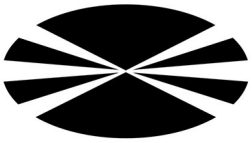


UNIVERSIDADE DA CORUÑA

Herramientas de Desarrollo

Cuarta iteración

Grupo 1.2

2021/2022

Jose Manuel Turnes Pazos	**jose.turnes**
Miguel Quiroga Rodriguez	**miguel.quiroga**
Brais Cibeira Pousa	**brais.cibeira**
Marcos Fernández Pol	**marcos.pol**
Antón Gendra Rodríguez	**anton.gendra**
Rita Cernadas Tubío	**rita.cernadas**

Índice

1. Introducción.

En esta iteración trataremos los diferentes tiempos medios de respuesta del servidor para los diferentes tipos de solicitudes y también estudiaremos la aplicación para cualquier tipo de pérdida de memoria o procesamiento.

2. Herramientas.

En este apartado trataremos las herramientas empleadas a lo largo del desarrollo de la práctica, desde la herramienta principal como es el IDE hasta herramientas de gestión de proyecto como es el Redmine.

2.1. Entornos integrados de desarrollo.

Aquí es donde empieza todo, el IDE es la herramienta más importante para poder realizar el desarrollo del código, en nuestro caso utilizamos **Eclipse**, que mejora el desarrollo con java gracias a las ayudas integradas mejorando la experiencia del desarrollador. Es el segundo IDE más usado y además de licencia libre. En nuestro caso las características que más empleamos son el uso de la integración de git con el IDE para poder hacer las actualizaciones de código y moverse entre ramas con mayor rapidez.

2.2. Sistemas de control de versiones.

Gracias al uso de **GitLab**, cada integrante del grupo podía crear una rama para cada tarea a partir de una rama concreta que usábamos para el desarrollo de cada iteración (develop). En esta rama vamos añadiendo el código de cada tarea justo después de testear que funciona, para intentar tener un entorno estable en develop.

Antes de incorporar nuevo código a la rama, hacemos uso de las merge request de Gitlab en las que mínimo otro compañero debe revisar que todo está bien y así evitamos futuros problemas. Además con el uso de merge request el revisor puede escribir comentarios para corregir fallos o aclarar dudas, los cuales quedan registrados y en todo momento cualquiera miembro del equipo podrá ver desde cualquier dispositivo en Gitlab.

Con todo esto, todos trabajamos a la vez de forma paralela en el proyecto cada uno en su tarea con lo que agilizamos el desarrollo.

2.3. Sistemas de gestión de proyectos.

Con la inclusión de **Redmine** en el proyecto fuimos capaces de coordinar mejor el equipo y tener una mejor planificación y organización en cada sprint. Nos ayudó a repartir tareas y a no perder tiempo en la selección de ellas debido a que tenemos un diagrama de Gantt con las dependencias entre tareas.

Además, al crear historias de usuario para las tareas, podíamos anotar información relevante dada por el cliente para tenerla a mano todo el equipo y podemos verlo en todo momento desde cualquier dispositivo ya que se trata de una plataforma web.

Por último, mencionar que con la ayuda de la wiki pudimos anotar lo que íbamos haciendo en cada iteración para tener un poco de contexto si en algún momento se necesitase. Además al conectar Redmine con GitLab, ahorramos tiempo en añadir información a las tareas ya que dándole un formato específico a los commits podemos por ejemplo imputar el tiempo en las tareas.

2.4. Automatización de Empaquetado.

Gracias a **Maven** podemos gestionar el proyecto entero, desde la etapa en la que comprobamos si el código es correcto hasta el despliegue de la aplicación y pasando por la ejecución de las pruebas. En resumen, realizamos la validación, compilación, test, empaquetamiento, install e incluso el despliegue.

2.5. Integración Continua.

Gracias a **Jenkins** podemos gestionar de manera remota las integraciones del proyecto, ya que está conectado con Gitlab, y cuando se hace un push o una merge request el propio Jenkins inicia una nueva integración con el código actualizado.

Además, jenkins también lanza tanto las ejecuciones de SonarQube, que sirve para comprobar la calidad del código mediante una serie de reglas, así como docker y kubernetes, los cuales nos permiten ejecutar nuestra aplicación en remoto sobre el servidor *https://deploy.fic.udc.es/medusa*.

