cobertura:

- 2 mutaciones correspondientes a el método getMessage() de dos excepciones propias que no checkean el caso de que este String pueda ser null
- 1 mutación correspondiente al método toString() redefinido por UserProfile que tampoco comprueba los valores null;
- 3 mutaciones correspondiente al método getTimeRemaing() de Product ya que no se testea esa funcionalidad por separada en los test. Se ha concluido en este punto ya que se considera que hemos obtenido una calidad considerable en los test

3.4. Cobertura

Como herramienta para comprobar la cobertura que realizan nuestros test sobre las líneas de código hemos utilizado la herramienta coberture de https://www.openhub.net/p/cobertura. Esta herramienta permite comprobar el número de líneas de código y de ramas que son comprobadas por nuestros test. Para incluir cobertura dentro del proyecto

hemos importado el plugin correspondiente de maven y hemos generado el informe a través del comando **mvn cobertura:cobertura**.



Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	55	48%	437/896	47%	93/194	1,615
es.udc.pa.pa007.auctionhouse.model.bid	3	100%	39/39	N/A	N/A	1
es.udc.pa.pa007.auctionhouse.model.bidservice	3	97%	41/42	100%	18/18	2,625
es.udc.pa.pa007.auctionhouse.model.category	3	100%	13/13	N/A	N/A	1
es.udc.pa.pa007.auctionhouse.model.product	3	96%	96/99	100%	32/32	1,5
es.udc.pa.pa007.auctionhouse.model.productservice	2	100%	23/23	100%	8/8	1,312
es.udc.pa.pa007.auctionhouse.model.userprofile	3	97%	34/35	100%	2/2	1,118
es.udc.pa.pa007.auctionhouse.model.userservice	4	97%	44/45	100%	8/8	1,562
es.udc.pa.pa007.auctionhouse.model.userservice.util	2	94%	147/155	89%	25/28	1,933
es.udc.pa.pa007.auctionhouse.model.util	1	0%	0/2	N/A	N/A	1
es.udc.pa.pa007.auctionhouse.web.components	1	0%	0/6	0%	0/2	2
es.udc.pa.pa007.auctionhouse.web.pages	1	0%	0/16	0%	0/6	1,75
es.udc.pa.pa007.auctionhouse.web.pages.bid	3	0%	0/58	0%	0/12	1,857
es.udc.pa.pa007.auctionhouse.web.pages.preferences	1	0%	0/5	N/A	N/A	1
es.udc.pa.pa007.auctionhouse.web.pages.product	5	0%	0/85	0%	0/16	1,632
es.udc.pa.pa007.auctionhouse.web.pages.user	4	0%	0/49	0%	0/14	2,25
es.udc.pa.pa007.auctionhouse.web.services	9	0%	0/128	0%	0/40	2,476
es.udc.pa.pa007.auctionhouse.web.util	7	0%	0/96	0%	0/8	1,438

Report generated by Cobertura 2.1.1 on 22/12/16 5:18.

Figura 2: *Captura del index.html generado por cobertura*

Esta captura se corresponde al index.html generado en el último análisis de cobertura. Como tanto la capa de la interfaz web como la clase serviceutil (proporcionada por el profesorado) no entran dentro de la fase testing, tras el último test hemos obtenido unos resultados reales de:

- Line Coverage: 290/296 97 %
- Branch Coverage: 68/68 100 %

Hemos obtenido una cobertura total de las ramas del proyecto y solo 6 líneas del proyecto correspondientes a métodos no testeados individualmente han quedado sin comprobar.

3.5. VisualVM

Como herramienta para realizar las pruebas de estrés sobre el proyecto hemos utilizado la herramienta VisualVm. Esta herramienta permite tras hacer un deploy del war del proyecto trabajar sobre una interfaz gráfica que permite ejecutar diferentes test de rendimiento que ponen a prueba la aplicación. Esta aplicación se ha instalado externamente al eclipse a través de la página `https://visualvm.github.io/`, desde donde se han realizado todas las pruebas.

