UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Evolución de un componente para reutilización de patrones de requisitos

Autor

Awais Iqbal Begum

Directora

Carme Quer Bosor

Codirectora

Cristina Palomares Bonache

Trabajo de Fin de Grado presentado bajo el marco del

Grado en Ingeniería Informática

en la especialidad de

Ingeniería del Software

Junio 2018

Resumen

La reutilización de requisitos de software puede ayudar a los ingenieros de requisitos a obtener, validar y documentar los requisitos de software y, como consecuencia, obtener especificaciones de requisitos de software de mayor calidad tanto en contenido como en sintaxis.

El sistema PABRE es una propuesta conjunta del grupo de investigación de *Ingenieria del Software y Servicios* (GESSI) de la UPC y del *Centre de Recherche Publique Henri Tudor* (TUDOR), que consiste en un framework de reutilización de requisitos basado en patrones. El desarrollo del sistema PABRE ha sido liderado por GESSI y la implementación de las distintas herramientas que lo componen se ha ido desarrollando gracias a distintos proyectos fin de master y grado de la *Facultad Informática de Barcelona* (FIB).

Mi proyecto se centrará en uno de los subsistemas del sistema PABRE, llamado PABRE-WS, que consiste en la implementación mediante servicios web de una API que proporciona consultas y actualizaciones de un catálogo de requisitos de software.

El primer objetivo de mi proyecto consiste en mejorar la calidad del código de los servicios web de PABRE-WS para que sean más mantenibles, fiables y robustos. Para ello se realizarán muchos cambios internamente sin afectar a las funcionalidades. El segundo objetivo consistirá en realizar cambios en la estructura de los patrones de requisitos que se requiera en los proyectos europeos OpenReq y Q-Rapids que son dos de los actores interesados en el proyecto. Para terminar, se añaden nuevas funcionalidades a los servicios web para dar mejor servicio a los usuarios de dichos servicios.

Finalmente, el resultado de este proyecto se incorporará a las herramientas que componen el sistema PABRE. Esto ampliará y otorgará mayor valor al sistema, mejorando así las herramientas que dan soporte al trabajo del grupo de investigación GESSI.

Resum

La reutilització de requisits de software pot ajudar als enginyers de requisits a obtenir, validar i documentar els requisits de software i, com a conseqüència , obtenir especificacions de requisits de software de major qualitat tant en contingut com en sintaxi.

El sistema PABRE és una proposta conjunta del grup de recerca *d'enginyeria del software i dels serveis* (GESSI) de la UPC i del *Centre de Recherche Publique Henri Tudor* (TUDOR), que consisteix en un framework de reutilització de requisits basat en patrons. El desenvolupament del sistema PABRE ha estat liderat per GESSI i la implementació de les diferents eines que ho componen s'ha anat desenvolupant gràcies a diferents projectes di de màster i grau de la *Facultat Informàtica de Barcelona* (FIB).

El meu projecte se centrarà en un dels subsistemes del sistema PABRE, anomenat PABRE-WS, que consisteix en la implementació mitjançant serveis web d'una API que proporciona consultes i actualitzacions d'un catàleg de requisits de software.

El primer objectiu del meu projecte consisteix a millorar la qualitat del codi dels serveis web de PABRE-WS, perquè siguin més mantenibles, fiables i robusts. Per això es realitzaran molts canvis internament sense afectar a les funcionalitats. El segon objectiu consistirà a realitzar canvis en l'estructura dels patrons de requisits que es requereixi en els projectes europeus OpenReq i Q-Rapids que són dos dels actors interessats en el projecte. Per acabar, s'afegeixen noves funcionalitats als serveis web per donar millor servei als usuaris d'aquests serveis.

Finalment, el resultat d'aquest projecte s'incorporarà a les eines que componen el sistema PABRE. Això ampliarà i atorgarà major valor al sistema, millorant així les eines que donen suport al treball del grup de recerca GESSI.

Abstract

The reuse of software requirements can help requirements engineers obtain, validate and document software requirements and as consequence, obtains higher quality software requirements specifications in both content and syntax.

The PABRE system is joint proposal of the research group of *Software and Service Engineering* (GESSI) of UPC and of the *Center of Recherche Publique Henri Tudor* (TUDOR), which consists of a framework for the reuse of requirements based on patterns. The development of the PABRE system have been led by GESSI and the implementation of the different tools that make it up has been developed thanks to different Master's Thesis and final degree projects from the *Facultad de informática de Barcelona* (FIB).

My project will focus on one of the subsystems of the PABRE system, called PABRE-WS, which consists of the implementation through web services of an API that provides queries and updates of the catalogue of software requirements.

The first objective of my project is to improve the quality of the PABRE-WS web services code in order to make them more maintainable, reliable and robust. To do this, I will make many changes internally without affecting the functionalities provided. The second objective is to make changes in the structure of the requirements patterns that are required in the two European projects: OpenReq and Q-Rapids, which are two of the stockholders interested in the project. Finally, the last objective is to add new features that will be incorporated to give better service to the users.

Finally, the results of this project going to be incorporate into the tools that make up the PABRE system. This will expand and give greater value to the system thus improving the tools that support the work of the GESSI research group.

Tabla de contenido

Tabla de ilustraciones

1 Contexto

En este capítulo se explica el contexto en el que se ha realizado el trabajo final de grado. Empieza con una introducción en la sección 1.1, seguidamente en la sección 1.2 de los actores involucrados en el proyecto.

1.1 Introducción

Este proyecto es un Trabajo Final de Grado (TFG) de la especialidad de Ingeniería del Software de la Facultad de Informática de Barcelona, UPC (Universitat Politècnica de Catalunya). Se trata de un proyecto de modalidad A en el cual se intenta adaptar el producto software PABRE-WS desarrollado bajo la dirección del grupo de investigación GESSI de la UPC [1] para su utilización en los sistemas resultantes de dos proyectos europeos, OpenReq [2] y Q-Rapids [3]. OpenReq tiene como meta desarrollar, evaluar y transferir métodos innovadores, algoritmos y herramientas para ser aplicados en Ingeniería de Requisitos en proyectos de software distribuidos y de gran escala. Q-Rapids, tiene como objetivo mejorar la calidad del software desarrollado mediante metodologías ágiles, aumentar la productividad del ciclo de vida del software y reducir el tiempo que tarda el software en llegar al mercado.

PABRE-WS es un producto software que implementa servicios web para el acceso y actualización de patrones de requisitos de software. El objetivo principal de este TFG es adaptar PABRE-WS para que cumpla las necesidades de los dos proyectos anteriormente mencionados, mejorando la calidad del código y en especial la mantenibilidad, extendiendo el dominio añadiendo nuevos atributos que se requieran e implementar búsquedas para que desde los sistemas desarrollados en OpenReq y Q-Rapids se pueda obtener patrones o partes de patrones que se necesitan para su funcionalidad.

1.2 Actores

Los Actores implicados son las partes interesadas en el desarrollo y resultado de este proyecto, es decir, cualquier persona o entidad que se vea afectada por el desarrollo y puesta en marcha de la nueva versión de los patrones y los servicios web.

1.2.1 OpenReq

OpenReq (Intelligent Recommendation & Decision Technologies for Community-Driven Requirements Engineering) es un proyecto europeo del programa Horizonte 2020 [4]. Los proyectos europeos están compuestos por un consorcio de entidades que colaboran en el desarrollo de un producto que surge de la investigación y desarrollo de los miembros del consorcio. El consorcio está compuesto por universidades y empresas. Entre las empresas, se definen problemas prácticos que tienen actualmente en dichas empresas, y donde el producto a desarrollar en el proyecto debe aportar mejoras. El grupo de investigación GESSI es una de las universidades del consorcio de OpenReq y son responsables del paquete de trabajo encargado de la implementación de las funcionalidades de recomendación y reúso de requisitos. Las empresas que definen los casos prácticos donde aplicar el producto resultante de OpenReq son Siemens, Qt, WinTre y Vogella.

Este TFG brindará soporte a uno de los objetivos buscados por OpenReq: “Aumentar la reutilización de los requisitos mientras se domina la complejidad de las interdependencias de los requisitos”. Por lo tanto, los miembros del consorcio de OpenReq, y en especial las empresas que definen los problemas a solucionar en el proyecto, serán los principales beneficiados con el desarrollo realizado en este TFG.

La nueva versión de PABRE-WS aportará a OpenReq la posibilidad de reusar requisitos. La clave de la nueva versión será el ofrecer la facilidad de realizar varias consultas sobre patrones, además de la nueva funcionalidad de importar y exportar un catálogo completo. Así, las organizaciones podrán tener una herramienta personalizada que les permita detectar relaciones entre sus requisitos y decidir qué acción tomar en cada situación. Esto traerá grandes beneficios, ya que mejorará la eficacia de la ingeniería de requisitos y adelantará posibles riesgos y defectos.

1.2.2 Q-Rapids

Q-Rapids (Quality-aware Rapid Software Development) es también un proyecto europeo del programa H2020. El grupo de investigación GESSI es en este caso coordinador del consorcio, así como co-responsable de la tarea de implementación de las funcionalidades de recomendación de requisitos [CQ1] . Las empresas que definen los casos prácticos donde aplicar el producto resultante de Q-Rapids son Nokia, Bittium, iTTi y SOFTTEAM [CQ2] .

Este proyecto brindará aporte a dos objetivos buscados por Q-Rapids: “Mejorar los niveles de calidad del software” y “Reducir el tiempo hasta que el software salga a producción”. Al igual que en el caso de OpenReq, los miembros del consorcio de Q-Rapids, y en especial las empresas que definen los problemas a solucionar en el proyecto, serán los principales beneficiados con el desarrollo realizado en este TFG.

La nueva versión de PABRE-WS aportará a Q-Rapids la posibilidad de obtener los patrones de requisitos de calidad que son necesarios para la generación de requisitos que mejoren la calidad de un producto software.

1.2.3 GESSI

GESSI (Grupo de Ingeniería del Software y de los Servicios) es uno de los grupos de investigación de la UPC. El grupo GESSI desarrolló el sistema PABRE [5] y todos los productos que lo componen actualmente gracias al trabajo de estudiantes en TFGs y trabajo final de master (TFMs) de estudiantes de la FIB. Existen múltiples publicaciones que describen los patrones de requisitos de PABRE y las herramientas del sistema PABRE, así como la tesis de Cristina Palomares [6]. El grupo GESSI propone los patrones de requisitos por las ventajas del uso de patrones de requisitos para mejorar la eficiencia de la licitación de requisitos y para mejorar la calidad de las especificaciones de requisitos.

El grupo GESSI está interesado en que el sistema PABRE sea usado a nivel práctico. Los proyectos europeos que han visto una utilidad en los patrones de requisitos son una oportunidad para este uso. Por tanto, se puede considerar al grupo de investigación GESSI como el cliente del proyecto. El resultado del TFG adaptará el sistema PABRE para que sea usado en los proyectos OpenReq y Q-Rapids y por lo tanto es de interés para el grupo ya que otorgará mayor valor al sistema.

1.2.4 Directora y codirectora

La directora del proyecto Carme Quer y la codirectora Cristina Palomares pertenecen al grupo GESSI y han sido las directoras de la mayoría de TFGs y TFMs. También han sido autoras junto con Xavier Franch de todas las publicaciones sobre los patrones de requisitos de software según la propuesta de PABRE. Además, Cristina realizó la tesis doctoral sobre la definición y uso de patrones de requisitos en las actividades de ingeniería de requisitos. Con lo cual las hace interesadas en que el TFG sea un éxito.

1.2.5 Desarrollador del proyecto

Este proyecto se trata de un TFG, con lo cual el desarrollador que soy yo, tendré que realizar todas las funciones durante el desarrollo, tendré que realizar las funciones de jefe de proyecto al realizar la planificación y costes, arquitecto de software al tener que planificar los refactor que se deben de aplicar para mejorar la calidad del código actual del proyecto, programador al tener que implementar todas implementaciones y, por último, tester al tener que definir y realizar los tests de todas las funcionalidades nuevas que implemente.

Al realizar todas esas funciones personalmente conseguiré mucha experiencia, aprenderé a gestionar un proyecto, practicaré técnicas de mejora de la calidad del software, y aprenderé a usar nuevas librerías.

2 Estado del arte

En este capítulo se explica el estado del arte actual, se pone en contexto, se realiza un estudio del mercado y se dan unas conclusiones de porque se debería de realizar el proyecto.

2.1 Contextualización

La reutilización de requisitos de software puede ayudar a los ingenieros de requisitos a obtener, validar, documentar los requisitos de software y como consecuencia, obtener especificaciones de requisitos de software de mayor calidad tanto en contenido como en sintaxis.

Hay diversos enfoques para la reutilización en la ingeniería de software. De entre todos ellos, los patrones ocupan una posición destacada. Un patrón es una solución genérica en forma de descripción o plantilla a un problema que aparece en múltiples y diversas situaciones. El término original proviene del arquitecto Cristopher Alexander el cual lo formuló de la siguiente forma: "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez." El ámbito del proyecto PABRE está centrado en la definición y uso de patrones en la etapa de ingeniería de requisitos, en concreto en la definición y uso de patrones de requisitos de software.

El grupo de investigación de Ingeniería del Software y Servicios (GESSI) [7]de la UPC junto con el *Centre de Recherche Publique Henri Tudor* (TUDOR) [8] propusieron hace unos años el framework PABRE [5] de reutilización de requisitos basada en patrones. Su método de estudio consistió primeramente en crear una versión preliminar de un catálogo de patrones de requisitos, así como su modelo conceptual subyacente. Para ello se basaron en las especificaciones de requisitos de sistemas de varios proyectos reales realizados por expertos en el campo de las TIC, en antecedentes en el campo de la ingeniería de requisitos (en particular sobre patrones de requisitos) y en el asesoramiento de algunos expertos en la materia. La primera versión del catálogo se mejoró y validó con la ayuda de ingenieros de requisitos que consideraron su aplicación en proyectos de la industria donde participaban.

Para facilitar la aplicación del framework PABRE, GESSI lideró el desarrollo del sistema PABRE para la construcción, gestión y uso de patrones de requisitos. Dicho sistema se desarrolló principalmente gracias a distintos proyectos de grado y máster de la FIB.

Actualmente el sistema PABRE (ver Ilustración 1), está formado por las siguientes herramientas o subsistemas:

- PABRE-Man: Aplicación que permite acceder y modificar el catálogo de patrones de requisitos, así como obtener estadísticas sobre el uso de estos.
- PABRE-Proj: Aplicación que permite acceder al catálogo de patrones de requisitos, definir y gestionar los requisitos de un proyecto de desarrollo de software en particular y producir documentación sobre la especificación de requisitos de un proyecto.
- PABRE-RW: Aplicación web que mediante la invocación de los métodos proporcionados por los servicios web PABRE-WS permite acceder y usar el catálogo de patrones de requisitos dentro de un editor que proporciona la funcionalidad de definir requisitos de un proyecto de desarrollo de software.
- PABRE-WS: Servicios web basados en una API REST que permite acceder al catálogo de patrones. Una API REST [9] es un conjunto de métodos que permite mediante el protocolo HTTP de transmisión de información vía web consultar, modificar, actualizar

y eliminar datos. En este caso, tales datos son los patrones de requisitos contenidos en el catálogo.

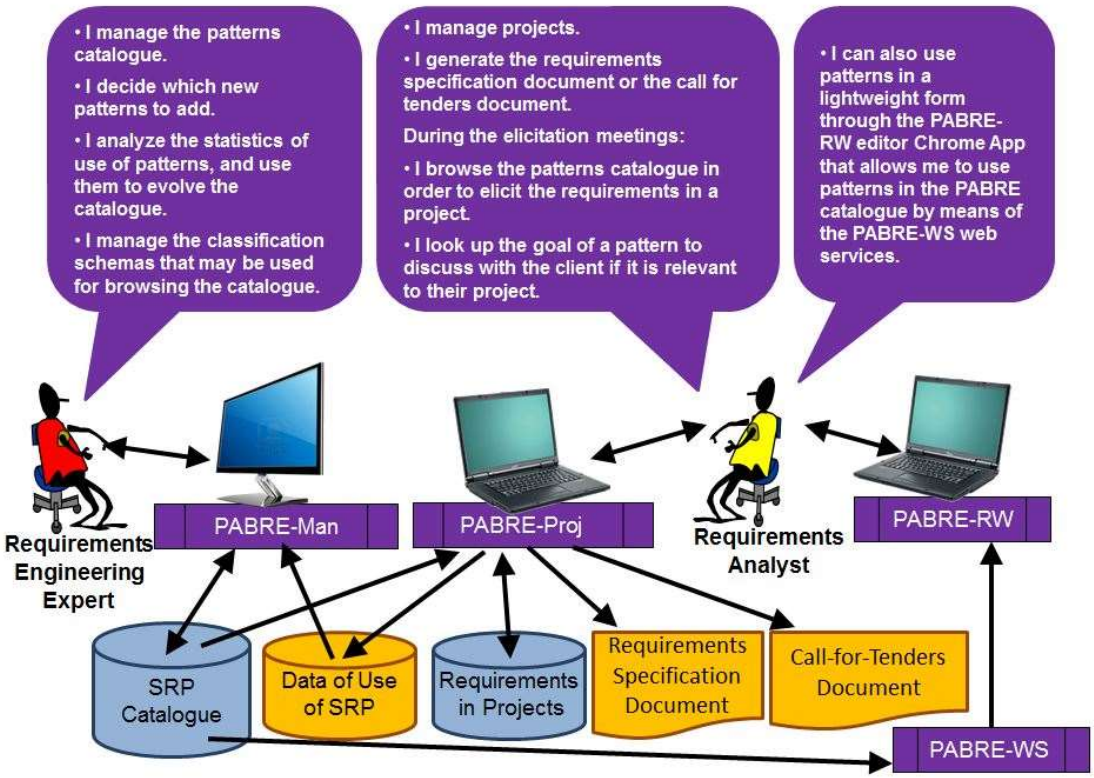

Ilustración 1: Sistema PABRE

2.2 Estudio del mercado

El estudio de mercado se ha realizado desde 2 puntos de vistas, como herramientas comerciales y herramientas que no están en el mercado, pero han sido propuestas por grupos de investigación y que, por el momento, no son usadas a nivel comercial. En este estudio de mercado se buscan alternativas a las herramientas que actualmente proporciona el sistema PABRE. Es decir, se buscan herramientas de gestión de requisitos que permitan definir y usar patrones de requisitos para la especificación de requisitos, que sean accesibles por parte de aplicaciones cliente mediante servicios web.

En cuanto a herramientas propuestas por grupos de investigación se puede decir que, según la tesis de Cristina Palomares [6] donde realizó un estudio sistemático de las propuestas existentes, existen múltiples propuestas de uso de patrones de requisitos, pero ninguna de ellas incluye un framework tan completo como PABRE. Por otra parte, las herramientas que dan soporte a las propuestas existentes tampoco servirían para ser adaptadas para gestionar patrones del nivel de complejidad propuesto por PABRE.

En cuanto a herramientas comerciales, hay más de 100 herramientas de gestión de requisitos actualmente en el mercado. Para hacer un estudio de mercado completo se debería estudiar todas ellas para ver si disponen de la posibilidad de definir y usar patrones de requisitos. Al no poder analizar todas ellas por cuestiones de tiempo, se realizó un estudio de las 8 herramientas clasificadas como las más usadas actualmente según los siguientes estudios:

- List of Requirements Management Tools [10]
- Market Guide for software Requirements [11]

- Modern software requirements management tools [12]
- Requirements Management tools evaluation report [13]

Después del estudio se ha podido ver que en ninguna de las herramientas se usan patrones de requisitos como tal, pero en algunas de ellas se habla de otros conceptos como: Requirement Templates y Requirement Libraries que pueden considerarse también artefactos para facilitar el reúso de requisitos

Requirement Templates: Esta opción nos permite definir plantillas por cada tipo de requisitos que se requiera. Los requisitos pueden ser creados mediante plantillas definidas anteriormente, o bien creados de forma totalmente independiente a las plantillas. Así pues, cuando se modifica una propiedad de un requisito la plantilla original no se ve afectada, pasa lo mismo cuando se realiza un cambio en una plantilla, este no se refleja en los requisitos creados antes de ese cambio.

Requirement Libraries: Algunas herramientas de gestión de requisitos permiten agrupar diferentes requisitos en grupos para poder reutilizarlos como un conjunto. Esta funcionalidad es especialmente útil cuando se trata de requisitos que se son utilizados en varios proyectos y cuando se cambia en uno se realiza el cambio en todos, por ejemplo, requisitos de regulaciones o seguridad. La ventaja que aportan estas herramientas es que las funcionalidades aportan consistencia al realizar cambios, ya que cuando se modifica algún requisito este cambio es aplicado en todos los lugares donde se ha utilizado esta librería.

De las 8 herramientas solo hay 3 que ofrecen alguno de estos dos tipos de artefactos de reúso. A continuación, se indica para cada una de las 3 herramientas que tipo de artefacto ofrece.

2.2.1 DOORS

DOORS (Dynamic Object Orient Requirements System) [14] es un software propiedad de IBM que sirve para realizar gestión de requisitos. Esta herramienta permite gestionar documentos de requisitos de forma centralizada para facilitar la colaboración en un proyecto, permite enlazar diferentes requisitos.

Ilustración 2: IBM DOORS logo

En el estudio se comenta que permite tener varias plantillas para poder reutilizar los requisitos descritos anteriormente, aunque no se ha podido comprobar si permite reutilizar las mismas plantillas en diferentes proyectos y según el estudio este software no permite tener librería de requisitos.

2.2.2 Modern Requirements4TFS

Modern Requirements [15] es un producto desarrollado por eDev Technologies, esta herramienta unas de las cosas que permite hacer es definir los requisitos mediante diagramas y casos de uso. También permite analizar el impacto que tendría cualquier cambio en un requisito entre otras cosas.

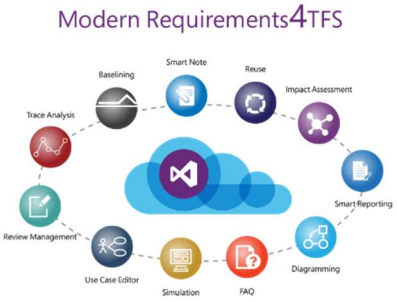

Ilustración 3: Modern Requirements4TFS

En el estudio se muestra que esta herramienta sí permite tener plantillas para poder reutilizar patrones,

permite reutilizarlos mediante búsquedas por palabras claves que se deben de usar en la fase licitación. A su vez también permiten tener una librearía de preguntar (FAQ) que permite al ingeniero a licitar los requisitos. Las preguntas están organizadas en 2 grupos funcionales y no-funcionales. Las preguntas se pueden modificar y guardarse para poder reutilizarse en otro proyecto diferente y así la próxima vez solamente hay que responder las preguntas para saber qué requisitos se necesitan.

2.2.3 JAMA

Por último, JAMA [16] es una herramienta desarrollada por JAMA Software, unas de las cosas que nos permite realizar son: crear, almacenar y consultar diferentes requisitos que se tienen dentro de un proyecto, también permite realizar todo el trabajo en colaboración con varias personas y realizar validaciones mediante tests.

Ilustración 4: Jama logo

Esta herramienta no tiene ningún tipo de plantilla para poder reutilizar los requisitos ya creados anteriormente, en cambio sobre el tema de librearía lo más parecido que tiene es que permite duplicar proyectos y poder reutilizar los requisitos del anterior proyecto. Aunque el concepto de librearía no está tratado por el software.

2.3 Conclusiones

Como conclusión se puede decir que no existe ninguna herramienta actualmente en el mercado que ofrezca la posibilidad de definir y usar patrones de requisitos como lo permite el framework PABRE. Dado que, tal como se ha comentado en los apartados anteriores, hay algunas herramientas que se relacionan con el tema de reutilizar patrones, pero ninguno es tan completo como PABRE.

PABRE nos permite ir más allá de las plantillas y/o librearías, permite tener catálogos donde se pueden tener patrones predefinidos e ir añadiendo contenido nuevo sobre la marcha. Eso permite tener más flexibilidad en la fase de licitación y ahorrando recursos a las empresas. Otra de las ventajas de PABRE sobre la competencia es que tiene un servicio web que nos permite acceder a todo el contenido desde una página web de forma remota.

Tal y como se ha comentado hasta ahora hay motivos más que suficientes para evolucionar el framework PABRE y añadir nuevas funcionalidades, para que tenga más ventajas sobre el resto de herramientas que se han comentado anteriormente, y proporcionar mejor servicio a los clientes que utilicen el framework.

3 Formulación del problema

En este capítulo vamos a hablar de la motivación por la que se realiza este TFG, los objetivos que tiene y por último de los requisitos a cumplir al final el proyecto.

3.1 Motivación

El primer motivo para la realización del proyecto es que el código del framework PABRE, y en concreto el del servicio web PABRE-WS, no tenían la calidad deseada para que pueda ser usado en proyecto europeos para la gestión y búsqueda de patrones de requisitos, en concreto en cuanto a robustez, mantenibilidad y extensibilidad. La mejora de la robustez hará que el software esté el máximo libre de fallos que pudieran afectar en la fase de licitación de requisitos. La mejora de la mantenibilidad y la extensibilidad harán que sea más fácil localizar errores que puedan existir y extender el software con nuevas funcionalidades.

El otro motivo por el cual se realiza el proyecto es que las empresas involucradas en los proyectos europeos que usan los servicios web de PABRE identificaron algunas funcionalidades e información que necesitaban guardar en el sistema que no se permitía de entrada en PABRE-WS.

3.2 Objetivos

Los objetivos de mi proyecto han sido, por una parte, la mejora de la calidad del código, que es algo muy importante, teniendo en cuenta que este framework se utilizará a nivel europeo, y que no habrá solamente un desarrollador involucrado en él, eso nos permitirá tener un código más robusto, mantenible y más fácil de extender en un futuro.

Por otro lado, también estaba el objetivo de ampliar la información guardada sobre un patrón en sistema para guardar más información que necesiten, los proyecto OpenReq y Q-Rapids, así como implementar diferentes búsquedas para facilitar el acceso a la información que requieran.

3.3 Requisitos

Al ser este proyecto desarrollado con una metodología ágil (ver capítulo 5 Metodología), se han producido cambios en los requisitos definidos inicialmente. A continuación, se indican primeramente, los requisitos definidos inicialmente para el proyecto. En el capítulo 4, de alcance del proyecto, se describe el alcance final del proyecto, debido a los cambios que ha habido en las necesidades de los actores, OpenReq y Q-Rapids, durante el durante el proyecto.

Los requisitos iniciales del proyecto eran los siguientes:

- Mejorar la calidad del código base.
    - Mejorar la mantenibilidad y extensibilidad realizando los *refactors* necesarios y documentando la API. En concreto, en el caso de la API, la API de partida no está documentada, así pues, se debe añadir descripciones de lo que hace cada una de las llamadas para guardar, obtener o borrar información.
    - Mejorar la robustez comprobando que todas las funcionalidades se comportan perfectamente como es de esperar y mejorando el entorno de pruebas automatizadas del código, para el posterior uso de este entorno en siguientes versiones y asegurar que añadiendo o cambiando el código las funcionalidades existentes no dejan de funcionar.
- Ampliar la información almacenada de los patrones, implementar los cambios en la información almacenada sobre los patrones de requisitos que se requieran por parte de los proyectos OpenReq y Q-Rapids.

- Implementar consultas complejas sobre los patrones guardados en el sistema: Desarrollar nuevas consultas sobre los patrones. Durante el transcurso del proyecto se ha solicitado implementar consultas algo complejas para pedir información de un patrón en concreto. Que al principio del proyecto no se permitía.

4 Alcance

Como se ha indicado en el capítulo 3, sección 3.3, al ser este proyecto desarrollado con metodología ágil, solo una parte del alcance del proyecto (requisitos) se puede definir desde el principio, pero después el alcance puede sufrir cambios determinados por los cambios en las necesidades de los actores que se van identificando en cada iteración debido a la mejor comprensión por parte de los actores del proyecto y el producto desarrollado conforme este avanza. El alcance al final del proyecto fue el siguiente:

- Ha mejorado la calidad del código. Se han realizado multitud de refactorizaciones del código para que las nuevas implementaciones sean mucho más fáciles mantener, también se ha mejorado para que ahora sea más entendible el código fuente ya que ahora está todo bien organizado, con un aumento de la cohesión y reducción del acoplamiento entre clases y packages. En cuanto a los tests se ha pasado de tener 880 tests a 1.860 tests en total contando las pruebas de las nuevas funcionalidades.
- Se ha extendido el dominio del software para añadir nuevas informaciones que se requerían almacenar sobre los patrones de requisitos. Estas informaciones eran las dependencias entre diferentes patrones y otros objetos del sistema.
- Se ha extendido el dominio del software para añadir funciones de coste a todas las versiones de los patrones.
- Se han extendido las funcionalidades añadiendo métodos de consulta para acceder a todas las dependencias de un patrón.
- Se ha implementado una funcionalidad para exportar todo el catálogo de patrones de requisitos existente.
- Se ha implementado una funcionalidad para importar un catálogo de patrones de requisitos siguiendo un formato en concreto.

4.1 Obstáculos

Al empezar el proyecto se identificó posibles obstáculos que se podrían encontrar durante el proyecto. En esta sección se analizan estos obstáculos que se hubiese podido encontrar durante el transcurso del TFG, y se indica si al final se encontraron o no, y en caso afirmativo como se resolvieron.

4.1.1 Restricción temporal

Dado que el TFG tiene una fecha límite fijada, la cual no se podía extender por las condiciones en las cuales se desarrolla, ante impedimentos que pudiesen surgir el plan era reaccionar de forma ágil adaptando los requisitos del proyecto realmente desarrollados para poder finalizar en el tiempo disponible, y continuar adelante para no comprometer la fecha de entrega.

Finalmente, no ha habido ningún impedimento que haya requerido cambios para reducir los requisitos definidos inicialmente.

4.1.2 Trabajar sobre código ya existente de otras personas

Dado que la parte existente del código perteneciente a los servicios web PABRE-WS lo habían escrito otras personas, se podía dar el caso que resultara complicado entenderlo.

Finalmente, no hubo partes del código suficientemente complicadas para retrasar el proyecto, y todas las dudas pudieron ser resueltas en el tiempo previsto mediante o bien la consulta de la documentación asociada, o bien consultado a la memoria de TFG y TFM de los anteriores desarrolladores.

4.1.3 Configuración del entorno de desarrollo

Para continuar con la implementación del PABRE-WS era necesario disponer de las herramientas específicas con las que se habían desarrollado las primeras versiones de los servicios web. Por tanto, se requería descargar las versiones de los programas apropiadas para no tener problemas a la hora de configurar. En el mundo del desarrollo de software esta tarea a menudo suele involucrar un cierto tiempo debido a la multitud de herramientas y parámetros que entran en juego.

Este si fue un obstáculo con el que me encontré debido a que la configuración con la que se había desarrollado el proyecto no estaba indicada en ninguna parte y no conseguía que funcionará el programa. Al final para poderlo resolver tuve que ponerme en contacto con el último desarrollador y me comento que el utilizó una versión más antigua y el cambio de versión me permitió avanzar.

5 Metodología

En este capítulo se explica la metodología que se ha utilizado durante la realización del proyecto. Se habla sobre la metodología de trabajo, donde se explica cómo se organizado el proyecto para realizar el proyecto. Después se habla de las herramientas para realizar el seguimiento y por último de los métodos de validación.

5.1 Metodología de trabajo

La metodología de trabajo que se eligió está basada en metodología agile [17], con sprints que tenían una duración de 2 semanas, donde al finalizar el *Sprint* se realizaba una reunión de seguimiento con la directora y la codirectora del proyecto. Igualmente, también realizaron reuniones adicionales cuando fueron necesarias. Las reuniones quincenales tenían como objetivo revisar el estado actual del proyecto, determinar las tareas que se tenían que desarrollar hasta la próxima reunión, resolver dudas que se tenían y obtener recomendaciones por parte de la directora y codirectora. Se eligió esta metodología dado que es un proyecto donde los requisitos podían ir cambiando dependiendo de las necesidades de los proyectos europeos interesados en el uso de PABRE-WS.

En cuanto al código, se ha utilizado un repositorio en Bitbucket [18] que utiliza el control de código utilizando la herramienta Git [19] que permite seguir la evolución del código y poder ir realizando entregas según se vaya realizando las tareas habladas durante la reunión que se realizaba antes del sprint. Durante el sprint se han ido creado diferentes ramas para ir realizando las tareas y así en caso de que no funcionara bien algo se podía borrar lo nuevo y poder empezar de nuevo a desarrollar y no estar bloqueado con el mismo problema. A su vez había una rama principal que tenía la versión estable que lo podía utilizar cualquier persona de forma segura ya que no tenía nada que no estuviese probado. Cuando se terminaba el desarrollo de alguna tarea se realizaba un *Merge* con la rama Development donde estaba la versión beta pendiente de la aprobación de la directora y codirectora, una vez dado el visto bueno se pasaba el código a la versión estable.

A su vez para gestionar las tareas realizadas y las pendientes he utilizado Trello [20], esta herramienta nos permite tener diferentes listas donde se pueden añadir tarjetas (como post-its), tenía una lista “Implementation TODO” donde se añadían tarjetas que explicaban las implementaciones que tenía que realizar, “Doing” donde indicaba que es lo que estoy implementando actualmente, “Waiting others” donde guardaba tarjetas con las dudas que tenía y estaban pendiente de responder por parte de otra persona, otra de las listas era “TODO write in memory” donde guardaba tarjetas que ya tenían la implementación realizada pero faltaban documentarlas en la memoria del proyecto y por último “Done” con las tareas ya completadas.