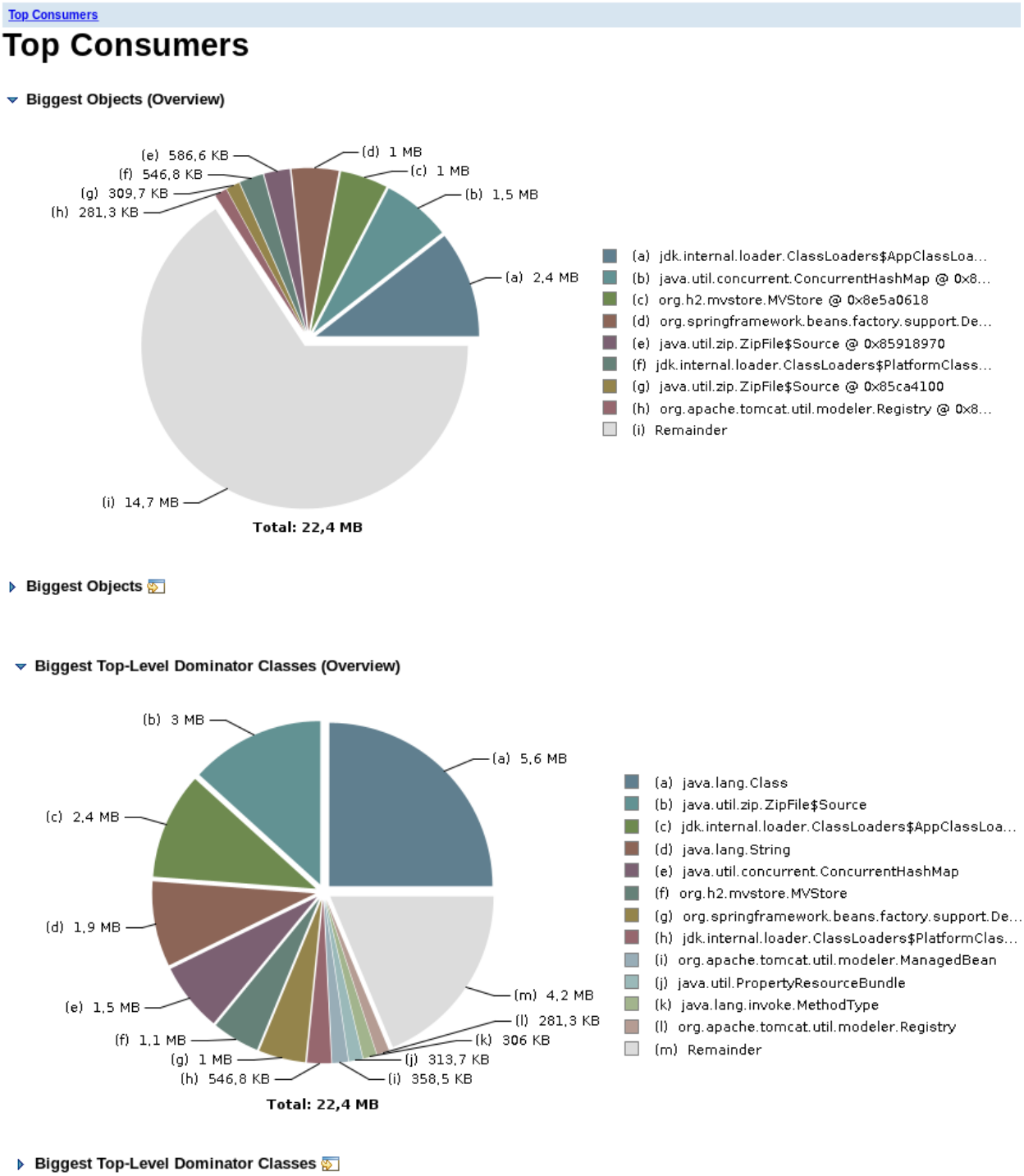
2.6. Inspección Continua.