Las características más importantes del ordenador sobre el que se realizaron las pruebas son:

- Procesador: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
- RAM: 16 GB
- Sistema Operativo: Windows10 64 bits
- Tarjeta Gráfica: Nvidia GeForce GTX 660M

Adjunto las capturas de pantallas tras realizar los diferentes test de estrés:

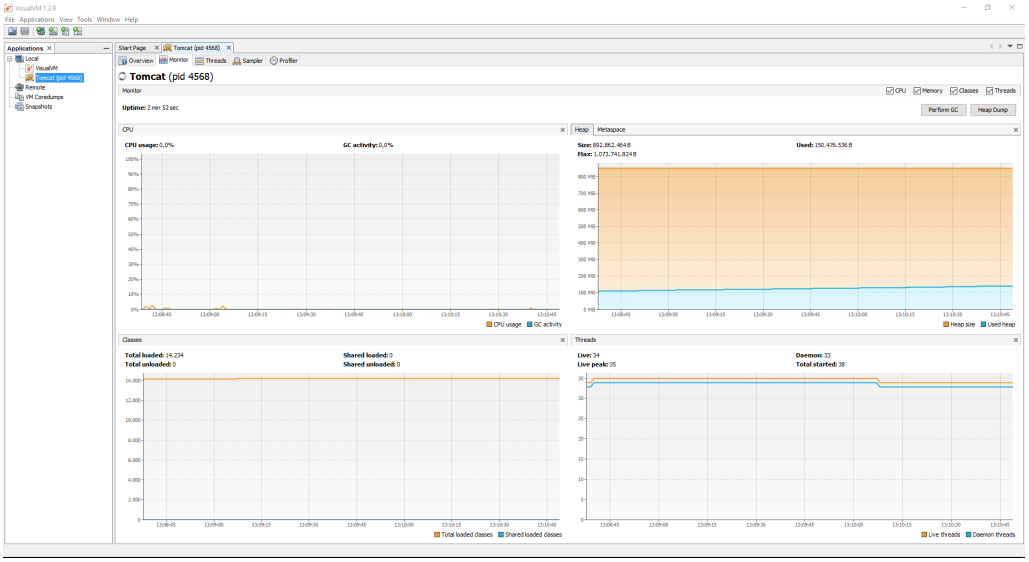


Figura 3: *Captura del estado inicial antes de las pruebas de estrés*

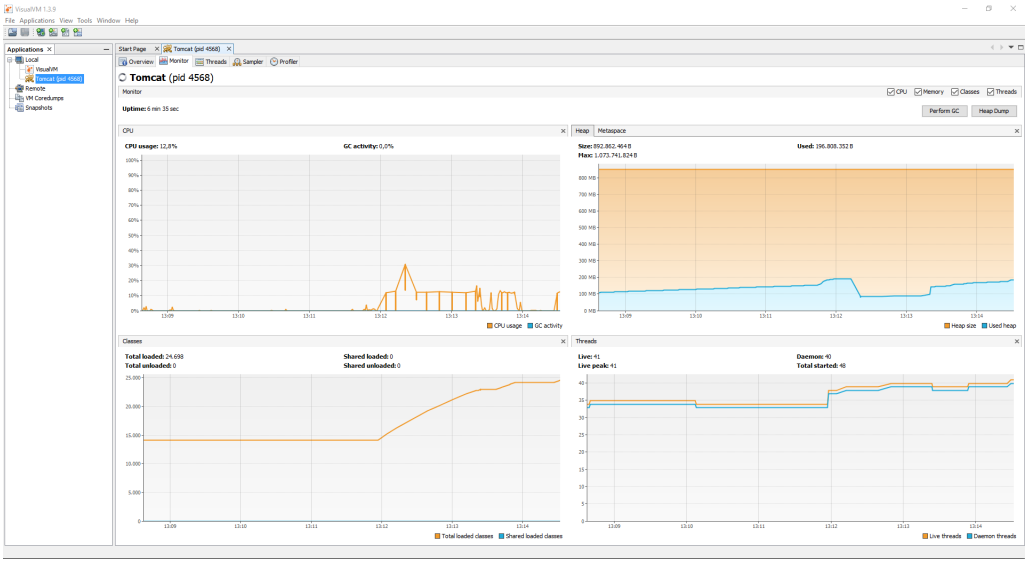
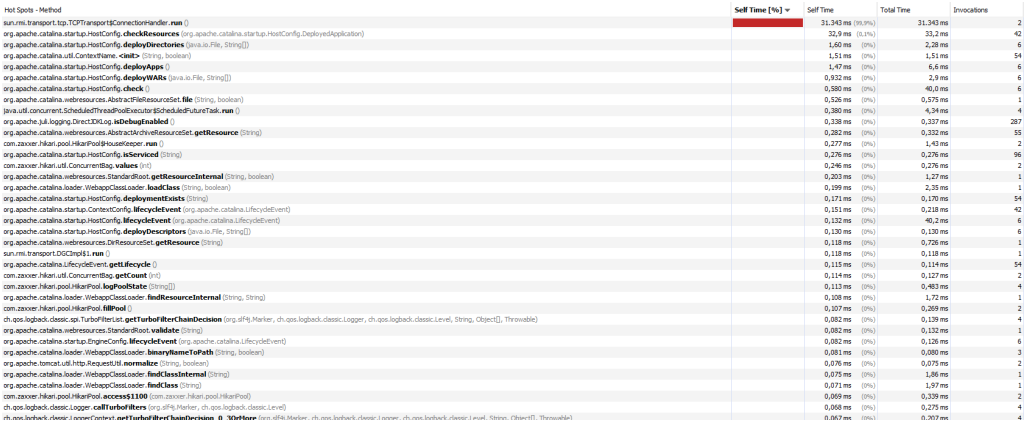