Para organizar mejor el trabajo y tener una perspectiva más amplia del trabajo a realizar, se planificó el proyecto como dividido en 3 etapas. Dentro de cada etapa se realizaban diferentes Sprints.

5.1.1 Etapa 1: Validación y mejora del código existente

En esta etapa se validaba el código existente del Web Service [21] implementado, lo primero que se realizó es comprobar que todo funcionaba correctamente, una vez comprobado se procedió a mejorar las pruebas automatizadas y añadir más pruebas para asegurar la robustez del código, una vez completado lo anterior se procedió a realizar *refactors* al código y para cada cambio que se realizaba se pasaban las pruebas automáticas para asegurar que todo seguia funcionando correctamente y por último se creaba una nueva documentación del PABRE-WS.

5.1.2 Etapa 2: Diseño y desarrollo de las nuevas funcionalidades

En esta etapa se analizaba las nuevas funcionalidades que se debían de añadir a los servicios web PABRE-WS durante la realización del proyecto. Una vez se concretaron los cambios a realizar, se procedía a implementarlos y cada vez que se implementaba algo nuevo se procedía a pasar las pruebas automáticas para asegurar que todas funcionalidades se comportaban como era esperado.

5.1.3 Etapa 3: Cierre del proyecto

En esta última etapa del proyecto, una vez terminadas todas las implementaciones y asegurándose que todo funciona correctamente, se procedió a escribir un manual de configuración para que la siguiente persona que tenga que trabajar en el mismo proyecto pueda configurar el entorno de desarrollo con facilidad y no tenga que perder mucho tiempo en su configuración.

Para finalizar se procedió a redactar la memoria final del TFG y preparar la defensa del proyecto.

5.2 Herramienta de seguimiento

Para la comunicación con la directora y codirectora del proyecto se utilizaron las herramientas habituales como el correo electrónico y en caso de que alguien no pudiese atender a la reunión se le informaba de lo hablado mediante un correo electrónico. Toda la documentación del proyecto se fue almacenando en Google Drive, antes de realizar alguna entrega de GEP se compartía el documento con la directora para poder recibir feedback y modificar los cambios oportunos.

En cuanto al código, como se ha dicho con anterioridad, se ha trabajado con herramientas de control de versiones Git, juntamente con el gestor de repositorios en red Bitbucket, con tal de garantizar la disponibilidad del código, la trazabilidad y facilitar la recuperación en caso de fallos.

5.3 Métodos de Validación

En esta sección se habla de los diferentes métodos de validación que se han utilizado durante el desarrollo del proyecto.

5.3.1 Validación de funcionalidades

Para realizar la validación de funcionalidades que se tenían que implementar, se implementaban los correspondientes tests automáticos que permitían comprobar que la funcionalidad se comportaba tal como era esperado. Además, antes de implementar una funcionalidad se disponía de tests para validar que todo lo realizado hasta ese momento funcionaba correctamente.

5.3.2 Validación de Iteraciones

En las reuniones que se realizaban cada 2 semanas se comentaban las funcionalidades implementadas. También se revisaban los distintos aspectos técnicos o decisiones de diseño tomadas por parte del desarrollador.

5.3.3 Validación de etapas

Al finalizar una etapa se revisaba que se habían alcanzado los objetivos propuestos durante cada una de las reuniones quinquenales. Se aseguraba que no quedaba ningún aspecto por resolver de las iteraciones que la componían.

5.3.4 Validación de aceptación

Al finalizar el proyecto, se realizó una reunión con algún miembro de los 2 proyectos que utilizan esta herramienta, para que den el visto bueno a los cambios.

6 Descripción de las Tareas

En este capítulo se describen las tareas que se han realizado durante el proyecto. Para facilitar la organización las tareas en las 3 etapas en las que se divide el proyecto.

6.1 Etapa 1: Validación y mejora del código existente

Esta primera etapa consistió en analizar el código inicial de PABRE-WS, mejorar la calidad de código, a su vez realizar la gestión del proyecto y realizar las entregas del curso de GEP [22].

6.1.1 Instalación del entorno de desarrollo

La primera tarea fue instalar todas las herramientas que utilizaba la herramienta, comprobar que todos los componentes funcionaban correctamente y que no había ningún problema en la instalación o ejecución.

6.1.2 Estudio el código

La tarea de estudio del código base dejado por Adrián Rambal [23] y Fernando Mora [24] fue necesarias para conocer el punto de partida del proyecto. En concretó se estudió el flujo del programa para entender cómo funcionaba y como interaccionaban las diferentes librerías.

6.1.3 Gestión inicial del proyecto

Entre las primeras tareas fue importante para el proyecto la definición y elaboración de los contenidos necesarios para hacer frente a las entregas demandadas por la asignatura de GEP. Esta tarea permitió establecer las bases para el proyecto y me hizo tener una visión más clara de los requisitos iniciales del proyecto, del contexto del proyecto, así como hacer una primera planificación. Las entregas fueron:

- Alcance del proyecto y contextualización
- Planificación temporal
- Gestión económica y sostenibilidad
- Presentación preliminar
- Pliego de condiciones
- Presentación y Documento Final

6.1.4 Refactorización del código

Una vez finalizado el estudio de código, se procedió a realizar los refactors oportunos en el código, con el fin de mejorar la mantenibilidad, hacer el código cumple los Java Code Conventions [25], también se realizaron cambios para hacer más comprensible el código y cambios en la organización del código.

6.1.5 Documentación de la API REST

En esta tarea se documentó toda la API REST en a la página web de Apiary [26]. En concreto se elaboró descripciones de cada método del servicio web indicando qué hace el método, qué valores de entrada necesita, qué devuelve y todos los posibles errores resultantes de su ejecución.

6.1.6 Análisis y mejora de los tests

El primer paso en esta tarea fue estudiar si era factible con el tiempo disponible en el proyecto pasar todos los tests que existían inicialmente, implementados en un TFG anterior en Postman [27], a Java Assured Rest Testing [28]. El tiempo marcado para saber si era factible o no, fue de

20 horas para realizar la instalación, estudiar el funcionamiento de la librería e implementar algunos tests básicos. Dado que la prueba realizada no fue satisfactoria, ya que se estimó que el tiempo necesario para el cambio de implementación de los tests era demasiado, seguimos con los tests en Postman y solamente revisé todos los tests y mejoré con las comprobaciones que realizaban, para que fueran más estrictas.

6.2 Etapa 2: Diseño y Desarrollo de las nuevas funcionalidades

En esta etapa se implementaron y testearon todas las nuevas funcionalidades que se tenían que añadir en el PABRE-WS y por último se realizó una reunión para que los equipos de OpenReq y Q-Rapids dieran el visto bueno.

6.2.1 Implementación y testeo de nuevos campos para almacenar información

Dentro de esta tarea se añadieron nuevos campos en el catálogo de patrones. Con ello se ha añadido que una versión de un patrón pueda tener una función de coste. También se ha cambiado el dominio para que las métricas no puedan tener dependencias.

6.2.2 Implementación y testeo de nuevas consultas

Esta tarea consistió finalmente en la implementación de una nueva consulta sobre un patrón. La consulta permite que el usuario puede obtener todas las dependencias que tiene un patrón, eso quiere decir que devuelve una lista de todos los objetos que dependen del patrón o de los cuales el patrón depende.

6.2.3 Implementación de nuevas funcionalidades de importar/exportar catálogo

Esta tarea no estaba incluida en los requisitos iniciales y surgió a petición de los proyectos europeos. Consistió en implementar una nueva funcionalidad, que permite al usuario extraer toda la información que hay dentro del catálogo de patrones en un fichero para poder ser exportado. A su vez también se requirió otra funcionalidad que permite importar toda la información extraída anteriormente o poder insertar información nueva directamente desde un fichero en un catálogo inicialmente vacío. Lo que aportará esto es que los proyectos que lo vayan a usar, puedan simplemente pasar su catálogo al completo sin tener que dar acceso a la base de datos.

6.2.4 Realización de reuniones para validar todo el trabajo

En la planificación inicial se contemplaba la realización tests de aceptación, pero debido que uno de los proyectos europeos aun no puede realizar deployment en sus servidores y que el otro proyecto europeo simplemente quería saber cómo funcionaba la API, se realizó simplemente una reunión para informar de cómo acceder a la información.

6.3 Etapa 3: Cierre del proyecto

Esta última etapa, que tuvo una duración de 2 semanas, consistió en finalizar toda la documentación del proyecto.

6.3.1 Redactado de un manual de configuración

En esta tarea se elaboró un manual de configuración para futuros desarrolladores. El objetivo fue que puedan instalar y configurar el proyecto con facilidad, incluyendo posibles fallos y como solucionarlos.

6.3.2 Redactado de la memoria final

En esta tarea se documentó en una memoria lo realizado a lo largo del proyecto. Esta documentación servirá para la defensa final del TFG.

6.3.3 Preparación de la defensa

En esta etapa se realizó la preparación de la defensa, incluye preparar el Powerpoint, realizar las presentaciones de pruebas para saber cómo se debe de dirigir exactamente y preparar posibles preguntas que se vayan a realizar.

6.4 Estimación de tareas

La Tabla 1 muestra la duración de cada una de las tareas descritas anteriormente. En la segunda columna de la tabla se indican las horas estimadas en la planificación inicial para cada tarea. En la tercera columna de la tabla se indican las horas reales dedicadas a cada tarea. Como se puede ver, las horas dedicadas a cada tarea han variado bastante dado que se aplica una metodología ágil y con eso nos permite ir modificando la planificación según las necesidades del equipo de desarrollo. A continuación, destaco algunas diferencias destacables entre el tiempo estimado y el real.

La primera diferencia es en el tiempo estimado para instalar y configurar el entorno. Debido a un problema con la configuración esta tarea me llevo 5 horas más de las esperadas.

En la tarea de refactorización del código se tardó muchas más horas de las estimadas inicialmente (45 horas más) dado que había muchas cosas que cambiar que no se vieron en la planificación inicial.

En la implementación de los nuevos campos de información a incluir en el catálogo de patrones se habría tardado menos tiempo, Pero debido a un problema de implementación, que tardé días en resolver, se acabó ajustando a la planificación inicial.

La implementación de la consulta tardé menos debido que no era tan difícil de implementar como se esperaba inicialmente.

La primera diferencia relevante son las horas dedicadas a la implementación de nuevas funcionalidades de importar/exportar catálogo de patrones. Como se puede observar no había estimación de horas inicial ya que este requisito no estaba entre los requisitos iniciales.

Respecto a los tests de aceptación se invirtió poco tiempo ya que solamente consistió en una reunión donde yo explicaba a los representantes de los proyectos europeos en la UPC, y a los desarrolladores, el cómo instalar y cómo funciona los PABRE-WS y no se hizo en realidad ningún test de aceptación.

Por último, en el cierre el manual de configuración no me llevo mucho tiempo ya que esta vez el proyecto está totalmente configurado para descargarlo y darle al *Run*. La redacción de la memoria se ha ajustado a la planificación inicial y la preparación de la defensa también.

Tarea	Duración inicial (horas)	Duración final (horas)
Instalación del entorno de desarrollo	25	30
Estudio del código	40	40
Gestión inicial del proyecto	90	90
Refactorización del código	15	60
Documentación de la API REST	40	50
Analizar y mejora los tests	70	70
Implementar y testear nuevos campos para almacenar información	70	70
Implementar y testear nuevas consultas	90	50
Implementar y testear nuevas funcionalidades de importar/exportar el catalogo	0	70
Realizar reuniones para validar el trabajo realizado	30	1
Redactado de un manual de configuración	25	1
Redactado de la memoria final	60	60
Preparación de la defensa	20	20
Total	575	617

Tabla 1: Resumen de duración para cada tarea

6.5 Diagrama de Gantt

En la Ilustración 5 se muestra el diagrama de Gantt inicial con las estimaciones de horas a dedicar a las tareas previstas y el grado de riesgo de cada tarea. El diagrama se hizo pensando en días de 5 horas de trabajo. Respecto al riesgo, tal como se puede ver en la leyenda, va de mayor a menor según el orden de los colores: rojo, naranja, amarillo y azul.

Las clasificaciones de riesgo se realizaron de acuerdo con las posibilidades de que las tareas llevaran más tiempo de lo esperado. Por ello todo lo relacionado con redactar la memoria, preparar defensa se estimaros como de bajo riesgo. Las tareas verdes eran las que tenían cierto riesgo como no gestión inicial del proyecto debido a mi poca experiencia en este tipo de actividad. Las tareas de color naranja como realizar tests de aceptación y estudiar el código, se marcaron de color naranja ya que en un principio no conocía como estaba escrito el código y el trabajo que me llevaría el trabajo de mejora de la calidad. Y por último las tareas de color rojo fueron calificadas como las de mayor riesgo ya que inicialmente no estaba del todo claro cuántas consultas y cambios de en la información a almacenar de los patrones habrían.

Como se puede observar en la ilustración indicada anteriormente, las tareas tienen precedencias entre ellas, eso es debido a que hay cosas que no se pueden realizar sin haber realizado otras antes, como, por ejemplo, no se puede redactar la memoria final sin haber terminado todas las implementaciones.

En la Ilustración 6 se puede ver el diagrama de Gantt final, después de haber realizado todo el proyecto.

Como se puede apreciar ahora los colores tienen otro sentido, en la otra ilustración los colores indicaban el grado de riesgo, pero en este diagrama los colores indican si se han superado las horas prevista, el color rojo indica que se han superado las horas esperadas, el color amarillo indica que se han tomado las horas que se esperaban, el color verde indica que al finalizar la tarea se ha tomado menos horas de las esperadas y por último, el color azul indica que es una tarea que no estaba prevista al inicio del proyecto.

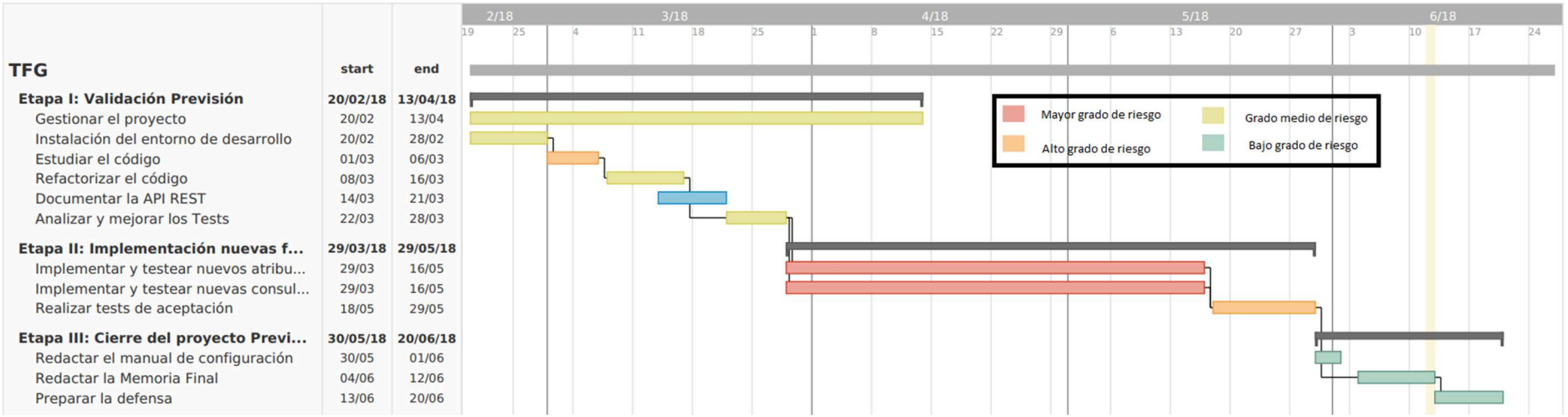

Ilustración 5: Diagrama de GANTT inicial

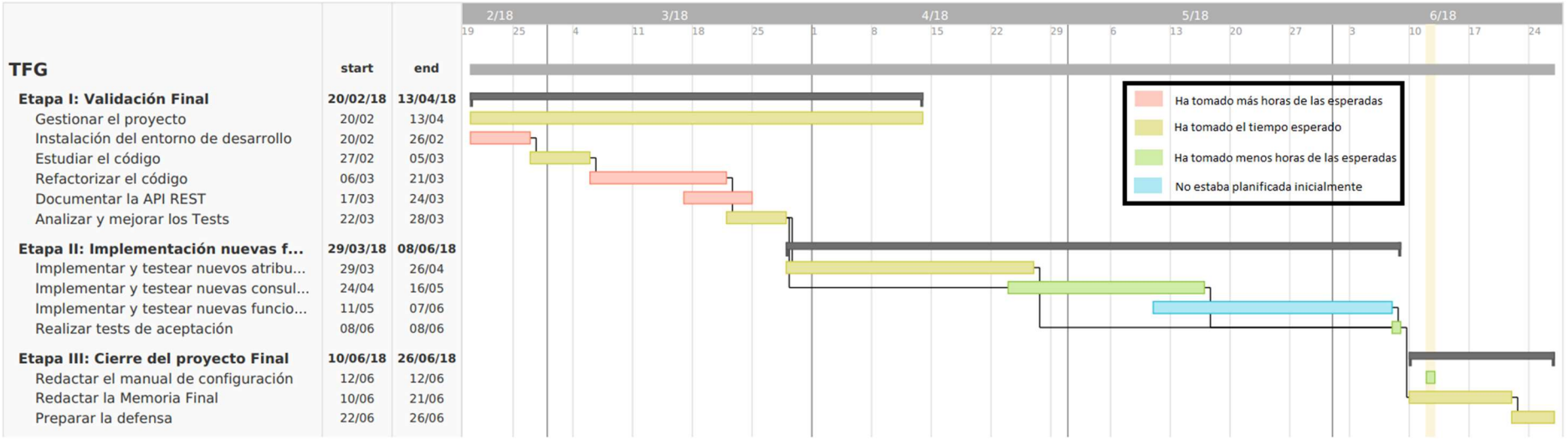

Ilustración 6: Diagrama de GANTT final

6.6 Matriz de Responsabilidades de Roles

En la Tabla 2 se indican los roles que intervendrán en cada una de las actividades y qué responsabilidad tendrán. Como se puede ver, el Jefe de proyecto debe de estar informado de todo lo que está pasando en el proyecto y en varias tareas será en responsable, en caso del analista, será el responsable de la tarea de estudiar el código y en algunas de las otras tareas de implementación será simplemente consultado para saber cómo realizar esa tarea, el programador será responsable de las tareas donde intervendrá y por último el tester será Subordinado-responsable en caso de implementaciones y será responsable a la hora de analizar y mejorar los tests ya existentes

Actividad	Roles			
	J. Proyecto	Analista	Programador	Tester
Instalación del entorno de desarrollo	I	C	R	-
Estudio del código	I	R	-	-
Gestión inicial del proyecto	R	-	-	-
Refactorización del código	I	C	R	-
Documentación de la API REST	I	-	R	-
Analizar y mejorar los Tests	I	-	-	R
Implementación y testeo nuevos campos para almacenar información	I	C	R	A
Implementación y testeo nuevas consultas	I	C	R	A
Implementación y testeo nuevas funcionalidades	I	C	R	A
Realizar reuniones para validar el trabajo	R	-	-	-
Redactado de un manual de configuración	I	-	R	-
Redactado de la memoria Final	R	-	-	-
Preparación de la defensa	R	-	-	-
R (Responsable) – A (Subordinado-responsable) – C (Consultado) – I (Informado)				

Tabla 2: Matriz de Responsabilidades de Roles

7 Riesgos identificados y plan de acción

Debido al uso de la metodología ágil utilizada en el TFG, durante el proyecto se podía ir adaptando dinámicamente la planificación inicial en caso de que surja algún problema.

Las reuniones quincenales con la directora y codirectora permitieron que haya tiempo suficiente para detectar posibles desviaciones de la planificación original y corregirlas. Además, varias tareas se pueden realizar en paralelo con lo cual permitieron dividir el tiempo total entre ellas según la prioridad que tenga esa tarea, ya que ha habido varios entregables sobre la planificación del proyecto, entonces se le dará prioridad a la gestión de proyecto sobre las implementaciones y una vez que este completado el entregable se procederá a seguir con la implementación del código.

Por todo lo mencionado anteriormente, con una dedicación estimada de 30 horas semanales y un total de 617 horas, la planificación del proyecto es factible.

A continuación, se mencionan posibles obstáculos detectados al inicio del proyecto:

Riesgo	Prob.	Impacto	Exposición	Plan de contingencia
Mala planificación	Media	Bajo	Baja	En las reuniones quincenales se revisará la planificación que se está llevando acabo y se harán los ajustes necesarios.
Desconocimiento de las librerías y herramientas de desarrollo	Baja	Alto	Media	Se dedicará tiempo extra al aprendizaje y se realizarán consultas a alguien que conozca cómo funcionan las librearías usadas.
Alguna propuesta para desarrollar no salgan rentable para desarrollar	Alta	Bajo	Baja	En caso de alguna de las cosas a implementar no salgan rentable cuanto a horas a dedicar con los beneficios que aportarán al proyecto se procederá a descartar la implementación

Tabla 3: Riesgos y Plan de contingencia

Debido al tercer riesgo, se descartó la migración de los tests realizados en Postman a Java Assured Rest testing. La acción de descartar se ha tomado ya que las horas dedicadas a instalar la librearía, configurar para que funcionara con todas las librearías instalas, aprender a utilizarlos y generar algunos de los muchos tests han tomado más de 35 horas, donde no deberían de haber tomado más de 20 horas y se decidió descartar la opción de migrar.

En caso de que hubiese habido algún otro imprevisto se podía ampliar la duración del proyecto o dedicando más horas de las necesarias.

8 Identificación de los costes

En este capítulo vamos a mostrar, en la sección 8.1, la estimación que se hizo al empezar el proyecto de su coste. A continuación, en la sección 8.2 se incluye el presupuesto total estimado para el proyecto y el que sería su coste final, una vez terminado. Finalmente, en la sección 8.3 se describe como se hizo la gestión de costes durante el proyecto.

8.1 Costes directos

En este apartado vamos a analizar los costes directos del proyecto

8.1.1 Recursos humanos

Al ser un proyecto de software, el principal coste directo es el recurso humano, en este caso, el yo he desempeñado todos los roles de manera gratuita, pero para realizar un análisis real se tomarán los salarios de referencia para cada rol de un informe del gobierno balear [29], sobre la adjudicación de un contrato donde se indicar los precios que ofrecen las diferentes empresas, de ellos he seleccionado la empresa que menos dinero pedía por cada rol.

Rol	Precio por hora (€)	Horas estimadas	Horas realizadas	Presupuesto (€)	Coste Final (€)
Jefe de Proyecto	45	170	170	7650	7650
Analista	40	79,5	108	3180	4320
Programador	34	177,5	206	6035	7004
Tester	34	148	128	5032	4352
Total				21897	23326

8.1.2 Costes directos por actividad

A continuación, se presenta un detalle de los costes desglosados entre las actividades del proyecto, teniendo en cuenta que solamente es de recursos humanos siendo este el predominante y el que varía según la actividad.

Actividad	Horas estimadas	Horas realizadas	Recursos	Coste (€) presupuesto	Coste final(€)
Instalación del entorno de desarrollo	25	30	Programador	850.	1020
Estudiar el código	40	40	Analista	1600	1600
Gestionar el proyecto	90	90	Jefe de Proyecto	4050	4050
Refactorizar el código	15	60	Analista 50% Programador 50%	300 255	1200 1020
Documentar la API REST	40	50	Programador	1360	1700
Analizar y mejorar los Tests	70	70	Tester	2380	2380
Implementar y testear nuevos atributos	70	70	Analista 20% Programador 50% Tester 30%	560 1190 714	560 1190 714
Implementar y testear nuevas consultas	90	50	Analista 20% Programador 50% Tester 30%	720 1530 918	400 850 510
Implementar y testear nuevas funcionalidades	0	70	Analista 20% Programador 50% Tester 30%	0 0 0	560 1190 714
Realizar tests de aceptación	30	1	Tester	1020	34
Redactar Manual de configuración	25	1	Programador	850	34
Redactar la Memoria final	60	60	Jefe de Proyecto	2700	2700
Preparar la Defensa	20	20	Jefe de Proyecto	900	900
Total				21897	23326

Tabla 4: Costes directos por actividad

8.1.3 Hardware

A nivel hardware se utilizará un portátil para realizar todas las tareas relacionadas con el proyecto.

Para calcular la amortización correspondiente al proyecto se realizan los siguientes cálculos:

$$\frac{Precio\ del\ recurso * Duración\ de\ proyecto\ en\ horas}{vida\ util\ en\ horas} = Precio\ a\ amortizar$$

Recurso	Precio (€)	Unidad	Vida útil	Amortización (€)
MSI CX72-6QD	930,50€	1	4 años	35.04
Total	930,50			35.04

Tabla 5: Costes de Hardware

8.1.4 Software

Durante la realización del proyecto ha sido necesario el uso de herramientas de software, la mayoría de ellas gratuitas.

Para realizar la amortización de la Tabla 6 se ha utilizado la siguiente formula:

$$\frac{Precio\ del\ producto * Duración\ de\ proyecto\ en\ horas}{vida\ util\ en\ horas} = Precio\ a\ amortizar$$

Producto	Precio (€)	Vida útil	Amortización (€)
Microsoft Office 2017	149	3 años	26.28
Adobe Acrobat Reader 2016	0	-	0
Google Drive	0	-	0
Bitbucket	0	-	0
Postman	0	-	0
Apache Tomcat	0	-	0
Total			26.28

Tabla 6: Costes de software

8.1.5 Costes indirectos

El ámbito de trabajo en el que se ha efectuado el proyecto ha sido el mismo piso donde vivo y la universidad donde estudio. Estos constan con los servicios necesarios como electricidad e internet, así como del mobiliario para trabajar cómodamente. En la Tabla 7 se pueden ver los costes asociados.

Servicio	Precio	Cantidad	Coste (€)
Transporte (T-Jove)	100 (€)	1 tarjeta	100
Total			100

Tabla 7: Costes indirectos

8.1.6 Contingencias

La manera elegida para el cálculo de estos costes es destinar un porcentaje, específicamente el 15% del coste de proyecto.

Tipo	Porcentaje	Coste (€)
Costes Directos	15%	3284.5
Costes Indirectos	15%	15
Total		3299.5

Tabla 8: Costes de contingencias

8.1.7 Imprevistos

Un coste improvisto sería que el ordenador utilizado en el proyecto se averíe, con una probabilidad de 5%. La reparación está estimada de media 150€.

Imprevisto	Precio (€)	Probabilidad (%)	Coste (€)
Avería en portátil	150	5	7,5
Total			7,5

Tabla 9: Costes de imprevistos

8.2 Presupuesto final y coste final

En esta sección se suma todos los costes estimados inicialmente para el proyecto, junto con los costes finales una vez terminado el proyecto. Es importante tener en cuenta las siguientes consideraciones:

- Los costes particulares tienen incluidos los impuestos correspondientes, por lo tanto, no hace falta hacer un desglose al final.
- Se presupone que los costes van a mantenerse constantes durante la vida del proyecto.
- No se añade ningún margen de beneficios, ya que el proyecto no tiene fin de lucro.

Concepto	Coste Presupuesto (€)	Coste Final(€)
Costes Directos	21958	23387
Costes Indirectos	100	100
Contingencias	3299.5	0
Imprevistos	7,5	0
Total	25365	23487

Tabla 10: Presupuesto final y el coste final

Como se puede ver en la Tabla 10 el presupuesto inicial era superior el coste final del proyecto.

8.3 Gestión de costes

Con el fin de controlar el presupuesto, al final de cada tarea del proyecto se actualizó el presupuesto con la cantidad efectiva de horas realmente dedicadas, el coste de los recursos utilizados y los gastos de los eventos inesperados que puedan haber ocurrido. Se comparaban estos números con los estimados previamente para obtener indicadores que muestren el desvió entre lo planificado y consumido.

Los indicadores que se fueron calculando durante el desarrollo para gestionar los costes, son los siguientes:

- **Desvió de recursos humanos en precio**: (Coste estimado – Coste real) * Horas reales
- **Desvió de horas consumidas**: (Horas estimadas – Horas reales) * coste estimado
- **Desvió total de horas en precio**: (Horas estimadas – Horas reales) * coste real

- **Desvió total en recursos humanos**: total coste estimado de recursos humanos – coste real de recursos humanos
- **Desvió total en Hardware**: total coste estimado en hardware – total coste real hardware
- **Desvió total en Software**: total coste estimado en software – total coste real software
- **Desvió total en Costes directos**: total coste estimado en costes directos – total coste en coste directos
- **Desvió total en Costes indirectos**: total coste estimado en costes indirectos – total coste en costes indirectos
- **Desvió total en Contingencias**: total coste estimado en contingencias – total coste en contingencias
- **Desvió total en Imprevistos**: total coste estimado en imprevistos – total coste en imprevistos

Además, se permitía un cierto margen de desviación, ya que el coste de imprevistos se ha tenido en cuenta y un porcentaje de contingencia se ha aplicado a la estimación del presupuesto.

9 Sostenibilidad y compromiso social

En este capítulo se describen los aspectos relacionados con la sostenibilidad y el compromiso social que se deben tener en cuenta relacionados con el TFG. Se considera la sostenibilidad y compromiso social desde las tres dimensiones siguientes: económica, social y ambiental. Pero primero se realiza una autoevaluación de mi dominio al inicio del TFG de la competencia de sostenibilidad.

9.1 Autoevaluación

Después de realizar la encuesta, he reflexionado sobre todos los conocimientos que he aprendido durante la realización de distintos cursos que me ayudaran a encontrar soluciones para que mi proyecto sea sostenible y cumpla un papel en el desarrollo social. A continuación, se enumera los principales aprendizajes que me ayudaran.

Hace 2 semestres que realice una asignatura optativa que se llama Software Lliure i Desenvolupament Social (SLDS [30]) donde nos enseñaban todo lo relacionado con el compromiso social. En concreto nos enseñaron como nos afectan todas las acciones que tomamos al comprar electrodomésticos nuevos, qué pasa si no se reciclan los componentes viejos y cómo afectan nuestras acciones al resto del mundo si no hacemos lo correcto. También se habló, aunque sin entrar mucho en detalle de como el desarrollo de un producto de TIC también afecta y no solamente su reciclaje.

El semestre anterior realice otra asignatura llamada Sistemes operatius per a aplicacions distribuïdes (SOAD [31]), donde la competencia transversal era sostenibilidad y compromiso social. En esta asignatura nos enseñaron como tener en cuenta la dimensión social, dimensión económica, dimensión ambiental del software y como llevar a cabo proyectos coherentes con el desarrollo humano y la sostenibilidad.

Considero que durante mis estudios he aprendido suficiente sobre sostenibilidad y compromiso social para tener en cuenta todos los elementos que pueden afectarme durante la realización de este proyecto, en los siguientes apartados se explica todos los puntos sobre sostenibilidad que se han tenido en cuenta.

9.2 Económica

El coste del proyecto viene dado por los recursos humanos, materiales, costes indirectos y otros costes descritos en los apartados anteriores. La mayor parte de los costes del proyecto provienen de los recursos humanos. El tiempo dedicado a cada tarea es el mínimo posible si se quiere tener el coste lo más bajo posible, ya que si se quisiera realizar el proyecto en menos tiempo entonces harían falta más recursos y materiales.

Una vez implantado el proyecto, cuando las organizaciones utilicen el software, mejorará la eficiencia de los ingenieros de requisitos. Esto provocará una mejora en todas las etapas del desarrollo, evitando la utilización de los recursos en tareas que serían desaprovechados por tener una definición mal elaborada.

Entrando en los riesgos, estos son muy pocos. El costo de aprendizaje y uso de la herramienta es bajo en relación al posible beneficio, aunque habrá un tiempo seguramente en que se desee probar la herramienta y no aplicarla, por lo tanto, allí no habrá beneficio.

9.3 Ambiental

Se utilizaron los mínimos recursos posibles para este proyecto, principalmente el eléctrico, requerido para la utilización del portátil y la luz ambiental. El proyecto, al ser un producto software, no requiere de ningún coste de fabricación ni se generará ningún tipo de contaminación, exceptuando la contaminación producida en la fabricación del portátil a utilizar.

Además, la solución planteada reduce el flujo de trabajo innecesario causado por requisitos repetidos o incompatibles que no pueden llevarse a cabo, y por lo tanto se reducen los costes de todo tipo asociados al mismo. Dentro de estos costes están los edificios donde se invertiría el tiempo para desarrollar todos los patrones desde cero o la electricidad que vayan a usarse para generar plantearse todos los patrones desde cero. También está el coste de mantener el servidor donde se alojará el PABRE-WS, y ya dependerá de las empresas que tiempo y recursos van a dedicar a tener activo el servicio.

En cuanto a los riesgos ambientales, puede ocurrir que en algún contexto del uso del WS no exista ningún patrón que cumpla lo que se necesite el cliente. En ese caso como no hay ahorro de trabajo por evitarse un requisito duplicado, no se presenta ninguno de los beneficios ambientales descritos anteriormente. Es más, se añaden gastos energéticos extra por la utilización del PABRE-WS sin encontrar resultado.

