



**UNIVERSIDAD DE CASTILLA-LA MANCHA
FACULTAD DE CIENCIAS SOCIALES DE TALAVERA**

GESTIÓN DE TÍTULOS PROPIOS: 2º PARTE

Miriam Fernández Osuna (Miriam.Fdez@alu.uclm.es)

Sergio García Muñoz (Sergio.Garcia75@alu.uclm.es)

Rubén Gómez Villegas (Ruben.Gomez10@alu.uclm.es)

Carlos Rincón González (Carlos.Rincon1@alu.uclm.es)



https://github.com/forSuin-ISOII/forSuin_ISO2


Asignatura: Ingeniería del Software II

Titulación: Grado en Ingeniería Informática

Nombre de la empresa: forSuin

Fecha: 21-12-2022

Ficha del trabajo:

Empresa: forSuin		
Apellidos, Nombre (Rol)	Firma	Sueldo (€)
Fernández Osuna, Miriam (CEO y desarrolladora)		
García Muñoz, Sergio (Desarrollador)		
Gómez Villegas, Rubén (Desarrollador)		
Rincón González, Carlos (Desarrollador)		
Remuneración total de la empresa (€):		1.400.000
Remuneración en el segundo entregable (€):		

CEO de la empresa: Miriam Fernández Osuna

Índice

1. Introducción:	4
2. Modificaciones primer entregable:	4
3. Sprints:	5
3.1 Cuarto sprint:	5
3.2 Quinto sprint:	6
3.3 Sexto sprint:	6
3.3.1 Chiqui sprint primero:	6
3.3.2 Chiqui sprint segundo:	7
4. Calidad:	7
4.1 Toma y configuración del proyecto en SonarCloud	7
4.2 Toma y configuración del proyecto en SonarCloud:	8
4.3. Análisis de problemas y soluciones aplicadas (sergio)	8
4.3.1 Código duplicado:	9
4.4 Análisis Sonar recurrentes en el proyecto:	9
4.5 Evolución del proyecto	11
4.6 Calidad de producto:	12
4.7 Integración SonarCloud automatizado:	12
5. Testing:	14
5.1 Integración de JUnit (Surefire) y EclEmma (JaCoCo) en el proyecto:	14
5.2 Plan de pruebas:	14
5.2.1 La problemática con las interfaces:	15
5.3 Análisis resultado:	15
6. Mantenimiento:	15

1. Introducción:

Tras una temporada más de desarrollo de funcionalidades, en esta segunda parte de entrega de valor el proyecto se ha visto analizado, tanto a nivel calidad, como probado por múltiples test y haber pasado por un proceso de mantenimiento. Todo esto de una manera planificada y sin olvidar lo que orquesta todas estas actividades, la gestión de la configuración.

2. Modificaciones primer entregable:

Durante el desarrollo de ésta segunda parte del proyecto, se han realizado una serie de cambios que afectaron al primer entregable, todo ello con la intención de mejorar sobre todo en la gestión de configuración, y avanzar en el desarrollo del software.

- Gestión de configuración: Si anteriormente se utilizaba la sección Projects en GitHub, ahora empleamos Trello, puesto que se encontraban dificultades para identificar las fechas en las que se realizaron partes del sprint, o incluso falta de funcionalidad.
- Nueva entidad Vicerrectorado: Con el fin de hacer más fácil la identificación en el inicio de sesión, se ha creado dicha entidad además de su propia tabla en la base de datos. Ésta entidad podrá ser jefe vicerrectorado (gracias a un booleano que lo indica) o no, es importante saberlo porque dependiendo de qué sea podrá realizar ciertas funcionalidades.

- Base de datos:

CursoPropio(*id*, nombre, ECTS, fechaInicio, fechaFin, tasaMatricula, edicion, estadoCurso, tipoCurso, **centro_FK**, **secretario_Prof_UCLM_FK**, **director_Prof_UCLM_FK**, requisitos)

Matricula(fecha, pagado, atributo, modoPago, **curso_PropioFK**, **estudiante_FK**)

Estudiante(*dni*, nombre, apellidos, titulacion, cualificacion)

Materia(*nombre*, horas, fechaInicio, fechaFin, **responsable_Prof_FK**, **curso_PropioFK**)

Centro(*nombre*, localizacion, atributo)

Profesor(*dni*, nombre, apellidos, doctor)

ProfesorUCLM(**profesor_dni_FK**, categoriaProfesor, **centro_FK**)

ProfesorExterno(**profesor_dni_FK** , titulacion)