Figura 4: *Captura durante el test de cpu*



Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.**run** ()		31.343 ms (99,9%)	31.343 ms	2
org.apache.catalina.startup.HostConfig.**checkResources** (org.apache.catalina.startup.HostConfig.DeployedApplication)		32,9 ms (0,1%)	33,2 ms	42
org.apache.catalina.startup.HostConfig.**deployDirectories** (java.io.File, String[])		1,60 ms (0%)	2,28 ms	6
org.apache.catalina.util.ContextName.**<init>** (String, boolean)		1,51 ms (0%)	1,51 ms	54
org.apache.catalina.startup.HostConfig.**deployApps** ()		1,47 ms (0%)	6,6 ms	6
org.apache.catalina.startup.HostConfig.**deployWARs** (java.io.File, String[])		0,932 ms (0%)	2,9 ms	6
org.apache.catalina.startup.HostConfig.**check** ()		0,580 ms (0%)	40,0 ms	6
org.apache.catalina.webresources.AbstractFileResourceSet.**file** (String, boolean)		0,526 ms (0%)	0,575 ms	1
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.**run** ()		0,380 ms (0%)	4,34 ms	4
org.apache.juli.logging.DirectJDKLog.**isDebugEnabled** ()		0,338 ms (0%)	0,337 ms	287
org.apache.catalina.webresources.AbstractArchiveResourceSet.**getResource** (String)		0,282 ms (0%)	0,332 ms	55
com.zaxxer.hikari.pool.HikariPool$HouseKeeper.**run** ()		0,277 ms (0%)	1,43 ms	2
org.apache.catalina.startup.HostConfig.**isServiced** (String)		0,276 ms (0%)	0,276 ms	96
com.zaxxer.hikari.util.ConcurrentBag.**values** (int)		0,246 ms (0%)	0,276 ms	2
org.apache.catalina.webresources.StandardRoot.**getResourceInternal** (String, boolean)		0,203 ms (0%)	1,27 ms	1
org.apache.catalina.loader.WebappClassLoader.**loadClass** (String, boolean)		0,199 ms (0%)	2,35 ms	1
org.apache.catalina.startup.HostConfig.**deploymentExists** (String)		0,171 ms (0%)	0,170 ms	54
org.apache.catalina.startup.ContextConfig.**lifecycleEvent** (org.apache.catalina.LifecycleEvent)		0,151 ms (0%)	0,218 ms	42
org.apache.catalina.startup.HostConfig.**lifecycleEvent** (org.apache.catalina.LifecycleEvent)		0,132 ms (0%)	40,2 ms	6
org.apache.catalina.startup.HostConfig.**deployDescriptors** (java.io.File, String[])		0,130 ms (0%)	0,130 ms	6
org.apache.catalina.webresources.DirResourceSet.**getResource** (String)		0,118 ms (0%)	0,726 ms	1
sun.rmi.transport.DGCImpl$1.**run** ()		0,118 ms (0%)	0,118 ms	1
org.apache.catalina.LifecycleEvent.**getLifecycle** ()		0,115 ms (0%)	0,114 ms	54
com.zaxxer.hikari.util.ConcurrentBag.**getCount** (int)		0,114 ms (0%)	0,127 ms	2
com.zaxxer.hikari.pool.HikariPool.**logPoolState** (String[])		0,113 ms (0%)	0,483 ms	4
org.apache.catalina.loader.WebappClassLoader.**findResourceInternal** (String, String)		0,108 ms (0%)	1,72 ms	1
com.zaxxer.hikari.pool.HikariPool.**fillPool** ()		0,107 ms (0%)	0,269 ms	2
ch.qos.logback.classic.spi.TurboFilterList.**getTurboFilterChainDecision** (org.slf4j.Marker, ch.qos.logback.classic.Logger, ch.qos.logback.classic.Level, String, Object[], Throwable)		0,082 ms (0%)	0,139 ms	4
org.apache.catalina.webresources.StandardRoot.**validate** (String)		0,082 ms (0%)	0,132 ms	1
org.apache.catalina.startup.EngineConfig.**lifecycleEvent** (org.apache.catalina.LifecycleEvent)		0,082 ms (0%)	0,126 ms	6
org.apache.catalina.loader.WebappClassLoader.**binaryNameToPath** (String, boolean)		0,081 ms (0%)	0,080 ms	3
org.apache.tomcat.util.http.RequestUtil.**normalize** (String, boolean)		0,076 ms (0%)	0,075 ms	2
org.apache.catalina.loader.WebappClassLoader.**findClassInternal** (String)		0,075 ms (0%)	1,86 ms	1
org.apache.catalina.loader.WebappClassLoader.**findClass** (String)		0,071 ms (0%)	1,97 ms	1
com.zaxxer.hikari.pool.HikariPool.**access$1100** (com.zaxxer.hikari.pool.HikariPool)		0,069 ms (0%)	0,339 ms	2
ch.qos.logback.classic.Logger.**callTurboFilters** (org.slf4j.Marker, ch.qos.logback.classic.Level)		0,068 ms (0%)	0,275 ms	4
ch.qos.logback.classic.LoggerContext.**getTurboFilterChainDecision_0_3OrMore** (org.slf4j.Marker, ch.qos.logback.classic.Logger, ch.qos.logback.classic.Level, String, Object[], Throwable)		0,067 ms (0%)	0,307 ms	4