9.4 Social

Desde el punto de vista social el proyecto es provechoso para todas las partes, en especial para el autor. He tenido la posibilidad de poner a práctica todos los conocimientos que he obtenido durante los 4 años de carrera que llevo estudiando. También es beneficioso para el grupo de investigación GESSI, dado que se evolucionará y mejorará una componente importante en el producto que tienen que realizar.

Este proyecto busca beneficiar al final del camino a las organizaciones, pero como llegar a ese punto implica un estudio y dedicación que van más allá del alcance de este proyecto. Entonces, se buscará beneficiar a este otro proyecto, que cuenta con más recursos y experiencia en el área de investigación. Ellos podrán exprimir al máximo este trabajo y agregar valor que luego llegue al usuario final como debería.

Una vez aclarado esto, decir que cuando se presentan requisitos defectuosos en un proyecto, se producen malestares en los equipos de trabajo. Esto se debe en principio a los contratiempos que surgen de las redefiniciones y aún peor, errores que pueden producirse una vez implementado el requisito solicitado.

Sobre los riesgos, para las organizaciones que no cuentan con un catálogo centralizado de requisitos y puedan realizar consultas sobre ello. Es habitual que sus trabajadores para compartir la información tengan que hablar y mejorar la comunicación en el equipo. Pero la probabilidad de que empeore el ambiente de trabajo es muy baja.

10 Entorno de desarrollo y seguimiento del proyecto

En este capítulo se describen los productos software necesarios para el entorno de desarrollo y seguimiento del proyecto. En la nueva versión que yo he realizado solamente se cambiado de versión a uno de los programas y cambiar un programa por otro que me proporcionaba más ventajas y facilidad de uso

10.1 Software empleado originalmente en PABRE-WS

Tal como se ha comentado anteriormente este proyecto toma como punto de partida el TFM de Fernando Mora [24] sobre el desarrollo del sistema PABRE-WS [5]. Después de este trabajo de máster, Adrián Rambal [23], en su trabajo de grado, añadió nuevas funcionalidades en el Web Service y ahora yo recogeré todas las herramientas que utilizaron ellos y las que se siguen utilizando hoy en día en el sistema PABRE.

Desde un punto de vista general la arquitectura de PABRE-WS consiste en un servidor que recibe peticiones HTTP [32] de un cliente y envía una respuesta HTTP hacia el cliente, definidas sobre los servicios web de la aplicación. Las respuestas contienen datos en formato JSON [33] que se han obtenido accediendo a la base de datos (BD), las respuestas contienen la información solicitada.

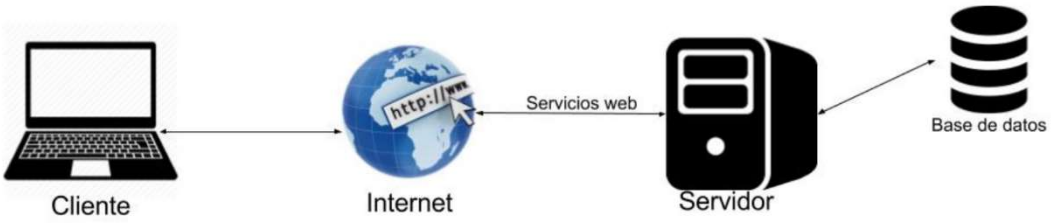

Ilustración 7: Arquitectura simplificada

Más específicamente, desde el punto de vista de las herramientas y tecnologías empleadas en el servidor, la aplicación está escrita en el lenguaje de programación Java [34] y se ejecuta sobre un servidor Apache Tomcat [35]. Una vez llega la solicitud a la aplicación este interacciona con el framework Jersey [36], para pasar la información de un formato plano de texto pasa a un objeto Java mediante el uso de la librearía Jackson. Cuando la aplicación quiere acceder a la base de datos utiliza la librearía de Hibernate [37] para acceder al base de datos.

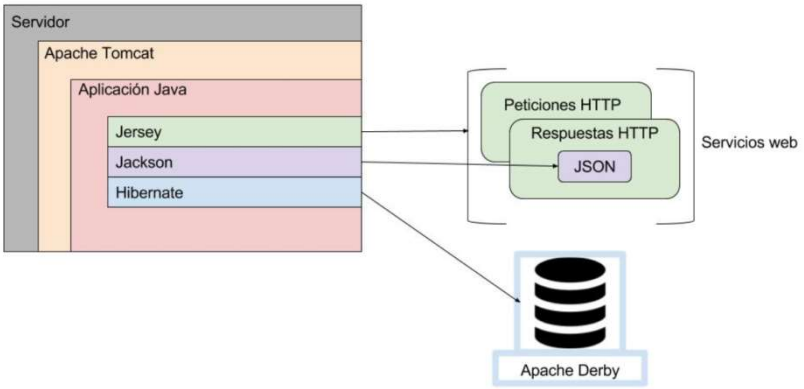

Ilustración 8: Arquitectura del servidor

El desarrollo del código del proyecto se llevó a cabo utilizando Eclipse [38]. Además, se han empleado otras herramientas para la documentación para programadores de los servicios web

(Apiary [26]) así como un cliente para conectarse con la BD (anteriormente en los otros trabajos se usó SquirrelSQL [39], en este proyecto he usado el DBeaver) y para el manejo de dependencias del software se instaló un plugin de Eclipse llamado Maven [40].

A continuación, se describirán más en detalle cada una de las tecnologías utilizadas.

Apache Derby

Apache Derby es una Sistema de gestión de base de datos (DBMS [41] por sus siglas en inglés) desarrollada por Apache Software Foundation [42], que puede ser incrustado dentro de nuestros programas Java, ocupando solo 2.6Mb de espacio, y funcionando directamente en memoria.

Como se ha comentado anteriormente ocupa solamente 2.6 MB de espacio y no requiere instalación ni en la parte de cliente ni en la parte de servidor, es una de los principales puntos a favor por la que se ha elegido en lugar de otra DBMS.

Este gestor de base de datos se puede empaquetar incluyendo el esquema y los archivos de datos del sistema para distribuir muy fácilmente todas las herramientas PABRE con el catálogo de patrones y la información de los proyectos existente a los clientes.

Apache Derby fue desarrollado como un proyecto Open source [43] y está bajo la licencia Apache 2.0 [44].

Apache Tomcat

Apache Tomcat es un servidor web y contener de servlets (aplicaciones Java que son ejecutadas en un servidor y devuelven un resultado) desarrollado por Apache Software Foundation.

Tomcat dispone de herramientas para su configuración y gestión. La forma de ejecutar una aplicación en el servidor consiste en poner su fichero correspondiente en el directorio webapps del servidor. Dado que este proyecto utiliza Eclipse como entorno de desarrollo el Eclipse realiza la tarea de poner el fichero en el directorio de forma automática.

Apiary

La documentación asociada al servicio web existente viene realizada con la plataforma Apiary, es una plataforma perteneciente a la empresa Oracle [45] desde 2017. Las herramientas que ofrece Apiary son de diversa índole: permite diseñar, emplear, implementar APIs, así como crear grupos de usuarios que trabajen en el mismo proyecto desde la misma plataforma. En este proyecto solamente se han empleado las herramientas de documentación de APIs REST.

DBeaver

DBeaver es una herramienta de administración de base de datos. Provee un entorno gráfico para visualizar y editar el contenido de base de datos asó como un editor para ejecutar scripts SQL. Además, soporta plugins para modificar, añadir sus funcionalidades y genera diagrama de relaciones de entidades. Es compatible con casi todas las bases de datos que hay en el mercado.

Es un software gratuito y está distribuido bajo la licencia Apache License 2.0

Eclipse

Eclipse es un IDE (Integrated Developmente Environment) que permite desarrollar programas para una gran variedad de lenguajes de programación, Además de ofrecer diversas herramientas que ayudan en esta tarea:

- Editor de código fuente

- Un debugger [46], que nos ofrece ir paso a paso en el código para saber qué valor tiene cada una de las variables.
- Ejecutar la aplicación de forma automática, ya que cuenta con varios plugins, entonces cuando se realiza un cambio en nivel de código Eclipse compila el código y lo ejecuta por nosotros.
- Etc.

Al inicio del proyecto se ha proporcionado una copia del proyecto en formato que pueda ser entendido por Eclipse, y que contenía todos los ficheros necesarios para que al abrirlo con Eclipse se pueda seguir desarrollando el software.

En el ámbito de este proyecto cabe destacar que además es necesario instalar dos plugins de Eclipse:

- M2eclipse: este plugin nos facilita la integración de Maven [40] con Eclipse.
- M2e-wtp: este plugin nos da compatibilidad con Eclipse Web Tools Platform (WTP), para realizar desarrollo de webs.

En el proyecto originalmente se utilizaba el Eclipse Java EE IDE para web Developmente versión Juno, pero luego yo he descargado la nueva versión la Oxygen 3 release.

Git

Git es un software de control de versiones enfocado a llevar a cabo un registro de los cambios en los archivos de código fuente de un programa. Git guarda estos cambios en un repositorio desde el que se es posible restaurar las distintas versiones de los ficheros que se hayan ido registrando.

El repositorio de Git que se emplea es Bitbucket, disponible a través de internet y ofrecido por la empresa Atlassian. La ventaja de usar un repositorio en la nube es que permite tener una copia de seguridad de todas las versiones del código fuente, incluida la actual. EL repositorio de Git de Bitbucket no tiene coste alguno, ya que es de uso personal.

Git está bajo la licencia GNU v2.0

Hibernate

Hibernate hoy en día es uno de las librearías más populares, es una librería de mapeo objeto/relacional (Object-related Mapping en inglés), que proporciona herramientas para asignar un modelo de objetos a una base de datos relacional.

Hibernate también encarga de manejar las consultas a la base de datos, así como del guardado de los datos. Desde el punto de vista del desarrollo de software un ORM proporciona una capa de abstracción entre la persistencia de los datos y el modelo del dominio que facilita la tarea de los desarrolladores.

La versión empleada en el proyecto original era 3.3.2 que más adelante yo le he cambiado por 3.6.10, por tema de compatibilidades con otros softwares. Hibernate es una librearía totalmente gratuita y está distribuida bajo la licencia LGPL 2.1 [47].

JSON

El formato con el que se representa la información que se retorna o recibe del servidor al realizar una petición HTTP a los servicios web es JSON.

JSON (JavaScript Object Notation) es un estándar que define un formato de transmisión de datos. Especifica la representación de dos tipos de estructuras: Colecciones (también referidas como objects) y Listas (arrays). Los objects están formados por conjuntos de pares nombre-valor, donde a ese determinado nombre le corresponde un valor ya sea un número, testo, una lista o incluso otro objeto. Las listas representan un conjunto de valores.

Es uno de los formatos más extendidos en APIs REST debido a que es fácilmente manipulable tanto por una persona como por una máquina. Además, cada lenguaje de programación tiene sus propias librerías para manipular JSONs.

Jackson

Jackson es una API que permite convertir objetos Java a formato JSON y viceversa. La gran ventaja de usar esta librearía es que es totalmente compatible con Jersey, con lo cual no hay que configurar nada más simplemente instalarla y usando una función se crea un objeto pasando un JSON.

La versión empleada originalmente era 2.8.8. Jackson está distribuido bajo la licencia Apache License 2.0.

Java

Java es un lenguaje de programación de propósito general, concurrente, y orientado a objetos (POO [48]). Fue desarrollado en la década de los 90 por la compañía Sun Microsystems. El objetivo de que tenían era tener un lenguaje orientado a objetos y que sea portable a todos los sistemas.

El objetivo de portabilidad significa que los programas escritos en un lenguaje Java se pueden ejecutar de manera similar en cualquier plataforma de hardware y sistema operativo. Esto se logra compilando el código Java a una representación intermedia llamada bytecode, en lugar de pasar a directamente código máquina. Java bytecode es interpretado por la máquina virtual de java que genera el código máquina de esa máquina en concreto.

Como se puede ver en la Ilustración 9 un programa MyProgram.java escrito en leguaje java se compila a un lenguaje bytecode y por último cada ordenador en su máquina virtual lo ejecuta lo más similar posible.

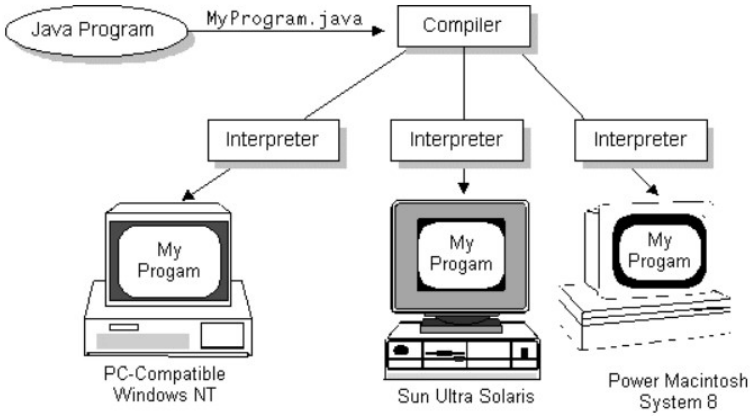

Ilustración 9: Portabilidad Java

Java tiene colector de basura automático para manejar el ciclo de vida de los objetos que se crean. El programador determina cuando los objetos con creado, pero la responsabilidad de eliminar los objetos no utilizados recae sobre Java Collector. Cuando no queda ninguna

referencia sobre el objeto el colector borra todo lo que tenía ese objeto y libera espacio para que ese espacio se pueda volver a utilizar.

Este lenguaje proporciona muchas funcionalidades como librearías que nos proporcionan colecciones, gráficos, hilos y acceso a la red.

Dado a todo lo mencionado anteriormente este leguaje es muy popular entre la comunidad de desarrolladores. Y dado a la gran comunidad de desarrolladores hay muchas librearías que se crean en Java.

Acorde al TFG de Cristina Palomares [6], quien ha desarrollado la segunda versión de PABRE-Man y PABRE-Proj, los siguientes puntos son las ventajas por las que se decidió desarrollar la aplicación en Java:

- Es un lenguaje orientado a objetos.
- Gran variedad de APIs disponibles para este lenguaje.
- Compatibilidad total en todo tipo de hardware/sistemas operativos.
- El bajo rendimientos en la ejecución de la aplicación no supone un problema para este tipo de aplicación.

Jersey

Java define el soporte a la arquitectura REST mediante la especificación JAX-RS. JAX-RS es una API que ofrece soporte para servicios web REST en Java mediante anotaciones. EL framework Jersey es una implementación de JAX-RS que además ofrece otras características que simplifican la programación de servicios REST.

Su funcionamiento en un servidor consiste en recibir llamadas HTTP, procesarlas para saber que parte de código se debe de ejecutar para esa llamada en concreto (indicado en un fichero llamada web.xml) y recoger la respuesta que se dé y devolverla mediante otra llamada HTTP.

La versión empleada en el proyecto es la 1.17 que es desarrollada por Oracle y está bajo la licencia GPL v2.

Maven

Maven es una herramienta de gestión y construcción de proyectos disponible para diversos lenguajes de programación, entre ellos Java llevado a cabo por Apache Software Foundation.

Maven emplea un fichero en formato XML [49], para describir diversos aspectos de un proyecto software: módulos y componentes externos de los que depende el proyecto o como aspectos sobre la compilación del programa.

Debido a que este proyecto emplea diversas librearías externas, esta herramienta simplifica la tarea de instalar las dependencias software especificado las librearais con sus respectivas versiones es un fichero de configuración llamado: pom.xml.

La versión de Maven empleada originalmente en el proyecto era la 3.0.5 que luego lo he actualizado a la versión 3.1 por temas de compatibilidad. Además, es un producto totalmente gratuito y está distribuido bajo la licencia Apache Software License v2.0.

Postman

Postman es un programa que ofrece un entorno de pruebas gráfico para APIs basadas en servicios web. Permite realizar peticiones a cualquier servicio web modificando los parámetros de ésta, así como poder visualizar rápidamente y de forma intuitiva las respuestas del servidor. También proporciona la posibilidad de guardar conjuntos de peticiones en lo que en el dominio

del programa se denominan como coleciones. Las coleciones permiten ejecutar todo un conjunto de peticiones de forma seguida. Además, cada vez que se efectúa una petición se pueden ejecutar pruebas mediante el lenguaje de programación JavaScript [50].

Finalmente, si se ejecuta una colección de peticiones al finalizar las pruebas nos hace un resumen indicando las pruebas que se han pasado y las que no.

Este programa es gratuito de forma personal, pero está bajo licencia Copyright [51].

Squirrel SQL Client

Squirrel SQL Client es una herramienta de administración de base de datos. Provee un entorno gráfico para visualizar y editar el contenido de base de datos asó como un editor para ejecutar scripts SQL. Además, soporta plugins para modificar y añadir sus funcionalidades. Es compatible con diversas bases de datos del mercado y en particular con la empleada en este proyecto Derby DB.

Es un software gratuito y está distribuido bajo la licencia GNU Lesser General Public License.

Decidí cambiar este cliente por otro llamado DBeaver [52], ya que el nuevo ofrecía un mejor entorno y me proporcionaba la funcionalidad de acceder y poder visualizar todos los datos e incluso acceder al diagrama de relación de tablas.

11 Estudio del sistema PABRE

Los patrones de requisitos software (Software Requirement Pattern, en singular, de ahora en adelante SRP) contribuyen a solucionar el problema que comúnmente aparece durante la etapa de definición de requisitos de un sistema. Específicamente este problema consiste en una definición ambigua, incompleta, incoherente y carente de pautas de estilo de los requisitos.

El uso de estos SRPs puede contribuir a crear un marco común de referencia que facilite la licitación de requisitos recurrentes en distintos proyectos y que ayude a mejorar la calidad de la especificación de los requisitos en los aspectos indicados anteriormente.

En este apartado se va a explicar en detalle la estructura de los SRP para facilitar la comprensión del resto del contenido de esta memoria. En concreto se presenta la estructura de los SRP mediante un ejemplo que la ilustra, pero también con el modelo conceptual que describe la estructura de los SRP en general. Para terminar también se muestra como en el sistema PABRE se propone clasificar los SRP dentro de sus catálogos.

11.1 Modelo conceptual de los patrones de requisitos

La primera pregunta fundamental que se debe de responder es cuál es la estructura de un SRP. La Ilustración 10 muestra un ejemplo de SRP que ilustra los atributos más significativos de un patrón.

Este SRP, llamado Failure Alerts, es un SRP que puede ser usado para definir requisitos relacionados con la existencia de alertas de fallo en productos software. Hay que tener en cuenta que el “Goal” del SRP pretende enunciar el problema que se pretende resolver en un producto software, en caso en que en su especificación se incluya requisitos definidos a partir del SRP. En el ejemplo se puede ver pues que el objetivo del patrón es satisfacer las necesidades de un cliente de tener un sistema (producto software) que proporcione alertas cuando ocurra un fallo en este sistema.

Se puede observar que el patrón tiene distintas formas (Pattern Form) que corresponden a las distintas formas de aplicar el patrón. En el ejemplo, un cliente pediría la primera (Hetereogeneous Failure Alerts) en caso que quisiera que el producto software implementado proporcionara la posibilidad de definir distintos tipos de alertas para distintos tipos de fallos. En cambio, pediría la segunda (Homogeneous Failure Alerts) en caso no necesite establecer què tipo de alerta se debe dar en caso de cada tipo distinto de fallo.

Las formas de los patrones, tiene simple una parte fija (Fixed Part), que siempre debe estar siempre definida, y pueden tener opcionalmente múltiples extensiones (Extended Part). En la aplicación de una forma de un SRP se deberá aplicar siempre la parte fija y se podrá aplicar o no las extensiones. En la Ilustración 10 se puede ver que la parte fija de la segunda forma, simplemente establece que el producto software especificado deberá lanzar alertas cuando ocurra algún fallo. Si el cliente, quiere establecer los tipos de alerta i/o los tipos de fallo, deberá usar la primera i/o segunda extensión.

Tanto las partes fijas como las extensiones tienen una misma estructura. Constan de una plantilla (Template) que se usaría para definir los requisitos, y en caso que sea necesario una definición de los parámetros que aparecen en la plantilla. Dichos parámetros deben tomar valor cuando un patrón se aplica en un proyecto. Por ejemplo, en el caso de la primera extensión de la segunda forma (Alert Types) existe un parámetro llamado %alerts%, en la definición de dicho parámetro se da la métrica que establece los posibles valores que puede tomar el parámetro en la aplicación del patrón.

<table>
<tr><th colspan="4">Requirement Pattern Failure Alerts</th></tr>
<tr><td colspan="4">Goal Satisfy the customer need of having a system that provides alerts when system failures occur</td></tr>
<tr><td rowspan="8">Pattern Form
Heterogeneous Failure Alerts</td><td rowspan="2">Fixed Part</td><td>Template</td><td>The system shall trigger different types of alerts depending on the type of failure</td></tr>
<tr><td>Extended Parts Constraint</td><td>Multiplicity (Alerts for Failure Types) = 0..*</td></tr>
<tr><td rowspan="4">Extended Part
Alerts for Failure Types</td><td>Template</td><td>The system shall trigger %alerts% alerts in case of %failures% failures</td></tr>
<tr><td>Parameter</td><td>Metric</td></tr>
<tr><td>alerts: non-empty set of alert types</td><td>alerts: Set(AlertType)
AlertType: Domain of possible types of alerts</td></tr>
<tr><td>failures: non-empty set of failure types</td><td>failures: Set(FailureType)
FailureType: Domain of possible types of failures</td></tr>
<tr><td rowspan="8">Pattern Form
Homogeneous Failure Alerts</td><td rowspan="2">Fixed Part</td><td>Template</td><td>The system shall trigger an alert in case of failure.</td></tr>
<tr><td>Extended Parts Constraint</td><td>multiplicity(AlertsTypes) = 0..1 and
multiplicity(Failure Types) = 0..1</td></tr>
<tr><td rowspan="3">Extended Part
Alert Types</td><td>Template</td><td>The solution shall trigger %alerts% alerts in case of failure</td></tr>
<tr><td>Parameter</td><td>Metric</td></tr>
<tr><td>alerts: non-empty set of alert types</td><td>alerts: Set(AlertType)
AlertType: Domain of possible types of alerts</td></tr>
<tr><td rowspan="3">Extended Part
Failure Types</td><td>Template</td><td>The system shall trigger alerts in case of %failures% failures</td></tr>
<tr><td>Parameter</td><td>Metric</td></tr>
<tr><td>failures: non-empty set of failure types</td><td>failures: Set(FailureType)
FailureType: Domain of possible types of failures</td></tr>
</table>

Ilustración 10: Ejemplo de un SRP

Una vez vista la estructura de un patrón con un ejemplo, es mucho más fácil entender el modelo conceptual de la Ilustración 11. Como se puede ver la clase SRP representa los patrones que habría en un catálogo de patrones de requisitos. La relación de la clase SRP con la clase SRP Form muestra el hecho de que un patrón puede tener diferentes formas (véase el 1..* en la multiplicidad de la asociación). A su vez la relación entre la clase SRP Form y la clase Fixed Part y Extended Part refleja que un patrón debe tener una parte fija (véase el 1 en la multiplicidad de la asociación) y puede tener definidas extensiones (véase el * en la multiplicidad de la asociación). Tanto la parte fija como la parte extendida son ambas una parte de un patrón (clase SRP Part) con una misma estructura, tal como se ha visto en el ejemplo, con su plantilla, parámetros y métricas (véase el atributo template y las clases Parameter, Metric).

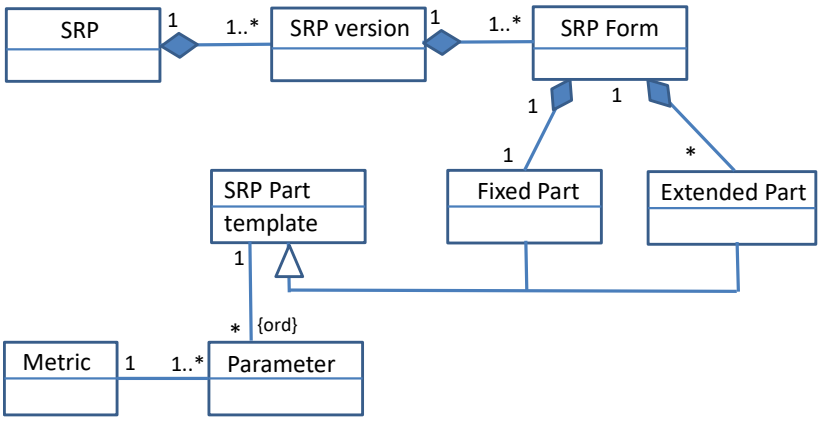

Ilustración 11: Modelo conceptual del sistema de patrones de requisitos

A continuación, se describe cada uno de los componentes que aparecen en el diagrama. Además, se indica los atributos de cada uno que por motivos de presentación no se han incluido en la ilustración.

SRP

Esta clase representa cada uno de los patrones de requisitos. Un patrón representa el problema específico que se quiere resolver. Aunque en el diagrama no muestra que información contiene un patrón (dado que es una versión simplificada).

El patrón contiene información como:

- El nombre del patrón
- El autor del patrón
- Una descripción
- Comentarios añadidos por el autor y los usuarios
- El objetivo de la creación del patrón
- El problema que soluciona
- Fuentes de procedencia
- Una serie de palabras (Keywords) que faciliten la búsqueda de este patrón
- Un conjunto de versiones

SRP Version

Para tener trazabilidad de los cambios sufridos por la evolución de un patrón de requisitos se mantienen versiones de los patrones de requisitos. Así por cada cambio realizado en el patrón se tendrá una versión diferente y con ellos se podrá llevar acabo la trazabilidad de quien ha realizado el cambio.

SRP Form

Un SRP Form representa los distintos modos o soluciones de a alcanzar el objetivo definido por un patrón. Un patrón atreves de sus diferentes versiones, debe de tener como mínimo un SRP Form.

La información que contiene un Form es la siguiente:

- Nombre
- Descripción

- Comentarios
- Fuentes de procedencia
- Autor
- Fecha última modificación

SRP Part

Un SRP Form está compuesto por dos tipos de partes (fijas y extendidas), que se explicarán más adelante. Pero los dos tipos de partes tienen información en común que se le llama SRP Part.

La información que guarda SRP Part es la siguiente:

- El texto del requisito a incluir en la especificación del sistema
- Una pregunta que se puede usar para determinar si un sistema cumple o no el requisito expresado en la parte
- Conjunto de parámetros (Parameter) que tomaran valor al utilizar el patrón de requisito en un proyecto concreto.

SRP Fixed

La parte fija se corresponde con el requisito que se deberá añadir a una especificación de requisitos para que el sistema software especificado cumpla el objetivo definido por el patrón.

Todo SRP Form tiene que tener un SRP Fixed de forma obligatoria

SRP Extended

La parte extendida corresponde con requisitos más concretos que dan más detalle sobre cómo alcanzar dicho objetivo.

Parameter

El texto de un requisito correspondiente a una parte es una frase en lenguaje natura con determinadas palabras clave que corresponden a los parámetros de la parte. Dichos parámetros tomarán valor cuando el patrón sea usado en un proyecto concreto. De este modo, manteniendo un texto de requisito en las partes del patrón se conseguirá requisitos adaptados a las características específicas de cada proyecto solo cambiando los valores de los parámetros.

Metric

Los posibles valores que pueden tomar los parámetros en una SRP Part, vienen definidos por una determinada métrica, la cual aporta una restricción sobre el tipo del valor que puede tomar. Los tipos de métrica son:

- Dominio (dado unos valores seleccionar uno de ellos)
- Números enteros
- Números reales
- Valores booleanos
- Fechas
- Conjunto de una métrica

11.2 Clasificadores

Para estructurar el catálogo de los patrones, el sistema PABRE usa una estructura jerárquica de clasificadores. En la Ilustración 12 se puede ver un ejemplo. Dicha clasificación se propone para simplificar y facilitar búsqueda de patrones durante la tarea de licitación de requisitos y la organización de patrones en el catálogo.

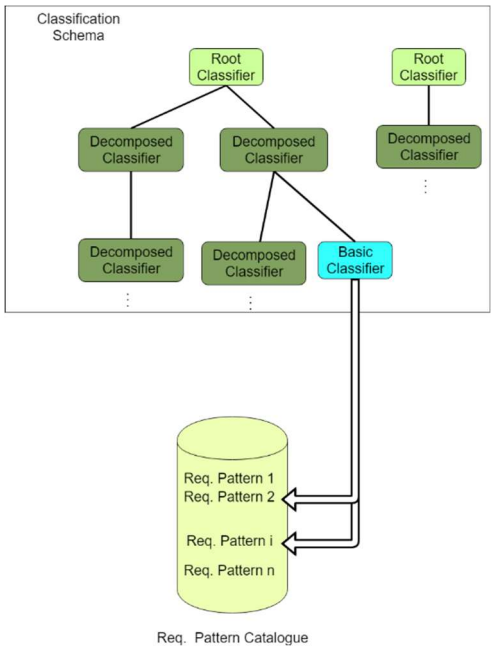

Ilustración 12: Clasificadores

Igual que antes, una vez vista la clasificación de los patrones mediante una ilustración, es mucho más fácil entender el modelo conceptual de la Ilustración 13. Los SRP pueden estar clasificados según distintos esquemas de clasificación (Classification Schema). Un esquema de clasificación consiste en una jerarquía de clasificadores generales (General Classifier). Los clasificadores generales finalmente serán, clasificadores básicos (Basic Clasifier), que son los que agrupan los SRP, y clasificadores descomponibles (Decomposed Classifier) que son los que agrupan otros clasificadores. Finalmente, el nivel más alto de clasificadores descomponibles son lo que llamamos clasificadores raiz (Root Clasifier).

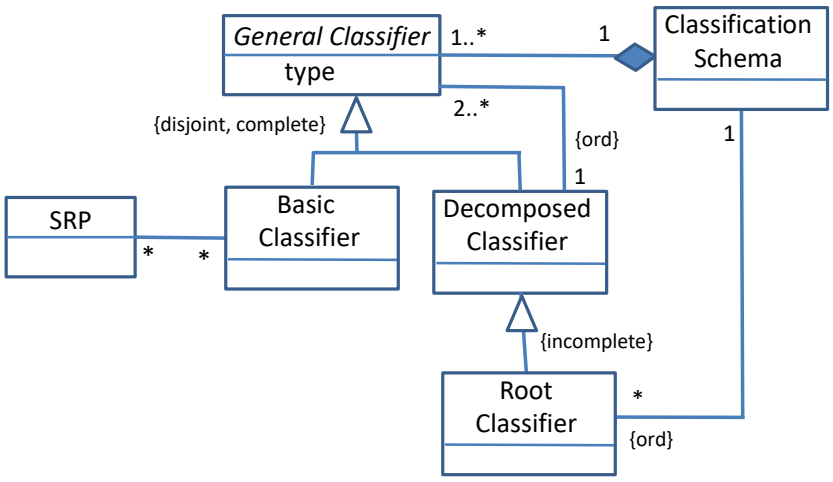

Ilustración 13: Modelo conceptual de la clasificación de requisitos

A continuación, se describe cada uno de los conceptos que aparecen en el diagrama con más detalle.

Classification Schema

Cada SRP puede estar clasificado e indexado de manera jerárquica en un esquema de clasificación (Classification Schema). El sistema PABRE permite que un mismo SRP esté presente en diversos esquemas al mismo tiempo, al fin y al cabo, los esquemas actúan como conjuntos, de forma que el ingeniero de requisitos pueda organizar los distintos SRPs disponibles en el catálogo para luego facilitar su aplicación en distintos proyectos. Internamente los esquemas de clasificación también emplean esquemas de clasificación para definir la jerarquía de los catálogos de una forma más precisa.

Cada esquema puede tener múltiples clasificadores raíz (Root Classifier).

Root Classifier

Los clasificadores que son Root, son el nivel más alto en la jerarquía y son clasificadores descomponibles, que como su nombre indica están descompuestos en un conjunto de clasificadores.

Decomposed Classifier

Un clasificador descomponible es un clasificador que estará descompuesto por clasificadores básicos i/o clasificadores descomponibles.

General Classifier

Se trata de una clase abstracta de clasificadores. En realidad, todos los clasificadores generales, serán o bien clasificadores descomponibles o clasificadores básicos.

Basic Classifier

En la jerarquía de clasificadores de un esquema de clasificación este tipo es el de nivel más bajo de la jerarquía, y no puede descomponerse en otros clasificadores. Es el tipo de clasificador que agrupa y organiza los SRP.

11.3 Dependencias

En esta sección vamos a presentar el sistema de dependencias que tiene PABRE. Las dependencias, como su nombre indica, permiten establecer las relaciones que existen entre distintos elementos u objetos de un catálogo de patrones de requisitos.

Las dependencias pueden existir entre los siguientes elementos:

- Los patrones de requisitos (SRP)
- Las versiones de patrones de requisitos (SRP Version).
- Las formas de los patrones de requisitos, i en concreto de las versiones de los patrones (SRP Form).
- Las partes dentro de una forma de un patrón (SRP Part), que como se ha explicado anteriormente son partes fijar (Fixed Part) o bien extensiones (Extended Part).
- Los parámetros de las plantillas que definen una parte de un patrón (Parameter).