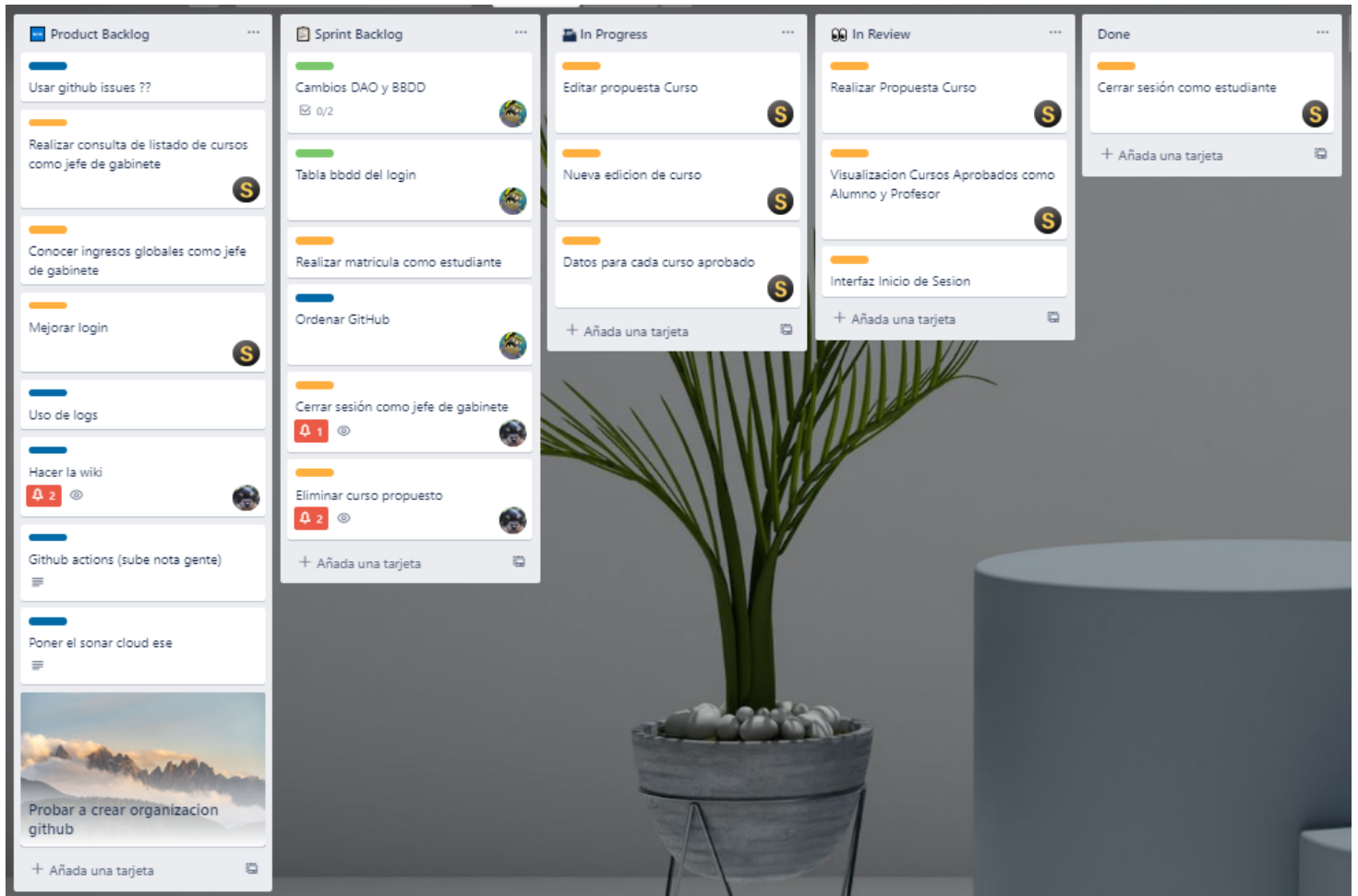
Vicerrectorado(*dni*, nombre, apellidos, jefe)

3. Sprints:

Durante el desarrollo de esta segunda parte del proyecto se han llevado a cabo 3 sprints.

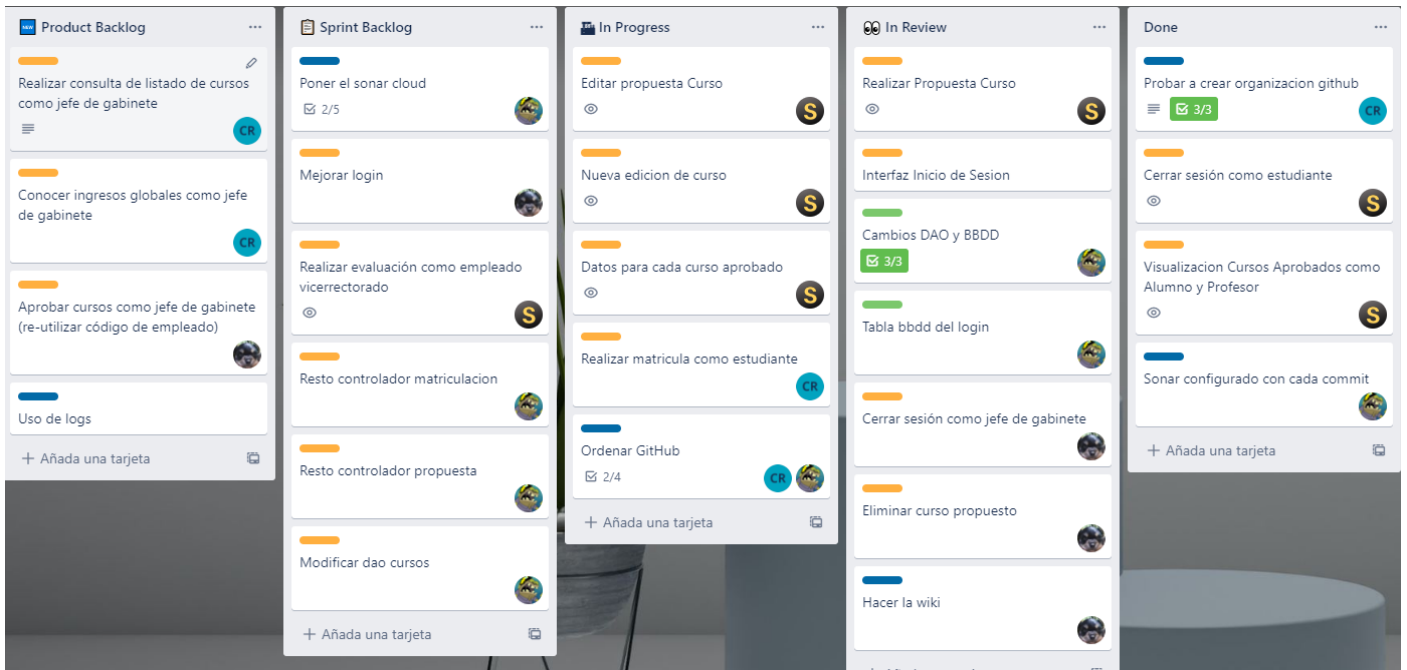
3.1 Cuarto sprint:

Con rango de fechas desde el 03 de Noviembre a 23 de Noviembre:



3.2 Quinto sprint:

Con rango de fechas desde el 23 de Noviembre a 15 de Diciembre:

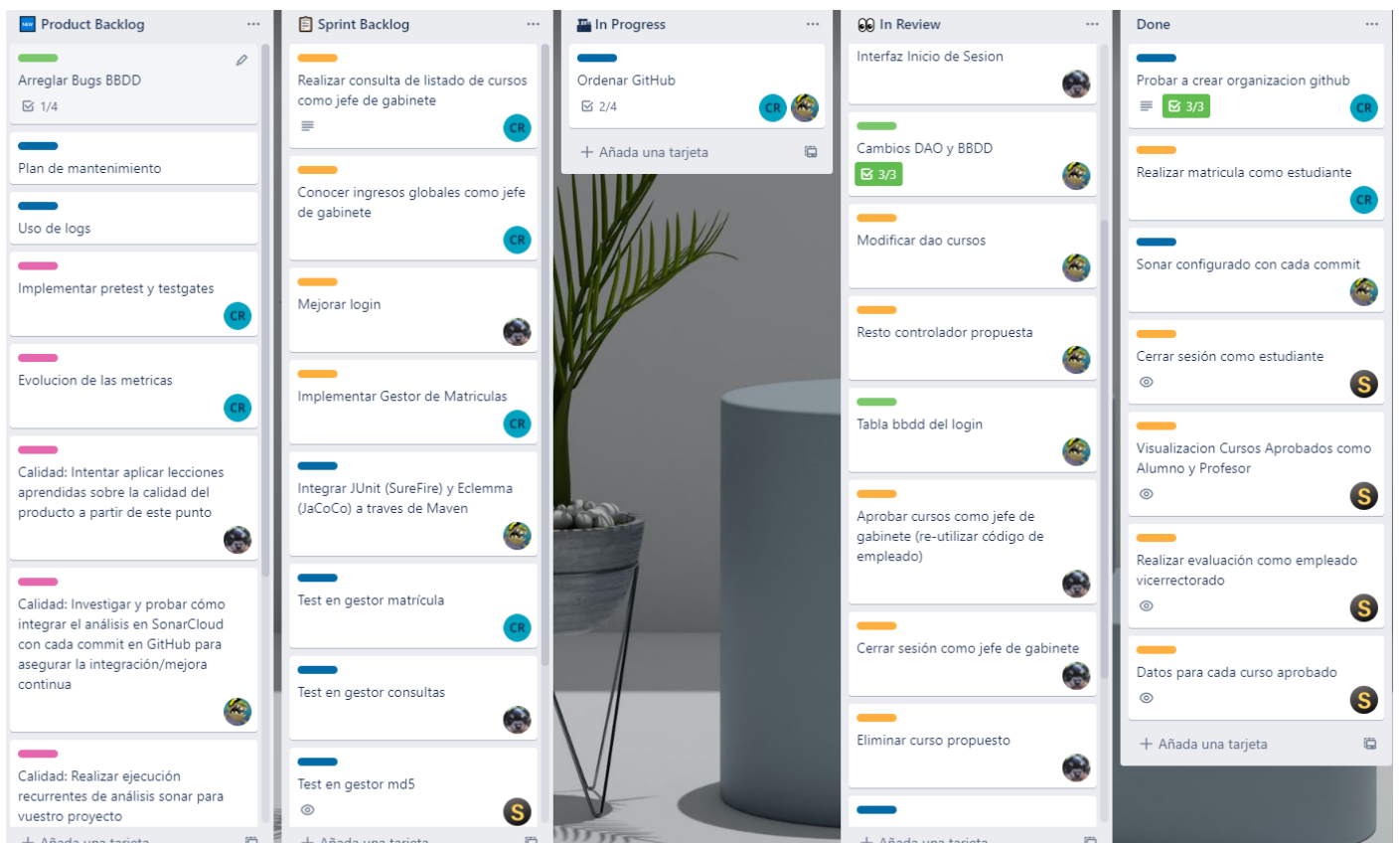


3.3 Sexto sprint:

Con rango de fechas desde el 15 de diciembre al 21 de diciembre.

