

UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA

HERRAMIENTAS DE DESARROLLO
GRADO EN INGENIERÍA INFORMÁTICA

Softfic Running

Manuel Daniel Gabín Brenlla

Alejandro Matos García

Brais Muñiz Castro

Paloma Piot Pérez-Abadín

Juan Manuel Rey Escobar

David Touriño Calvo

David Vila Sánchez

A Coruña a 18 de diciembre de 2017

Índice

1. Ejecución del proyecto	2
2. Información de las herramientas	2
2.1. Eclipse	2
2.1.1. Vistas	2
2.1.2. Acciones para ahorrar tiempo de codificación	2
2.1.3. Debugging	3
2.1.4. Puntos de parada	3
2.2. Git	3
2.2.1. Estrategia de ramas	3
2.3. Redmine	4
2.3.1. Issues	4
2.3.2. Wiki	4
2.3.3. SCRUM en Redmine	4
2.3.4. Redmine y Git	5
2.4. Maven	5
2.4.1. Pom	5
2.4.2. Perfiles	5
2.4.3. Dependencias	5
2.4.4. Proyecto Multimódulo	6
2.5. Jenkins	7
2.6. SonarQube	7
2.6.1. Análisis con Jenkins	7
2.6.2. SonarLint	7
2.6.3. Resultados	7
2.7. JMC	7
2.8. JMeter	7
2.9. Tecnologías	8
2.9.1. JUnit	8
2.9.2. Mockito	8
2.9.3. JaCoCo	8
2.9.4. Cargo	8
2.9.5. Selenium	8
3. Planificación y Desviaciones	9
3.1. Planificación	9
3.2. Desviaciones	11
4. Pruebas de carga y Profiling	12
4.1. Entorno de ejecución de las pruebas	12
5. Estadísticas del código y proyecto	17
6. Referencias	20

1. Ejecución del proyecto

Para la ejecución del proyecto con el perfil de desarrollo, debe estar corriendo la base de datos de **MySQL** y, como se trata de un proyecto **Maven**, se realizará de la siguiente forma:

```
1 mvn install
```

Si se desea ejecutar la aplicación en el entorno de producción, habrá que indicarlo:

```
1 mvn install -Pprod
```

Esto generará un *war* que habrá que desplegar en un servidor de aplicaciones como, por ejemplo, **Tomcat**. El servidor de **Jenkins** se encarga de realizar esto.

Como el proyecto está en un repositorio de **GitLab**, habrá que bajarse, primero, la última versión (en nuestro caso, la correspondiente al *tag* 4.0):

```
1 git checkout tags/4.0
```

2. Información de las herramientas

2.1. Eclipse

Eclipse [1] es un Entorno de Desarrollo Integrado (IDE) que dispone de cientos de *plugins* libres, de un editor de texto con un analizador sintáctico, la compilación es en tiempo real, tiene pruebas unitarias con **JUnit**, control de versiones con **CVS**, integración con **Ant**, asistentes (*wizards*) para creación de proyectos, clases, tests, etc. y refactorización.

2.1.1. Vistas

Se ha hecho uso de distintas vistas para programar de una forma más cómoda. Al realizar un *debug* se ha usado la vista de **debugging**, y a la hora de desarrollar la aplicación la de **Java EE**.

2.1.2. Acciones para ahorrar tiempo de codificación

Se ha personalizado el editor con *save actions* para que cada vez que se guarden los cambios realizados sobre una clase haga un *refactor* del código y añada los *import* necesarios.

Se ha hecho uso de los generadores automáticos de *getters* y *setters*, del constructor, del *toString*, para facilitar el *debugging*, etc. Asimismo, al hacer un *renaming* no se ha ido palabra a palabra, sino que se ha usado la opción de sustitución automática.

También se ha hecho uso de los *shortcuts* para así ahorrar tiempo a la hora de programar.

2.1.3. Debugging

Debuggear permite correr un programa mientras vemos el código fuente y el valor de las variables.

2.1.4. Puntos de parada

Lo más utilizado han sido los *breakpoints*, para parar la ejecución en el punto indicado. Los *watchpoints* fueron utilizados menos veces, ya que solo paran la ejecución si una variable es usada.

Al realizar el *debugging* se han utilizado *shortcuts* F5, F6, F7, F8 y Ctrl + R para correr el *debug*.

2.2. Git

Durante el desarrollo del proyecto se ha utilizado **Git** [2] como herramienta de control de versiones y **GitLab** [3] como servicio web de control de versión y desarrollo de software colaborativo, basado en **Git**.

Cuando comenzó el proyecto, la idea original era instanciar un arquetipo de **Maven** y subirlo al repositorio para inicializarlo. Pero, por problemas técnicos, no fue posible. Entonces se nos facilitó un arquetipo ya instanciado en un repositorio ya inicializado, de modo que, simplemente se tuvo que clonar.

2.2.1. Estrategia de ramas

Cuando comenzó el proyecto existía una única rama **master** y a partir de esta rama se creó otra en local llamada **develop**, donde se realizaba el desarrollo. Desde esta rama se iban creando las diferentes ramas **feature** (cuando se realizaba una funcionalidad), **hot-fix** (para arreglar fallos detectados) y **task** (para realizar una tarea) y luego se mergeaban a **develop**. Se ha seguido esta aproximación en el *Sprint 1*.

A partir del *Sprint 2*, se ha comenzado a usar *Gitflow* que consiste en crear una rama de desarrollo, **develop**, en remoto, de la cual se creaban las distintas ramas ya mencionadas. También, para mantener las distintas ramas en remoto se realizaban los merge con la opción *-no-ff*.

Para el *Sprint 4*, el último, se ha seguido la filosofía del *merge request*, realizados a través de **GitLab**, cuyo objetivo es que otro miembro del equipo de desarrollo revise la tarea o funcionalidad realizada, si está correcta, aprueba el *merge*, en caso contrario, se rechaza justificando el porqué.

En todos los sprints, a excepción del primero, se ha gestionado una rama **release**, para liberar la versión. Si en este punto se detectaba algún *bug*, se creaba una rama desde **release** y se corregía. A continuación, cuando el producto no tenía errores, se mergeaba en **master** asociado a un *Tag*.

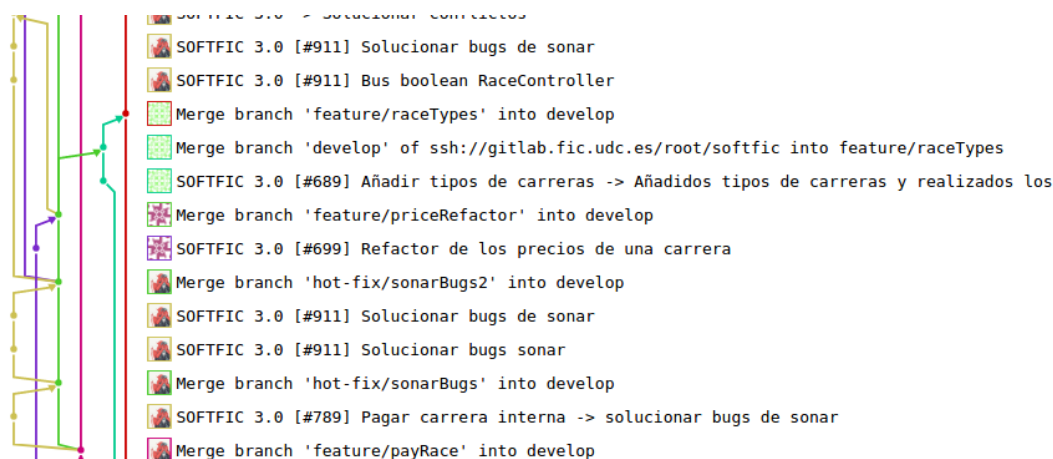


Figura 1: Grafo de una parte del proyecto, donde se refleja la estrategia de ramas aplicada.