Algunos ejemplos de dependencias son los siguientes:

- Dependencia entre un parámetro donde se indican los tipos de fallo que debe rastrear un producto software que en un catálogo de patrones de requisitos puede estar definidos en un patrón Log Capabilities, con el valor del parámetro %failures% de las

partes del patrón Failure Alerts donde se indican los tipos de alerta que se debe dar según el tipo de fallo que se produzca en el patrón Failure Alerts.

- Dependencia entre el patrón Failure Alerts y el patron Platform, ya que para poder ofrecer ciertos tipos de alertas en función de ciertos fallos deberá disponerse de software específico que de soporte a tales alertas.

Una vez presentados los objetos para los que se puede definir dependencias en un catálogo de patrones de requisitos, y haber mostrado algunos ejemplos de dependencias,, es mucho más fácil entender el modelo conceptual de la Ilustración 11. Como se puede ver la clase Pattern Object corresponde a una clase que representa todos los objetos entre los que se puede definir dependencias.

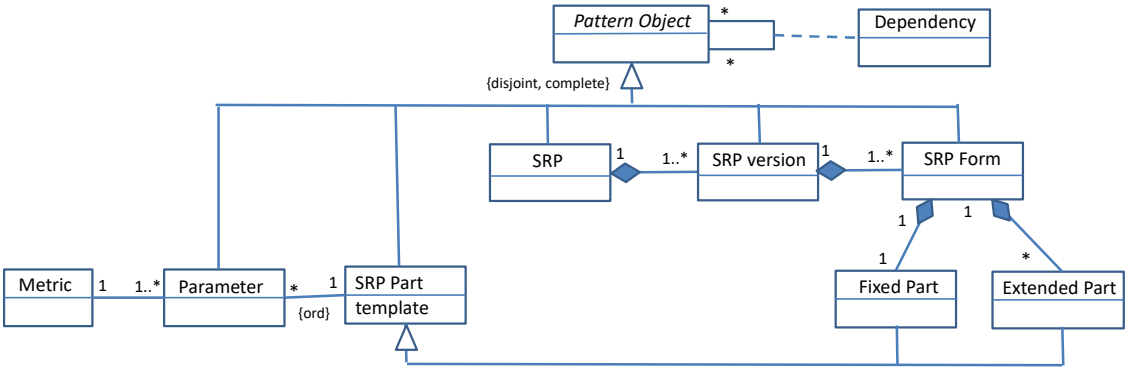

Ilustración 14: Modelo conceptual de las dependencias

En el modelo conceptual de dependencias existen dos conceptos nuevos que se describen con detalle a continuación.

Pattern Object

Se trata de una clase abstracta que agrupa todos los objetos de un catálogo de patrones de requisitos entre los que se pueden definir dependencias. Los tipos de objetos para los que se puede definir dependencias son: SRP, SRP version, SRP Form, SRP Part, Parameter.

Dependency

Las dependencias en un catálogo de patrones pueden existir entre cualquier objecto que sea Pattern Object.

Las dependencias pueden ser de 4 tipos:

1. **Implica** (implies): Un elemento A implica el uso de un objeto B
2. **Excluye** (excludes): Un elemento A excluye el uso de un elemento B
3. **Constribuye** (contributes): El uso de un elemento A contribuye en el funcionamiento del elemento B
4. **Daña** (damages): El uso de un elemento A daña el funcionamiento del elemento B

Por último, cabe destacar que las dependencias pueden ser unidireccionales o bidireccionales.

12 Mejorar la calidad de código

Tal como se ha indicado anteriormente uno de los objetivos de mi proyecto es mejorar la calidad de código.

Aunque se explique de forma secuencial los cambios a nivel de código no fueron así, ya que a medida que se iba avanzando en el proyecto se iban haciendo nuevos cambios para mejorar el código y reutilizar el código ya existente.

Para asegurar que todo funcione correctamente y que al realizar un cambio en el código no afecte a las funcionalidades, después de cada cambio a nivel de código se pasó todos los tests de integridad.

En este capítulo empezaremos hablando de tres técnicas y convenciones que ayudan a mejorar la calidad del código. En la sección 12.1 se explica el objetivo de las Java Code Conventions y la conveniencia de aplicar estas convenciones. Seguidamente, en la sección 12.2 hablaremos de en qué consiste el proceso de Refactoring y cómo aplicarlo. Como tercer aspecto hablaremos de los principios de diseño SOLID en la sección 12.3. El capítulo terminará con la sección 12.4 donde se presentará los cambios realizados en el código del proyecto. Debido a que ha habido muchos cambios, y a que no se quiso crear una memoria muy larga explicando una y otra vez lo mismo, para cada tipo de cambio se explica el tipo de cambio, el objetivo de ese tipo de cambio, y por último un ejemplo de una aplicación concreta del tipo de cambio en el código.

12.1 Java Code Conventions

Java Code Conventions (JCC de ahora en adelante) es una forma estándar de programar en Java, desarrollada por Oracle. Los motivos por los cuales se debe de tener un estándar a la hora de programar son:

1. El 80% del coste total de la vida útil de un software va en mantenimiento.
2. Difícilmente cualquier software es mantenido por el autor que había desarrollado el software
3. Las convecciones mejoran la comprensión del código, permitiendo a los desarrolladores entender nuevo código más rápido.

A continuación, nombraré algunas de las convecciones JCC que no se cumplían en el proyecto:

- Indentación: El código desarrollado en Java tiene que estar indentado a cuatro espacios.
- Longitud de líneas: Las líneas de código no debe de superar los 80 caracteres.
- Sentencias if: Las sentencias if siempre deben de tener ‘{}’ para evitar posibles errores
- Convenciones de nomenclatura de los paquetes: El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel (actualmente com, edu, gov, mil, net, org) o uno de los códigos ingleses de dos letras que identifican cada país como se especifica en el estándar ISO 3166, 1981. Los siguientes componentes del nombre del paquete variarán de acuerdo a las convenciones de nomenclatura internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos o máquinas.
- Convenciones de nomenclatura de las clases: Los nombres de las clases deben ser sustantivos. Cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Se debe intentar mantener los nombres de las clases simples y descriptivos, usar palabras completas, y evitar acrónimos y abreviaturas.

Dado que es un estándar que se debe de tener cuando se desarrolla en Java, se va a modificar el código para que cumpla el JCC.

12.2 Refactoring

El *Refactoring* del código heredado (legacy) es una técnica que aconseja hacer diversas modificaciones el diseño del código para mejorar aspectos como su mantenibilidad.

Por concepto, un *refactor* es un proceso que realiza una modificación en el software pero que a la vez no tiene repercusión a nivel funcional. Dado que estos procedimientos son efectivos y testeados, se presentan altamente necesarios en el momento de reestructurar el código.

La Ilustración 15 muestra como luego de haber reorganizado la estructura interna de un programa se conseguirá el mismo resultado, pero para el desarrollador es mucho más entender la estructura refactorizada que la que no está refactorizada.

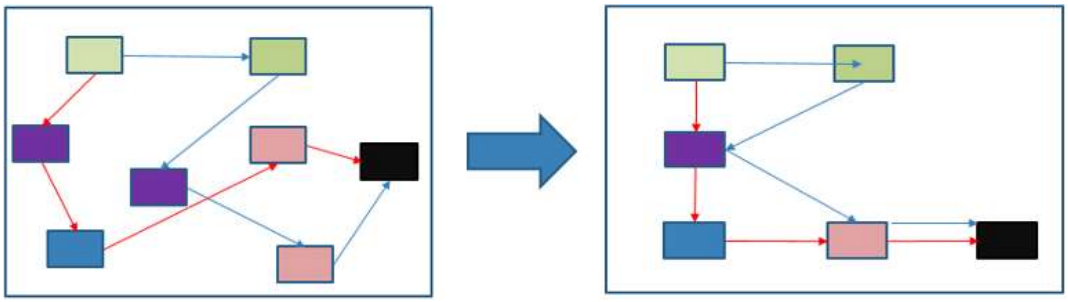

Ilustración 15: Refactoring

A continuación, se incluye una lista de algunos de los métodos que ofrece realizar el IDE Eclipse pulsando solamente un botón:

Método extraer (Extract Method): Si hay fragmentos de código que pueden ser agrupados en nuevo método que contenga este código, debe utilizarse el método extraer.

Método mover (Move method): Crear un método con un código similar en la clase en la que el método es más utilizada.

Clase extraer (Extract Class): Cuando una clase realiza el trabajo de dos, se debe en su lugar crear una nueva clase y colocar los campos y métodos responsables de la función relevante.

Inline Method: Si el cuerpo de un método es más obvio que el método en sí, reemplazar las llamadas al método con su contenido y borrar el método original.

Extract variable: Cuando se tiene una expresión que es difícil de entender, colocar el resultado de la expresión o sus partes en variables separadas que sean más entendibles.

Rename method: Cuando el nombre de un método no explica o no tiene relación con lo que el método realiza, se ha de renombrar el método.

Remove Parameter: Si un parámetro no es usado en código de un método, se ha de borrar el parámetro no usado.

Remove Parameter: Si un parámetro no es usado en código de un método, se ha de borrar el parámetro no usado.

12.3 Principio de SOLID

Los principios SOLID son una agrupación de principios de diseño de los que se partirá como base y en los que radicará gran parte de los conceptos específicos de arquitectura de software.

12.3.1 Principio de responsabilidad única

El principio radica en el hecho de que una clase sólo debería tener una y solo una única causa por la cual puede ser modificada. Es importante destacar que cuando se habla de “una única responsabilidad”, se entiende como “una única razón para cambiar”. Así entonces, si una clase tiene dos responsabilidades, entonces asume dos motivos por los cuales puede ser modificada.

12.3.2 Principio de abierto a extensión, cerrado a modificación

En el segundo de los principios SOLID, se afirmaba que: “Una clase debe estar abierta a extensiones, pero cerrada a las modificaciones”. Dicho de otra manera, OCP argumenta que deberíamos diseñar clases que nunca cambien, y que cuando un requisito cambie, lo que debemos hacer es extender el comportamiento de dichas clases añadiendo código, no modificando el existente.

Las clases que cumplen con abierto a extensión y cerradas a modificación tienen dos características:

- ✓ Son abiertas para la extensión; es decir, que la lógica o el comportamiento de esas clases puede ser extendida en nuevas clases.
- ✓ Son cerradas para la modificación, y por tanto el código fuente de dichas clases debería permanecer inalterado.

12.3.3 Principio de substitución de Liskov

Este principio se resume muy bien con la conocida frase: “las funciones que usen objetos de una determinada clase padre, deben ser capaces de usar instancias de clases que hereden de esta sin verse alteradas”. En otras palabras, en lo que se basa este principio es en el hecho de que los contratos que definamos para nuestras interfaces o clases abstractas tienen que ser fuertes y consistentes.

12.3.4 Principio de segregación de interfaces

El principio de Segregación de Interfaces, trata sobre las desventajas de las interfaces "pesadas", y guarda una estrecha relación con el nivel de cohesión de las aplicaciones. Básicamente lo que nos quiere decir este principio es que "las clases que implementen una interfaz o una clase abstracta no deberían estar obligadas a utilizar partes que no van a utilizar". La forma de solucionar el problema consiste en segregar las operaciones en pequeñas interfaces. Una interfaz es un contrato que debe cumplir una clase, y tales contratos deben ser específicos, no genéricos; esto nos proporcionará una forma más ágil de definir una única responsabilidad por interfaz - de otra forma, violaríamos además el Principio de Responsabilidad Única.

12.3.5 Principio de inversión de dependencias

El *Dependency Inversion Principle* es el último de los principios SOLID.

Básicamente lo que nos dice este principio es que:

- ✓ Las clases de alto nivel no verían depender de las clases de bajo nivel.
  Ambas deberían depender de las abstracciones.

- ✓ Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

12.4 Cambios realizados

En este apartado voy a listar todos los tipos de refactors realizados y luego en cada sección explicare en que consistió y un ejemplo de código.

Los refactors realizados que se van a continuación no están organizados según JCC ni el principio SOLID, ya que hay varios de ellos que se realizan los cambios para cumplir el JCC y el principio SOLID, por ello se van a comentar todos los tipos de cambios que se han realizado (pueden haberse repetido en varias partes del código, pero para no repetir solo se comenta una vez):

- Reorganizar los paquetes
- Relocalizar las excepciones
- Creación de un Log
- Deserialización con una sola configuración
- Consistencia entre excepciones
- Borrado de métodos innecesarios
- Extracción de métodos a la clase correspondiente
- Simplificación de los métodos
- Evitar el uso de la instrucción instaceof
- Cambio de nombres de fichero de mapeo
- Separar todo lo relacionado con las estadísticas

Además de los tipos de cambios indicados también se han arreglado los 303 avisos que nos daba Eclipse sobre los “Warnings” que tiene el código (variables que se han usado, imports no utilizados, etc.) al final de este capítulo el código ya tenía 0 warnings.

12.4.1 Reorganizar los paquetes

Los paquetes en Java (packages) son la forma en la que Java nos permite agrupar de alguna manera lógica los componentes de nuestra aplicación que estén relacionados entre sí.

Los paquetes permiten poner en su interior: clases, interfaces, archivos de texto, entre otros. De este modo, los paquetes en Java ayudan a darle una buena organización a la aplicación ya que permiten modularizar o categorizar las diferentes estructuras que componen nuestro software.

Los paquetes en Java, a parte del orden que nos permite darle a la aplicación., también nos brindan un nivel adicional de seguridad para nuestras clases, métodos o interfaces, dado que usando modificadores de acceso(usar public, private o protected, para indicar quién puede acceder a esa clase o método) [53] se puede limitar el acceso a ciertas partes del código.

Objetivo:

El objetivo de reorganizar los paquetes, es que el código esté más organizado. Así, si en un futuro, alguien buscará alguna funcionalidad, clase, pues lo tenga fácil simplemente siguiendo la ruta de los paquetes, ya que son descriptivos y se entiende lo que contiene.

Ejemplo:

La estructura inicial de los paquetes es la indica en la figura siguientes:

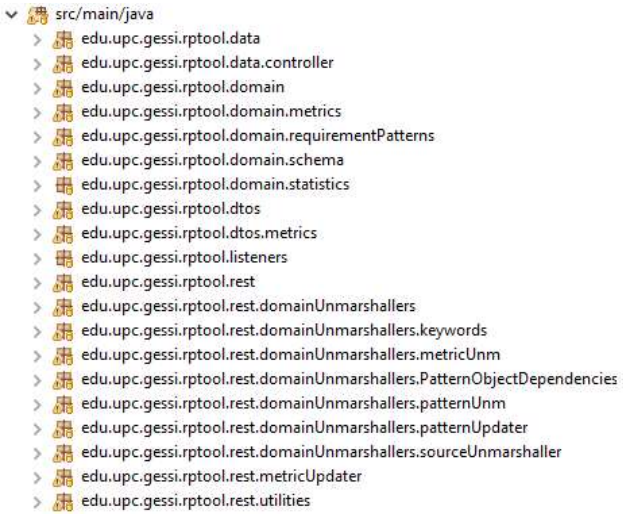

Ilustración 16: Antigua organización de paquetes

El primer paso que se ha tomado es quitar todas las mayúsculas de los nombres ya que JCC indica que los nombres de los paquetes tienen que ser SIEMPRE en minúscula.

El siguiente paso es cambiar los nombres para que sean más descriptivos, por ejemplo el package edu.upc.gessi.rptool.domain.requirementPatterns (que contiene todo lo relacionado con patrones) se ha hecho un refactor a su nombre y ahora es: edu.upc.gessi.rptool.domain.patternelements, dado que contiene elementos relacionados con los patrones y no solamente patrones de requisitos.

Al final del proceso de reorganización de los paquetes, la nueva estructura de paquetes es la siguiente:

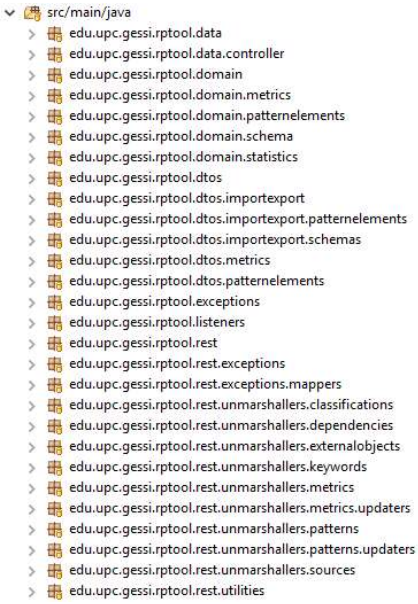

Ilustración 17: Nueva organización de paquetes

12.4.2 Relocalizar las excepciones

En Java los errores en tiempo de ejecución (cuando se está ejecutando el programa) se denominan excepciones, y esto ocurre cuando se produce un error en alguna de las

instrucciones del programa, como por ejemplo cuando se hace una división entre cero, cuando un objeto es 'null' y no puede serlo, cuando no se abre correctamente un fichero, etc. Cuando se produce una excepción se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa.

Objetivo:

El objetivo de relocalizar las excepciones es que siempre estén en un solo paquete lo más genérico posible y donde el nombre indique que las excepciones están localizadas ahí.

Ejemplo:

Originalmente las excepciones relacionadas con la lógica del software, estaban localizadas en mismo paquete que el resto de las clases como se puede ver en la Ilustración 18.

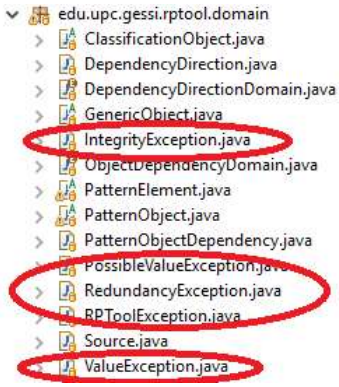

Ilustración 18: Excepciones de dominio

Para organizar mejor se ha procedido a crear un nuevo package edu.upc.gessi.rptool.exceptions como se puede ver en la Ilustración 19, así ahora se pueden ir añadiendo todas las excepciones que tengan que ver con el rptool en el mismo sitio.

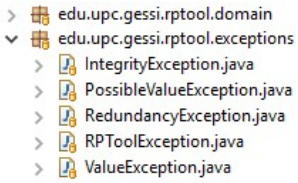

Ilustración 19: Excepciones de dominio organizadas

12.4.3 Creación de un Log

El uso de logging o de bitácoras dentro del contexto de desarrollo de aplicaciones constituye insertar sentencias dentro de la aplicación, que proporcionan un tipo de información de salida que es útil para el desarrollador. Un ejemplo de logging son las sentencias de rastreo o de seguimiento que colocamos en una aplicación para asegurarnos de que esta haya pasado por los flujos adecuados. Esto normalmente es realizado usando "System.out.println".

El problema con este tipo de sentencias, es que algunas veces olvidamos quitarlas después de que han servido a su propósito específico, y terminamos con una aplicación que genera cientos o miles de salidas innecesarias a la consola. Otro problema es que todas las llamadas tienen el mismo significado, es decir, no podemos saber exactamente cuál sentencia representa una salida de debug (que seguramente olvidamos quitar), cuál representa una salida de información

útil, y cuál representa un error que se ha generado; las tres se ven exactamente igual para nosotros.

Para ello se debe de crear un log que puedas indicar la prioridad de ese mensaje, y por donde quieres “imprimir” ese log.

Objetivo:

El objetivo de crear un log único es que se pueda llamar desde cualquier parte del código y solamente se tenga que configurar una única vez. Así si se le indica que lo escriba en un único fichero, se escribirá en ese fichero, además se indicará desde qué clase se ha llamado y qué línea de código ha generado ese mensaje en concreto.

Ejemplo:

Originalmente el software no tenía un Log, con lo cual cuando se quería imprimir algo se tenía que utilizar System.out.printf(“”) y llamando a esa función se imprimía por consola y no nos permitía desviar el log a un fichero.

Ahora se ha implementado una clase que se puede ver en código de la Ilustración 20

```
/**
 * Class responsible of logging
 */
public abstract class Log {

	public static Logger LOGGER;

	private Log() {
	}

	private static void initLogger() {
		Log.LOGGER = Logger.getLogger(Log.class.getName());
		Log.LOGGER.setLevel(Level.ALL);
		ConsoleHandler handler = new ConsoleHandler();
		handler.setLevel(Level.ALL);
		Log.LOGGER.addHandler(handler);
	}


	/*	PRINTS BY THE GIVEN LEVEL	*/
	public static void logByLevel(Level l, String s) {
		if (Log.LOGGER == null) {
			initLogger();
		}
		LOGGER.log(l, s);
	}

	/*	PRINTS AT DEBUG LEVEL		*/
	public static void printDebug(String message) {
		Log.logByLevel(Level.FINEST, message);
	}

	// PRINTS DIFERENT LEVELS
	// ...
}
```

Ilustración 20: Nueva clase Log

La nueva clase está en el paquete edu.upc.gessi.rptool.rest.utilities se llama Log.java, para guardar cualquier información solamente hace falta llamar a Log.printDebug(“”) y con eso se guardara mensaje como debug en el fichero/consola que se le indique en la configuración el proyecto. En total se han borrado más de 105 llamadas a System.out.println(), que servían para compobar valores de las variables.

12.4.4 Deserialización con una sola configuración

Deserialización consiste en pasar un JSON a un objeto Java. Cuando un cliente envía una cadena de caracteres en formato JSON esta llega como una corriente de bytes (Stream of bytes), entonces, para que el programa pueda manipular dicho JSON, hay que usar una librería (Jackson) que realiza el proceso de Deserialización (pasar un stream a un objeto) y luego se guardará en la base de datos (ver Ilustración 21).

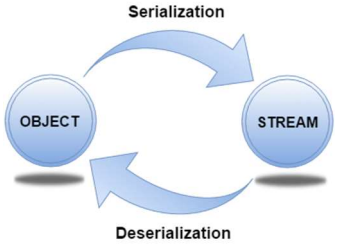

Ilustración 21: Serialización – Deserialización

Inicialmente en el proyecto cada vez que se quería deserializar un JSON se creaba una nueva instancia [54] de ObjectMapper (como se puede ver en la Ilustración 22) y se configuraba indicando que fallara si había algún campo desconocido (es decir, sino existía el campo dentro del objeto que se iba a crear). La solución funcionaba, pero era algo ineficiente ya que generaba muchas instancias innecesarias del ObjectMapper.

```
ObjectMapper mapper = new ObjectMapper();

mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, true);

unnmParent = mapper.readValue(metricJson, RequirementPatternUnmarshaller.class);
```

Ilustración 22: ObjectMapper instance

Objetivo:

El objetivo es crear una clase única que se pueda reutilizar en todo el contexto del proyecto para poder realizar el proceso de serialización y deserialización sin tener que ir clase por clase creando instancias. Con la creación de la clase única también se evitaría tener que hacer muchos cambios en el código cuando se decida cambiar algo de la configuración (por ejemplo, que no falle cuando reciba algo desconocido).

Ejemplo:

Con el refactor ahora se ha creado una nueva clase que aplica el patrón Singleton [55] (que solamente puede tener una única instancia y se puede llamar todas las veces que se quiera). En la Ilustración 23Ilustración 17: Nueva organización de paquetes se puede ver el nuevo código para realizar la misma función que el anterior.

```
unnmParent = Deserializer.deserialize(metricJson, RequirementPatternUnmarshaller.class);
```

Ilustración 23: Nueva configuración de ObjectMapper

La Ilustración 24 muestra la nueva clase Deserializer.

```
public class Deserializer {
	private static ObjectMapper mapper;
	private static void initMapper() {
	mapper = new ObjectMapper();
	mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
true);
	}

	public static <T> T deserialize(String json, Class<T> Unmarshall)
			throws JsonParseException, JsonMappingException,
			IOException
	{
		if (mapper == null) {
			initMapper();
		}
		return mapper.readValue(json, Unmarshall);
	}

	public static <T> T convertObject(Object obj, Class<T> unmarshall) {
		if (mapper == null) {
			initMapper();
		}
		return mapper.convertValue(obj, unmarshall);
	}

}
```

Ilustración 24: Deserializer

Este cambio afectaba a 31 clases diferentes, en algunas solamente se pasaba como referencia el ObjectMapper y otras las instanciaba. En total, se inicializaba este ObjectMapper 26 veces con la misma configuración.

12.4.5 Consistencia entre excepciones

En la sección 12.4.2 Relocalizar las excepciones se ha descrito el uso de excepciones en Java. En esta sección se hablará de cómo tratar las excepciones para que den una información consistente. Con eso quiere decir que, por ejemplo, en caso de los servicios web de búsqueda en el catálogo de patrones siempre devuelva una estructura de mensaje tipo "OBJECT [IDENTIFICADOR_OBJECT] not found" y que no vaya cambiando ese formato entre cada tipo de objeto a buscar. Por ejemplo, si se busca un Patrón devuelva "RequirementPattern [35231] not found", pero si busco un Form me devuelva "Error 404: not found ".

Objetivo:

El objetivo es que cada vez que se llame a los servicios web para buscar algún objeto del catálogo, se trate el caso de que no se encuentre el objeto. Además, en caso de que se quiera modificar el mensaje de que el objeto buscado no se encuentra en un futuro, no se tenga que buscar en todas las líneas del código para ver donde se genera dicho mensaje.

Ejemplo:

Inicialmente en el proyecto cada vez que se accedía a la base de datos desde el mismo fichero se llamaba y se hacía la comprobación conforme si existía algún resultado de la consulta o no. Como se puede ver en la Ilustración 25, no se reutilizaba el código para el tratamiento de error.

```
@GET
@Path("{id}")
@Produces({ MediaType.APPLICATION_JSON })
public RequirementPatternDTO getPattern(@PathParam("id") long id) {
    RequirementPattern rp = PatternDataController.getPattern(id);
    if (rp == null)
        throw new NotFoundException("Requirement Pattern [" + id
                + "] not found");
    return new RequirementPatternDTO(rp);
}

@GET
@Path("{patternId}/versions")
@Produces({ MediaType.APPLICATION_JSON })
public Set<RequirementPatternVersionReducedDTO> getPatternVersions(
        @PathParam("patternId") long patternId) {
    RequirementPattern rp = PatternDataController.getPattern(patternId);
    if (rp == null)
        throw new NotFoundException("Requirement Pattern [" + patternId
                + "] not found");
    SortedSet<RequirementPatternVersionReducedDTO> rpvSet = new TreeSet<RequirementPatternVersionReducedDTO>();
    for (RequirementPatternVersion rpv : rp.getMyVersions()) {
        rpvSet.add(new RequirementPatternVersionReducedDTO(rpv));
    }
    return rpvSet;
}
```

Ilustración 25: Acceso Inicialmente

Lo que he realizado en todas las clases que se llama a los controladores es que el acceso se realice mediante solamente una función retrieveXXXXXX, donde XXXXXX puede ser cualquier objeto de la base de datos, donde XXXXXX puede ser cualquier clase guardado en la base de datos. Este cambio se ha realizado en 8 clases diferentes, se han cambiado 60 llamadas que se hacían a la Base de datos para obtener algún elemento que ahora se hacen mediante una llamada que controla que exista lo buscado y si no se devuelve un error cambiando.

En la Ilustración 26 se puede ver la nueva forma en que se controla el error.

```
@GET
@Path("{id}")
@Produces({ MediaType.APPLICATION_JSON })
public RequirementPatternDTO getPattern(@PathParam("id") long id) {
    RequirementPattern rp = retrieveRequirementPattern(id);
    return new RequirementPatternDTO(rp);
}

@GET
@Path("{patternId}/versions")
@Produces({ MediaType.APPLICATION_JSON })
public Set<RequirementPatternVersionReducedDTO> getPatternVersions(@PathParam("patternId") long patternId) {
    RequirementPattern rp = retrieveRequirementPattern(patternId);
    SortedSet<RequirementPatternVersionReducedDTO> rpvSet = new TreeSet<RequirementPatternVersionReducedDTO>();
    for (RequirementPatternVersion rpv : rp.getMyVersions()) {
        rpvSet.add(new RequirementPatternVersionReducedDTO(rpv));
    }
    return rpvSet;
}

/** JAVADOC COMMENTS */
private RequirementPattern retrieveRequirementPattern(long patternId) {
    RequirementPattern m = PatternDataController.getPattern(patternId);
    if (m == null)
        throw new NotFoundException("RequirementPattern [" + patternId + "] not found");
    return m;
}
```

Ilustración 26: Acceso Actualmente

12.4.6 Borrado de métodos innecesarios

Muchas veces cuando se está desarrollando nuevo código, el desarrollador crea métodos para poder realizar comprobaciones para ver si lo que está implementando funciona bien o no. Una vez terminado el desarrollo del código estos métodos continúan en el código a pesar de que no se usan en el producto final.

Objetivo:

Borrar todos los métodos que no tengan importancia en la funcionalidad del software.

Ejemplo:

En varias clases hay métodos creados solamente para mirar que valores llegan a la hora realizar una operación cono se puede ver en la Ilustración 27.

```
public class Patterns {

    private void myPrintForm(RequirementForm rf)
    {
        System.out.println(rf.getName());
        System.out.println(rf.getMyFixedPart().getPatternText());
        System.out.println("pos " + rf.getRank());
        System.out.println("tiene " + rf.getMyExtendedParts().size() + " extended parts");
        System.out.println("++++");
    }
    private void myPrintVersion(RequirementPatternVersion rpv)
    {

        System.out.println(rpv.getAuthor());
        System.out.println("forms");
        for (RequirementForm rf: rpv.getForms())
        {
            myPrintForm(rf);
        }
        System.out.println(".....");
    }
    private void myPrintRP(RequirementPattern rp)
    {
        System.out.println("pattern - > " + rp.getName());
        System.out.println("versions");
        for (RequirementPatternVersion rpv: rp.getMyVersions())
        {
            myPrintVersion(rpv);
            System.out.println("-----");
        }
    }
```

Ilustración 27: Métodos sobrantes

Lo que hay que realizar es borrarlo ya que no aporta nada a nivel de funcionalidad y en lugar de imprimir los valores por pantalla se debe de utilizar el Debugger que trae el IDE Eclipse.

12.4.7 Extracción de métodos a la clase correspondiente

En el código inicial del proyecto había un par de métodos que estaban en clases en las que no debían. Esto es una práctica que perjudica a la cohesión de las clases, ya que cada clase debe de tener su propia responsabilidad de gestionar sus atributos.

Objetivo:

Extraer todos los métodos de una clase B que manipulen o comprueben atributos de clase A y llevar esos métodos a la clase A.

Ejemplo:

Como se puede ver en la Ilustración 28 dentro de la clase SchemaUnmarshaller hay un método que llama getRootFromUnmarshaller(…), donde lo único que se hace es obtener valores de otra clase llamada RootClassifier y realizar comprobaciones sobre ello, con ello se viola el principio de 5612.3.1 Principio de responsabilidad única.

```
public class SchemaUnmarshaller {

    //SOME CODE...

    private RootClassifier getRootFromUnmarshaller(RootClassifierUnmarshaller aux, boolean isRoot) throws Exception {
        RootClassifier rc = aux.getRootClassifier();

        try {
            rc.setSources(IdToDomainObject.getSources((aux.getIdsSources())));
        } catch (NotFoundException e) {
            throw new SemanticallyIncorrectException("invalid source id");
        }
        Set<RootClassifierUnmarshaller> s = aux.getUnmInternalClassifiers();
        session.save(rc);
        boolean[] positions = new boolean[s.size()]; // inicializado a false
        for (RootClassifierUnmarshaller tmp : s) {

            int myPos = tmp.getPos();
            if (myPos < 0 || myPos >= s.size()) {
                throw new SemanticallyIncorrectException("Incorrect pos value");
            } else
                positions[myPos] = true;
            rc.addInternalClassifier(getRootFromUnmarshaller(tmp, false));
            session.update(rc);
        }
        for (boolean b : positions) {
            if (!b)
                throw new SemanticallyIncorrectException("Incorrect pos value");
        }
        int type = rc.getType();
        // estamos en un internal classifier, no puede ser root
        if (isRoot && (type == 3 || type == 1 || type == 2)) {
            throw new SemanticallyIncorrectException("Incorrect type value");
        }

        // MORE CODE...

        return rc;
    }

    // MORE CODE...
```

Ilustración 28: Métodos en lugar incorrecto

Lo que se ha realizado al final en este caso es quitar el método de esa clase y llevarlo a la clase RootClassifierUnmarshaller y así esa clase tendrá la responsabilidad de realizar las comprobaciones pertinentes.

12.4.8 Simplificación de los métodos

Uno de los indicios de mala calidad de código es crear métodos muy largos. En este tipo de métodos, al intentar realizar un cambio hay que modificar por varias partes. En estos casos la mejor solución es descomponer los métodos en otros más pequeños con menos líneas de código. Con probabilidad habrá métodos más pequeños que podrán ser reutilizados en diferentes partes del mismo código.

Objetivo:

Crear métodos más cortos, en cuanto a líneas de código, para así luego poder reutilizarlos en diferentes partes del proyecto.