Al añadir **SonarQube** al proyecto pudimos ver la calidad del código que estábamos desarrollando, es decir estábamos realizando un análisis estático sobre el código. Gracias a este análisis pudimos ver una evaluación de nuestro código y descubrir el código duplicado, los code smell, vulnerabilidades, bugs, etc. y aprendimos a cómo solucionarlos. A mayores, también podemos comprobar la cobertura de los test.

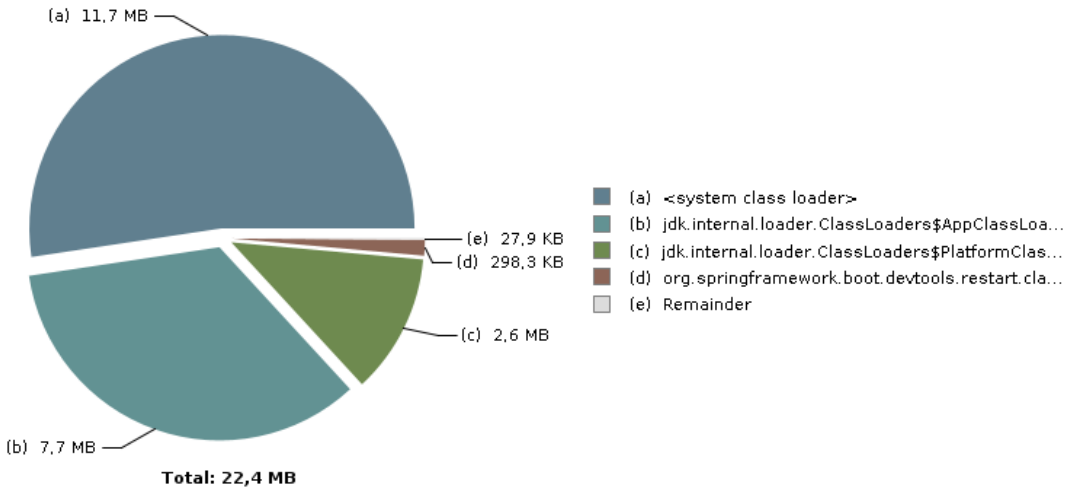
3. Resultados test.

En este apartado trataremos los resultados de los distintos test, desde los test de pérdida de memoria hasta los test de carga.

Empezamos con los test de pérdida de memoria, que para ello utilizamos Eclipse MAT.



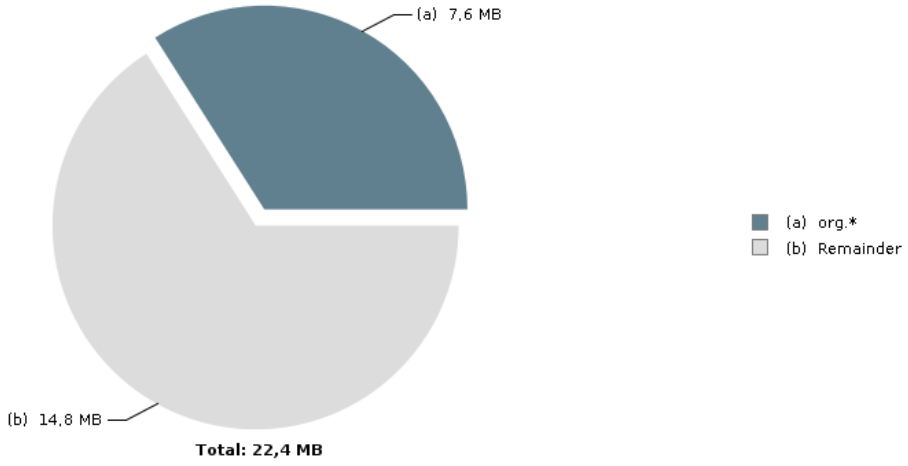
Biggest Top-Level Dominator Class Loaders (Overview)



Component Report org.*

Component Report org.*

Size: **7,6 MB** Classes: **3,3k** Objects: **177,2k** Class Loader: **11**



Possible Memory Waste

Duplicate Strings

Found 1 occurrences of char[] with at least 10 instances having identical content. Total size is 2.240 bytes.

Top elements include:

- 28 × **\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u...** (2.240 bytes)

Details »

Empty Collections

No excessive usage of empty collections found.