Figura 5: *Captura del resultado final de las pruebas de estrés*

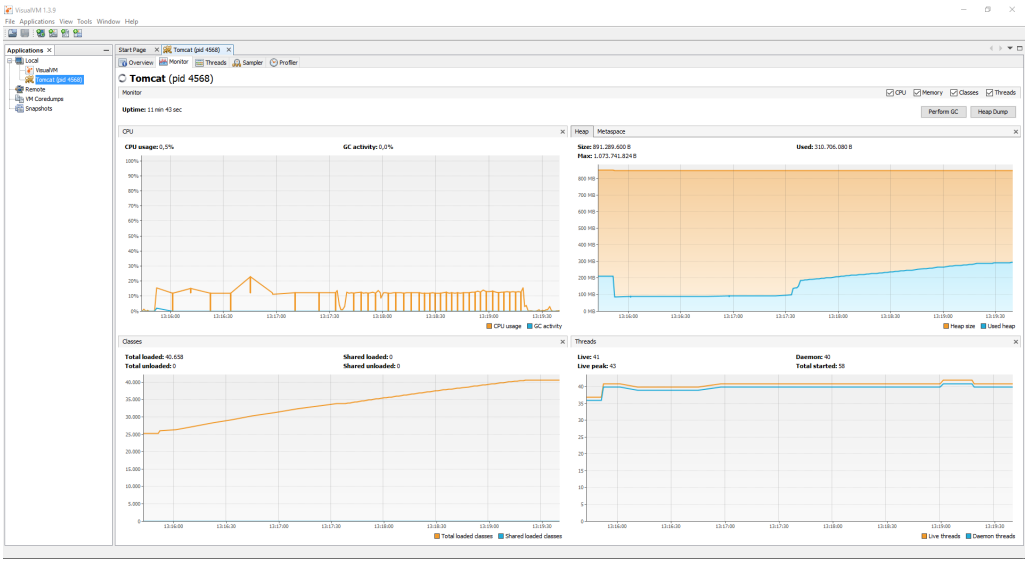


Figura 6: *Captura durante el test de memoria*

Figura 7: *Captura del resultado del test de memoria*

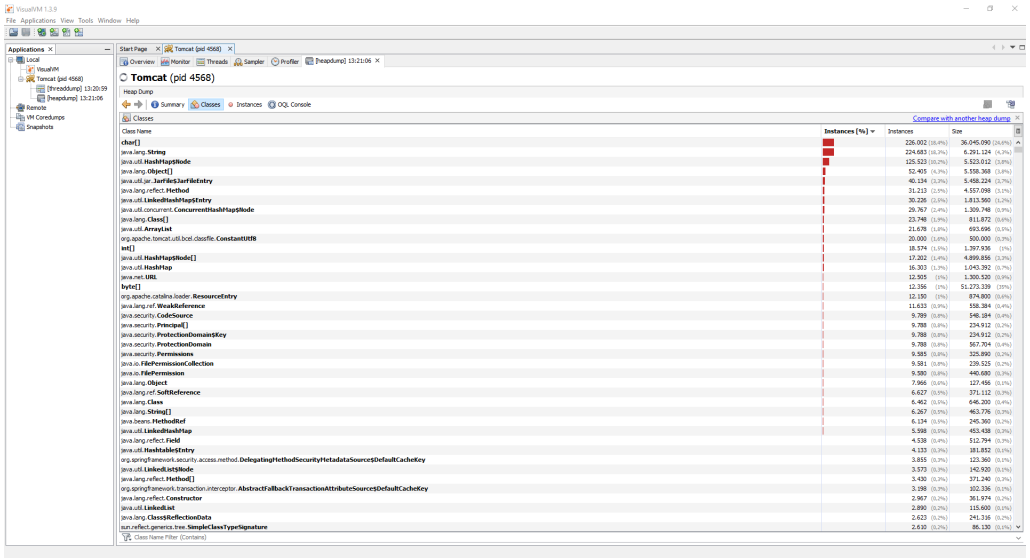


Figura 8: *Captura de las clases más empleadas de la aplicación durante las pruebas de estrés*

Todas estas capturas están almacenadas en https://github.com/ivangonzalezcotelo/practicaVVS/tree/master/doc/diagrams.

3.6. JUnit-QuickCheck

Para realizar pruebas empleando valores aleatorios nos hemos apoyado en la herramienta JUnit-QuickCheck que permite crear test que autogeneran valores aleatorios para comprobar como actuan nuestras clases ante diferentes valores. Para poder utilizarlo hemos incluido las dependencias maven y utilizado posteriormente los import en clases test.

4. Rexisto de probas

La mayoría de retrasos en la evolución de las iteraciones son producidas por el solapamiento con otras entregas de fecha de vencimiento más urgente que nos impidieron cumplir con el trabajo ideal semanal que corresponde a la evaluación continua de la asignatura. También hemos tenido retrasos debido a errores con las descargas de las distintas herramientas bien via maven o a través del propio marketplace de eclipse.

5. Rexistro de erros

Las pruebas tanto de integración como de unidad nos han permitido revelar errores de implementación de la aplicación sobre todo en los casos de los llamados valores frontera. Estos valores no estaban comprobados en los test ya existentes por lo que a pesar de que los test mostraban que la aplicación funcionaba sin problemas, no era así para dichos valores.

Las herramientas **coberture** y **piTest** también han permitido comprobar lineas y funcionalidades de la aplicación que no estaban siendo comprobados por lo que en algunos casos existían ciertos casos que presentaban un error en la implementación que no era descubierto ya que dicho código nunca era ejecutado por los test existentes.