Ejemplo:

Como ejemplo se puede utilizar él mismo que el de 12.4.7 Extracción de métodos a la clase correspondiente, el mismo código de Ilustración 28 , se puede refactorizar creados métodos más cortos donde podamos reutilizar luego esos mismos métodos, y en caso de alguna de las subclases quieran sobrescribir [56] podrán realizarlo sin ningún problema. En la Ilustración 29 se puede ver el nuevo código que he sustituido para que él código tengo más calidad.

```
public RootClassifier buildAndSave(boolean isRoot, Session session)
		throws SemanticallyIncorrectException, RedundancyException,
		IntegrityException
{
	setSources(session);
	checkPosAndAddRootClassifier(session);
	checkType(isRoot);
	getPatternAndUpdate(session);
	return rc;
}

protected void setSources(Session session) throws
SemanticallyIncorrectException {
	try {
		rc.setSources(IdToDomainObject.getSources((sources)));
		session.save(rc);
	} catch (NotFoundException e) {
		throw new SemanticallyIncorrectException("invalid source id");
	}
}

protected void checkPosAndAddRootClassifier(Session session)
	throws SemanticallyIncorrectException,
IntegrityException,RedundancyException {
	Set<RootClassifierUnmarshaller> s = this.internalClassifiers;
	checkPosInRootClassifierSetAndAdd(s, session);
}

protected void checkType(boolean isRoot) throws SemanticallyIncorrectException
{
	checkType(getIdsReqPatterns(), getUnmInternalClassifiers(), isRoot);
}

protected void getPatternAndUpdate(Session session) throws
	SemanticallyIncorrectException, RedundancyException
{
	if (rc.getNpatterns() != this.getIdsReqPatterns().size()) {
	throw new SemanticallyIncorrectException("npatterns != patterns size");
	}
	Set<RequirementPattern> auxReqPatterns;
	try {
	auxReqPatterns =
IdToDomainObject.getReqPatterns(this.getIdsReqPatterns());
	} catch (NotFoundException e) {
		throw new SemanticallyIncorrectException("invalid req pattern
id");
	}
	for (RequirementPattern rp : auxReqPatterns) {
		rp.addBasicClassifier(rc);
		session.update(rp);
	}
}
```

Ilustración 29: Nuevo código que sustituye

12.4.9 Evitar el uso de la instrucción instaceof

El operador *instaceOf* nos permite comparar si una instancia es de una clase o no. Normalmente se puede utilizar este método para saber exactamente de qué tipo es una instancia. En el código inicial se abusaba mucho de este operador. Eso indicaba que el diseño no era muy bueno, dado que la gran mayoría de las veces es muy fácil cambiar este operador usando polimorfismo para mejorar la reutilización del código.

Objetivo:

El objetivo es evitar el uso de este operador ya que es un indicio de que el diseño es muy malo y no se ha utilizado el polimorfismo para resolver el problema dado.

Ejemplo:

Una de las formas para evitar usar este operador en nuestro código es usar un método abstracto que se puede observar en la Ilustración 30, el método getType() es abstracto con lo cual cuando se extienda esta clase se deberá indicar exactamente qué tipo de métrica es esta clase

```
public abstract class Metric extends MetricObject {

	public abstract TYPE getType();

	public static enum TYPE {
		DOMAIN("domain"),
		FLOAT("float"),
		INTEGER("integer"),
		STRING("string"),
		TIME("time"),
		SET("set");

		private final String name;

		private TYPE(String s) {
			name = s;
		}

		public boolean equalsName(String otherName) {
			return name.equals(otherName);
		}

		@Override
		public String toString() {
			return this.name;
		}

	}
	//SOME CODE
}
```

Ilustración 30: Nuevo código en Metric

Un ejemplo de cómo se debería extender esta clase es la Ilustración 31, donde se puede ver que la clase FloatMetric, realiza un extends de SimpleMetric(que a su vez hereda de Metric), con lo cual está obligado a realizar la anotación de @Override para indicar que va a sobrescribir un método de alguna clase padre, a la vez el nombre del método tiene que ser exactamente como se indica en la clase Metric.

```
public class FloatMetric extends SimpleMetric {

	@Override
	public TYPE getType() {
		return TYPE.FLOAT;
	}
	//SOME CODE
}
```

Ilustración 31:Ejemplo implementación nuevo método

Antes de realizar el refactor en la clase Metrics.java se utilizaba varias veces el operador instaceOf (Ilustración 32) que ahora se ha cambiado utilizando el método getType como se puede ver en la Ilustración 33.

```
82      @GET
83      @Path("{id}")
84      @Produces({ MediaType.APPLICATION_JSON })
85      public MetricDTO getMetric(@PathParam("id") long id) {
86          Metric m = retrieveMetric(id);
87
88          MetricDTO metricDTO;
89          if (m instanceof DomainMetric) {
90              metricDTO = new DomainMetricDTO((DomainMetric) m);
91          } else if (m instanceof FloatMetric) {
92              metricDTO = new FloatMetricDTO((FloatMetric) m);
93          } else if (m instanceof IntegerMetric) {
94              metricDTO = new IntegerMetricDTO((IntegerMetric) m);
95          } else if (m instanceof SetMetric) {
96              metricDTO = new SetMetricDTO((SetMetric) m);
97          } else if (m instanceof StringMetric) {
98              metricDTO = new StringMetricDTO((StringMetric) m);
99          } else if (m instanceof TimePointMetric) {
100             metricDTO = new TimePointMetricDTO((TimePointMetric) m);
101         } else {
102
103             metricDTO = new MetricDTO(m);
104         }
105
106         return metricDTO;
107     }
```

Ilustración 32: uso de instaceOf antes de refactor

```
83      @GET
84      @Path("{id}")
85      @Produces({ MediaType.APPLICATION_JSON })
86      public MetricDTO getMetric(@PathParam("id") long id) {
87          Metric m = retrieveMetric(id);
88
89          MetricDTO metricDTO;
90          if (m.getType() == TYPE.DOMAIN) {
91              metricDTO = new DomainMetricDTO((DomainMetric) m);
92          } else if (m.getType() == TYPE.FLOAT) {
93              metricDTO = new FloatMetricDTO((FloatMetric) m);
94          } else if (m.getType() == TYPE.INTEGER) {
95              metricDTO = new IntegerMetricDTO((IntegerMetric) m);
96          } else if (m.getType() == TYPE.SET) {
97              metricDTO = new SetMetricDTO((SetMetric) m);
98          } else if (m.getType() == TYPE.STRING) {
99              metricDTO = new StringMetricDTO((StringMetric) m);
100         } else if (m.getType() == TYPE.TIME) {
101             metricDTO = new TimePointMetricDTO((TimePointMetric) m);
102         } else {
103             metricDTO = new MetricDTO(m);
104         }
105
106         return metricDTO;
107     }
```

Ilustración 33: Nueva forma de comprobar el tipo

12.4.10 Cambio de nombres de fichero de mapeo

La librearía de Hibernate, se le tiene que indicar que fichero se utiliza para mapear, que al inicio del proyecto se llamaba Mapeo.hbm.xml, ahí se indicaban todos los mapeos de la base de datos a objetos se realizan, dado que PABRE va ser utilizado por proyectos europeos donde todo se realiza en inglés se ha decidido cambiar el nombre del fichero Mapeo.hbm.xml a Mapping.hbm.xml

Objetivo:

El objetivo de este cambio es conseguir que todo aquel que se mire el código entienda que realiza este fichero y no limitarlo solamente a hispano-hablantes.

12.4.11 Separar todo lo relacionado con las estadísticas

En el código inicial del proyecto, había varias clases que no se utilizaban. Se trataba de clases usadas en proyectos anteriores y que gestionaban el almacenado de estadísticas de uso de los patrones. Los servicios web implementados en PABRE-WS no deben gestionar datos sobre el uso de patrones, con lo cual se ha decidido separar todo el código de estadísticas para que luego se haga una revisión al completo y adaptar las clases para que funcionen como se espera.

Objetivo:

El objetivo de este cambio es tener el código más ordenado sin tener clases sueltas de por medio que no tengan ninguna utilidad. Para ello se creará un paquete separado donde se guardará todo para si en un futuro puede ser necesaria esta funcionalidad.

Ejemplo:

Para guardar estas clases se habló con la codirectora y se decidió separar todas esas clases en un nuevo Package de Java llamado “edu.upc.gessi.rptool.domain.statistics”, ahí ahora están todas las clases, sean del tipo que sean (Controller, clases de dominio, DTO, etc.)

13 Documentar la API

Tal como se ha indicado anteriormente uno de los objetivos de mi proyecto es mejorar la calidad de código, pero también incluye mejorar la documentación existente, para ello hay que mejorar la documentación de la API que está en la página web de Apiary.

Una API (Application Programming Interfaces) es una especificación formal sobre cómo un módulo de un software se comunica o interactúa con otro.

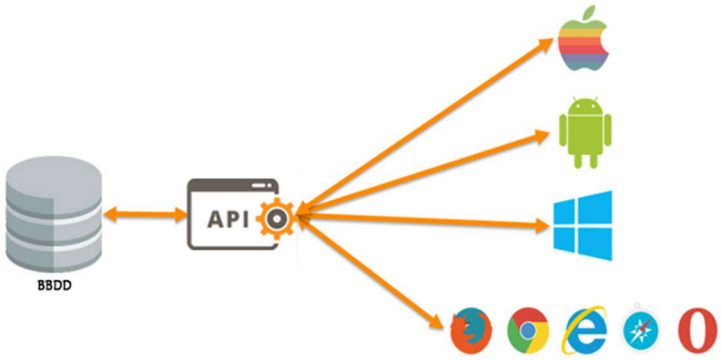

Ilustración 34: interacción de una API

La API de PABRE-WS está creada en Apiary. Esta página web nos permite indicar a los desarrolladores que van a utilizar nuestro servicio, como utilizar nuestro software, ya que hay que especificar que ruta se debe acceder, qué parámetros se necesitan para realizar alguna operación.

Realizar una buena documentación es muy importante para este proyecto, dado que hay dos proyectos europeos que usarán esta herramienta, para ello hay que redactar toda la documentación de la API en inglés, explicando que haca cada operación, que condiciones mínimas se tiene que cumplir para ello.

La Ilustración 34 se puede ver que una API hace de intermediario entre la base de datos (BBDD en la imagen) y los clientes (los iconos de la derecha de Apple, Android, Windows, etc.), con ello permite a los desarrolladores poder acceder a la base de datos desde cualquier plataforma.

En la Ilustración 35 se puede ver cómo era la documentación que se le daba a los desarrolladores para interaccionar con PABRE-WS.

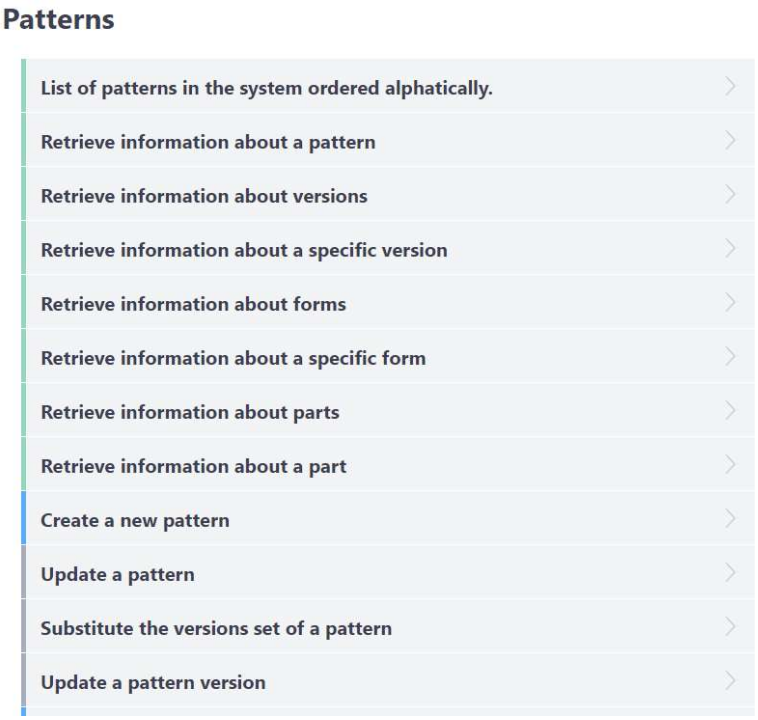

Ilustración 35: Documentación de la API al inicio

La redacción de la API consistió en documentar 23 operaciones relacionadas con Patrones, 10 operaciones relacionados con esquemas y clasificadores, 5 operaciones sobre dependencias, 5 operaciones sobre Sources, 5 operaciones de Keywords, 10 operaciones de métricas, 7 operaciones nuevas sobre funciones de costes que se implementarán más tarde en la memoria y por último 2 operaciones de importar y exportar todo el catalogo existente.

En la redacción que se realizó de cada operación, se incluyó los tipos de contenido siguientes:

1. Titulo
2. Descripción
3. Ruta de la operación
4. Tipo de solicitud
5. Los parámetros necesarios para la operación
6. Ejemplo de JSON a enviar
7. Ejemplo de JSON devuelto
8. Todos los posibles retornos con los errores

A continuación, se muestra un ejemplo de una de las operaciones, donde se muestra con números cada uno de los tipos de contenido indicados en la lista anterior.

En la Ilustración 36 se pueden los campos 1, 2, 3, 4 y 5. En la Ilustración 37 se puede ver un ejemplo de posible JSON que se podría enviar al servidor. En la Ilustración 38 se puede ver un ejemplo de la solicitud procesada de forma satisfactoria y otro ejemplo cuando la solicitud no ha ido bien.

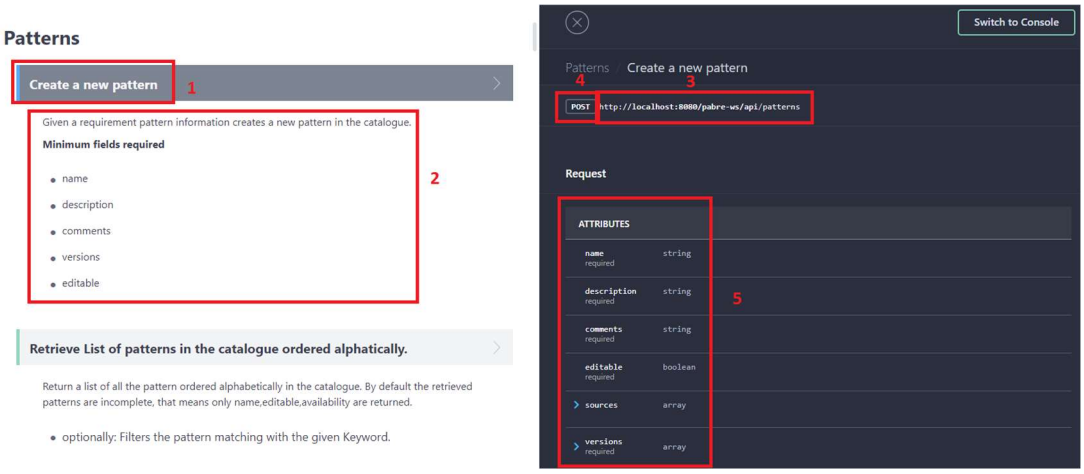

Ilustración 36: Apiary con título y descripción

Ilustración 37: Apiary con JSON de ejemplo

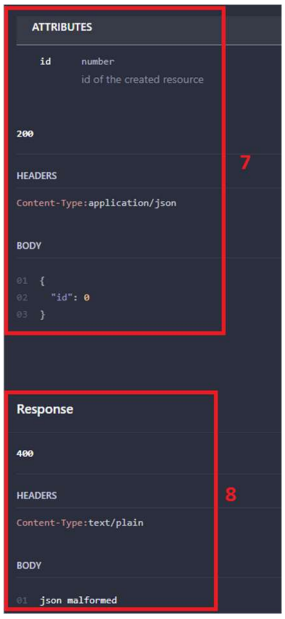

Ilustración 38: Apiary ejemplo de respuesta del servidor

Una vez se ha mostrado como se ha realizado la documentación de la API, se puede comparar la Ilustración 35 con la información inicial disponible a través de Apiary , donde no había ninguna descripción ni nada, con la Ilustración 39 donde ya está la API documentada.

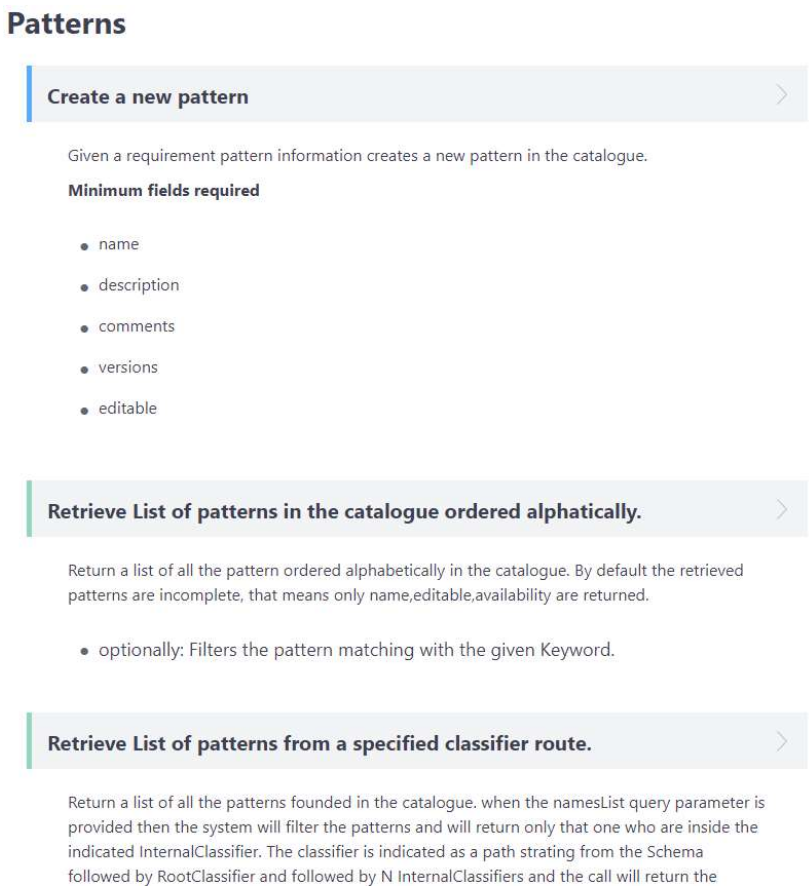

Ilustración 39: Documentación de la API al finalizar

14 Mejora de los Tests

Tal como se ha indicado anteriormente uno de los objetivos de mi proyecto es mejorar la calidad de código. En este capítulo primeramente, en la sección 14.1, voy a explicar cómo funciona Postman, la herramienta que se ha utilizado para realizar los tests. A continuación en la sección 14.2 voy a describir los cambios realizados en la organización de los tests. Y por último, en la sección 14.3, voy a mostrar como los cambios realizados pueden ayudar a mejorar la calidad de las pruebas.

14.1 Como funciona Postman

Postman funciona mediante colecciones (Collections), además para tener variables globales, se puede tener un entorno de desarrollo(Environment), las colecciones son un conjunto de carpetas donde se almacenan todos los tests. En la Ilustración 40 se pueden ver los diferentes elementos que tiene Postman.

1. Son las diferentes colecciones que tenemos importadas en el Postman.
2. Las carpetas internas para tener una mejor organización interna de una colección
3. Son los diferentes tests que hay dentro de la carpeta, en cada tests se indica con un poco si la solicitud es un GET, POST, PUT, DELETE, etc.
4. Pestaña actual abierta, donde se pueden abrir varios tests a la vez para poder modificarse
5. Tipo de operación que realiza el tests (puede ser GET, POST, PUT, DELETE, etc.)
6. La ruta que debe de acceder Postman para realizar la solicitud (baseUrl és una variable de entorno)
7. El entorno de desarrollo usado actualmente.

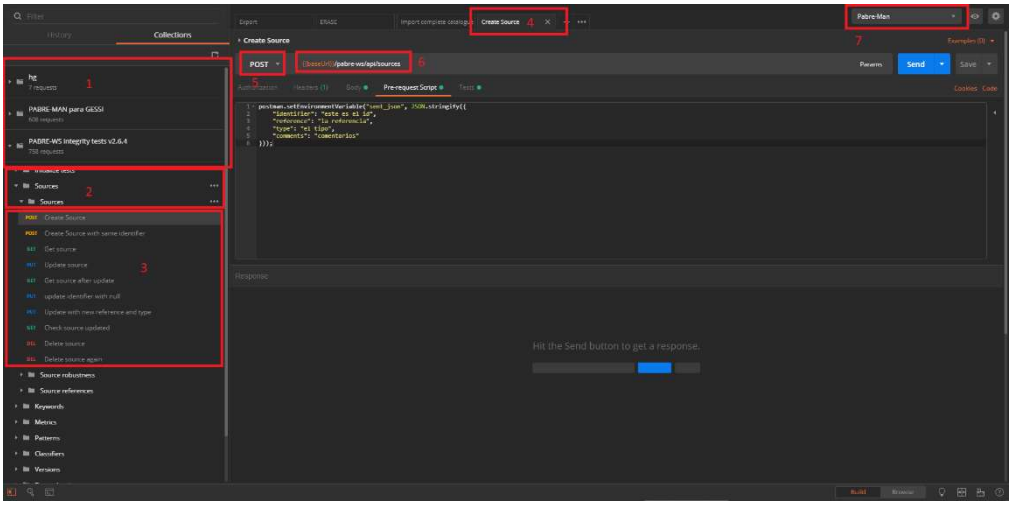

Ilustración 40: Pantalla Postman

Si clicamos en Body podemos acceder al contenido de la solicitud y se mostrará como la Ilustración 41 , en este caso se envía la variable de entorno *sent_json*, que anteriormente se ha definido en el apartado Pre-request Script.

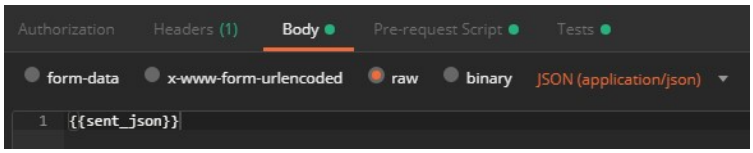

Ilustración 41: Body de la solicitud

En el apartado de Pre-request Script, se puede añadir código en lenguaje JavaScript como se puede ver en la Ilustración 42, se guarda un JSON dentro de la variable global del entorno de desarrollo *sent_json*.

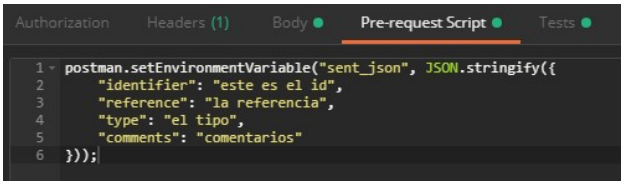

Ilustración 42: Pre-request Script

Si vamos al apartado de Tests, el código de ahí se ejecuta una vez realizada la solicitud y se haya recibido la respuesta por parte del servidor. Como se puede ver en la Ilustración 43 una vez se ha recibido la respuesta, la primera línea comprueba que la respuesta recibida ha devuelto el código 200, en la segunda línea se guarda en la variable jsonData la respuesta recibida convirtiéndola en JSON, en la línea 3 se comprueba que el id devuelto es un entero y por último se guarda ese id para los próximos tests.

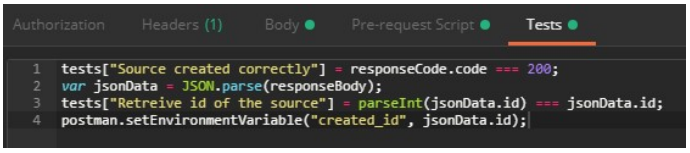

Ilustración 43: Tests

Por último, vamos a hablar de las variables de entorno, en las variables de entorno se puede guardar cualquier cosa, desde un objeto hasta un entero, como se puede ver en la Ilustración 44 hay varias variables ya guardadas para poder realizar diferente tipos de tests.

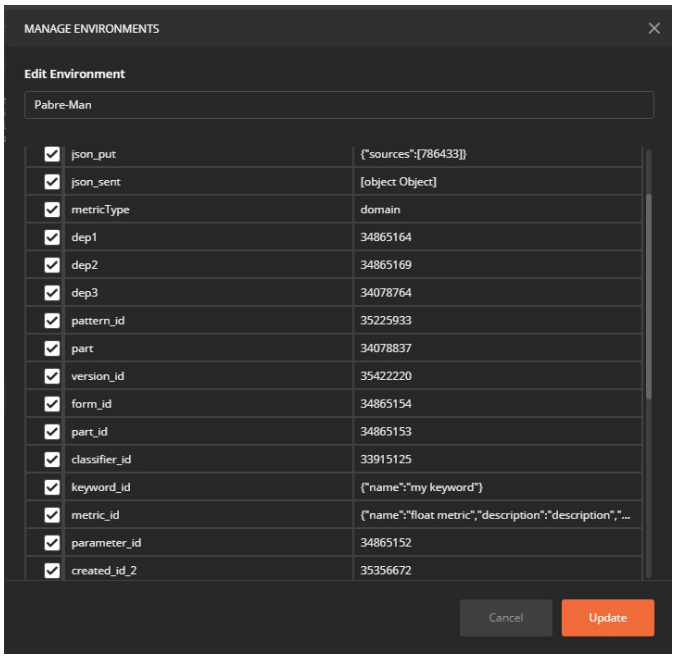

Ilustración 44: Manejo de variables

Una funcionalidad más importante de Postman, es el Collection Runner, que nos permite ejecutar todos los tests que hay en una collection siguiendo un orden. En la Ilustración 45 se puede ver la funcionalidad, donde se indica las pruebas que se han pasado y han fallado.

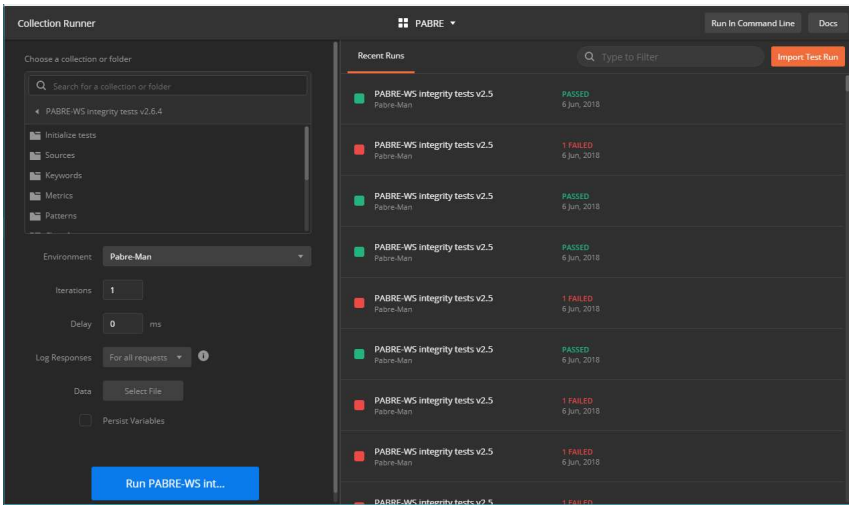

Ilustración 45: Collection runner

14.2 Cambios en la organización

En los tests se han realizado varios cambios, inicialmente había muchas carpetas de primer nivel, como se puede ver en Ilustración 46 donde había 41 carpetas, era muy difícil buscar los tests que queríamos modificar. En cambio, en la Ilustración 47 se puede ver la nueva organización donde hay carpetas con diferentes niveles.

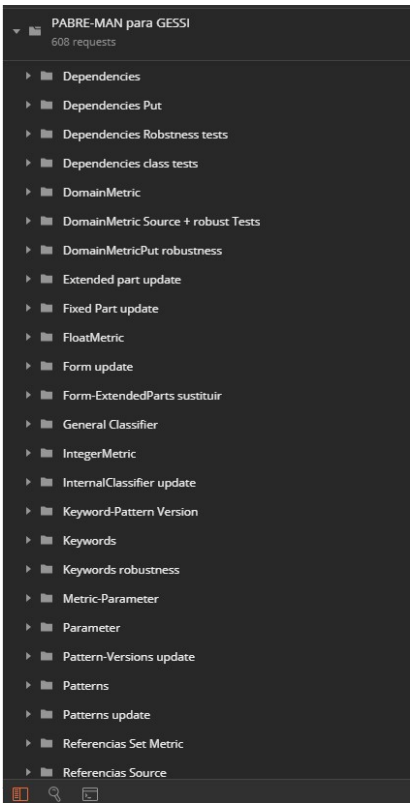

Ilustración 46: Organización de tests antes de reorganizar

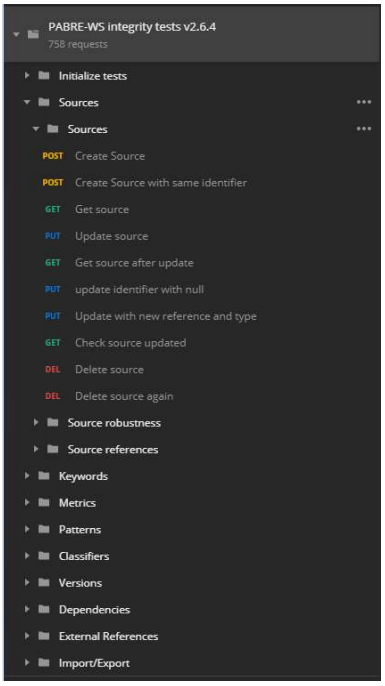

Ilustración 47: Organización de tests después de reorganizar

En la Ilustración 48 se puede ver el flujo que siguen las pruebas, que primero ejecutan el código que se encuentra dentro de Collection Prerequest, seguido de Prerequest de carpeta, Prerequest del test y después se ejecuta los tests de la colección, carpeta y test de la solicitud.

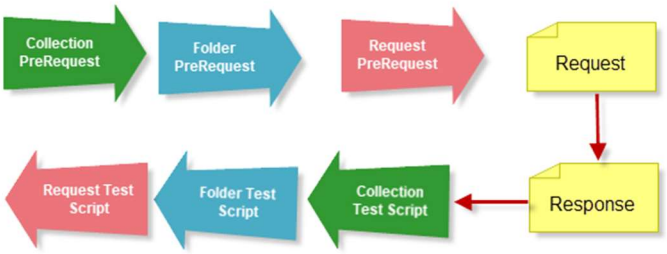

Ilustración 48: Flujo de las pruebas

14.3 Cambios para mejorar la calidad de las pruebas

Para mejorar la calidad de los tests, se han revisado todos los tests que había disponible, había muchos tests que simplemente no realizaban ninguna comprobación. Antes de mejorar todos los tests en total se realizaban 876 y una vez finalizado todo el TFG han quedado 1895 comprobación en todos los tests.

15 Cambiar la información almacenada de los patrones

Tal como se ha indicado anteriormente uno de los objetivos de mi proyecto es realizar los cambios requeridos en la estructura de los patrones de requisitos por los actores interesados en PABRE-WS.

En este capítulo se va a describir los cambios realizados. Al usar una metodología agile, si alguno de los proyectos europeos interesados en PABRE-WS necesitaba algún cambio a nivel de dominio del sistema, ese cambio se efectuaba en el siguiente Sprint dependiendo de la prioridad que del cambio.

Finalmente, se pidieron dos cambios a nivel de dominio de PABRE-WS:

- Quitar las dependencias en que estuvieran involucradas las métricas. En la sección 11.3 definir dependencias en PABRE pueden ser entre: SRP, SRP version, SRP Form, SRP Part, Parameter. Sin embargo, en la implementación de partida de PABRE-WS
- Crear funciones de coste para las versiones de patrones, eso quiere decir que cada SRP versión, puede tener más de una función de coste, almacenando el nombre de la función y un campo para guardar la fórmula de la función.

En la sección 15.1 se describe el cambio en las dependencias, en la sección 15.2 se describe el cambio de añadir funciones de coste a las versiones de los patrones de requisitos. En cada sección se explica inicialmente a nivel bastante abstracto el cambio sin entrar en detalle. A continuación, se incluyen 3 subsecciones donde primeramente se muestra el cambio a nivel de base de datos, después el cambio a nivel de código y por último el cambio en el mapeo con la base de datos.

15.1 Las Métricas no pueden tener dependencias

Debido a un malentendido en la implementación del anterior TFG, se realizó un cambio que no se debería de permitir, en este caso el cambio a realizar es quitar que una métrica pueda tener dependencia con otro de los objetos del sistema. Como se puede observar en la Ilustración 49 la clase Metric hereda de PatternElement, pero no debería de ser así, aunque tiene que tener los mismos campos no puede tener dependencia con ningún otro PatternObject.

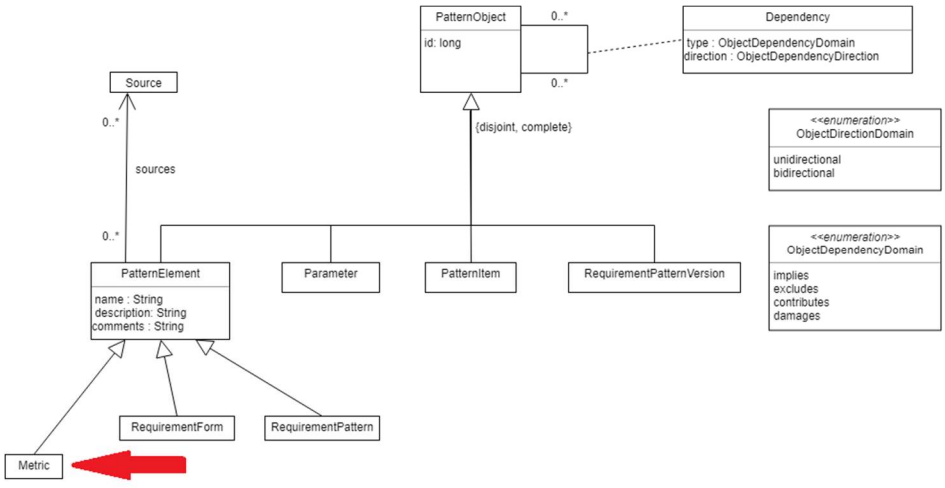

Ilustración 49: Sistema PABRE con Métricas

Para arreglar esto, se realizó una reunión con Cristina Palomares y Carme Quer, revisando como se debería de comportarse se llegó a la conclusión de que en lugar de heredar de PatternElement, se debe de crear una nueva clase MetricObject con los atributos de PatternElement y luego Metric los heredaba. De esta forma las métricas no tenían dependencias.