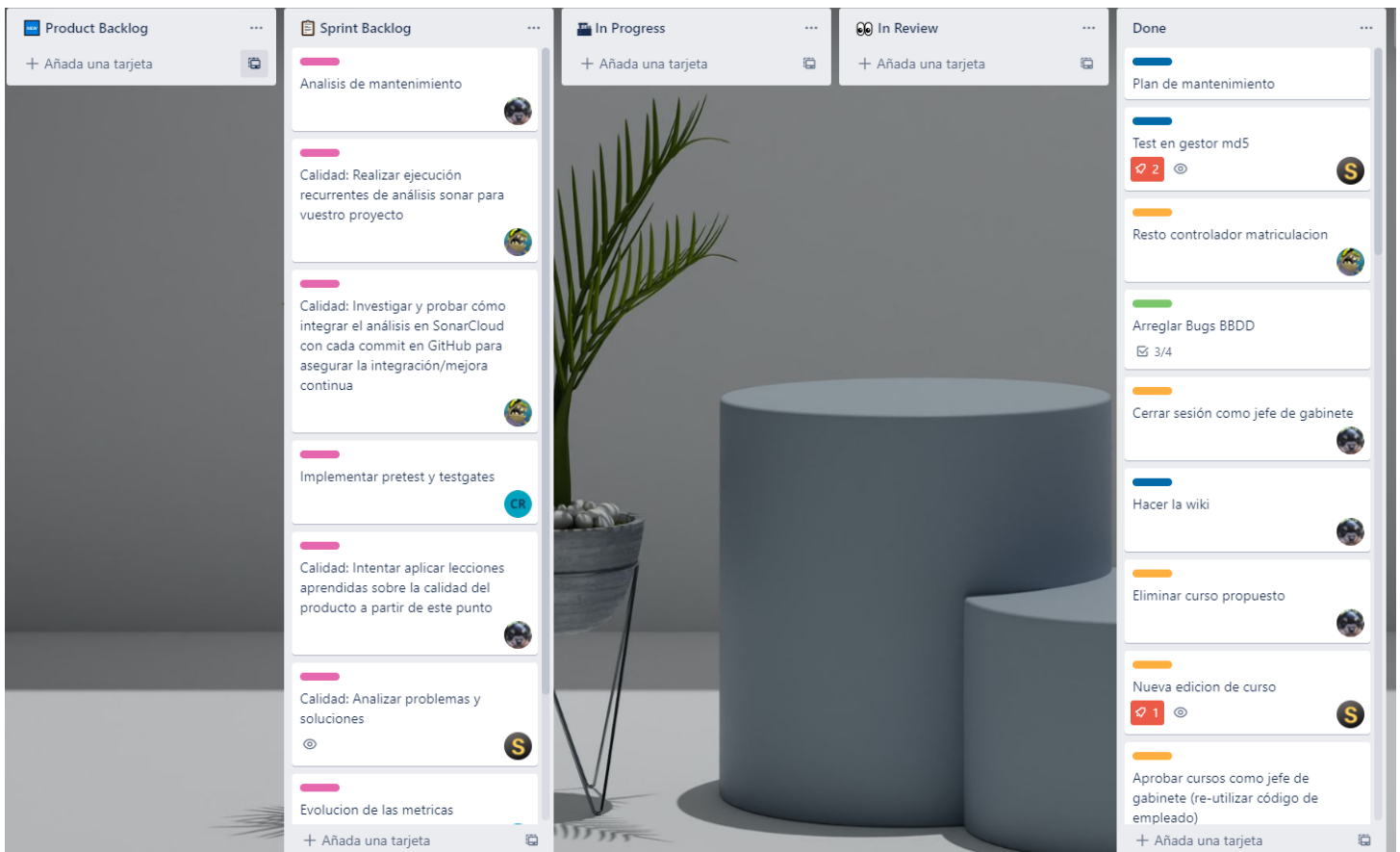
3.3.1 Chiqui sprint primero:

Orientado para testing, con rango de fechas desde el 15 de Diciembre al 18 de Diciembre.



3.3.2 Chiqui sprint segundo:

Orientado para la calidad y el mantenimiento del software, con rango de fechas desde el 18 de diciembre al 21 de diciembre.



4. Calidad:

4.1 Toma y configuración del proyecto en SonarCloud

A la hora de determinar la calidad del proyecto, nos hemos basado en *SonarCloud*, se han llevado a cabo las siguientes configuraciones:

Se han producido cambios en lo que respecta al repositorio y todo el contenido de GitHub. En primer lugar se ha creado una organización para permitir a *SonarCloud* el análisis, y todos los miembros se han unido a dicho entorno (común entre GitHub y SonarCloud)

Siguiendo las transparencias prácticas de la asignatura, se ha producido añadiendo unas líneas al archivo `pom.xml`. Por último, se ha comprobado que *Sonar* ya es funcional con el comando `mvn verify sonar:sonar`.

Con el fin de no usar la configuración por defecto de la plataforma, hemos querido proporcionar una configuración personalizada, creando así varios quality gates:

- *forSuin Quality pre-test.*
- *forSuin Quality.*

En próximas secciones profundizaremos en su configuración.

En relación a la seguridad, la contraseña y el usuario ya se encuentran incluidas en la plataforma. Pero no supone un riesgo porque la base de datos está embebida.

4.2 Toma y configuración del proyecto en SonarCloud:

En nuestro proyecto hemos tomado 2 quality gates para SonarCloud, que creemos que nos permiten tener una buena visión de la calidad de nuestro proyecto.

Hemos elegido estas teniendo en cuenta los requisitos de calidad del código y de calidad del software final.

- *forSuin Quality pre-test.*

Es la línea base sobre la que vamos a trabajar. En él hemos establecido un coverage del 0%, ya que no es nuestro objetivo en este punto del proyecto.

Sin embargo, nuestro enfoque está en obtener un código limpio, sostenible y confiable, sobre el que poder trabajar, arreglando pronto los errores que nos pueden suponer una mayor pérdida de tiempo según avance el proyecto y se vayan apilando los problemas.