2.3. Redmine

Para la gestión del proyecto se ha utilizado **Redmine** [4], una herramienta que a través de *issues* permite dividir el proyecto en componentes más pequeños, organizar tareas y medir el progreso, investigar y resolver *bugs* tras el desarrollo y mejorar la comunicación entre los integrantes del equipo.

2.3.1. Issues

Se ha seguido la siguiente filosofía de uso de **Redmine**. Para cada tarea se ha creado una *issue* especificando su tipo (*Errores*, *Tareas* y *Soporte*), su duración estimada, las fechas de comienzo y fin estimadas, la versión a la que pertenece, su prioridad y una descripción de trabajo a realizar.

Una vez definidas todas las tareas del *sprint*, cada integrante del equipo escogía una para realizar, asignándosela en **Redmine**.

A medida que la tarea iba evolucionando, se imputaban las horas dedicadas, indicando el tipo de trabajo realizado (*Diseño* o *desarrollo*) y el porcentaje de completitud de la tarea.

Cuando la tarea se finalizaba, se marcaba como *Resuelta* y realizada al 100 %. En este punto, si quedan tareas sin asignar, escogería otra y repetiría el proceso.

2.3.2. Wiki

Se ha mantenido una *Wiki* del proyecto, donde, por cada versión, se han hecho páginas con información relativa a la misma.

2.3.3. SCRUM en Redmine

A lo largo del último *sprint* se ha trabajado siguiendo la metodología **SCRUM** [5], por lo tanto se ha habilitado en **Redmine** un panel *SCRUM* y se han creado *Historias de Usuario* dentro de las cuales se creaban las tareas relativas a las mismas.

2.3.4. Redmine y Git

Al realizar un *commit*, se ha indicado el número de tarea correspondiente, precedido de una almohadilla, para así enlazarlo a la *issue* de **Redmine**.

Latest revisions

#		Date	Author	Comment
e5e210bf	●	12/17/2017 03:43 PM	Paloma Piot Pérez-Abadín	Merge branch 'support/SeleniumTests' into 'develop' Support/selenium tests See merge request root/softfic/28
c1bcbaac	●	12/17/2017 03:11 PM	David Touriño Calvo	SOFTFIC 4.0 [#1264] Tests automatizados con selenium -> Resolucion de conflictos + adaptacion a cambios
efc16960	●	12/17/2017 12:05 PM	Manuel Daniel Gabín Brenlla	Merge branch 'hot-fix/fixSonar' into develop
8a1d7426	●	12/17/2017 03:08 AM	Paloma Piot Pérez-Abadín	SOFTFIC 4.0 [#1320] Arreglar bugs de Sonar -> Solucionar bugs de sonar + mejorar interfaz
c8a7c1f9	●	12/17/2017 12:50 AM	David Touriño Calvo	SOFTFIC [#1264] Tests automatizados con Selenium ->Adaptados tests a las nuevas funcionalidades, e intento de jacoco
0d1bb842	●	12/16/2017 08:37 PM	Paloma Piot Pérez-Abadín	SOFTFIC 4.0 [#1320] Arreglar bugs de Sonar -> Solucionar fallos detectados
b84c4a06	●	12/16/2017 07:40 PM	Paloma Piot Pérez-Abadín	SOFTFIC 4.0 [#1320] Arreglar bugs de Sonar -> Crear enumerado para el estado de seguimiento + arreglar fallo provocado tras solucionar un bug
caa4a49a	●	12/16/2017 06:34 PM	Paloma Piot Pérez-Abadín	SOFTFIC 4.0 [#1320] Arreglar bugs de Sonar -> arreglar parte de los bugs
9a3b14f7	●	12/16/2017 02:57 PM	David Touriño Calvo	Merge branch 'develop' of ssh://gitlab.fic.udc.es/root/softfic into support/SeleniumTests
db1533c7	●	12/16/2017 02:56 PM	David Touriño Calvo	SOFTFIC [#1264] Tests automatizados con Selenium Arreglados tests en linux

Figura 2: Vista de Redmine donde se asocia el comentario de un commit a una tarea.

2.4. Maven

Se ha usado **Maven** [6] como herramienta para la gestión y la construcción del proyecto.

Maven no solo proporciona la capacidad de construir de forma rápida y eficiente un proyecto, sino que también proporciona la capacidad de gestionar aspectos tan relevantes como son las dependencias del proyecto, proporcionar una estructura consistente o generar un site.

2.4.1. Pom

Es un fichero XML que describe el proyecto. En él se ha incluido el nombre y versión del proyecto (que iba incrementando cuando se realizaban funcionalidades nuevas), datos necesarios para el *Maven Site*, como los nombres de los desarrolladores, todo lo necesario para las dependencias, plugins y perfiles.

2.4.2. Perfiles

Se han creado dos perfiles en **Maven**: en uno de ellos, se han definido las propiedades de la base de datos para usar *MySQL*, mientras que en el otro perfil, se utiliza *HSQldb*. Esta división es debido a que en la máquina donde se hace *deploy* no hay una instancia de *MySQL* y en su lugar utilizamos *HSQldb*. En un entorno real, lo más correcto habría sido realizarlo a la inversa.

2.4.3. Dependencias

Para los tests automatizados, era necesario añadir unas dependencias, que entraban en conflicto por la versión definida en el *pom* padre. Para solucionarlo, se añadieron las dependencias necesarias dentro del *tag* de *DependencyManager*.

2.4.4. Proyecto Multimódulo

Maven permite separar un proyecto en distintos módulos. Esto nos permite separar los componentes de la aplicación según la funcionalidad, ejecutar la construcción solamente del módulo en el que se está trabajando, reduciendo el tiempo de construcción del proyecto, mejorar la organización de las dependencias del proyecto, la legibilidad de la configuración en el *pom*, etc.

Se ha dividido el proyecto en dos módulos:

- **Controller:**

Contiene los controladores de la aplicación web y los *templates* de la vista.

- **Model:**

Contiene la funcionalidad independiente de la web: los servicios, la capa de acceso a datos y las entidades.

Se han repartido los *tests* en cada proyecto de forma que cada proyecto prueba la funcionalidad de la que se ocupa.

Para dividir el proyecto en módulos se ha procedido de la forma siguiente:

- Se han creado dos directorios en la raíz del proyecto: *softfic-controller* y *softfic-model*.
- Se ha copiado el *src* del proyecto en cada módulo y se les ha quitado a cada uno la parte no relevante en su contexto.
- Después hubo que modificar los ficheros *pom* de cada proyecto:
 - Se ha cambiado la configuración del *pom* padre:
 - *Packaging: pom*.
 - Se inserta la etiqueta `<modules>` enumerando los módulos de la aplicación.
 - Se ha añadido a este *pom* lo que se ha considerado general de ambos submódulos para que obtengan dicha configuración por herencia (la configuración de *maven site*, perfiles y algunas dependencias).
 - En los submódulos:
 - Se han cambiado los `<artifactId>` para cuadrar con lo referenciado en el padre.
 - Se ha cambiado `<parent>` para que referencia al proyecto padre.
 - Se ha eliminado todo lo que no era necesario para cada uno, aislando la configuración individual de cada proyecto.

Se han dejado algunas cosas sin realizar por falta de tiempo:

- Las dependencias en el *pom* podrían haberse optimizado más para que cada proyecto solo declarase las que necesita.
- Se intentó conseguir que solo hubiese un punto en la configuración donde cambiar la versión global de la aplicación. Se creó una `<propertie>` en el *pom* padre e intentó usarse la variable en los hijos para configurar las versiones, pero daba problemas al compilar algunos módulos.