15.1.1 Implementación a nivel de base de datos

En primer lugar, se debe implementar una nueva tabla que represente MetricObject siguiendo una de las estrategias que Hibernate emplea para representar una relación entre una superclase y sus respectivas subclases. En particular Hibernate soporta diversas estrategias de mapeo de herencia, pero se ha decidido mantener la forma en la que se ha realizado en el proyecto para ofrecer una solución coherente. En particular se emplea la estrategia tabla por subclase.

Se procede a crear la nueva tabla METRIC_OBJECT de la siguiente forma:

```
CREATE TABLE PATTERNS.METRIC_OBJECT
(
	ID BIGINT PRIMARY KEY NOT NULL,
	NAME VARCHAR (255) NOT NULL,
	DESCRIPTION CLOB NOT NULL,
	COMMENTS CLOB NOT NULL
)
```

Se crea una tabla análoga a la combinación antigua de PatternElement y PatternObject. Las columnas se llaman igual que los atributos de clase. La columna ID es de tipo *bigint*, que corresponde al tipo *long* de Java. El resto de columnas son de tipo texto. La diferencia entre los tipos *varchar* y *clob* es que el primero está pensado para textos cortos mientras que el segundo para textos más largos. Ambos corresponden al tipo String en Java

El siguiente paso es modificar la METRIC. Es necesario eliminar la clave foránea hacia PATTERN_ELEMENT (tabla correspondiente a PatternElement) y reemplazarla por una nueva hacia METRIC_OBJECT.

```
ALTER TABLE PATTERNS.METRIC DROP CONSTRAINT FK8758E9B03AFA14DE;

ALTER TABLE PATTERNS.METRIC
ADD CONSTRAINT FK8758E9B03AFA14DE
FOREIGN KEY (ID)
REFERENCES PATTERNS.METRIC_OBJECT;
```

La primera sentencia realiza un *alter table* sobre METRIC para eliminar la clave foránea hacia PATTERN_ELEMENT. Mientras que la segunda declara una nueva clave foránea de METRIC hacia METRIC_OBJECT sobre la columna ID. El cliente DBeaver, para una determinada tabla, muestra sus claves foráneas, así como las otras posibles tablas que la referencian. De este modo se obtiene el nombre de las restricciones. En este caso, el nombre de la clave es un identificador que genera Hibernate en el momento de crear de cero el esquema de la base de datos y se ha decidido mantener.

```
INSERT INTO PATTERNS.METRIC_OBJECT (ID, NAME, DESCRIPTION, COMMENTS)
      SELECT ID, NAME, DESCRIPTION, COMMENTS
      FROM PATTERNS.PATTERN_ELEMENT
      WHERE ID IN (SELECT ID FROM PATTERNS.METRIC)
```

Como se pretende mantener los datos se ha realizado una migración de datos de las filas de la antigua PATTERN_ELEMENT a METRIC_OBJECT con la siguiente sentencia:

El Último paso a realizar sobre la BD es crear una tabla asociativa para relación de MetricObject con Source. La estrategia a seguir para un tipo de relación muchos a muchos es declarar dos columnas (una referenciando a un extremo de la relación y la otra al otro mediante claves foráneas) y hacer que este par sea la clave primaria de la tabla. El resultado es el siguiente código SQL:

```
CREATE TABLE PATTERNS.OBJECT_SOURCE_METRIC_OBJECT (
OBJECT_ID BIGINT NOT NULL,
SOURCE_ID BIGINT NOT NULL,
CONSTRAINT OBJECT_SOURCE_METRIC_OBJECT_PK PRIMARY KEY
(OBJECT_ID, SOURCE_ID),
CONSTRAINT OBJECT_SOURCE_METRIC_OBJECT_FK_SOURCE FOREIGN KEY
(SOURCE_ID) REFERENCES PATTERNS."SOURCE"(ID),
CONSTRAINT OBJECT_SOURCE_METRIC_OBJECT_FK_METRIC FOREIGN KEY
(OBJECT_ID) REFERENCES PATTERNS.METRIC_OBJECT(ID)
)
```

También se ha migrado la información de la relación con Source a esta nueva tabla.

15.1.2 Implementación a nivel de código

A nivel de código en primer lugar es necesario crear una nueva clase de dominio: MetricObject. En el código se omiten los setters y getters dado que son obvios:

```
public abstract class MetricObject {

	/*
	 * ATTRIBUTES
	 */
	private long id;
	private String name;
	private String description;
	private String comments;
	private Set<Source> sources;

	// MORE CODE...

}
```

Es una clase abstracta debido que no puede instanciar sin indicar el tipo de métrica que es.

Finalmente se declara en la clase Java del dominio que Metric es subclase de MetricObject en lugar de PatternElement. En Java se realiza mediante la palabra *extends* de la siguiente forma:

```
public abstract class Metric extends MetricObject {
// CODE...

}
```

15.1.3 Cambios en el Mapeo

Una vez implementadas los cambios a nivel de código, se procede a actualizar el fichero de mapeo.

Para indicar que la clase METRIC_OBJECT corresponde a la clase Java MetricObject, se tiene que crea un tag nuevo *class* donde se indica el nombre la ruta completa de la clase y como tabla el nombre de la tabla en la BD.

```
<class name="edu.upc.gessi.rptool.domain.MetricObject"
table="METRIC_OBJECT">
. . .
</class>
```

Dentro de ese tag, se tiene que indicar que el atributo Java *id* corresponde a la columna ID y que se generará con el algoritmo Hi-Lo (Es el que se ha usado en todo el ámbito del proyecto).

```
<id name="id" column="ID" type="long">
<generator class="hilo" />
</id>
```

A continuación, se indican los atributos Java correspondiente a cada campo de la tabla:

```
<property name="name" column="NAME" not-null="true" />
<property name="description" type="text" length="2000"
                    column="DESCRIPTION" not-null="true" />
<property name="comments" type="text" length="2000"
                    column="COMMENTS" not-null="true" />
```

Lo siguiente es indicar que el atributo sources de la clase Java es un conjunto mapeado en la tabla OBJECT_SOURCE_METRIC_OBJECT definido por las columnas OBJECT_ID y SOURCE_ID. El tag *key* corresponde a la columna que contiene el identificador de la instacia Java de METRIC_OBJECT mientras que el elemento many-to-many define el tipo de relación, así como la clase del otro extremo de la relación.

```
<set name="sources" table="OBJECT_SOURCE_METRIC_OBJECT"
       lazy="false">
       <key column="OBJECT_ID" />
       <many-to-many column="SOURCE_ID"
       class="edu.upc.gessi.rptool.domain.Source" />
</set>
```

Después se declara la subclase con el tag *joined-subclass* indicando así que se sigue la estrategia tabla por subclase. Donde se debe de indicar como name hay que indicar la ruta completa con los package y nombre de la clase y por último indicar el nombre de la tabla.

```
<joined-subclass name="edu.upc.gessi.rptool.domain.metrics.Metric"
table="METRIC" lazy="false">
    <key column="ID" />
    <property name="name" column="NAME" not-null="true" unique="true"/>
   <!–Here goes more subclasess of Metric -->
</joined-subclass>
```

Una vez realizado todos los cambios el sistema acaba teniendo el UML como el de la Ilustración 50, con eso hemos conseguido que las métricas no puedan tener ningún tipo de dependencia.

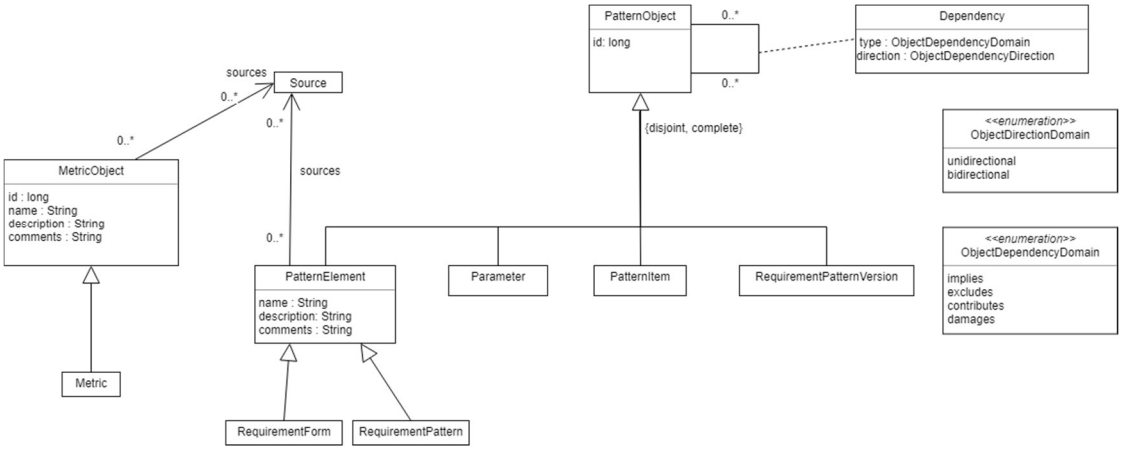

Ilustración 50: Esquema UML después de añadir MetricObject

15.2 Crear función de coste

Durante la realización de este proyecto, se dio el caso que en el proyecto europeo Q-rapids necesitaban que cada versión de un patrón tenía que guardar más información. Más en concreto tenía que guardar la función de coste de esa versión, y en un futuro se guardaría más información sobre las versiones.

Para ello Cristina Palomares dio un Esquema UML de cómo debería la relación y que clases deberían de existir, en la Ilustración 51 se puede ver el diagrama propuesto y que se ha implementado finalmente.

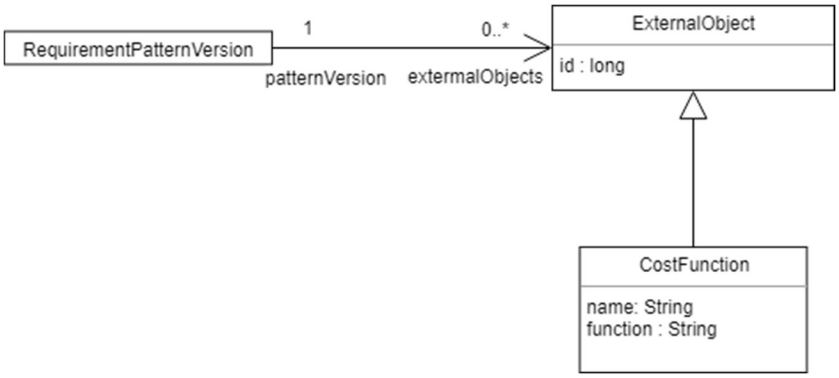

Ilustración 51: Esquema a implementar

15.2.1 Implementación a nivel de base de datos

En primer lugar, se debe implementar una nueva tabla que represente ExternalObject siguiendo una de las estrategias que Hibernate emplea para representar una relación entre una superclase y sus respectivas subclases. En particular Hibernate soporta diversas estrategias de mapeo de herencia, pero se ha decidido mantener la forma en la que se ha realizado en el proyecto para ofrecer una solución coherente. En particular se emplea la estrategia tabla por subclase.

Se procede a crear la nueva tabla EXTERNAL_OBJECT de la siguiente forma:

```
CREATE TABLE PATTERNS.EXTERNAL_OBJECT
(
	ID BIGINT PRIMARY KEY NOT NULL,
	PATTERN_ID BIGINT,
	CONSTRAINT FKFA1F763339533A9A FOREIGN KEY (PATTERN_ID)
	REFERENCES PATTERNS.REQUIREMENT_VERSION(ID)
)
```

Se crea una tabla que representa a ExternalObject, con lo cual él único atributo que tiene es ID, además contiene como foreign key una referencia a REQUIREMENT_VERSION. La columna ID es de tipo *bigint*, que corresponde al tipo *long* de Java y PATTERN_ID también es tipo bigint dado que también es un ID de otro objeto.

Ahora crearemos la tabla COST_FUNCTION de la siguiente forma:

```
CREATE TABLE PATTERNS.COST_FUNCTION
(
	ID BIGINT PRIMARY KEY NOT NULL,
	NAME VARCHAR (255) NOT NULL UNIQUE,
	COMPLETE_FUNCTION CLOB NOT NULL,
	CONSTRAINT FKC0B6640A353CC9D8 FOREIGN KEY (ID)
	REFERENCES PATTERNS.EXTERNAL_OBJECT(ID)
)
```

Esta tabla se utilizará para almacenar la información de la función de coste. La columna ID es tipo *bigint*, la columna NAME es de tipo *varchar (255)* dado que el nombre no se espera que vaya a superar a ese número de caracteres y por último la columna COMPLETE_FUNCTION es donde se guardará la función de coste, es de tipo *clob* ya que no sabemos exactamente qué tamaño puede tener una función de coste. Además, se ha añadido una contraint para que el ID de COST_FUNTION deba de existir en EXTERNAL_OBJECT.

15.2.2 Implementación a nivel de código

A nivel de código es necesario crear dos nuevas clases de dominio: ExternalObject y CostFunction. Para mejorar la visualización se omitirán los getters y los setters.

```
public abstract class ExternalObject {

	/*
	 * ATTRIBUTES
	 */
	protected long id;
	protected RequirementPatternVersion patternVersion;

	// MORE CODE...

}
```

ExternalObject es una clase abstracta debido que no se puede instanciar un objeto externo en general, sino que habrá que instanciar alguno de sus hijos (que por ahora solamente hay uno), tiene dos atributos, el primero es el ID que lo va a identificar y el segundo es la versión de un patrón al que va a ser asignado.

```
public class CostFunction extends ExternalObject{

	/*
	 * ATTRIBUTES
	 */
	private long id;
	private String name;
	private String function;

	// MORE CODE...

}
```

CostFunction hereda de ExternalObject, con lo cual heredará los dos campos de ExternalObject, pero el atributo id será "sobrescrito" ya que CostFunction ahora tiene otro campo llamado ID. Además de los campos de ExternalObject esta nueva clase tiene 2 campos nuevos *name* y *function* que son de tipo string.

Ahora vamos a añadir un atributo a la clase RequirementPatternVersion (dentro de edu.upc.gessi.rptool.domain.patternelements) y añadimos un atributo nuevo de la siguiente forma (con sus correspondientes getters y setters):

```
public class RequirementPatternVersion extends PatternObject {
	/*
	 * ATTRIBUTES
	 */
	// Here goes more atributes of this class

	private Set<ExternalObject> externalObjects;

}
```

15.2.3 Cambios en el Mapeo

Una vez implementadas los cambios a nivel de código, se procede a actualizar el fichero de mapeo.

Para indicar que la clase EXTERNAL_OBJECT corresponde a la clase Java ExternalObject, se tiene que crea un tag nuevo *class* donde se indica el nombre la ruta completa de la clase y como tabla el nombre de la tabla en la BD.

```
<class name="edu.upc.gessi.rptool.domain.ExternalObject"
table="EXTERNAL_OBJECT">
. . .
</class>
```

Dentro de ese tag, se tiene que indicar que el atributo Java *id* corresponde a la columna ID y que se generará con el algoritmo Hi-Lo (Es el que se ha usado en todo el ámbito del proyecto).

```
<id name="id" column="ID" type="long">
<generator class="hilo" />
</id>
```

Ahora vamos a indicar en el mapeo, la existencia de la relación many-to-one que existe entre RequirementPatternVersion con ExternalObjetc, para ello se añade el siguiente trozo en el fichero de mapeo:

```
<many-to-one name="patternVersion" column="PATTERN_ID"
class="edu.upc.gessi.rptool.domain.patternelements.RequirementPatternVersion"
not-null="false" lazy="false" />
```

Lo siguiente a realizar es mapear la clase CostFunction, que se realiza de la siguiente forma:

Después para mapear CostFunction, se declara la subclase con el tag *joined-subclass* indicando así que se sigue la estrategia tabla por subclase. Donde se debe de indicar como name hay que indicar la ruta completa con los package y nombre de la clase y por último indicar el nombre de la tabla.

```
<joined-subclass name="edu.upc.gessi.rptool.domain.CostFunction"
table="COST_FUNCTION">
    <key column="ID" />
    <property name="name" column="NAME" not-null="true" unique="true"/>
    <property name="function" type="text" length="2000"
                        column="COMPLETE_FUNCTION" not-null="true" />
</joined-subclass>
```

Con las líneas indicadas anteriormente, se mapea la clase, indicando cual la clave que lo identifica, y los dos atributos como el name y function.

Por último, se indica a Hibernate que actualice el atributo que tiene la clase RequirementPatternVersion, para ellos se tiene que indicar de la siguiente forma:

```
<set name="externalObjects" lazy="false" cascade="all-
delete-orphan" >
      <key column="PATTERN_ID" />
      <one-to-many
class="edu.upc.gessi.rptool.domain.ExternalObject" />
</set>
```

Con esto ya funciona la nueva funcionalidad, más tarde en las siguientes secciones se creará las llamadas REST para gestionar (TODO).

16 Estudio de la Tecnología de los servicios web de PABRE-WS

En este capítulo se explican las tecnologías que conforman los servicios web de PABRE-WS. En primer lugar, se proporciona una definición general de servicio web basado en el TFG de Adrián Rambal para luego explicar el tipo específico de servicios web (servicios web REST) y el software empleado para su implementación en PABRE-WS. Además, se proporciona un ejemplo de uso de servicios web REST dentro de PABRE-WS.

16.1 Definición de servicio web

Un servicio web es un sistema software que emplea una determinada tecnología para interoperar en la web. Los servicios web ofrecen métodos para que remotamente los sistemas clientes puedan emplear las funcionalidades que ofrecen. En general estas funcionalidades corresponden a peticiones y respuestas que permiten a los clientes modificar y recibir datos procesados desde el servidor.

La ventaja a la hora de definir los servicios web de un sistema es que son independientes del cliente que los vaya a emplear. Es decir, es responsabilidad de las aplicaciones cliente adaptarse a los servicios web que ofrece un sistema y no al revés. Por esta razón un mismo conjunto de servicios web se puede emplear para proporcionar funcionalidades a distintos tipos de aplicaciones y sistemas software ya sean smartphones, páginas web, etc.

Existen distintas tecnologías y estándares que definen el funcionamiento de los servicios web. En particular en el ámbito del proyecto PABRE-WS se ha empleado la arquitectura REST para desarrollarlos.

Finalmente, un conjunto de servicios web REST también se denomina API REST. En el mundo del software, una API (Application Programming Interface) define como un programa debe comunicarse con otro (interfaz) mediante las funciones que puede realizar.

16.2 Servicios web REST

REST (Representational State Transfer) es un conjunto de pautas que definen un estilo de arquitectura para servicios web que suele emplear el protocolo HTTP de transmisión de información vía web. Cabe recalcar que estas pautas no conforman ninguna clase de estándar, si no que expresan una serie de recomendaciones y características que pueden aplicarse de una forma más o menos estricta a la definición de un sistema.

Las características que definen un sistema REST son las siguientes:

- **Separación entre cliente y servidor**: tanto el cliente como el servidor (encargado principalmente del almacenamiento de los datos) son independientes entre sí. Esto permite que ambos componentes puedan evolucionar de forma individual siempre y cuando la interfaz que empleen ambos para comunicarse se mantenga.
- **Interfaz uniforme**: define cómo deben interaccionar cliente y servidor basándose en los siguientes aspectos:
    - **Recursos**: un recurso representa información (generalmente corresponde a una entidad) que se puede crear, consultar, modificar o borrar y que está guardada en el servidor. Además, cada uno se puede identificar de forma única a través de su URI (Uniform Resource Identifier), similar a una dirección web, correspondiente. Cabe destacar que la representación del recurso en el cliente es independiente de la del servidor. Esto implica que un mismo recurso puede ser representado desde el servidor mediante distintos formatos de datos.

  - **Manipulación de recursos a través de sus representaciones:** la información que recibe el cliente es suficiente para permitir acciones de modificado o borrado sobre ese recurso del servidor siempre y cuando esa funcionalidad esté permitida.
  - **Mensajes auto descriptivos:** cada mensaje intercambiado entre cliente y servidor (en ambas direcciones) contiene la información para poder procesarlo correctamente. Por ejemplo: si el mensaje contiene unos datos que representan una imagen, el mensaje también contendrá información referente al formato de estos.
  - **Hypermedia as the Engine of Application State (HATEOAS):** un recurso devuelto por el servidor incluye URIs a otros recursos (*hypermedia*, vínculos) que están relacionados con este. La idea subyacente es que el cliente no debe conocer como están estructurados los recursos, en su defecto únicamente debería emplear estos vínculos.
- **Sin estado:** El servidor no necesita guardar información referente al estado del cliente. Cada mensaje recibido contiene suficiente información para responder correctamente esa petición.
- **Cacheable:** Las respuestas previas del servidor pueden guardarse completa o parcialmente (si así lo define el servidor) en el cliente para disminuir la comunicación entre ambas partes.
- **Sistema por capas:** El cliente no debe conocer si se está conectado directamente al servidor o algún intermediario. Estos servidores intermedios se emplean para mejorar el rendimiento del sistema, así como para aumentar las medidas de seguridad en el servidor final.

16.3 Realizar operaciones mediante una API REST

El siguiente ejemplo emplea algunos de los métodos definidos en los servicios web PABRE-WS para aclarar los conceptos de la arquitectura REST.

Supongamos que el servidor disponible a través de la siguiente dirección web www.ejemplo.com ofrece la API REST del catálogo de patrones de requisitos de PABREWS. La API permite a múltiples recursos (se detallarán más adelante) pero para este ejemplo se van a usar unos pocos. En particular se emplearán los recursos que representan el catálogo de patrones, un patrón del catálogo y las versiones de un patrón del catálogo.

Para hacer la API accesible a los programadores que desarrollen aplicaciones clientes es recomendable emplear nombres que sean consecuentes con la información que se está tratando. Además, la estructura de la URI que define un recurso debe seguir una jerarquía lógica. En este caso el recurso identificado por la URI www.ejemplo.com/patterns representa de forma única el catálogo de patrones, mientras que www.ejemplo.com/patterns/1234 representa el patrón con identificador 1234 dentro del catálogo. Si ahora se quiere hacer referencia al recurso "versiones del patrón 1234 del catálogo" la URI resultante será www.ejemplo.com/patterns/1234/versions siguiendo una jerarquía lógica.

Respecto a la manipulación de los recursos en el cliente usualmente se emplean los métodos GET, POST, PUT y DELETE definidos en el protocolo HTTP para obtener, crear, modificar y borrar, respectivamente, recursos en el servidor. De este modo si se quiere obtener la información asociada al recurso "catálogo de patrones" se realizará una petición HTTP con el método GET al recurso /patterns. O bien si se quiere modificar un patrón del catálogo se realizará la petición

HTTP con el método PUT, al recurso /patterns/1234 juntamente con la información necesaria para crear un patrón (toda esta información en la misma petición).

16.4 Software empleado

Actualmente los servicios web que ofrece la API PABRE-WS dispone solamente de métodos de consulta sobre el catálogo de patrones. De acuerdo con la definición usual de los métodos de consulta se emplea el método HTTP GET sobre la URI del recurso para acceder a él y el formato escogido para representar el recurso en la respuesta es JSON.

Asimismo, se han empleado los códigos de estado HTTP para añadir información sobre la petición que se ha recibido. Este código va incluido en la respuesta que el servidor devuelve al cliente al recibir una petición y en el proyecto se ha empleado para definir si la petición era correcta o para determinar si ha habido algún tipo error. La información que se ha enviado representando a los recursos para todas las peticiones está en formato JSON.

Desde el punto de vista la implementación en los servicios web interviene las tecnologías que permiten tratar peticiones y respuestas HTTP (librería Jersey), generar los recursos en formato JSON para añadirlos a las respuestas (librería Jackson) y por último las relacionadas con la base de datos (librería Hibernate y Apache Derby). El esquema de funcionamiento se puede ver en la Ilustración 52. Consiste en tratar la petición HTTP que se recibe en el servidor, recuperar el recurso (entidad) que se está pidiendo mediante su consulta en la base de datos (Hibernate y Apache Derby), convertir esta entidad a formato JSON (Jackson) y finalmente devolver una respuesta HTTP con esta información (Jersey).

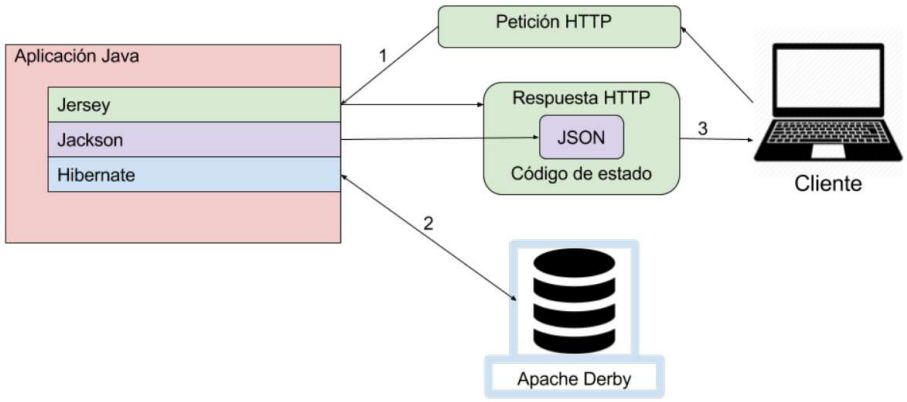

Ilustración 52: Funcionamiento PABRE-WS

16.5 Servicios web existentes

Los servicios web existentes de PABRE-WS se pueden consultar online en esta ruta: https://pabrews31.docs.apiary.io/#

16.6 Transmisión de información

Para transmitir la información entre cliente y servidor se utilizan DTOs, DTO son las siglas de Data Transfer Object, un patrón de diseño de software empleado para representar los datos de una entidad del dominio en la capa de presentación. La ventaja de emplear el patrón DTO es que la representación de una entidad del dominio queda desligada de su implementación. Esto nos permite añadir campos dentro del DTO, por ejemplo, en el dominio no existe ningún campo

llamado URI, pero en los DTO se puede añadir muy fácilmente (se verá más adelante), también nos permite no enviar toda la información de dominio.

En el proyecto se ha empleado el DTO para representar los recursos accesibles a través de la API, en particular cada entidad del catálogo tiene asociado un DTO. Los DTOs están implementado empleando la librearía Jackson para que automáticamente se realice el proceso de serialización que consiste en pasar los objetos Java a un formato JSON.

La librearía Jackson requiere que los atributos que se quieren serializar al JSON resultante puedan ser accedidos a través de métodos setters y getters públicos definidos en la clase. Además, el nombre con el que aparecerán estos atributos en su representación JSON depende como se llame el set y get (Por ejemplo: getNombre se representará como “Nombre” y getBla se representará como “Bla”) independientemente de cómo se llame el atributo.

La librearía Jackson nos permite usar anotaciones para definir diferentes aspectos de la serialización.

- @JsonInclude:
  Esta anotación se indica en la cabecera de la clase y indica a Jackson de solo añadir propiedades que cumplan ciertas condiciones. Mediante uso de parámetro se puede definir su comportamiento, en este proyecto solamente nos interesan 2 parámetros:
  1. Include.ALWAYS: Con este parámetro se indica a Jackson que siempre represente todos los campos de esa clase
  2. Include.NON_NULL: Con este parámetro se indica a Jackson que solamente se muestren los campos que no sean “null”.
- @JsonIgnore:
  Esta anotación se indica antes de un atributo, con ello Jackson simplemente ignorará ese campo y no lo mostrará.
- @JsonFormat:
  Esta anotación se ha empleado para dar formato a diversos atributos en el momento de la serialización. Principalmente se ha empleado en dar formato a las fechas
- @Ref:
  Se ha utlizado para generar la URI que acompaña al recurso devuelto. Esta anotación se emplea sobre un atributo Java de tipo URI, proporcionado por la librearía java.net para construirá con el formato deseado. Tomando el código de la Ilustración 53, ahí la anotación @Ref cogerá el valor de id y creara el objeto URI y lo mostrará en el JSON.

```
private long id;

@Ref(value = "schemas/{id}", style = Style.ABSOLUTE)
private URI uri;
```

Ilustración 53: Ejemplo @Ref

17 Cambios en las llamadas REST

Tal como se ha indicado anteriormente uno de los objetivos de mi proyecto es añadir nuevas funcionalidades, para ello hay que explicar que son las llamadas REST y cómo funciona el framework Jersey. En este capítulo se va empezar explicando cómo funciona la anotación dentro del framework Jersey, seguidamente se hablará de los cambios que, realizado durante el proyecto, todos los cambios consisten en implementar nuevas llamadas o cambiar las existentes.

17.1 Explicación de las llamadas REST

Las clases que controlan las llamadas REST contienen el código a ejecutar cuando se recibe una petición mediante el protocolo HTTP. Pueden acceder a la información que se le ha pasado mediante la URI a la que se ha accedido, así como como diferentes parámetros internos del protocolo HTTP. Siempre acaban devolviendo una Respuesta HTTP, si tiene que devolver una respuesta se devuelve un JSON con la respuesta.

Los diferentes controladores REST son los siguientes:

- Catalogue: Este controlador gestiona las llamadas relacionadas con el catalogo al completo, por ejemplo, para importar o exportar el catalogo
- Metrics: contiene todas las operaciones relacionadas sobre las métricas
- Parameters: Este controlador gestiona los parámetros que contiene el catalogo
- PatternFunctionCost: Este controlar gestiona las operaciones sobre funciones de coste
- PatternObject: Este controlador gestiona las operaciones sobre las dependencias entre diferentes PatternObject
- Pattern: contiene todas las operaciones relacionadas con los RequirementPattern, RequirementPatternVersion, RequirementForms, Fixed part y Extended Part
- Schemas: Este controlador gestiona todo el sistema de clasificación
- Sources: gestiona las fuentes de información almacenadas en el sistema
- VersionKeyword: Este controlador gestiona todas las palabras clave para realizar búsquedas más eficientes

La implementación de estos controladores se ha realizado mediante las anotaciones de Java que proporciona la librearía JAX-RS. En la librearía estas clases se conocen como *root resources*. Para definir diversos aspectos sobre el funcionamiento de estos se han empleado las siguientes anotaciones:

- @Path:
  Se emplea para definir la ruta, a nivel de clase o a nivel de método. De este modo al recibir una petición HTTP sobre una determinada URI el framework determinará el método a ejecutar. Si se indica una ruta a nivel de Clase la librearía primero buscará la clase donde va la solicitud y una vez dentro mirará por cada método cuál de los métodos debe ser llamado. Por ejemplo, cuando el servidor recibe una solicitud con la URI "{baseUrl}/api/metrics/6325632" (donde 6325632 es la id de la métrica a buscar) buscará por todas las clases si hay alguna clase que coencida con "/metrics" y encontrará Metrics.java y una vez dentro buscará algún método que acepte un parámetro que en este caso lo acepta el getMetric que acepta un parámetro.

```
@Path("/metrics")
public class Metrics {

	@GET
	@Path("{id}")
	@Produces({ MediaType.APPLICATION_JSON })
	public MetricDTO getMetric(@PathParam("id") long id) {

		//SOME CODE
	}
}
```

Ilustración 54: Ejemplo anotaciones

- @GET, @POST, @PUT, @DELETE:
  Se emplean para definir el método de comunicación de un protocolo HTTP, el uso de @GET indica que la solicitud que va a llamar a ese método tiene que ser del tipo "GET", pasaría lo mismo con POST, PUT y DELETE, dado que solamente sirven para filtrar las solicitudes por el tipo de método de la llamada HTTP.

- @Produces:
  Se utiliza para especificar el formato de la información que contendrá la respuesta que devolverá el servidor. Tiene que ser uno de los MIME (Multipurpose Internet Mail Extesions) disponible por Jersey. Durante este proyecto solamente se utiliza 2 MIMEs diferentes el "TEXT_PLAIN" y "APPLICATION_JSON", se utiliza el texto plano para devolver los errores y para el resto se utiliza el JSON.

- @Consumes:
  Esta anotación se utiliza para filtrar y asegurarse el tipo de información que se va a procesar en una operación, normalmente se utiliza para filtrar según el tipo de información que envíe el cliente, pero en este proyecto solamente se acepta el MIME en todas las operaciones que reciban información (POST y PUT).

- @PathParam:
  Esta anotación nor permite acceder al valor de alguno de los parámetros que se encuentran en la URI, por ejemplo, en el ejemplo de Ilustración 54 se puede ver que getMetric accede al parámetro de la solicitud recibida, para ello indica que quiere recibir el parámetro "id" que en ese caso sería 6325632.