Hemos conseguido este objetivo, ya que hemos obtenido un código limpio, según los estándares que nos habíamos propuesto:

- *forSuin Quality*

Esta quality gate está diseñada con el objetivo de generar un código que, además de mantener los estándares de calidad ya establecidos en el pre-test, nos proporcione una cobertura frente a los errores lo suficientemente grande como para asegurarnos de que no hay errores que nos rompan la ejecución del software.

4.3. Análisis de problemas y soluciones aplicadas (sergio)

PROBLEMAS	SOLUCIONES
Código sin usar.	Eliminando el código señalado (la mayoría tratándose de <code>import</code> de paquetes).
Comentarios <i>TODO</i> .	Realizar el propio <i>TODO</i> y eliminar el comentario.
Atributos públicos en clase.	Pasar de público a privado.

Ausencia de tipo en los genéricos.	Añadir el tipo dentro de los < > .
Sustituir el tipo por < > .	Quitar el tipo donde se señala.
Cambiar & por && .	Se han hecho las variables antes para que se ejecuten las funciones y más adelante se hace la comprobación (referencia al caso de pantallaRealizarPropuesta).
Bloque de código comentado.	Simplemente eliminarlo.
Constructor con más de 7 parámetros.	Ignorar code smell, debido a que era necesario (referencia al caso de cursoPropio).
Declaración de variables en una sola línea.	Se hizo que todas las declaraciones de variables estuvieran en líneas distintas (aunque sean del mismo tipo).
Líneas exactamente iguales.	Hacer uso de constantes.
Mal uso de la expresión <code>size() == 0</code> .	Usar funciones tales como <code>isEmpty()</code> .
ECTS mal escrito.	Se ha cambiado a eCTS
Posibilidad de que <code>null</code> sea un valor.	Tener un mejor control para que no ocurra
Guardar en variable temporal algo innecesario	Devolverlo directamente con <code>return</code>
<code>Null</code> a la hora de acceder a elementos de la base de datos en el gestor de matrículas.	Comprobar si tiene título y estudiante la matrícula y lanzar la excepción correspondiente.

4.3.1 Código duplicado:

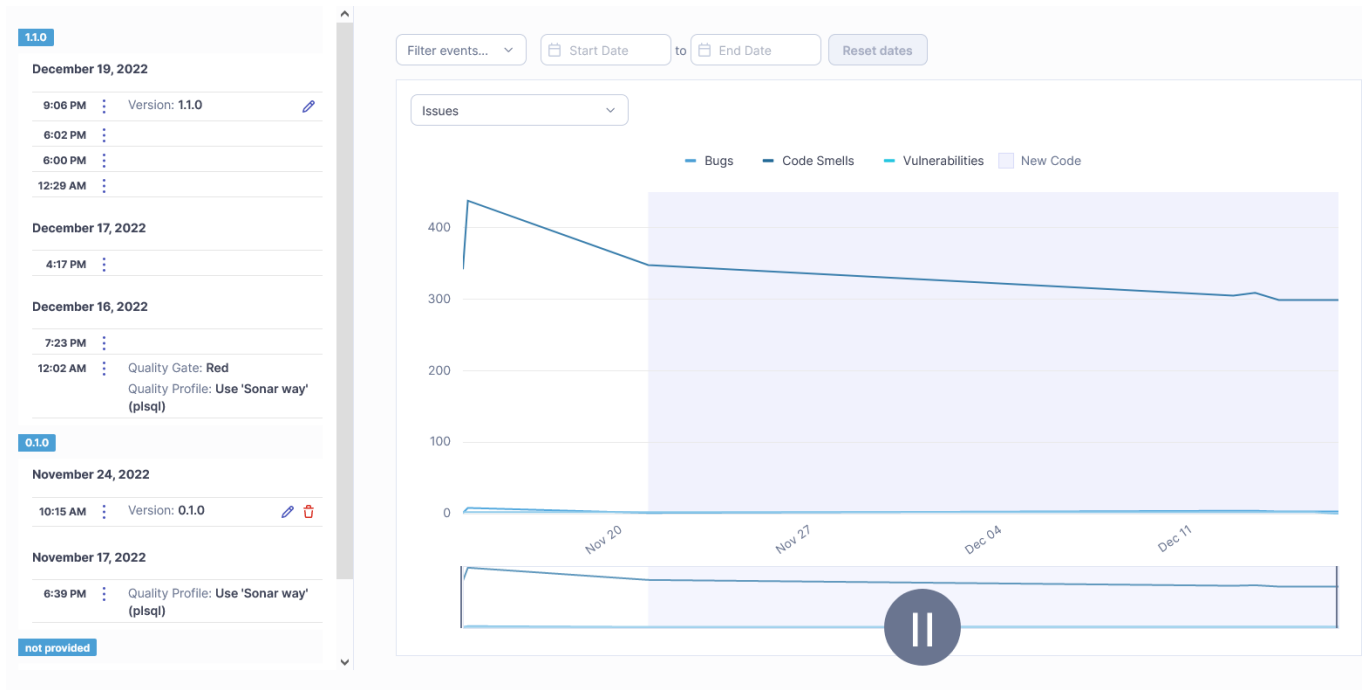
Es otro de los problemas que nos tuvimos que enfrentar, ya que en un principio, las funcionalidades de `realizarPropuesta`, `editarPropuesta` y `nuevaEdición` se iban a hacer en ventanas distintas. Sin embargo, con el uso de la herramienta de *SonarCloud* nos dimos cuenta que esto provocaba, no sólo código duplicado, sino archivos duplicados. Por lo que, se tomó la decisión de tratar de hacer las tres funcionalidades en una única ventana, consiguiendo eliminar este aviso de duplicado.