Para que **SonarQube** analizase de forma correcta los distintos módulos, hubo que indicar en la configuración de **Jenkins** la distribución de los módulos.

2.5. Jenkins

Jenkins [7] proporciona integración continua para el desarrollo software. Es un sistema corriendo en un servidor que es un contenedor de servlets como *Apache Tomcat*. Soporta herramientas de control de versiones y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como *scripts* de shell y programas batch de Windows.

Se ha trabajado con un proyecto **Maven** y se ha creado indicando un nombre, una descripción, la ubicación de **Redmine**, el **SCM** para el código fuente y diferentes acciones pre y post build y el deploy remoto.

2.6. SonarQube

SonarQube [8] es una plataforma para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente como *Checkstyle*, *PMD* o *FindBugs* para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.

2.6.1. Análisis con Jenkins

Se ha configurado **Jenkins** para que cada vez que se realice una integración, **SonarQube** analice el proyecto. Hemos implementado esto para las ramas **master**, **develop** y **release**. También, en algún momento puntual, creamos alguna rama para solucionar *bugs* o *code smells* de **SonarQube** y en estos casos, también se analizaba dichas ramas.

2.6.2. SonarLint

Se ha incorporado un *plugin* de **SonarQube** para **Eclipse: SonarLint** [9]. No ha sido posible enlazarlos, ya que existe un *bug* en la versión utilizada. Pero ha sido de gran utilidad para ver los *bugs* directamente desde **Eclipse**. También, se comprobó que algunos *bugs* que se mostraban en el *plugin* eran distintos a los de **SonarQube**.

2.6.3. Resultados

Resultados del análisis de **SonarQube** en la rama **develop** con la última actualización del código fuente.

2.7. JMC

Java Mission Control [10] (JMC) es una herramienta para recopilar continuamente información detallada sobre el tiempo de ejecución y permite el posterior análisis de incidentes.

Se ha utilizado para realizar el *profiling* de la aplicación mientras se ejecutaban las pruebas de carga con **JMeter**. Se explicará en el apartado de **Pruebas de Carga y Profiling**.

2.8. JMeter

JMeter es una herramienta de software diseñada para ejecutar pruebas de carga y medir su comportamiento.

Se ha utilizado para lo propio. La configuración de **JMeter** para las pruebas de carga se explicará en el apartado de **Pruebas de Carga y Profiling**.

2.9. Tecnologías

2.9.1. JUnit

JUnit [12] es un *framework* utilizado para programar tests. Se ha empleado para realizar todas las pruebas unitarias del proyecto y realizar aserciones para comprobar el correcto funcionamiento de los servicios.

2.9.2. Mockito

Mockito [13] es un *framework* de *mocking* que permite escribir test con un API simple. La librería de *Mockito* permite simular creaciones, verificación y troquelado. Se ha empleado para mockear en los tests de unidad.

2.9.3. JaCoCo

JaCoCo [14] es un librería Java libre para controlar la cobertura de código. Genera distintos informes de cobertura tanto de los tests unitarios como de integración.

Se ha configurado el *plugin* de **JaCoCo** para **Maven** en el *pom.xml* para analizar la cobertura del proyecto. Cuando se ejecutan los *tests* se genera un informe de la cobertura de los mismos. Además, se ha integrado con el *maven site* de manera que se refleja la cobertura en los reportes.

En la configuración del trabajo de **Jenkins** se ha tenido que configurar algunas variables de **SonarQube** para integrarlo con **JaCoCo**, para que aparezca la cobertura en los informes de **SonarQube**.

2.9.4. Cargo

Cargo [15] es un *plugin* que permite manipular contenedores de aplicaciones como *tomcat*. Se ha utilizado para que en la ejecución de las fases *pre-integration-test* y *post-integration-test* ejecute un servidor y lo mate, respectivamente. Este servidor se ha utilizado para los tests de aceptación de **Selenium**.

2.9.5. Selenium

Selenium [16] automatiza los navegadores. En nuestra aplicación se ha utilizado para ejecutar los tests de aceptación pedidos por el cliente para que se efectúen de forma automatizada.

Los tests implementados con **Selenium** son los test de aceptación acordados con el cliente:

- Test Carballiño:

El navegador arranca, y busca carreras en un radio de 30km de O Carballiño. Las carreras deberán tener un recorrido con una distancia de al menos 15km, no podrán costar más de 30€ y no tendrán test médico. Al buscar, encuentra una carrera en Longoseiros, el test comprueba que esa carrera es la buscada y termina.

- Test dar de alta:

El test dará de alta una carrera en el sistema, como para dar de alta una carrera hace falta ser un usuario registrado, lo que primero se hace es autenticarse con una cuenta de ejemplo (user@example.com). Al añadir la carrera, se introducen los campos necesarios, concretamente, la carrera será en Ferrol el 30 de diciembre de 2017, tendrá chip que se podrá alquilar de forma opcional por 5€ y costará 80€ para los mayores de 40 años y 100€ para el resto. Al añadir la carrera, el test comprueba que redirige a la página de detalles de la misma, donde se comprueba que los campos son los introducidos.

- Test de Paca:

Lo primero que hace el test es autenticarse con la cuenta de `dani_valcarce`. Una vez autenticado, se comprueban las carreras a las que van a ir Paca (la prima de Dani Valcarce) y Javier Gómez Noya, por lo que lo primero que hace el test es seguirlos. Primero sigue a Javier, donde comprueba en su perfil que asistirá a un triatlón. Después, sigue a Paca, y observa en su perfil que también está anotada en ese triatlón. Además, comprueba que Paca quedó en quinta posición en una media maratón en Bilbao. Después, busca más información sobre dicha triatlón, y ve que hay 23 usuarios anotados, de los cuales cuatro son usuarios a los que sigue.

Para su ejecución, están configurados como tests de integración, que se ejecutan con el *plugin failsafe* en **Maven** en la fase *integration-test*. Como para que funcione **Selenium** hace falta tener arrancado el servidor, utilizamos el *plugin* de **Cargo** para iniciar un servidor **Tomcat** en la fase *pre-integration-test* y matarlo en la fase *post-integration-test*.

El *driver* para el navegador utilizado es *geckodriver*, para *Firefox*, que se encuentra en la raíz del proyecto.

3. Planificación y Desviaciones

Durante el desarrollo del proyecto, se han seguido cierto criterios a la hora de planificar las funcionalidades a desarrollar en cada iteración y también a la hora de programar la duración de cada una de las *issues* creadas en el gestor de proyectos **Redmine**.

3.1. Planificación

A la hora de realizar la planificación de las distintas iteraciones, se ha tenido en cuenta la **prioridad** asignada por el usuario. Asimismo, se ha intentado hacer ver al usuario el punto de vista del equipo de desarrollo en cuanto a la prioridad de las *historias de usuario*.