- @QueryParam:
  Esta anotación nos permite acceder a los valores de una *query*, una query es una lista de clave-valor, que se indica después del símbolo de interrogante '**?**'. Por ejemplo, cuando se indica el URI: http://localhost:8080/api/metrics?complete=false mediante el código se puede obtener el valor del complete como en el ejemplo de la Ilustración 55, donde además se indica que si el cliente no indica este campo en la ruta por defecto se tomará como falso.

```
@GET
@Produces({ MediaType.APPLICATION_JSON })
public List<MetricDTO> getMetrics(@DefaultValue("false")
@QueryParam("complete") boolean complete) {
}
```

Ilustración 55: Ejemplo de QueryParam

17.2 Obtener todas las dependencias que existan sobre un patrón

En esta sección, se va implementar una llamada para obtener todas las dependencias de un patrón, tal como se puede ver en la Ilustración 56, todos los objetos que desciendan de PatternObject (ver Ilustración 49) pueden tener dependencia de otro PatternObject, en este caso Cristina Palomares ha solicitado implementar una llamada que devuelva todas las dependencias de un patrón. Por ejemplo: si miramos la Ilustración 56 se puede ver que RequirementPattern, Form tienen una dependencia de RequirementPattern2. FixedPart depende RequirementPattern3 y por último RequirementPattern3 y ExtendedPart2 dependen de ExtendedPart1. La llamada a implementar nos tiene que devolver una lista donde se incluyan todos los objetos nombrados anteriormente e indicando el tipo de dependencia y la dirección.

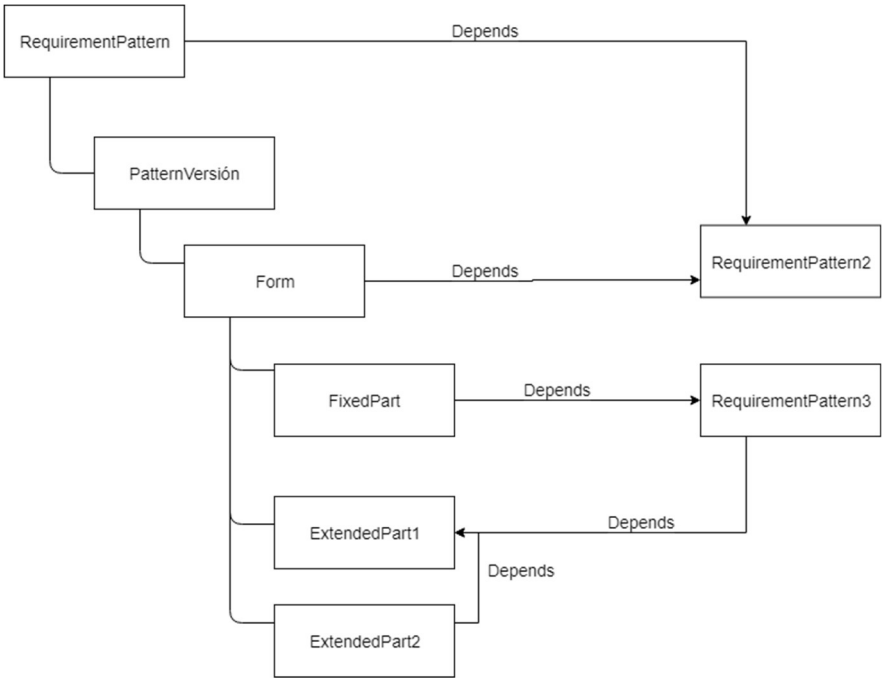

Ilustración 56: Ejemplo de dependencias

La ruta a implementar es la siguiente:

{{baseUrl}}/pabre-ws/api/patterns/{{patternID}}/dependencies

Si realizáramos una solicitud de tipo GET, nos tiene que devolver algo parecido a Ilustración 57.

```
{
  "id": 35127303,
  "dependenciesUri": "http://localhost:8080/pabre-ws/api/patternObjects/35127303/dependencies",
  "dependencies": [
    {
      "uri": "http://localhost:8080/pabre-ws/api/patternObjects/35127298/dependencies/35127308",
      "dependingObjectId": 35127298,
      "dependingObjectClass": "RequirementPattern3"
      "dependingObjectName": "mi pattern 3",
      "dependencyType": "IMPLIES",
      "dependencyDirection": "BIDIRECTIONAL",
      "dependentObjectId": 35127308,
      "dependentObjectClass": "ExtendedPart1",
      "dependentObjectName": "extended 1"
    },
    {
      "uri": "http://localhost:8080/pabre-ws/api/patternObjects/35127303/dependencies/35127313",
      "dependingObjectId": 35127303,
      "dependingObjectClass": "RequirementPattern",
      "dependingObjectName": "mi pattern",
      "dependencyType": "CONTRIBUTES",
      "dependencyDirection": "UNIDIRECTIONAL",
      "dependentObjectId": 35127313,
      "dependentObjectClass": "RequirementPattern2",
      "dependentObjectName": "mi pattern 2"
    },
    {
      "uri": "http://localhost:8080/pabre-ws/api/patternObjects/35127296/dependencies/35127313",
      "dependingObjectId": 35127296,
      "dependingObjectClass": "FixedPart",
      "dependingObjectName": "-",
      "dependencyType": "IMPLIES",
      "dependencyDirection": "BIDIRECTIONAL",
      "dependentObjectId": 35127313,
      "dependentObjectClass": "RequirementPattern",
      "dependentObjectName": "mi pattern 3"
    },
    ...
}
```


Ilustración 57: Ejemplo devolución llamada dependencia

17.2.1 La implementación

La implementación empieza modificando la clase edu.upc.gessi.rptool.rest.Pattern.java, es donde se implementan todas las llamadas relacionadas con los patrones, para ello se crea un nuevo método dentro de esa clase, le método se ha llamado getPatternCompleteDependencies y se le pasa como parámetro la identificación del patrón.

```
200    @GET
201    @Path("{id}/dependencies")
202    @Produces({ MediaType.APPLICATION_JSON })
203    public CompletePatternDependenciesDTO getPatternCompleteDependencies(@PathParam("id") long id) {
204        RequirementPattern rp = retrieveRequirementPattern(id);
205        Set<PatternObjectCompleteDependency> l = rp.getAllPatternDependencies();
206        CompletePatternDependenciesDTO pcd = new CompletePatternDependenciesDTO(rp, l);
207        return pcd;
208    }
```

Ilustración 58: implementación de la llamada

En la Ilustración 58 se puede ver la implementación del método, donde en la línea 200 se indica que la llamada tiene que ser tipo GET usando la anotación @GET del framework Jersey, en la siguiente línea se utiliza la anotación @Path indica la ruta que se tiene que marcar (en este caso

sería {{baseUrl}}/api/Pattern/{id}/dependencies), en la línea 202 se indica que lo que produce este método es un JSON, en la línea 204 se obtiene que el patrón que se requiere desde la base de datos, se realiza un llamada al patrón para que devuelva todas las dependencias que tenga y todos elementos como versión que contenga, seguidamente se crea un DTO y por último, se devuelva el DTO que Jersey conjuntamente con Jackson se convertirá en una respuesta HTTP con el contenido en formato JSON.

Dado que como se ha podido ver en la Ilustración 50, todo objeto que desciende de PatternObject, contiene un conjunto de dependencias, para ello he decidido crear un método en PatternObject llamada getAllDependencies() como se puede apreciar en la Ilustración 59.

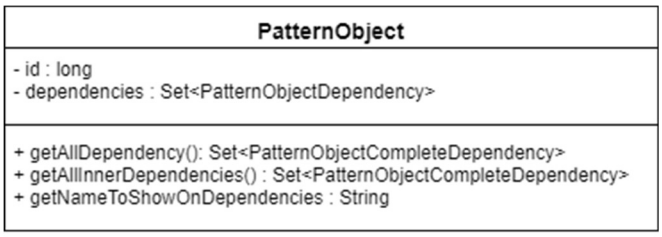

Ilustración 59: Nuevos métodos de PatternObject

Este método crea un nuevo PatternObjectCompleteDependency que contiene más información que PatternObjectDependency que se tiene guardado en PatternObject, una vez que este método guarde todas sus dependencias, llama a getAllInnerDependencies() que es un método abstracto eso quiere decir que todo los que hereden de esta clase deben de implementar este método. Igual que también se ha creado un nuevo método para mostrar en todas las dependencias, ya que hay algunos elementos como Parámetros, versiones, etc. Todo lo explicado anteriormente se puede ver en Ilustración 60.

```
45      /**
46       * This method returns all the dependencies of this object, includes this object
47       * dependencies and the all the inner childs.
48       *
49       * @return Set of all the dependencies of this {@link PatternObject}
50       */
51      public Set<PatternObjectCompleteDependency> getAllDependencies() {
52          Set<PatternObjectCompleteDependency> s = new HashSet<>();
53          for (PatternObjectDependency patternObjectDependency : dependencies) {
54              PatternObjectCompleteDependency pocd = new PatternObjectCompleteDependency(this, patternObjectDependency);
55              s.add(pocd);
56          }
57          s.addAll(getAllInnerDependencies());
58          return s;
59      }
60
61      /**
62       * This method returns complete dependencies inside a patternObject
63       *
64       * @return Set of all the dependencies inside of the {@link PatternObject}
65       */
66      protected abstract Set<PatternObjectCompleteDependency> getAllInnerDependencies();
67
68      public String getNameToShowOnDependencies() {
69          return "-";
70      }
```

Ilustración 60: Nuevos métodos implementados

Para visualizar mejor el flujo que sigue esta llamada a continuación mostrare los diagramas de secuencia [57] que sigue la llamada.

Como se puede ver en la Ilustración 61 , todo el proceso para obtener las dependencias empieza con el método getPatternCompleteDependencies() que se encuentra dentro de la clase

Patterns.java, este método es llamado por el framework Jersey, ya que se le ha indicado una ruta a ese método (“/patterns/{id}/dependencies”).

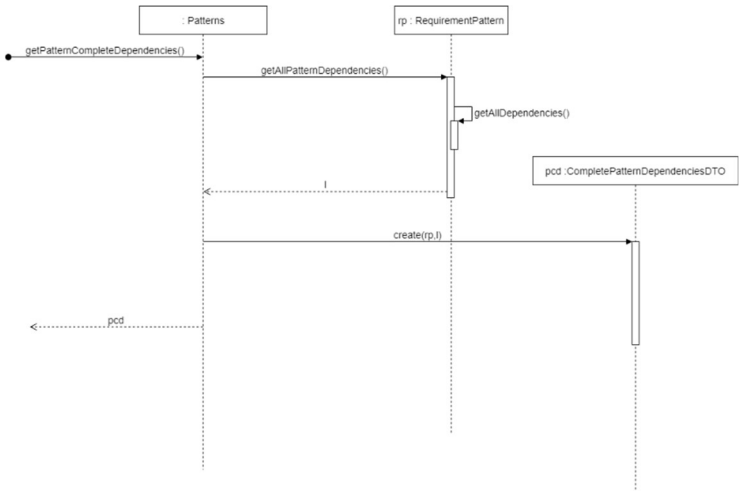

Ilustración 61: Llamada para obtener todas las dependencias

Una vez llamado a ese método, se obtiene el patrón en concreto al que se le quiere conocer todas las dependencias y se llama a su método getAllPatternDependencies(). A su vez RequirementPattern realiza una llamada getAllDependencies, que no lo tiene implementado, con lo cual se acabará ejecutando el código del objeto padre PatternObject.

Una vez que el patrón devuelve un conjunto *l*, se crea un nuevo CompletePatternDependenciesDTO, parando como parámetro el patrón obtenido para que el DTO pueda extraer información del patrón y pasando a la vez el conjunto de dependencias que se han encontrado, por último, se devuelve este DTO que luego Jersey-Jackson lo convertirán en un JSON.

Cuando el RequirementPattern llamo a getAllDependencies, ejecuto todo lo que se puede ver en la Ilustración 62, primero se crea un conjunto vacío donde se guardarán todas las dependencias, una vez creado el conjunto se empieza a recorrer todas las dependencias que contenga ese patrón y por cada uno de ellos se crea una instancia de PatternObjectCompleteDependencies, donde se almacenará toda la información necesaria y por último se guarda esa instancia en el conjunto que hemos creado anteriormente.

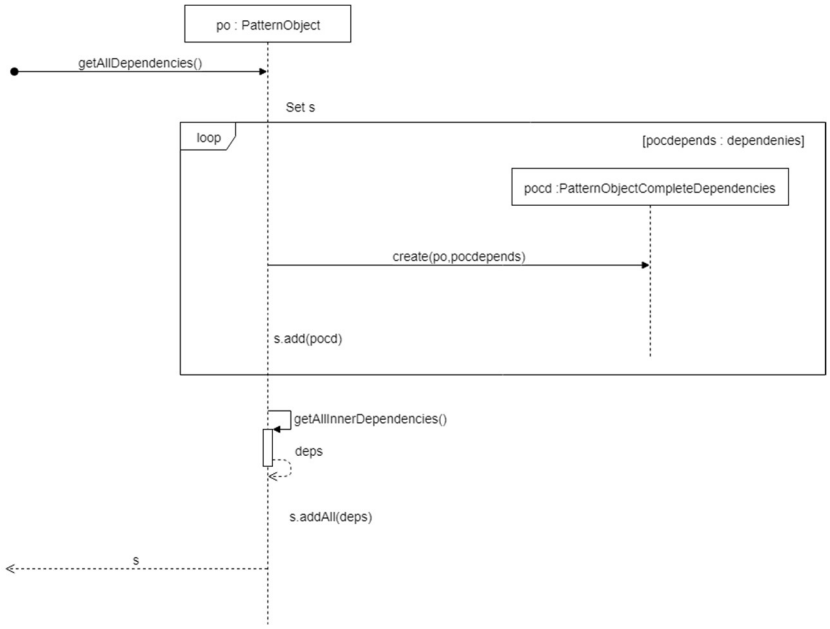

Ilustración 62: Obtener todas las dependencias del PatternObject

Una vez añadida esas dependencias al conjunto, se realiza una nueva llamada al método abstracto getAllInnerDependencies, que cada uno que extienda de PatternObject debe implementar y con ello debe añadir las dependencias que tengan los objetos que tenga dentro, por último, se añaden todas esas dependencias al mismo conjunto y se devuelve ese conjunto.

Siguiendo con el flujo que tiene la llamada, ahora vamos a ver la implementación de getAllInnerDependencies, dentro de RequirementPattern, eso se puede ver en la Ilustración 63.

Lo primero que realiza el método es crear un nuevo conjunto vacío donde se añadirán todas las dependencias. Una vez creado se recorren todas las versiones que hay de un patrón, se llama a getAllDependencies en cada uno de ello, todas las dependencias que obtiene se añaden a ese conjunto creado anteriormente, y por último se devuelve el conjunto con todas las dependencias.

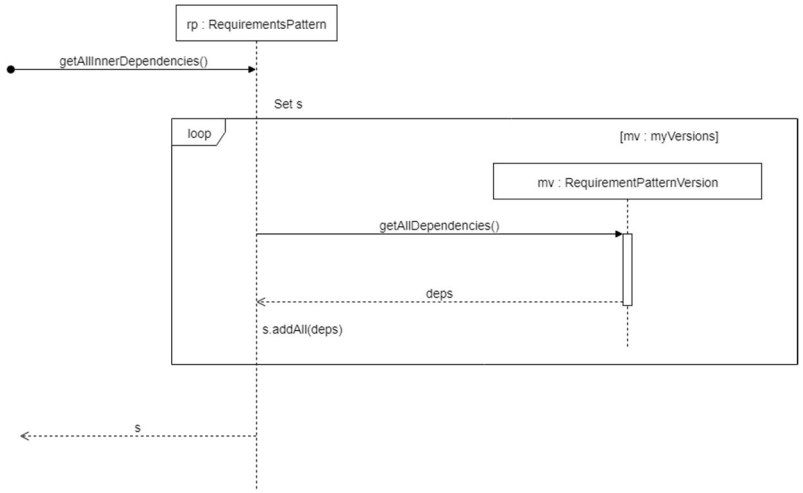

Ilustración 63: getAllInnerDependencies RequirementPattern

La llamada getAllDependencies de RequirementPatternVersion sigue el mismo proceso que el RequirementPattern, primero se llama al PatternObject para obtener todas las dependencias de

esa versión, para luego llamar a getAllInnerDependencies de esa versión, en la Ilustración 64 se puede ver que diagrama de secuencia sigue RequirementPatternVersion.

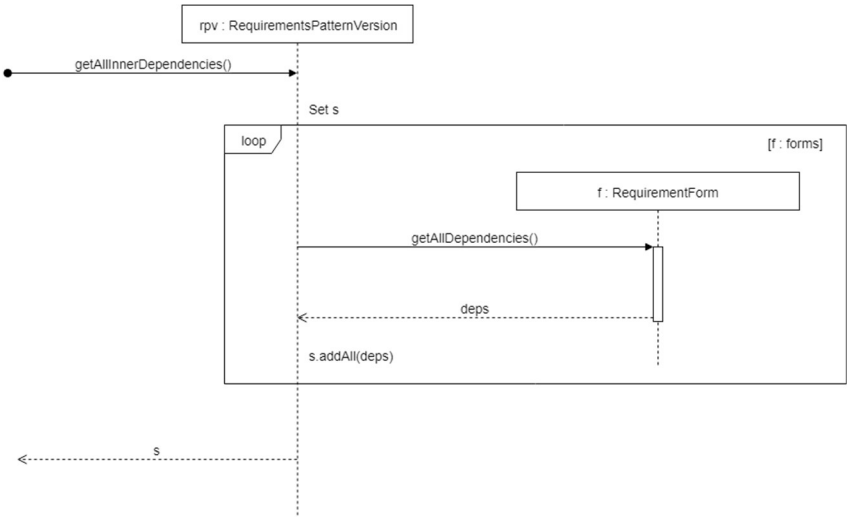

Ilustración 64: Llamada getAllInnerDependencies de una versión

Se repite el mismo proceso que antes pero ahora se llama a getAllDependencies de RequirementForm que hay en cada versión, al final en lo único que cambiará es en el GetAllInnerDependencies del RequirementForm. En la Ilustración 65 se puede verla implementación del método.

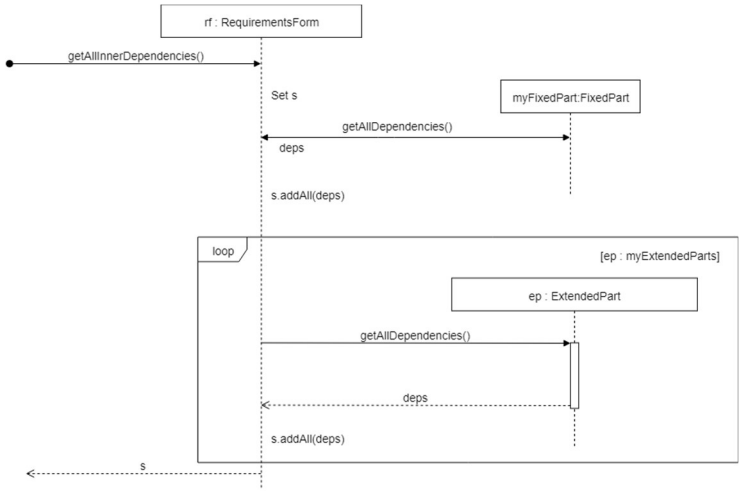

Ilustración 65: Llamada getAllInnerDependencies de un Form

En el caso de RequirementForm cambia un poco la implementación, ya que un Form tiene una parte fija y un conjunto de partes extendidas. Para ello primero se realiza un getAllDependencies de la parte fija y el resultado se guarda en el conjunto, después por cada parte extendida se obtienen sus dependencias.

A continuación, miraremos como se ha implementado getAllInnerDependencies, pero esta vez este método no se ha implementado dentro de FixedPart ni ExtendedPart, ya que las dos clases comparten la misma clase padre PatternItem, esa clase implementa la llamada getAllInnerDependencies ya que tiene que devolver las dependencias de los parámetros que contiene.

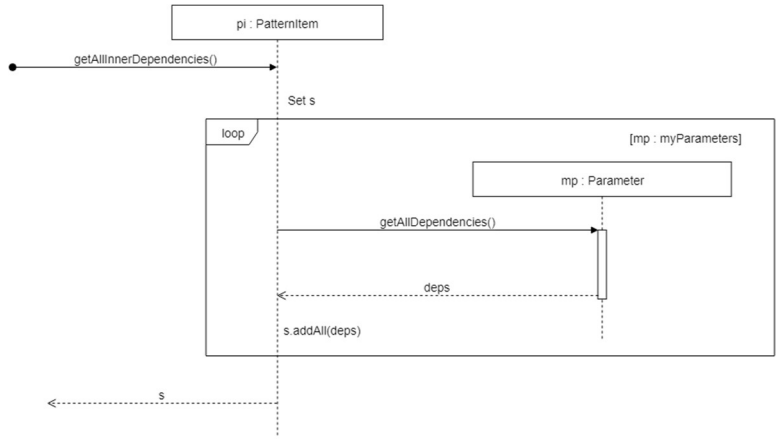

Ilustración 66: Implementación getAllInnerDependencies de PatternItem

La implementación de lo comentado se puede ver en la Ilustración 66, ahí se ve que se recorren todos los parámetros y se devuelven todas las dependencias de esos parámetros. Por último, Parámetro implementa la llamada getAllInnerDependencies, y lo único que devuelve es un conjunto vacío ya que no contiene ninguna dependencia interna aparte de las de PatternObject.

17.3 Obtener todos los patrones que estén en una ruta de clasificadores

En esta sección, se va a implementar una llamada REST para obtener los patrones de un clasificador mediante la ruta de su esquema. Dado un esquema como el de la Ilustración 67 se quiere obtener todos los patrones que están dentro de InternalCLassifier3 del Schema1, para ello se va implementar una llamada en la siguiente ruta:

```
{{baseUrl}}/pabre-ws/api/patterns?{{queryParams}}
```

Donde queryParams será la ruta hasta los patrones a obtener, por ejemplo, para obtener los patrones “Pattern1” y “Pattern2” se debería de realizar un GET a la siguiente ruta:

```
{{baseUrl}}/pabre-ws/api/patterns?namesList=Schema1
&namesList=RootClassifer&namesList=InternalClassifer1&namesList=InternalClassifier3
```

Con ello obtendremos una lista complete de todos los patrones que hay en el último nivel de los clasificadores (InternalClassifier3 en este caso).

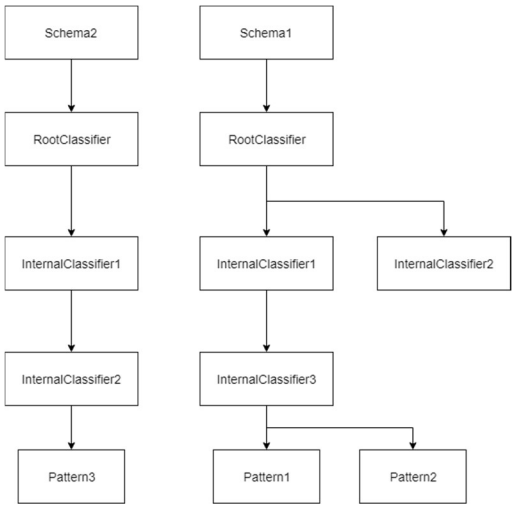

Ilustración 67: Ejemplo llamada clasificadores

17.3.1 La implementación

Decidí implementar esta búsqueda dentro de la ruta de los patrones, ya que esta búsqueda lo único que hace es coger todos los patrones y realizar un filtro para obtener solamente los patrones que estén dentro de un clasificador.

La implementación empieza en la clase edu.upc.gessi.rptool.rest.Patterns.java, lo primero que se tiene que hacer es cambiar el método que ya existe, llamado getPattern que se ejecuta cuando se solicitan todos los patrones, anteriormente se podía solicitar los patrones indicando si se quieren los patrones con todos los campos o no, también se podían filtrar los patrones para que contengan una palabra clave. La Ilustración 68 muestra la nueva cabecera del método indicado anteriormente, ahora contiene un lista que se obtendrá como "namesList" donde se indicarán los clasificadores.

```
@GET
@Produces({ MediaType.APPLICATION_JSON })
public Set<RequirementPatternDTO> getPatterns( @DefaultValue("") @QueryParam("keyword") String keyword, QueryParam("namesList")
List<String> namesList,@DefaultValue("false") @QueryParam("complete") boolean isComplete) throws ValueException {
// CODE HERE
}
```

Ilustración 68: Nueva cabecera getPatterns

Una vez dentro del método se añadió una condición mirando si el usuario pasaba la lista con los nombres o no, en caso de que pasará una lista, lo primero que se hace es obtener el Contexto actual de Jersey, dado que Jersey instancia estas clases *root resources* se requiere obtener esa instacia para poder solicitar los diferentes clasificadores. Una vez obtenida la instacia de root resource de Schemas.java (sirve para todo el sistema de clasificación), se llama al método getLastInternalClassifier pasando la lista que ha pasado el cliente, ese método nos devolverá la instacia del último clasificador dentro de la lista (En el ejemplo de la Ilustración 67 sería la instancia de InternalClassifier3). Una vez que se obtiene el clasificar el siguiente paso sería obtener todos los identificadores de los patrones de ese clasificador mediante PatternDataController con la llamada listPatternIdWithNameEditableAvailable, pasando como parámetro el clasificador. Todo lo comentando anteriormente se puede ver como código en la Ilustración 69.

```
// Another if with given keyword
} else if (namesList != null && namesList.size() != 0) { // Case when someone wants patterns
                                                          from a specified internalClassifier
        Schemas sc = resourceContext.getResource(Schemas.class);
        InternalClassifier ic = sc.getLastInternalClassifier(namesList);
        if (ic == null) {
                throw new NotFoundException("One of the indicated classifiers in the route
are not found");
        }
        lrp = PatternDataController.listPatternsWithThisClassifier(ic.getId());
}  else { // Case when dosen't exists any filter
        requirementPatterns =
PatternDataController.listPatternIdWithNameEditableAvailable();
}
// More Code
```

Ilustración 69: Nueva condición con nombre de clasificadores

Una vez se han obtenido todos los Id de los patrones el siguiente paso es mismo que ya se realizaba cuando se hacía una solicitud filtrando los patrones con una keyword. Que consiste en por cada Id que se ha obtenido crea una instancia de RequiermentPatternDTO y añadirla en el conjunto que se va a devolver.

Ahora miremos la implementación de getLastInternalClassifier que se ha implementado en *root resource* Schemas, este método usa la recursividad para llegar al nivel más bajo de clasificadores, por ello a continuación explicaré mediante diagrama de secuencia para que se entienda mejor.

En la Ilustración 70 se puede ver que llega la llamada SchemaDataControlle desde Schemas, pasando una lista llamada namesList, cuando llega esa lista el controlador llama a método que hay dentro de la misma clase llamada getInternalClassififerByNames pasando la misma lista.

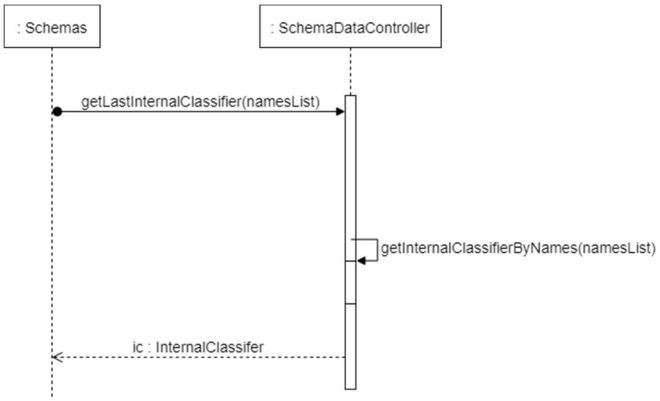

Ilustración 70: flujo en el root resource

La Ilustración 71 muestra cómo llega la llamada al controlador, lo primero que se hace es comprobar que como mínimo haya 3 elementos en la lista (nombre del Schema, nombre del root y nombre de un internalClassifer), si no hay 3 elementos se devuelve un error indicando que cómo mínimo debe de haber 3 nombres. Después de comprobar se obtiene el esquema que debe de ser el primer elemento de la lista, una vez obtenido el esquema se recorren todos los root Classifier que contiene ese esquema, y comprobar el nombre si es igual al segundo elemento de lista, en caso de que sea igual, se guarda una referencia a ese clasificador y se sale del bucle.

En último paso de este método es iniciar la llamada recursiva, pasando una sublista de la que se le ha pasado, descartando los primeros 2 elementos que ya hemos utilizado, Lo que nos devuelve ese método ya será el clasificador que estábamos buscando.

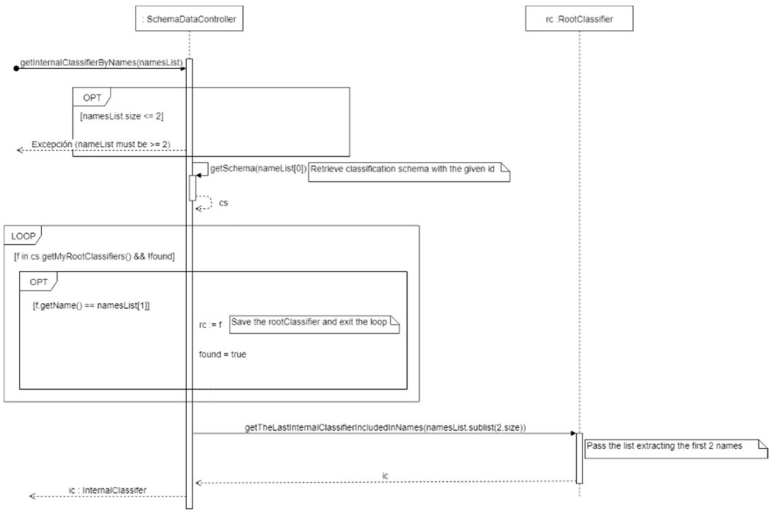

Ilustración 71: llamada que obtiene el esquema y recorre los root

Dado que RootClassifier no implementa la llamada que se le ha realizado, ese método será llamado en la clase padre que es InternalClassifier, la ejecución de esa llamada se puede ver en la Ilustración 72, donde lo que se realiza es recorrer todos los clasificadores que tiene ese método, y comprobar cuál de ellos tiene el mismo nombre que el primer elemento de la lista que se le ha pasado y cual se encuentre guardarlo y salir del bucle.

Después de obtener e clasificar se comprueba que la lista de nombres contenga más elementos para ir a más profundidad a buscar el clasificador (comprueba que sea >= 2 ya que el primer elemento es el que acabamos de buscar), en ese caso se llama otra vez a este método dentro del clasificador que acabamos de obtener, le pasamos una sublista quitando el primer elemento y el resultado ya será el clasificador que se busca. Todo este se ejecuta de forma recursiva hasta que no queden elementos en la lista de nombre.

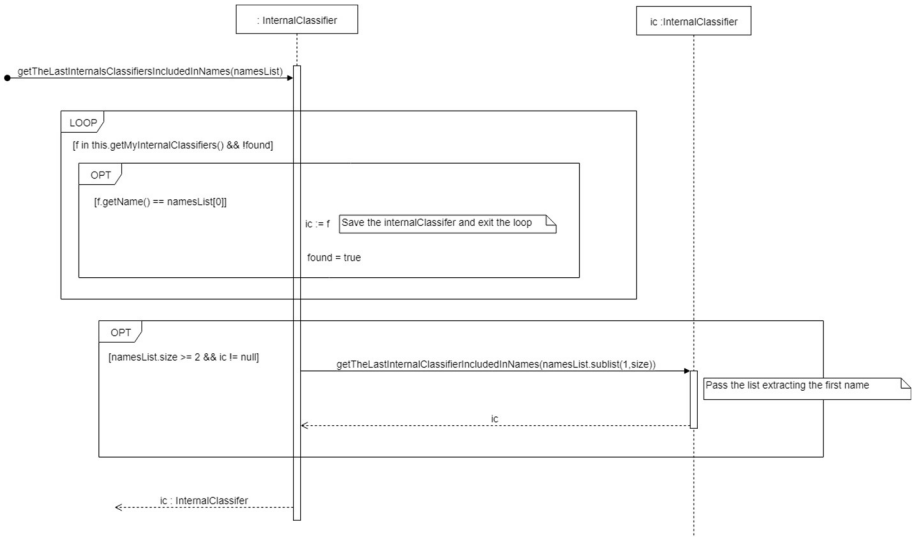

Ilustración 72: Llamada recursiva de InternalClassifier

Con todo lo comentado anteriormente esta llamada funciona y devuelve el resultado esperado, una vez implementado se procedió a crear los tests correspondientes y luego comentarlo en la API.

17.4 Implementar Importar Catalogo

En esta sección, se va a implementar la llamada REST para importar Sources, Keywords, Metrics, Patterns y Schemas, que se pasarán mediante un mensaje en formato JSON. Está funcionalidad permitirá tener un catálogo a partir de un fichero JSON.

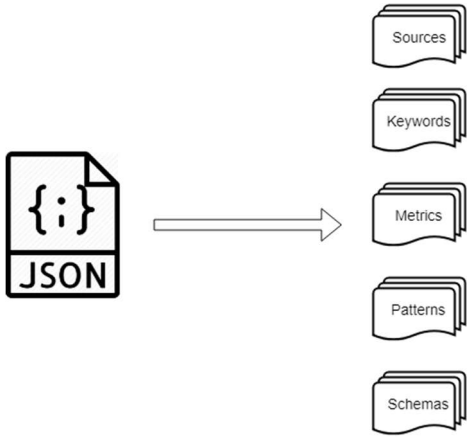

Ilustración 73: importar un catalogo

Se ha decidido que la llamada para importar sea en una nueva ruta para catálogo, para ello para importar se tendrá que realizar una llamada POST en la siguiente ruta:

{{baseUrl}}/pabre-ws/api/catalogue/

17.4.1 Diseño del JSON a importar

Antes de implementar el código hay diseñar el JSON que va enviar el cliente para importar toda la información, antes para importar algunas cosas como métricas tipo Set, se le pasaba la identificación de la métrica simple, pero ahora toda la información que nos pasan no está guardada y no tiene identificadores, tenemos que usar los nombre (en caso de Source usaremos identifier).