4.4 Análisis Sonar recurrentes en el proyecto:

Para observar nuestro desempeño como desarrolladores software con cierto criterio de calidad, nos hemos basado en la realización de distintos análisis durante la evolución del proyecto, los cuales han sido facilitados gracias a la automatización (explicado en el apartado [4.7](#)) y han permitido realizar un código más limpio y mantenible. En ellos se puede ver la mejora de nuestro código a lo largo de este último mes de muchas modificaciones y commits. Desde la página de Sonar no se puede observar la cantidad exacta de análisis realizados, pero a partir de los automatizados se pueden estimar unos 60 análisis, cada uno con mejores resultados que el anterior. Se indaga más en la evolución del proyecto en el siguiente apartado.

<p>Testing ✔ Passed</p> <p>December 20 at 10:04 PM d1c8882c Arreglados bugs hashcode</p> <p>5 Fixed Issues 0 New Issues ▲ -0.6% Coverage ● -0.2% Duplications</p> <p style="text-align: right;">+117 Lines of Code</p>
<p>Testing ✘ Failed</p> <p>December 20 at 7:21 PM feb11553 ProfesorDaoTest + CentroDaoTest terminados</p> <p>0 Fixed Issues 0 New Issues ● 0.0% Coverage ● -0.2% Duplications</p> <p style="text-align: right;">-1 Lines of Code</p>
<p>Testing ✘ Failed</p> <p>December 20 at 6:03 PM e16181c7 Bug Fix daos + tests</p> <p>3 Fixed Issues ▲ +2 New Issues ● +0.8% Coverage ● -0.1% Duplications</p> <p style="text-align: right;">+112 Lines of Code</p>
<p>Testing ✘ Failed</p> <p>December 20 at 5:01 PM d090b3ae Update DateLabelFormatterTest.java</p> <p>1 Fixed Issues 0 New Issues ● 0.0% Coverage ● -0.0% Duplications</p> <p style="text-align: right;">0 Lines of Code</p>
<p>Testing ✘ Failed</p> <p>December 20 at 4:39 PM daf58ee3 Merge branch 'Testing' of https://github.com/forSuin-ISOII/forSuin_ISO2 into Testing</p> <p>0 Fixed Issues 0 New Issues ● +7.4% Coverage ▲ +0.4% Duplications</p> <p style="text-align: right;">+134 Lines of Code</p>

<p>Interfaces ✘ Failed</p> <p>November 24 at 2:09 PM 0ead4aee Pequeña interfaz + Pequeños errores</p> <p>54 Issues 0.0% Coverage 1.8% Duplications</p> <p style="text-align: right;">2.5k Lines of Code</p>
<p>Main Branch — Not computed</p> <p>November 24 at 10:15 AM a214ca29 Merge branch 'Desarrollo' of https://github.com/forSuin-ISOII/forSuin_ISO2 into Desarrollo</p> <p>47 Fixed Issues ▲ +17 New Issues ● 0.0% Coverage ● -2.5% Duplications</p> <p style="text-align: right;">+171 Lines of Code</p>
<p>Main Branch — Not computed</p> <p>November 17 at 6:39 PM e0b9926e Sonar implementado</p> <p>0 Fixed Issues ▲ +103 New Issues ● 0.0% Coverage ● -1.8% Duplications</p> <p style="text-align: right;">+163 Lines of Code</p>
<p>Main Branch — Not computed</p> <p>November 17 at 2:19 PM 0faac002 Create README.md</p> <p>0 Fixed Issues 0 New Issues ● 0.0% Coverage ● 0.0% Duplications</p> <p style="text-align: right;">0 Lines of Code</p>
<p>Main Branch — Not computed</p> <p>November 17 at 1:10 PM 0faac002 Create README.md</p> <p>345 Issues 0.0% Coverage 34.2% Duplications</p> <p style="text-align: right;">2.6k Lines of Code</p>



4.5 Evolución del proyecto

En el inicio del proyecto, no hicimos ningún tipo de test ni de control. Suponemos que esto es lo normal, ya que no se habían creado las funcionalidades básicas para ser probadas siquiera.

A mitad del proyecto, cuando ya teníamos una base sólida con la que trabajar incorporamos *SonarCloud* a nuestro proyecto, de forma que cada push al repositorio generase un reporte de los problemas que había con él.

En un principio, la integración de *SonarCloud* no nos fué completamente útil por dos razones principales:

- 1.- Teníamos una *quality gate* muy estricta, por lo que al mínimo error que teníamos nos daba fallo.
- 2.- Al ser un código en desarrollo aún, teníamos muchos fallos como para que la herramienta nos fuera útil del todo.

Cuando nos dimos cuenta de esto generamos una *quality gate* menos exigente, lo cual se juntó con un código más depurado, haciendo que ahora si pudiéramos utilizar la herramienta para depurar los pequeños errores que había.

Posteriormente, y a fin de crear un código listo para el despliegue, creamos una *quality gate* más avanzada, con unos requisitos mayores.

Con ello, creamos una serie de tests, los cuales nos permitieron mejorar la obtención de errores y excepciones, de forma que minimizamos los errores inesperados al máximo, haciendo uso del porcentaje de cobertura de los tests.