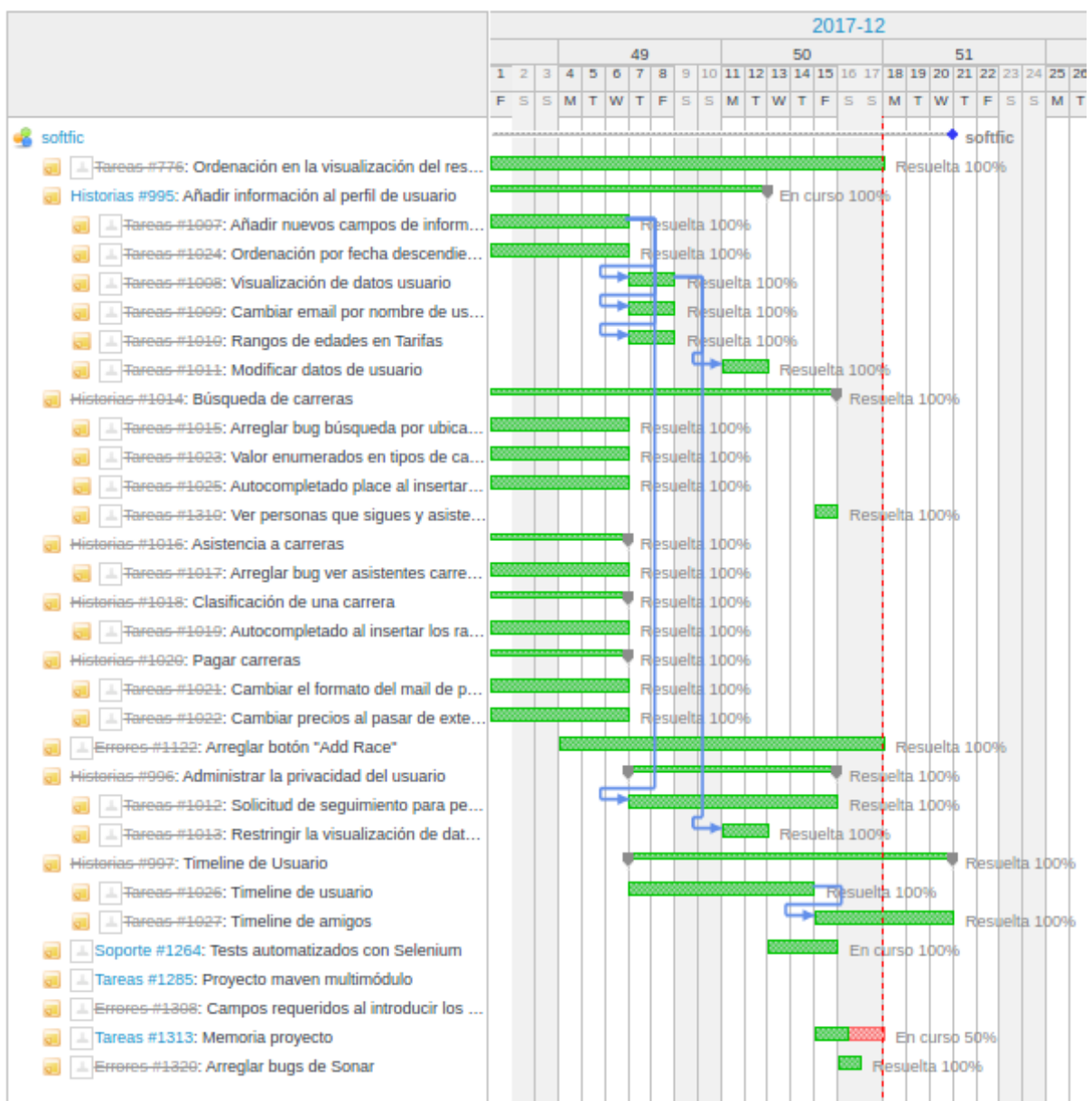
Con respecto a la creación de *issues*, se ha realizado una estimación de la duración de cada una de ellas y su prioridad, así como una planificación de las fechas en las que cada *issue* debería empezar y terminar.

Priority 🗨

	open	closed	Total
Inmediata	-	2	2
Urgente	-	4	4
Alta	-	17	17
Normal	-	50	50
Baja	-	4	4

Figura 3: Prioridades de las *issues* del proyecto.

Además, se han establecido relaciones entre las *issues* creadas, si existían dependencias entre las mismas (FC, CC, FF, CF).

Figura 4: Diagrama de *Gantt* con las dependencias de la cuarta iteración.

La planificación se ha visto ligeramente modificada durante el último *sprint*, con la introducción de la metodología de desarrollo **SCRUM**, incluyendo un *SCRUM Panel* en **Redmine**, la realización de estimaciones en las *Historias de Usuario* mediante *Puntos de Historia* y la reestimación al añadir tiempo en una *issue*.

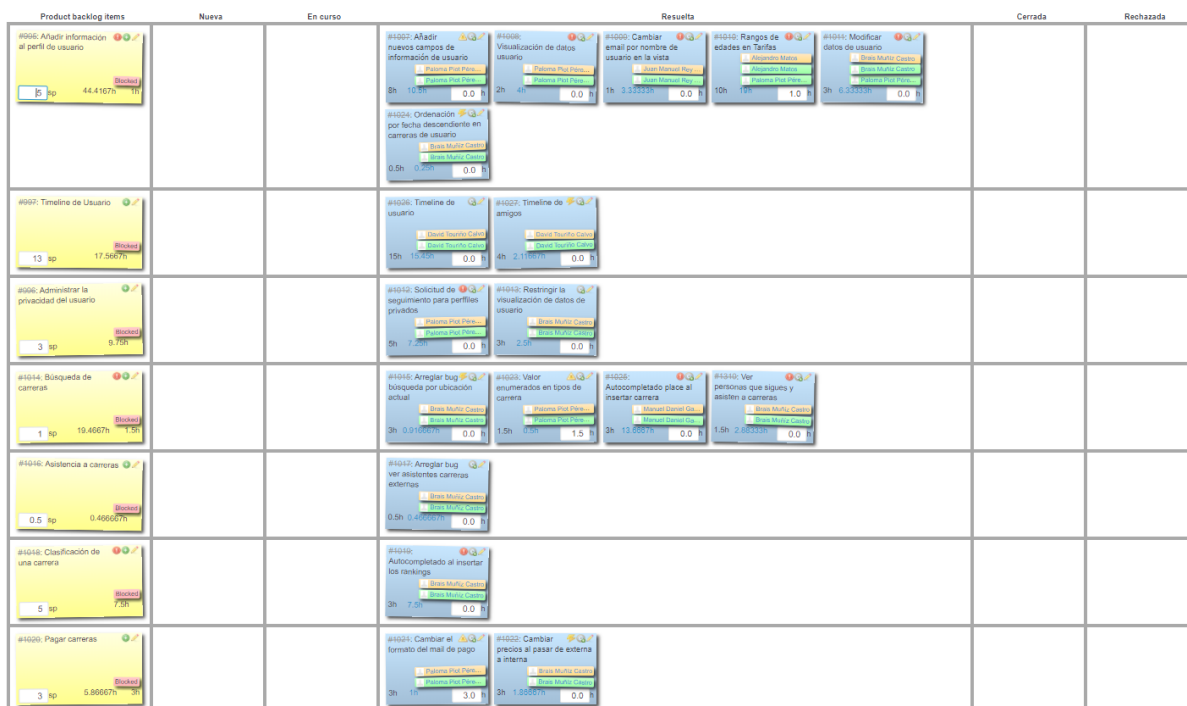


Figura 5: *SCRUM Panel* con las *Historias de Usuario* de la cuarta iteración.

En general se ha tratado de llevar un desarrollo equilibrado en cada *sprint*, sin grandes diferencias de carga de trabajo entre los mismos, a excepción del primer *sprint*, que no tenía tanta carga de trabajo, pero aún no se utilizaba **Redmine**.

Total time: 420.22 hours

Version	2017-10	2017-11	2017-12	Total time
2.0	119.75			119.75
[none]		4.00	179.23	179.23
3.0		121.23		121.23
Total time	119.75	125.23	175.23	420.22


Figura 6: Horas empleadas por el equipo de desarrollo en cada *sprint*.

3.2. Desviaciones

En cuanto a las desviaciones, debido a que el equipo de desarrollo no cuenta con una amplia experiencia en este terreno, la estimación de algunas *issues* puntuales no ha sido la más precisa. Sin embargo, en general, no han quedado *issues* planeadas para un *sprint* sin resolver en el mismo.