Details »

Collection Fill Ratios

Detected the following collections with fill ratios below 20%:

- 1.356 instances of **java.util.LinkedHashMap** retain **>= 446.136** bytes.

Details »

Zero-Length Arrays

No excessive usage of zero-length arrays found.

Details »

Array Fill Ratios

No serious amount of arrays with low fill ratios found.

Details »

Primitive Arrays with a Constant Value

No excessive usage of primitive arrays with a constant value found.

Details »

Miscellaneous

Soft Reference Statistics

A total of 1.206 java.lang.ref.SoftReference objects have been found, which softly reference 85 objects.
No objects totalling 0 B are retained (kept alive) only via soft references.
No objects totalling 0 B are softly referenced and also strongly retained (kept alive) via soft references.

Details »

Weak Reference Statistics

A total of 1.221 java.lang.ref.WeakReference objects have been found, which weakly reference 332 objects.
No objects totalling 0 B are retained (kept alive) only via weak references.
No objects totalling 0 B are weakly referenced and also strongly retained (kept alive) via weak references.

Details »

Finalizer Statistics

A total of 1 object implement the finalize method.

Details »

Map Collision Ratios

No maps found with collision ratios greater than 80%.

Details »

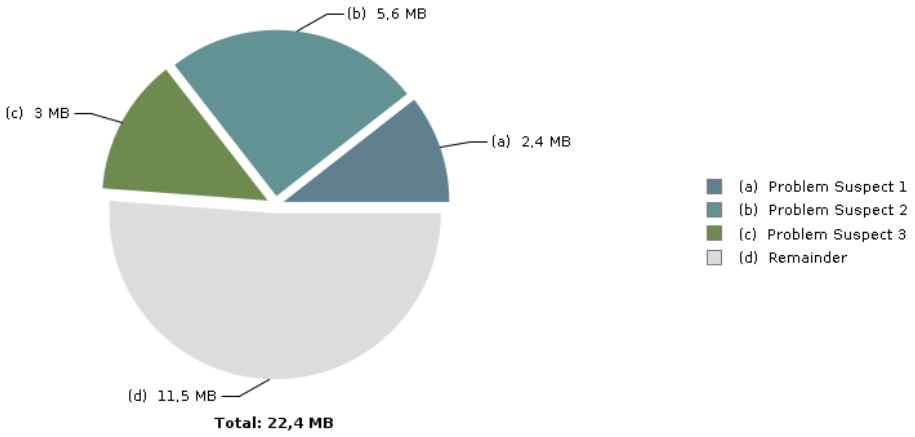
Table Of Contents

Leak Suspects

System Overview

Leaks

Overview



Problem Suspect 1

The classloader/component **"jdk.internal.loader.ClassLoaders$AppClassLoader @ 0x85b05590"** occupies **2.477.768 (10,55 %)** bytes. The memory is accumulated in classloader/component **"jdk.internal.loader.ClassLoaders$AppClassLoader @ 0x85b05590"** which occupies **2.477.768 (10,55 %)** bytes.

Keywords

jdk.internal.loader.ClassLoaders$AppClassLoader @ 0x85b05590

Details »

Problem Suspect 2

7.336 instances of **"java.lang.Class"**, loaded by **"<system class loader>"** occupy **5.869.056 (24,98 %)** bytes.

Keywords

java.lang.Class

Details »

Problem Suspect 3

66 instances of **"java.util.zip.ZipFile$Source"**, loaded by **"<system class loader>"** occupy **3.122.800 (13,29 %)** bytes.

Biggest instances:

- java.util.zip.ZipFile$Source @ 0x85918970 - 600.656 (2,56 %) bytes.
- java.util.zip.ZipFile$Source @ 0x85ca4100 - 317.120 (1,35 %) bytes.

These instances are referenced from one instance of **"java.util.HashMap$Node[]"**, loaded by **"<system class loader>"**, which occupies **2.640 (0,01 %)** bytes.

Keywords

java.util.zip.ZipFile$Source
java.util.HashMap$Node[]

Details »

Y por último tenemos los test de carga, para ello usamos Apache JMeter. Estos test los podemos ver en la carpeta añadida al repositorio.