6. Estatísticas

- **Número de erros encontrados diariamente e semanalmente:** Se han encontrado una media de 2 errores semanales.
- **Nivel de progreso na execución das probas:** Las pruebas ejecutan casi la totalidad del código(Comprobado gracias a la herramienta coberture)
- **Análise do perfil de detección de erros:** Los errores más comunes se corresponden a errores en comprobaciones (if) con motivo de los valores frontera y a problemas con zonas del código no testeadas.
- **Informe de erros abertos e pechados por nivel de criticidade:** No se han encontrado errores críticos en el proyecto ya que se encontraba en un estado de calidad aceptable antes de la ejecución de la fase testing, más que un problema en el primer build por la falta de un script que se tuvo que recrear. Los errores encontrados se han reportado como issues dentro del repositorio y se corresponden a errores menores de implementación, estilo o errores con los plugins de las heramientas descargadas.
- **Avaliación global do estado de calidade e estabilidade actuais:** La aplicación ha pasado los test de estrés con solvencia lo que permite ver que puede trabajar con peticiones concurrentes (VisualVM), tiene unos test que comprueban la totalidad de las funciones del código (PiTest y Coberture) y tiene el estilo correspondiente al estándar (CheckStyle), por lo que prodíamos decir que la aplicación es estable y tiene un nivel de calidad alto.

7. Outros aspectos de interese

A pesar de que ya ha sido mencionado, se han excluido de todos los test la clase serviceutil por ser algo previamente proporcionado por el profesorado y sobre la que no se han aplicado cambios y la interfaz web ya que ha sido recomendado apartarla de esta fase por la profesora de la asignatura de VVS.

No se ha utilizado la herramienta GraphWalker para generar pruebas a través del modelado de grafos porque encontramos muchos errores realizando el modelo y acabamos por abandonarlo por falta de tiempo.