Estimated time: 405.83 Spent time: 420.22

Figura 7: Horas estimadas frente horas empleadas en el desarrollo del proyecto.

Tracker 

	open	closed	Total
Errores	-	10	10
Tareas	-	54	54
Soporte	-	6	6
Historias	-	7	7

Figura 8: *Tracker* de Redmine

4. Pruebas de carga y Profiling

4.1. Entorno de ejecución de las pruebas

Para llevar a cabo las pruebas se lanzó la aplicación en un equipo con las siguientes características:

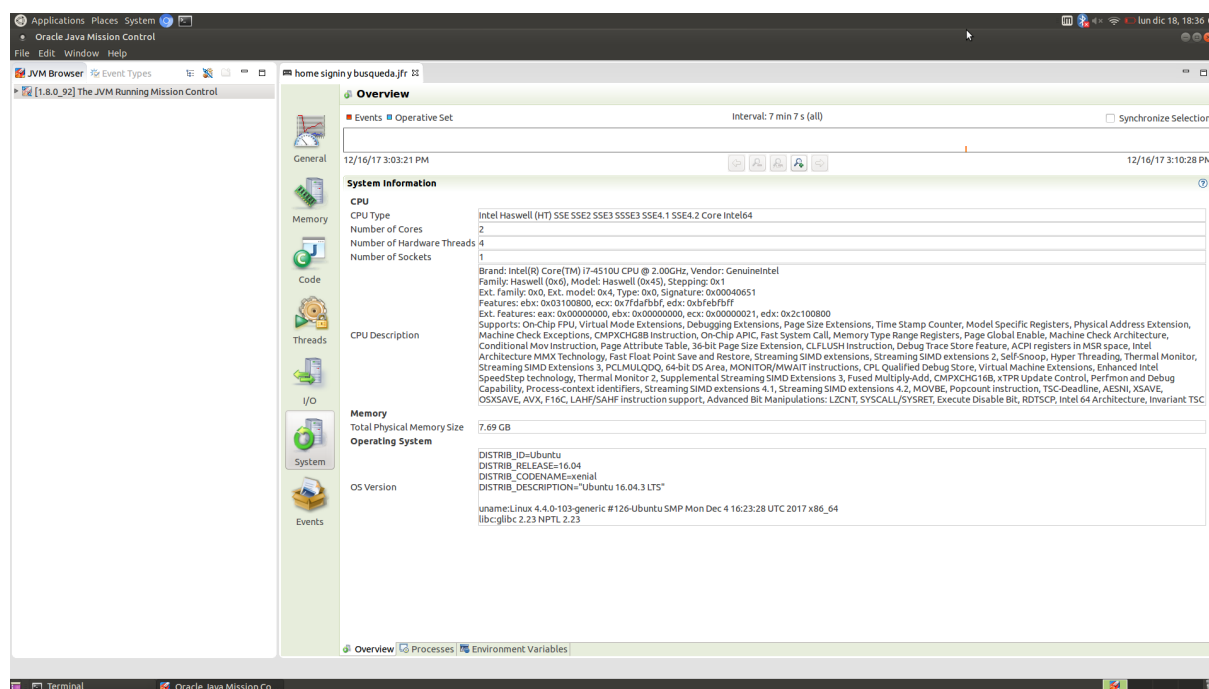


Figura 9: Especificaciones del entorno de pruebas.

Las **Pruebas de Carga** se lanzaron desde un equipo contra la aplicación ejecutándose en el otro. El mismo equipo que hacía de servidor llevaba a cabo monitorización y *profiling* durante las pruebas de carga. Lo ideal habría sido que el equipo que realiza el *profiling* no sea el mismo que lanza la aplicación, porque incluye en las mediciones. Lo más correcto habría sido realizar el *profiling* en remoto en otro equipo. La comunicación entre las máquinas tuvo lugar en una LAN doméstica (120 Mbits/s), en la que se encontraban conectados los equipos anteriormente descritos y 4 equipos más, pertenecientes al resto de miembros del equipo de desarrollo.

Para la ejecución de las pruebas de carga se utilizó la herramienta **JMeter**, mientras que para realizar el *profiling* se utilizó **Java Mission Control**.

El diseño de las pruebas de carga se centró en el número de usuarios concurrentes (peticiones concurrentes) y el tiempo de respuesta del servidor. Los resultados que se presentan a continuación son relativos a tres pruebas que utilizan la misma configuración:

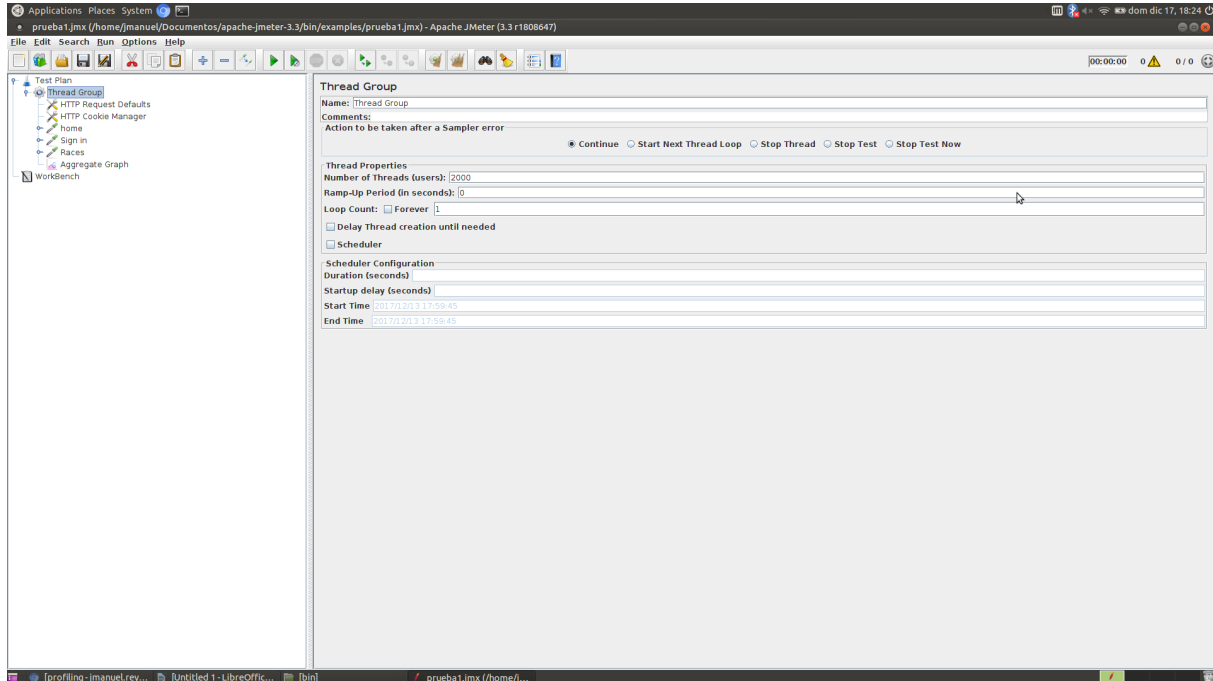


Figura 10: Number of threads: 2000
Ramp-Up Period (seconds): 0
Loop Count: 1

Se estableció la cifra de 2000 usuarios debido a que durante la puesta en marcha de las pruebas, se comprobó durante los ensayos realizados que a partir de 3000 *threads*, en el entorno descrito anteriormente, las pruebas tardaban demasiado tiempo sin aportar resultados que variasen de modo sustancial respecto de las pruebas con 2000 usuarios. Del mismo modo, con menos de 1000 usuarios las pruebas eran demasiado rápidas y prácticamente en todos los casos se obtenía un 100 % de rendimiento. Cabe mencionar que a partir de 10000 usuarios, la ejecución de las pruebas comenzaba a informar de errores de falta de espacio en el *heap* (*OutOfMemoryError: Java Heap Space*). Se utilizó un *ramp-up period* de 0 para simular que todos los usuarios accedían a la vez al sistema (en una aplicación empresarial real no es extraño que lleguen más de 1000 peticiones concurrentes) y un *Loop count* de 1 para que cada usuario realizase una vez cada petición configurada.