Después de este proceso se nos ha quedado un código más limpio, el cual carece de duplicidades hasta el máximo exponente que nos es posible. La mayor parte de los errores que pueden suceder están controlados, de forma que no provoquen errores fatales en el programa y hemos capturado muchas excepciones nuevas, las cuales no habríamos sido capaces de reconocer sin estas herramientas.

4.6 Calidad de producto:

Conociendo todo aquello que caracteriza una alta calidad de producto, hemos llegado a las siguientes conclusiones:

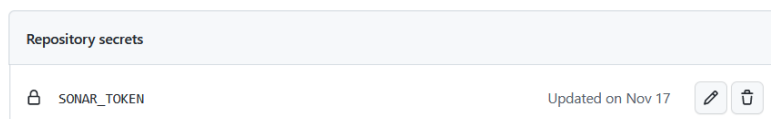
- Es multidimensional: Siendo flexible a modificaciones y grandes implementaciones en vista de un futuro.
- No es absoluta: Mantiene un equilibrio entre el esfuerzo dedicado, los riesgos asumibles y las coberturas previstas.
- Está sujeta a restricciones, que condicionan y valoran gran mayoría de posibilidades a manejar tanto a nivel de usuario como a nivel de desarrollador.
- Ligada a compromisos aceptables: No sólo adecuarse a los requisitos pedidos por el cliente final, sino adaptar tanto al equipo de desarrolladores como al proyecto ante adversidades o contratiempos.
- Los criterios de calidad no son independientes, dependen de otros aspectos como el desarrollo, el testing y el mantenimiento.
- Política / sistema / planificación de la calidad: Frente a problemas en el código que se han podido observar en anteriores secciones, se han impuesto unas soluciones, ¿pero cómo ha surgido esta gestión? Principalmente observando los patrones que siguen los errores, evaluando su importancia, y consultando con todo el equipo de desarrollo en búsqueda de algo en común para declarar qué justificación puede tener dicho error.
- Acción Preventiva: Gestionar de forma correcta los conflictos que surjan a la hora de compartir nuevas implementaciones del código, tener copias de seguridad con el objetivo de no perder el desarrollo ya implementado, o trabajar correctamente con las versiones de maven, podrían ser varias de las prácticas seguidas.
- Defecto: No es totalmente funcional.

Por otro lado, ¿podemos decir que nuestro proyecto es maduro? Si lo comparamos con la primera muestra entregada al cliente, el proyecto no sólo ha mejorado considerablemente, sino que aspira a avanzar y mejorar más.

4.7 Integración SonarCloud automatizado:

Durante el desarrollo, hemos considerado la opción de integrar el análisis de SonarCloud con cada commit, así facilitando la tarea de realizar análisis y comprobar la mejora de nuestro código. Para ello se han seguido los siguientes pasos:

- 1) Dentro de los ajustes del repositorio (Settings>Secrets>Actions), se ha puesto el SONAR_TOKEN adecuado.



- 2) El fichero pom.xml ha sido adaptado con las siguientes propiedades:

```
<properties>
```

```
<sonar.organization>forsuin-isoi</sonar.organization>
<sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

- 3) Por último se ha añadido una configuración base de SonarCloud al fichero `.github/workflows/build.yml`, indicando las ramas con las cuales realizar un análisis tras cada commit/push.