Se diseñaron y ejecutaron tres pruebas de carga. A continuación se exponen los resultados obtenidos en cada una:

- Primera prueba:

Cada usuario lleva a cabo dos peticiones HTTP, una petición GET para acceder a la *homepage* y una petición POST para iniciar sesión en la aplicación. El rendimiento es considerablemente bueno para el número de usuarios concurrentes, teniendo en

cuenta que estamos ejecutando la aplicación en un ordenador portátil de gama media. El porcentaje de errores es del 0.13 % y todos se corresponden con “*Connection Reset*”, debido a que es inevitable que el servidor se sature y tenga que rechazar o no sea capaz de responder algunas peticiones.

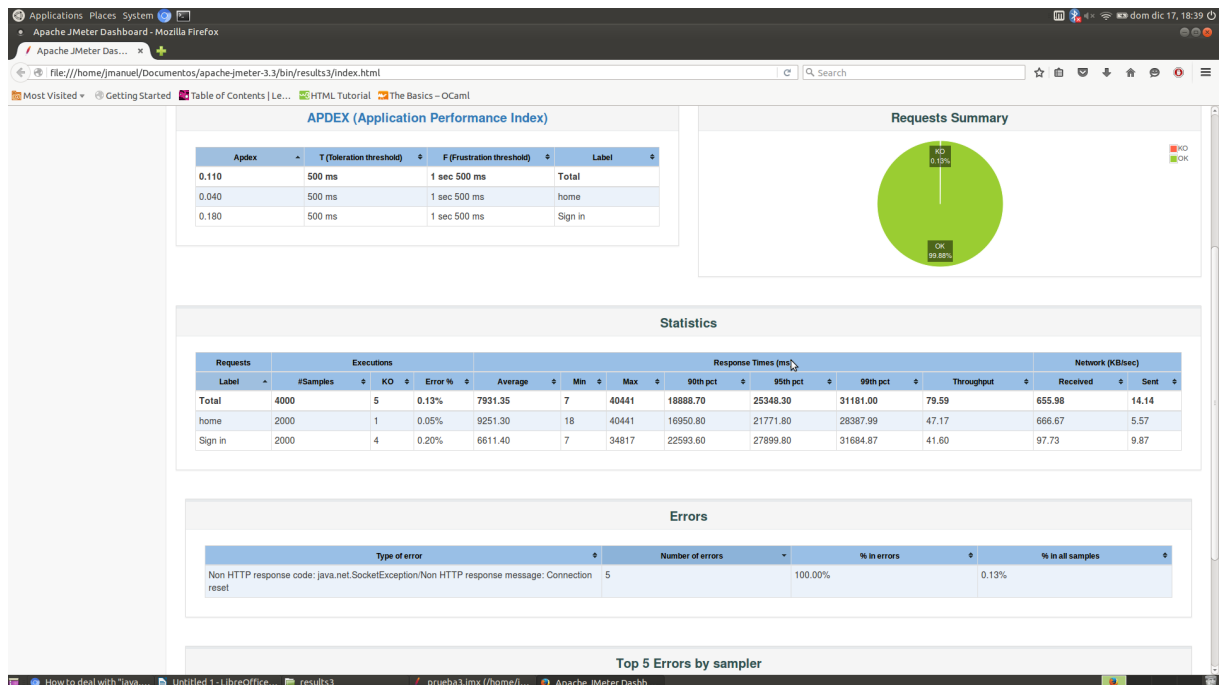


Figura 11: Primera Prueba de Carga

■ Segunda prueba:

Cada usuario ejecuta dos peticiones POST. Primero intenta iniciar sesión y, acto seguido, procede con una búsqueda de carreras por distintos criterios. En este caso, se obtiene mejor rendimiento que en la prueba anterior debido, entre otras cosas, a que ahora no se está intentando acceder a la *homepage* de la aplicación, la cual incluye un “*gift*” animado, que introduce una ralentización en el renderizado de la página.

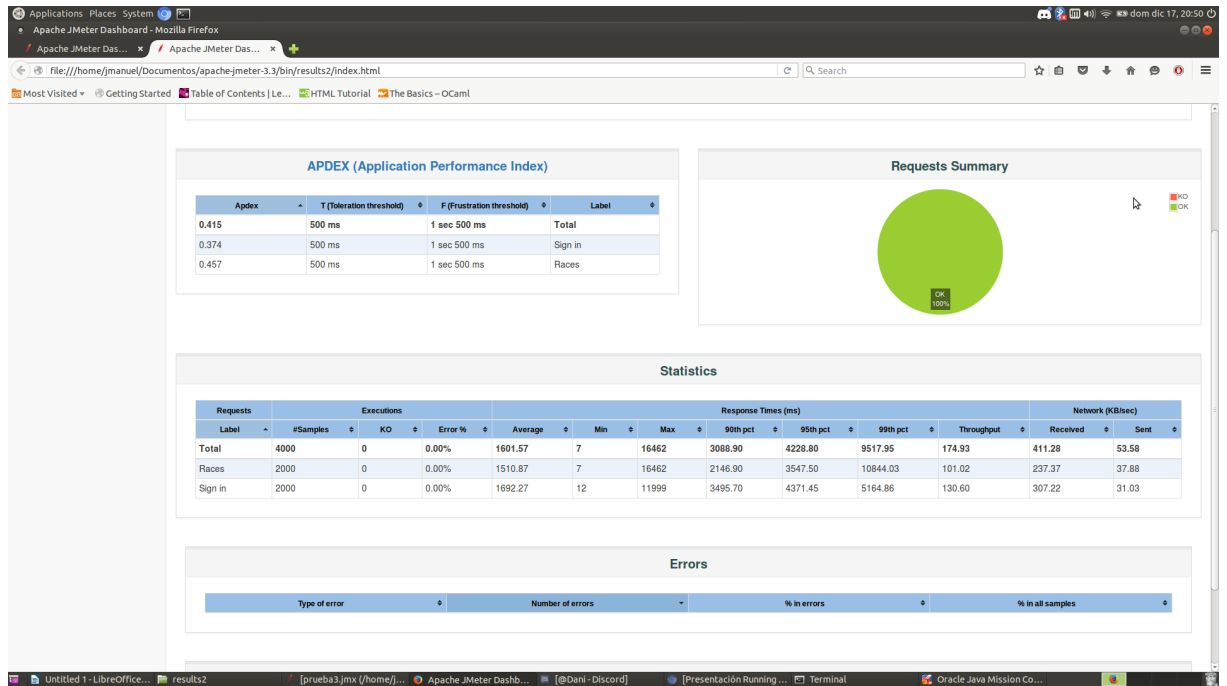


Figura 12: Segunda Prueba de Carga

■ Tercera prueba:

Cada usuario lleva a cabo todas las peticiones de las pruebas anteriores, es decir, primero una petición GET para acceder a la *homepage*, después una petición POST para iniciar sesión, y finalmente otra petición POST para hacer una búsqueda de carreras. En este caso, podemos observar que el rendimiento se resiente un poco más que en la primera prueba, como era de esperar, obteniendo un 0.27% de errores, de los cuales la mayoría se corresponden con “*Connection Reset*” y también hay un error de fallo de respuesta del servidor, debido de nuevo, al elevado número de peticiones concurrentes y a las condiciones de tráfico en la red del entorno de pruebas.

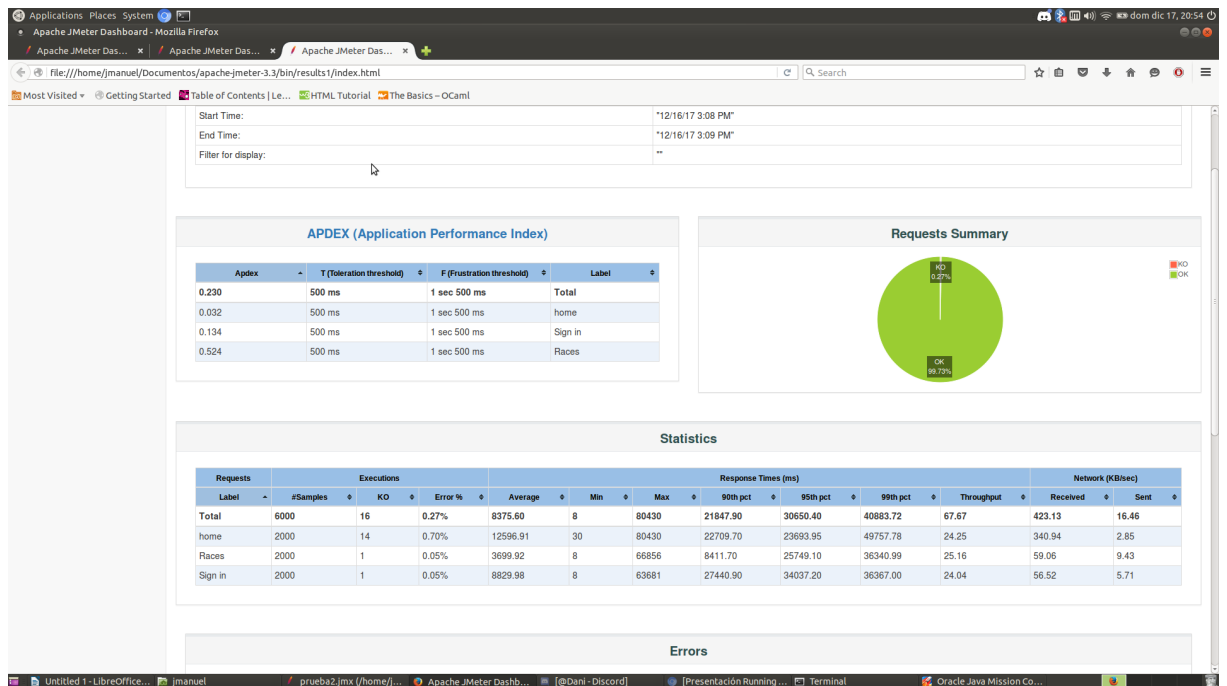


Figura 13: Tercera Prueba de Carga

Finalmente podemos poner como ejemplo los resultados de *profiling* obtenidos durante esta última prueba:

Se puede apreciar que el mayor porcentaje de código ejecutado corresponde con parsers, debido al HTML que tiene que renderizar la aplicación.

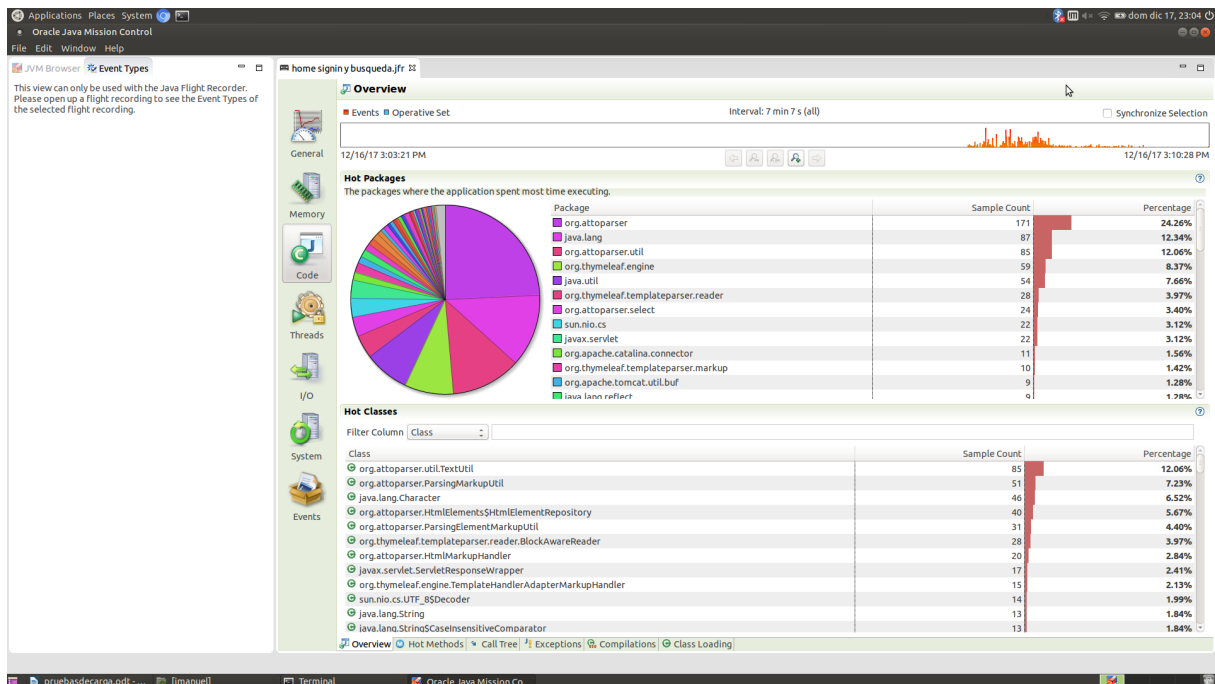


Figura 14: Resultados de *profiling* obtenidos durante la tercera prueba.

Podemos observar también las estadísticas de uso de la memoria. El tamaño máximo

del *heap* es de 1GB y la gráfica señala que nunca llega a llenarse del todo, actuando antes del *Garbage Collector* hasta en 16 ocasiones.

La memoria no tiene tendencia a crecer tras pasar el *Garbage Collector*, por lo que deducimos que no existen *Memory Leaks*, al menos en las funcionalidades que se ejecutan en estas pruebas.