```
name: SonarCloud
on:
  push:
    branches: # Aquí se indican las ramas
      - Desarrollo
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Set up JDK 11
        uses: actions/setup-java@v1
        with:
          java-version: 11
      - name: Cache SonarCloud packages
        uses: actions/cache@v1
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v1
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify
    org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
    -Dsonar.projectKey=forsuin-ISOII_forSuin_ISO2
```

Desde el siguiente enlace se pueden observar la cantidad de análisis realizados:
https://github.com/forsuin-ISOII/forsuin_ISO2/actions?query=workflow%3ASonarCloud+is%3Asuccess++

5. Testing:

Testing consiste en ejecutar pruebas, valorando todas aquellas restricciones y opciones posibles que se producen, jugando con la base de datos, y los tipos de dato que manejan.

5.1 Integración de JUnit (Surefire) y EclEmma (JaCoCo) en el proyecto:

JUnit ha sido usado en el proyecto, más en concreto *JUnit 4*, para poder ejecutar clases de forma controlada, evaluando así el funcionamiento esperado de cada uno de los métodos. Para ello, se añadió *JUnit* a las dependencias de *Maven* y el equipo de desarrollo hizo uso del plugin de *Eclipse MoreUnit*, este último permitió crear y ejecutar las pruebas de una manera sencilla.

Para crear las pruebas, se ha tomado referencia de los apuntes del laboratorio 4 ofrecidos en clase, los cuales nos han ayudado a familiarizarnos con la herramienta.

Ya con los test creados, el *IDE* nos permitía ver el cubrimiento de las líneas al ejecutarlo en local, lo cual no daba mucha seguridad del resultado. Esto se soluciona con la integración de distintos plugins de *Maven*.

Primero se ha configurado el `pom.xml` para que *Maven* genere informes (en el apartado de reporting), y un *Maven* site (en build). Tras esto, se añade también el plugin *Surefire* (en reporting), que genera un informe de la ejecución de los test *JUnit*, junto a *JXR*, el cual nos indica las líneas donde se producen errores. Finalmente *JaCoCo* evaluará la cobertura de sentencia tras estar insertado en reporting, y debe ser configurado, siendo de lo más importante el mínimo de la cobertura. En nuestro caso, lo hemos puesto al 20%, para que *JaCoCo* no lance ningún error en el caso de que no se alcance, y pueda continuar perfectamente con el análisis.

A partir de aquí, tenemos el proyecto listo para generar informes usando el comando `mvn site:site`. La salida de este se puede observar desde el siguiente enlace:

https://forsuin-isoii.github.io/forSuin_ISO2/target/site/index.html

5.2 Plan de pruebas:

Para el plan de las pruebas se han seguido los siguientes principios principalmente:

- Se ha usado la técnica de *each-use* en la mayoría de pruebas, además de tratar de cubrir todas las bifurcaciones, condiciones y decisiones que sean posibles.
- Se ha tomado la decisión de probar todos los métodos posibles de un gestor en un mismo caso de prueba, ya que encontrábamos ventajas como la reducción del número de funciones, variables y esfuerzo en los test.
- Si en el propio test, se necesitaba crear una nueva entidad en la base de datos, este se eliminará al final del test, para que de esta forma la base de datos vuelva a su estado original y poder ejecutar otra prueba sin errores.

Hemos decidido probar todos los gestores, ya que son los encargados de unir todas las partes funcionales de nuestro software. Haciendo esto nos encargamos de que las

interfaces gráficas realicen las acciones necesarias en la base de datos, viendo que se controlan los posibles errores de forma correcta.

Además hemos realizado tests en algunos DAO, los cuales no eran alcanzados por la cobertura realizada en los gestores, pero en los cuales también queríamos garantizar un funcionamiento correcto, ya que son cruciales para el funcionamiento de nuestro software.

Se han realizado las siguientes [tablas de pruebas](#) para un plan mucho más específico .

5.2.1 La problemática con las interfaces:

Durante el proceso de testing, tomamos la decisión inicial de realizar tests de todos los apartados del software, los cuales incluyen en una gran parte las interfaces gráficas, ya que conforman la mayor cantidad de la base del código del proyecto.

En un inicio configuramos el *Sonar y EclEmma/Jacoco*, de forma que ejecutase todos nuestros test. Pero surgía el problema de que ninguna de las herramientas ejecutaban los test, llegaba a un punto en el que saltaba un error, y nunca llegaba a calcularse la cobertura. Tras muchas pruebas nos dimos cuenta de que el error estaba con las interfaces ya que *Sonar y EclEmma* no pueden hacer el test sobre la interfaz (que era más para otras plataformas como *Selenium*).

Hemos probado a eliminar el test de la pantalla para que sea capaz de ejecutar el resto de los test, pero se queda en un 20% de cobertura, y nunca llegará ni al 50% por las densas líneas de las pantallas (es más de la mitad de nuestro proyecto).

Como solución, hemos excluido el paquete de presentación del coverage, para así tener una representación más real de las coberturas de nuestros tests. Por otra parte, la presentación se ha probado a mano por personas reales, sin estar automatizado.

5.3 Análisis resultado:

Al final de los tests mencionados previamente y tras solucionar la problemática con las Interfaces conseguimos llegar al punto que nos propusimos en cuanto a optimización y calidad del software, obteniendo finalmente los siguientes resultados:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
persistencia		76%		79%	29	100	122	534	17	64	3	11
negocio.entities		50%		40%	111	162	122	321	55	106	2	13
presentacion		8%		100%	43	47	47	55	43	46	41	42
negocio.controllers		80%		74%	24	78	34	140	9	41	8	19
Total	1,946 of 5,630	65%	101 of 260	61%	207	387	325	1,050	124	257	54	85

6. Mantenimiento:

El plan de mantenimiento que se ha llevado a cabo ha sido ejecutar sonar cloud para poder analizar los distintos problemas y corregirlos (detallado previamente en el apartado de calidad). Cada vez que se solucionan dichos problemas, se volvería a realizar un nuevo análisis de calidad. Convirtiendo esto, en un ciclo para mantener el software.

Además de ir solucionando estos problemas se ha tratado de seguir los principios de clean code, en el caso de los principios SOLID:

- Principio Single Responsibility → Todas nuestras clases tienen una única responsabilidad. Por ejemplo, CursoPropio es la clase para todos los cursos o Profesor para los profesores.
- Principio Open Closed → Para cumplir este principio hemos dejado abiertas las clases a posibles extensiones pero no a modificaciones. Volviendo a tomar como ejemplo la clase CursoPropio, esta clase no se modificará pero si se puede extender, usando herencia para formar nuevos tipos de curso.
- Principio Liskov's Substitution → Siguiendo este principio cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. En concreto las clases de Profesor, ProfesorUCLM y ProfesorExterno.
- Principio Interface Segregation → Siguiendo este principio conseguimos que las clases implementadas, no implementen métodos que no necesiten realmente.

Por otro lado, para la mantenibilidad de los tests se ha seguido los principios F.I.R.S.T:

- Fast → Se ejecutan de manera rápida.
- Independent → Son independientes entre sí.
- Repeatable → Se pueden ejecutar en cualquier entorno.
- Self-Validating → Pasan una respuesta booleana.
- Timely → Este es el único principio que no se ha llegado a cumplir, sin embargo, a futuro se piensa implementarlo haciendo primero los tests y luego producir el código.

Tras la realización del plan de mantenibilidad y el seguimiento de los principios de "clean code", se consiguió reducir los bugs, code smells, código duplicado así como los problemas de seguridad. Consiguiendo de esta forma una mejora en los indicadores de calidad.