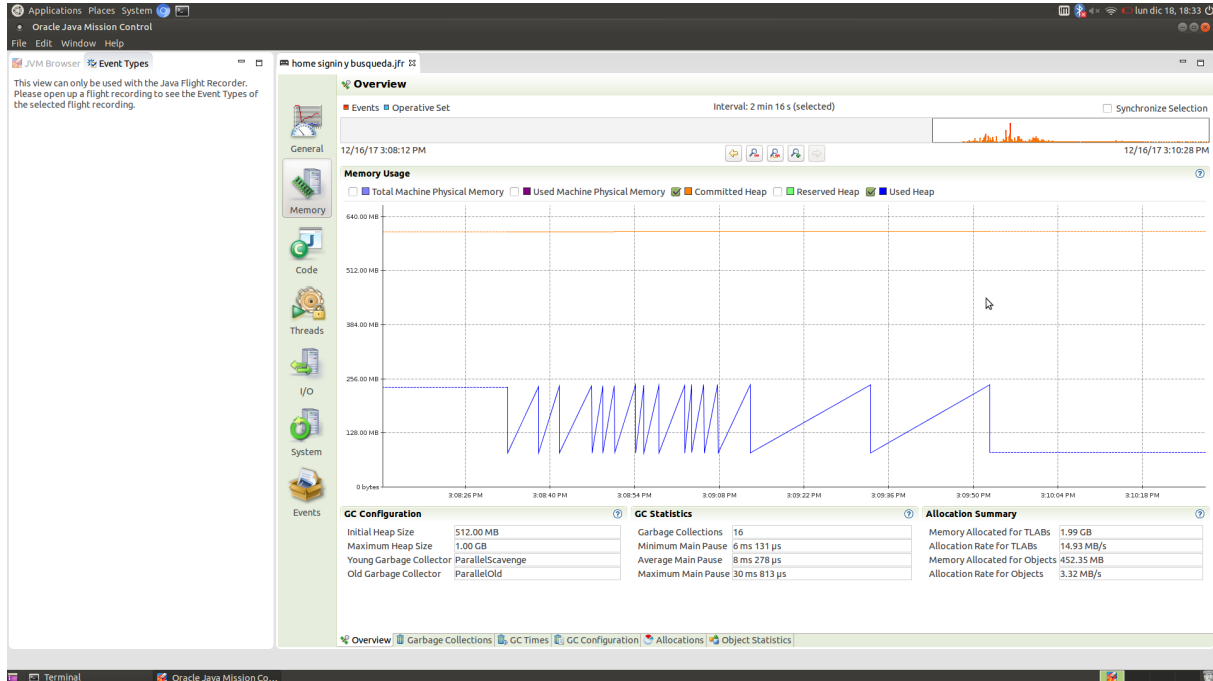


Figura 15: Uso de la memoria.

5. Estadísticas del código y proyecto

■ Líneas de código:

Al ser una aplicación web, utilizando tecnologías **Java**, distinguimos la mayor parte de líneas de código en **Java** y **HTML**.

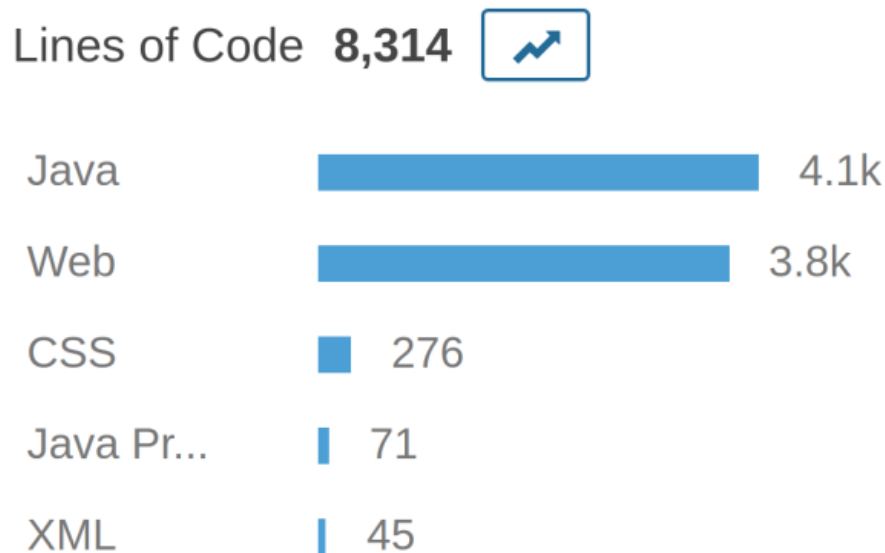


Figura 16: Uso de la memoria.

- **Cobertura:**

Se obtuvo un 59,2% de cobertura en todo el proyecto.

- **Duplicados:**

Existe un 13,4% de duplicados, pertenece principalmente código **HTML**, reutilizado para agilidad.

- **Estimaciones:**

El tiempo estimado y gastado se entiende como la suma de los tiempos estimados y gastados en cada tarea.

- Versión 2.0:

Tiempo estimado del proyecto: 118 horas.

Tiempo gastado en el proyecto: 119,75 horas.

- Versión 3.0:

Tiempo estimado del proyecto: 172,67 horas.

Tiempo gastado en el proyecto: 121,23 horas.

- Sprint 4.0:

Tiempo estimado del proyecto: 114,17 horas.

Tiempo gastado en el proyecto: 177,9 horas.

- Total:

Tiempo estimado del proyecto: 405,83 horas.

Tiempo gastado en el proyecto: 420,22 horas.

- **Tareas por *sprint*:**

En la siguiente figura se muestra el número de tareas de cada *sprint*, aumentando al principio del *sprint* después de la reunión con el cliente, y disminuyendo poco a poco hasta el final del *sprint*.

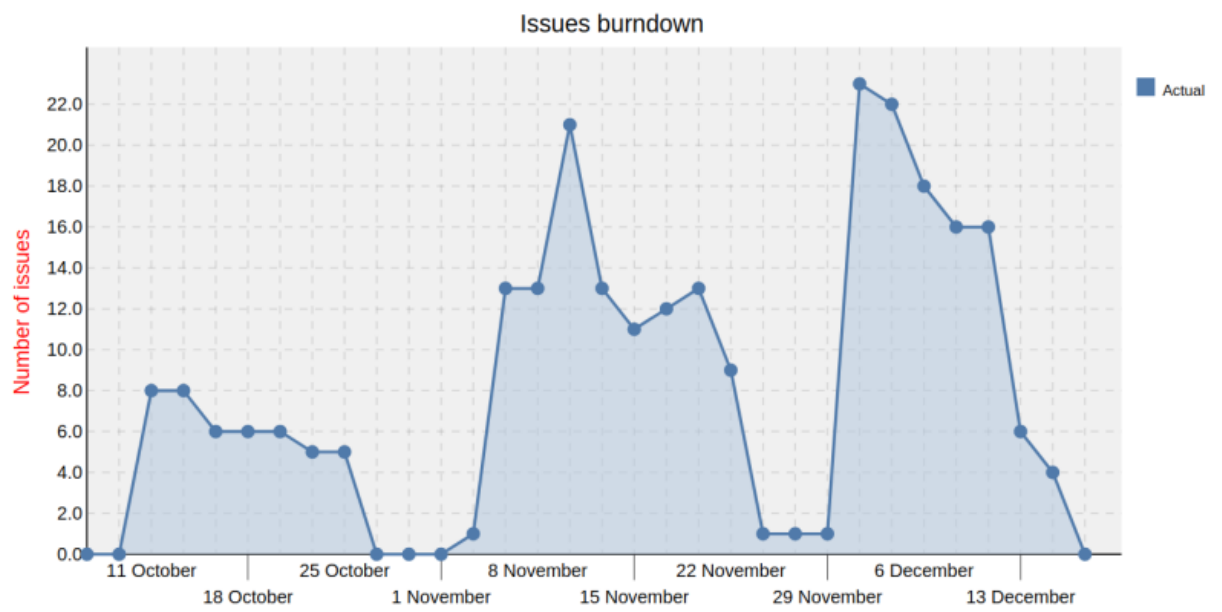


Figura 17: Uso de la memoria.

6. Referencias

Referencias

- [1] <https://www.eclipse.org/>
- [2] <https://git-scm.com/>
- [3] <https://about.gitlab.com/>
- [4] <https://www.redmine.org/>
- [5] <https://www.scrum.org/>
- [6] <https://maven.apache.org/>
- [7] <https://jenkins-ci.org/>
- [8] <https://www.sonarqube.org/>
- [9] <https://www.sonarlint.org/eclipse/>
- [10] <http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html>
- [11] <http://jmeter.apache.org/>
- [12] <http://junit.org/junit5/>
- [13] <http://site.mockito.org/>
- [14] <http://www.eclEmma.org/jacoco/>
- [15] <https://codehaus-cargo.github.io/cargo/Home.html>
- [16] <http://www.seleniumhq.org/>