El diseño del JSON consiste en 5 listas, ninguna de ella será obligatoria.

1. Lista de fuentes de información(Sources):
   Las fuentes de información sigue con el mismo formato con el que se crea normalmente, por ejemplo, el código de la Ilustración 74 muestra cómo tiene que ser el formato.

```
"sources": [{
    "identifier": "ISO/IEC 9126-1334",
    "reference": "ref1",
    "type": "12",
    "comments": "1245"
  }]
```

Ilustración 74: Ejemplo sources

2. Lista de palabras claves(Keywords):
   La importación de las palabras es igual que la creación, la Ilustración 75 muestra un ejemplo de cómo se debe enviar.

```
"keywords": [{
    "name": "My keyword"
  }]
```

Ilustración 75: Ejemplo importación de Keywords

3. Lista de métricas (Metrics):
   La importación de métrica es diferente a la creación, dado que hay que indicar el tipo de métrica que se quiere importar y no hay parámetros para indicar. Para ello como se puede ver en la Ilustración 76, se indica primero con una clave “type” el valor del tipo de métrica que es (float, integer, time, etc.), luego se indica con una clave “Metric” y se indican los campos de esa métrica como en la creación, una de las diferencias con el caso cuando se quiere crear una métrica, es cuando la métrica es de tipo “Set”, para ello en lugar de dar el campo “simpleMetricId” ahora se pasa el campo “simpleMetricName”, donde se indica el nombre de la métrica que forma parte de ese conjunto, y la otra diferencia respecto a la creación es que ahora las fuentes de información en lugar de pasar la identificación de las fuentes como “source” se pasa como “sourceByIdentifier.

```
"metrics": [{
    "type": "float",
    "metric": {
      "name": "cambiado",
      "description": "h",
      "comments": "h",
      "minValue": 2,
      "maxValue": 6
    }
  }, {
    "type": "set",
    "metric": {
      "name": "nuevo set metric",
      "description": "h1",
      "comments": "h2",
      "simpleMetricName": "cambiado",
      "sourcesByIdentifier": ["sourceIden1"]
    }
  }]
```


Ilustración 76: Ejemplo import métricas

4. Lista de patrones (Patterns):
   En la importación de los patrones algunos campos han cambiado respecto a la creación, Los cambios son los siguientes:
   - Todas las fuentes de información que tengan alguno de los elementos internos del patrón tiene que ser enviado como “sourcesByIdentifier”, en lugar de sources.
   - Dentro de los parámetros ahora en lugar de pasar la métrica con su identificación como “metricId” ahora se pasará el nombre de la métrica con el nombre de “metricName”.
   - Las palabras claves de una versión ahora se pasarán con su nombre en lugar de la identifiación, aunque el campo se sigue llamando igual “Keywords”.

   La Ilustración 77 muestra un ejemplo de cómo debe de ser el JSON que se envía para poder importar solamente un patrón al catálogo, este patrón deberá de importar anteriormente lo siguiente: Un source llamado “sourceIden1”, una métrica llamada “metrica1” y otra llamada “metrica2”.

```
"patterns": [{
        "name": "mi pattern",
        "description": "descripcion pattern",
        "comments": "comentarios pattern",
        "editable": false,
        "sourcesByIdentifier": [
                "sourceIden1"
        ],
        "versions": [{
                "versionDate": "23-12-1995 22:22:22",
                "author": "autor version 2",
                "goal": "goal version 2",
                "numInstances": 0,
                "available": true,
                "statsNumInstances": 0,
                "statsNumAssociates": 0,
                "artifactsRelation": "artifacts rel v2",
                "forms": [{
                        "name": "n2",
                        "description": "description form 2 version 2",
                        "comments": "comments f2 v2",
                        "author": "author f2 v2",
                        "modificationDate": "23-12-1995 22:22:22",
                        "numInstances": 0,
                        "statsNumInstances": 0,
                        "statsNumAssociates": 0,
                        "pos": 1,
                        "available": true,
                        "fixedPart": {
                                "patternText": "pattern text fp f2 v2",
                                "questionText": "q fp f2 v2",
                                "numInstances": 0,
                                "available": true,
                                "statsNumInstances": 0,
                                "artifactsRelation": "art fp f2 v2"
                        },
                        "extendedParts": [{
                                "name": "b",
                                "patternText": "pattern text ep1 f2 v2",
                                "questionText": "q ep1 f2 v2",
                                "numInstances": 0,
                                "available": true,
                                "statsNumInstances": 0,
                                "parameters": [{
                                        "name": "my p",
                                        "correctnessCondition": "my cc",
                                        "description": "my d",
                                        "metricName": "metrica1"
                                }, {
                                        "name": "my p2",
                                        "correctnessCondition": "my cc2",
                                        "description": "my d 2",
                                        "metricName": "metrica2"
                                }
                        ],
                        "artifactsRelation": "art ep1 f2 v2",
                        "pos": 0
                        }]
                }]
        }],
}]
```


Ilustración 77: Ejemplo importación de patrón

5. Lista de Esquemas de clasificadores (Schemas):
   En la importación de los esquemas algunos campos han cambiado respecto a la creación, Los campos que han cambiado son los siguientes:
   - Todas las fuentes de información que tengan alguno de los elementos internos del patrón tiene que ser enviado como “sourcesByIdentifier”, en lugar de sources.
   - Todos los clasificadores ahora en lugar de tener un campo que se llame “requirementPatterns” que pasaba el id del patrón ahora se pasa solamente el nombre del patrón.

   La Ilustración 78 muestra un ejemplo del JSON que se debe de enviar, para importar un esquema al catálogo.

```
"schemas": [{
        "name": "nombre schema",
        "description": "descripcion schema",
        "comments": "",
        "sourcesByIdentifier": ["sourceIden1"],
        "rootClassifiers": [{
                "name": "root2",
                "description": "d root2",
                "comments": "",
                "sourcesByIdentifier": [],
                "type": 0,
                "npatterns": 0,
                "pos": 1,
                "requirementPatterns": [],
                "internalClassifiers": [{
                        "name": "interno root 2",
                        "description": "d interno root 2",
                        "comments": "",
                        "sourcesByIdentifier": [],
                        "type": 1,
                        "internalClassifiers": [],
                        "npatterns": 1,
                        "pos": 0,
                        "requirementPatterns": ["mi pattern"]
                }]
        }]
}]
```

Ilustración 78: Ejemplo importación de un esquema

17.4.2 La implementación

La implementación de esta nueva funcionalidad empieza en la clase Catalalogue.java que está ubicado en el siguiente paquete: edu.upc.gessi.rptool.rest, en esta clase se crea un nuevo método llamada importarCatalogo al que se le pasa un String que sería el JSON que ha enviado el usuario mediante el cuerpo de mensaje HTTP, lo primero que se realiza es Deserializarlo usando la clase Deserializer pasando el nuevo ImportUnmarshaller (que se comentará más adelante), con ello obtendremos una instancia de la clase ImportUnmarshaller, que se lo pasaremos a ObjectDataController, para que importe todo su contenido a nuestro catálogo, y una vez importado devolveremos una respuesta con el código 201 (creado correctamente).

La Ilustración 79 muestra el código que realiza todo lo comentado anteriormente.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response importCatalogue(String metricJson) throws Exception {
        ImportUnmarshaller iu = null;
        try {
                iu = Deserializer.deserialize(metricJson, ImportUnmarshaller.class);
                ImportDTO i = ObjectDataController.importCatalogue(iu);
                return Response.status(201).entity(i).build();
        } catch (Exception e) {
                e.printStackTrace();
                // When Jackson library throws a exception while deserializing a
                //object isalways JsonMapping or something related to that library,
                //so when we make our custom exception for semantically incorrect
                //object we have to cast it to Exception so Jersey don't use
                //JsonMappingExceptionMapper instead of
                        //SemanticallyIncorrectExceptionMapper
                if (e.getCause() instanceof SemanticallyIncorrectException) {
                        throw (Exception) e.getCause();
                }
                throw e;
        }
}
```

Ilustración 79: Llamada rest para importar

El ImportUnmarshaller contiene 5 listas de unmarshallers más 5 listas de objetos y una instancia a la sesión donde se va a guardar todos los objetos instanciados. En la Ilustración 80 se puede ver un diagrama UML con los atributos que tiene este nuevo unmarshaller y los 5 métodos principales para deserializar esos elementos y guardarlos en las correspondientes

listas de objetos instanciados. En el UML no se muestran los getters y setters para visualizar mejor todo, pero cada atributo tiene sus getters y setters.

Ilustración 80: Nueva clase ImportUnmarshaller

Para realizar la implementación de importar Sources y Keywords se han reutilizado los unmarshallers ya existentes, con lo cual todo funciona igual que en la creación de uno de ellos.

Para implementar la importación de las métricas se ha creado un nuevo unmarshaller llamado GenericMetricImportCreatorUnmarshaller (ilustración XX) donde se guardará toda la información recibida, y para generar la métrica se llama al método build() que nos devuelve una instancia de Metric, con la métrica ya creada lista para ser guardada en la base de datos.

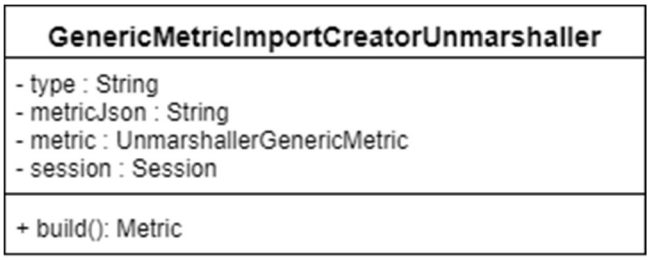

Ilustración 81: Nuevo unmarshaller para métricas

Para reutilizar al máximo las clases ya existentes, a la hora de importar algún patrón, se ha implementado de tal forma que se extiende el código anterior y se le añaden los campos que han cambiado. En la Ilustración 82 (se han quitado los métodos para simplificar) se puede ver que se han creado por cada uno una nueva clase que incorpora la palabra Import en el nombre, extiende la clase ya existente para crear, lo único que realiza es añadir nuevos atributos cambiando los tipos y así utilizar la clase padre, con ello se crea la instancia del objeto en la clase padre, pero desde la clase hijo se pueden realizar los cambios de todas formas. Tal como se comentó en el capítulo 12 Mejorar la calidad de código se han hecho cambios en el código para tener métodos más pequeños y así luego se pueden sobrescribir, se utiliza mucho el tema de sobrescribir métodos de la clase padre, por ejemplo, en la clase FixedPartImportUnmarshaller, se sobrescribe el método assignParameters para que en lugar de formar un parámetro a partir de ParameterUnmarshaller se creé a partir de ParameterImportUnmarshaller. Con ello se ha reutilizado el código al máximo posible y está abierto a nuevas extensiones en un futuro.

Una de las decisiones que he tomado es pasar la instancia de Session por todos los unmarshallers, dado que, si en un futuro se quiere ejecutar todo el código en paralelo, pasando las sesiones nos aseguramos que estamos buscando y creando instancia en la misma sesión de la base de datos.

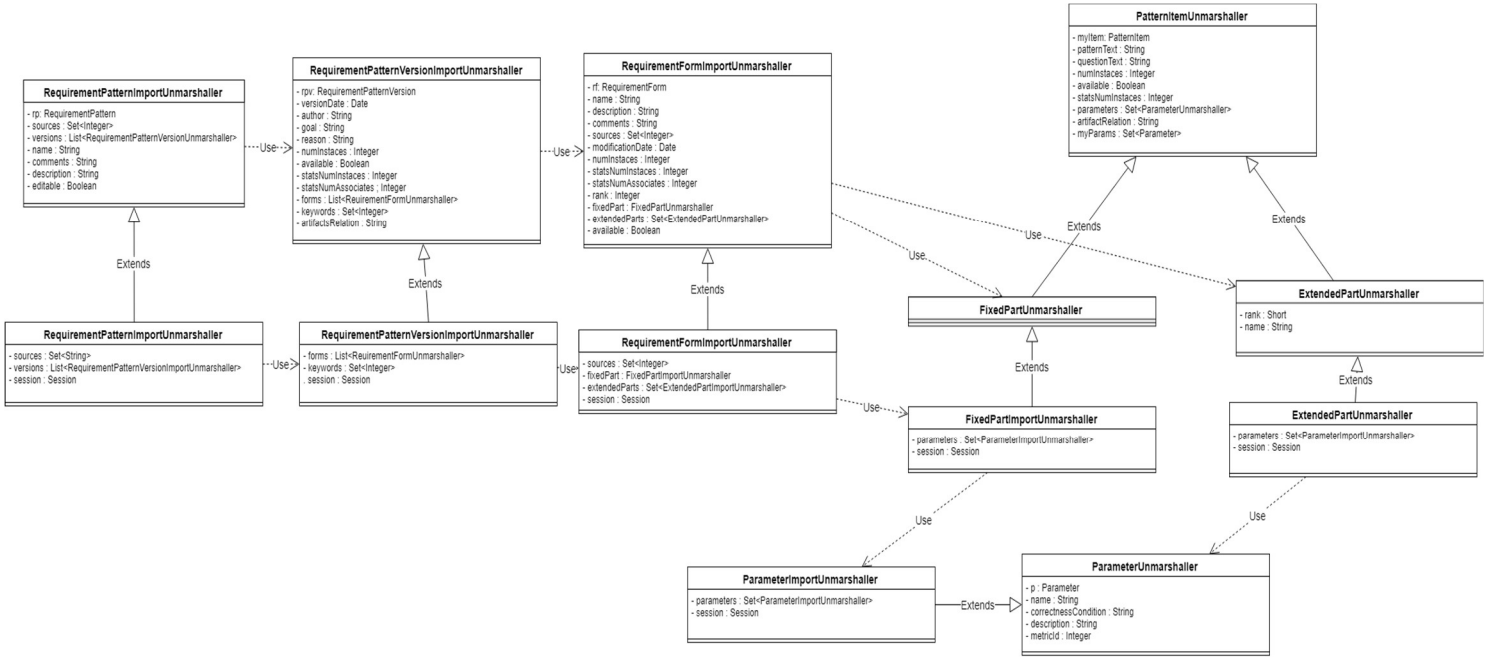

Ilustración 82: Diagrama UML de los unmarshallers de patrones

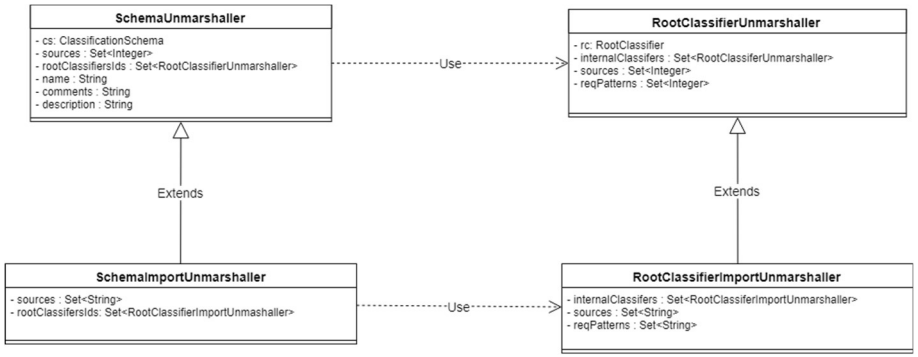

Ilustración 83: Diagrama UML de los unmarshallers de esquema

En el tema de importar Schema se ha seguido el mismo planteamiento extendido las clases ya existentes, añadiendo los nuevos atributos y sobrescribiendo los métodos pertinentes para que se utilicen las clases import en lugar de las clases normales para crear un esquema. La Ilustración 83 se puede ver el diagrama UML de la importación de los esquemas.

17.5 Implementar Exportar Catalogo

En esta sesión, se va a implementar la llamada REST para poder realizar un export de todo el catalogo existente en la base de datos, todo ello tomará forma en un fichero JSON para luego poder importar-lo de nuevo. Esta funcionalidad nos permitirá extraer toda la información de un catálogo a un fichero JSON.

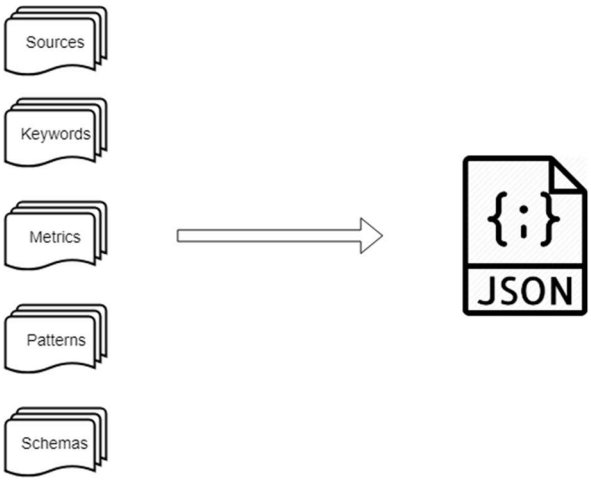

Ilustración 84: Exportar un catalogo

Se ha decidido que la llamada para exportar sea en la ruta general de catálogo, para ello para exportar se tendrá que realizar una llamada GET en la siguiente ruta:

```
{{baseUrl}}/pabre-ws/api/catalogue/
```

17.5.1 Diseño del JSON a exportar

Dado que el JSON a exportar debe de coincidir con el JSON que se puede importar, se va a utilizar el mismo formato explicado en el apartado 17.4.1 Diseño del JSON a importar.

17.5.2 La implementación

Para realizar la implementación de exportar el catálogo he decidido crear todos los DTOs (Data Transfer Objects) desde cero. Lo he decidido dado que la librearía Jackson que se utiliza en el proyecto, no permite ocultar los campos a devolver de forma dinámica, con eso quiero decir que no permite que un campo llamada URI no se pueda ocultar a la hora de exportar el catalogo. He realizado varias pruebas para ver si había alguna forma, encontré una forma, pero para lo que necesitamos ahora mismo de exportar no compensa el tiempo y esfuerzo para implementar esa solución.

Explicado ya lo anterior a continuación voy a explicar cómo funciona la funcionalidad de export, en la XX se puede ver el código a nivel de controlador REST que se ha implementado en la clase Catalogue, donde se puede ver que es un método GET del protocolo HTTP, además lo que devuelve este método tiene el MIME de type APPLICATION_JSON, en este método llamado exportCatalogue se llama al ObjectDataController para que nos devuelva una instancia de ExportDTO con todo el contenido listo y lo que se hace es devolver ese DTO.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response exportCatalogue() {
	ExportDTO export = ObjectDataController.exportCatalogue();
	return Response.status(Status.OK).entity(export).build();
}
```

Ilustración 85: llamada rest para exportar

Internamente lo que realiza ObjectDataController es obtener todos los elementos de esa clase, una vez que obtiene todos, los recorre creado el DTO pertinente para poder devolver. En la Ilustración 86 se puede ver un ejemplo de como el controlador obtiene todos los patrones del catálogo, los recorre y crea por cada uno de ello una instancia de RequirementPatternExportDTO y guarda la lista dentro del ExportDTO que será devuelto quien lo haya llamado.

```
ExportDTO exportDTO = new ExportDTO();

// MORE CODE

List<RequirementPattern> listPatterns =
MediatorGeneric.list(RequirementPattern.class);
List<RequirementPatternExportDTO> listPatternDTO = new ArrayList<>();
for (int i = 0; i < listPatterns.size(); i++) {
	listPatternDTO.add(new RequirementPatternExportDTO(listPatterns.get(i)));
}
exportDTO.setPatterns(listPatternDTO);

// MORE CODE

return exportDTO;
```

Ilustración 86: Obtención de los patrones y añadir a la lista

Ahora vamos a hablar de todos los DTOs que se han creado y se relacionan.

Sources:

Para exportar los Source se ha creado una clase llamada SourceExportDTO, que tiene los campos necesarios para exportar toda la información, en la Ilustración 90 se puede ver el diagrama UML de la implementación de la clase indicada anteriormente.

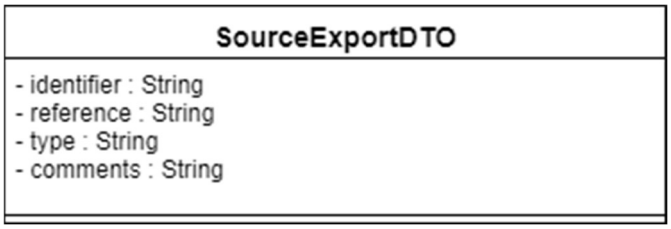

Ilustración 87: Diagrama UML exportación Sources

Keywords:

Para exportar las palabras claves se ha creado una clase llamada KeywordExportDTO, que tiene un único campo que es el nombre, en la Ilustración 88 se puede ver el diagrama UML de la implementación.

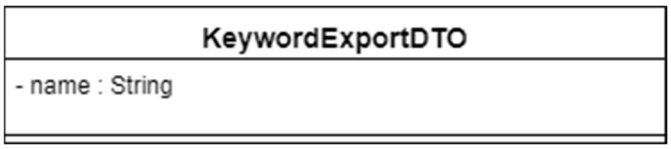

Ilustración 88: Diagrama UML exportación Keywords

Metrics:

Para exportar las métricas, se han creado 2 clases DTO, la primera de ella sirve para hacer el primer nivel indicando el tipo de métrica que es, la segunda clase sirve para exportar la métrica en sí, con todos los campos dependiendo del tipo de métrica que se quiera exportar.

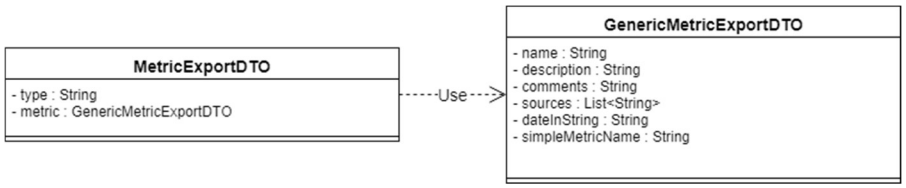

Ilustración 89:Diagrama UML exportación Metrics

Patterns:

Para exportar los patrones, se han creado un total de 8 clases, todo empieza cuando se crea la instancia de un RequirementPatternExportDTO, ya que se le pasa la instancia de un RequirementPattern y a partir de ahí se van generando todos los DTO de forma automática en los constructores de las clases.

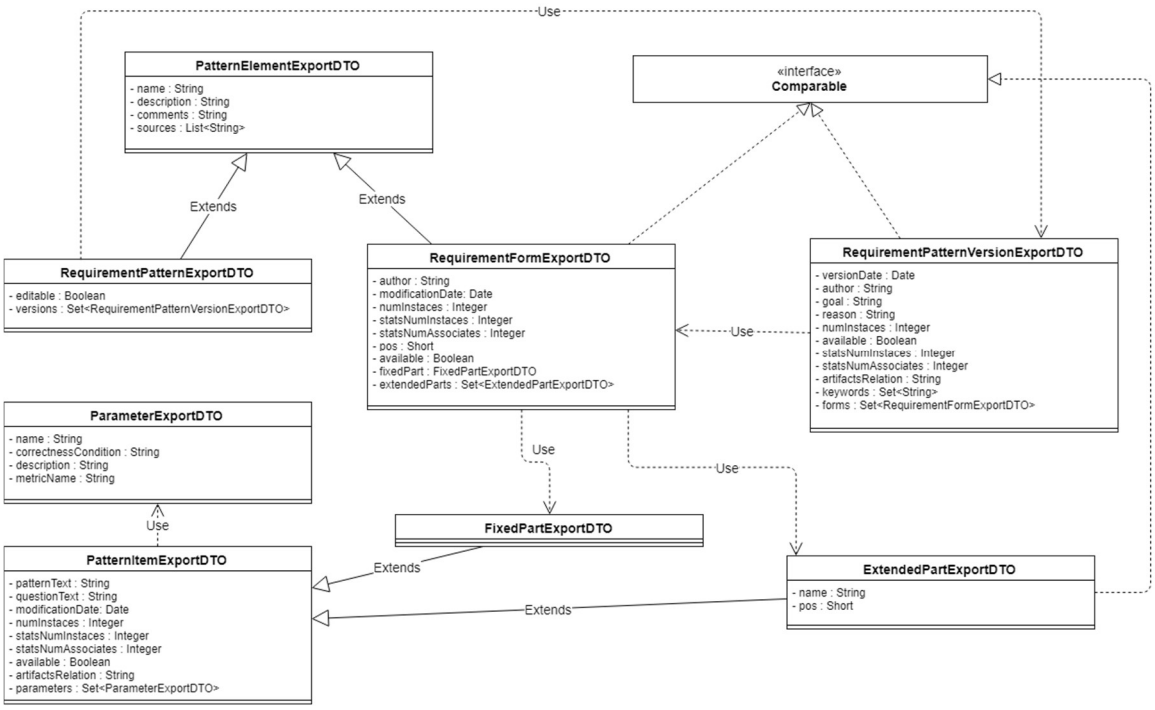

Ilustración 90: Diagrama UML exportación Patterns

Schemas:

La exportación de los esquemas que existen en el catálogo se han creado un total de 4 clases DTO, toda la estructura de DTO se genera automáticamente dentro den constructor de la clase SchemaExportDTO. La Ilustración 91 muestra las 4 clases implementadas.

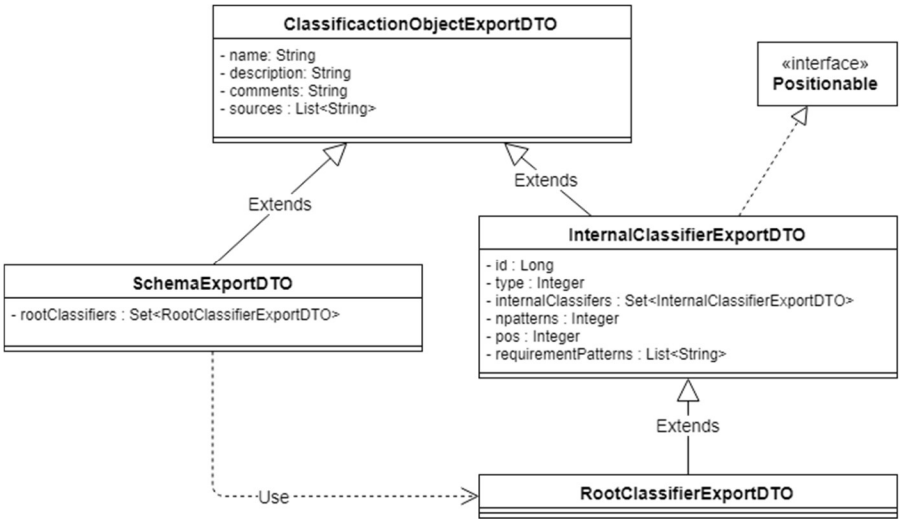

Ilustración 91: Diagrama UML exportación Schemas

18 Conclusiones

En este capítulo se hablará de los resultados de este proyecto, daré una valoración personal sobre el proyecto y por último trabajo futuro en el proyecto.

18.1 Resultado del proyecto

Desde el punto de vista de los resultados del proyecto, se podría considerar que ha sido un éxito.

El refactor realizado en PABRE, ayudará mucho a los futuros desarrolladores de la herramienta, ya que la calidad de código ha mejorado mucho y está todo más documentado. También será de mucha utilidad la documentación creada para los proyectos que van a utilizar la herramienta.

Las consultas implementadas ayudarán permitirá a los analistas de requisitos a buscar toda la información con mucha más facilidad.

Los nuevos tests asegurarán que el programa sea robusto.

La nueva funcionalidad de importar/exportar todo el catalogo ayudará mucho, ya que antes de tener esa implementación había que enviar una copia de la base de datos a todas partes, donde ahora solamente hará falta pasar un fichero con información en formato JSON, con ello el destinatario podrá tener todo el catalogo.

18.2 Valoración personal

La valoración global sobre este proyecto es muy positiva, dado que era mi primer proyecto donde partía con código que lo habían hecho otros estudiantes, con ello he podido ver la forma de plantear los problemas y las soluciones que han aportado a esos problemas.

He podido poner en práctica todos los conocimientos que he obtenido durante los 4 años de carrera, he podido poner en práctica todo el conocimiento sobre patrones que obtuve en la asignatura de Arquitectura de Software(AS), he podido poner en práctica el conocimiento obtenido sobre los servicios web y API REST de la asignatura de Aplicaciones y Servicios Web (ASW).

He podido realizar un proyecto sobre servicios web, es algo que me llamo mucho la atención cuando lo explicaron en clase y ahora he podido desarrollar una parte de un servicio web y puedo entender perfectamente cómo funcionan los servicios web.

Gracias al proyecto he aprendido a usar nuevas herramientas/ librearías que no había utilizado anteriormente y no sabía cómo utilizarlas al principio. En concreto, he aprendido usar Hibernate, Maven, y Jersey que no los había utilizado nunca. En cambio, había otras herramientas/librearías que, aunque si había tocado, sabía muy poco sobre ellas, por ejemplo: Apache Tomcat, Postman, DBeaver, Git y Restful APIs. Ahora con el conocimiento que tengo considero que se mucho más sobre ellas.

También he aprendido a resolver problemas sobre servicios web, he mejorado mi habilidad para buscar soluciones a los problemas de forma autónoma, se cómo y dónde buscar la información para resolver los problemas. Otra de las cosas que he aprendido es gestionar mejor el tiempo y, por último, también he aprendido mucho de ofimática para preparar la memoria final del proyecto.

En conclusión, considero que todo el conocimiento que he adquirido durante el desarrollo de este proyecto me será de mucha utilidad en mi carrera profesional, donde podré decir que tengo experiencia en ciertas herramientas y me interesa trabajar en ellas.

18.3 Futuro trabajo

En este proyecto se ha mejorado mucho la herramienta PABRE, pero se pueden implementar muchas cosas, que no se han podido realizar por la falta de tiempo y tener que dedicar el tiempo a otras funcionalidades que tenían más prioridad.

Una de esas cosas que se han quedado pendiente es implementar una llamada que cambie la base de datos para poder pasar las pruebas, mejorar el sistema de Log, actualmente el Log que he creado permite guardar toda la información en un único fichero, pero considero que es mejor separar la información guardada por el tipo donde se ejecuta el código (información de Apache, información del código, información de la base de datos, etc.), con ello será más fácil buscar donde ha ocurrido el problema.

Para futuro una de las cosas que hay que probar es realizar solicitudes de forma simultanea (concurrencia), ya que todas las pruebas realizadas hasta ahora son en secuencia, y durante el procesamiento de una solicitud no hay interferencia de ninguna otra solicitud.

Aunque en este proyecto se han implementado varias consultas, considero que en un futuro se necesitarán más consultas para facilitar el trabajo, a quienes quieran acceder al catálogo.

En conclusión, PABRE después de este proyecto está mucho mejor que la versión inicial del PABRE-WS, ya que permite realizar muchas más funcionalidades, pero como todo programa software siempre se le pueden añadir más funcionalidades y con este proyecto PABRE está listo para recibir más mejoras con mucha más facilidad.

19 Bibliografía

[1] Facultad de Informática de Barcelona, «FIB,» [En línea]. Available: https://www.fib.upc.edu. [Último acceso: 20 06 2018].

[2] OpenReq, «Proyecto OpenReq,» [En línea]. Available: http://openreq.eu. [Último acceso: 20 06 2018].

[3] Q-Rapids, «Proyecto Q-Rapids,» [En línea]. Available: http://www.q-rapids.eu. [Último acceso: 20 06 2018].

[4] Horizonte 2020, «Proyecto Horizonte 2020,» [En línea]. Available: https://ec.europa.eu/programmes/horizon2020/. [Último acceso: 20 06 2018].

[5] GESSI, «PABRE,» [En línea]. Available: http://www.upc.edu/gessi/PABRE/. [Último acceso: 20 06 2018].

[6] C. Palomares, «Tesis doctoral Cristina Palomares,» [En línea]. Available: http://www.essi.upc.edu/~cpalomares/publicacions/2016/Definition%20and%20Use%20of%20Software%20Requirement%20Patterns%20in%20Requirements%20Engineering%20-%20C.%20Palomares%20PhD%20Thesis.pdf. [Último acceso: 20 06 2018].

[7] GESSI, «Página principal GESSI,» [En línea]. Available: https://gessi.upc.edu/en. [Último acceso: 20 06 2018].

[8] Wikipedia, «Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/CRP_Henri_Tudor.

[9] Wikipedia, «REST,» [En línea]. Available: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional. [Último acceso: 20 06 2018].

[10] Software Testing Help, «45 Best Requirements Management Tools,» [En línea]. Available: https://www.softwaretestinghelp.com/requirements-management-tools/. [Último acceso: 20 06 2018].

[11] GARTNER, «Market GUide for Software Requirements Definition and Management SOlutions,» [En línea]. Available: https://www.gartner.com/doc/3356317/market-guide-software-requirements-definition. [Último acceso: 20 06 2018].

[12] TechRadar, «Modern Software Requirements Management Tools,» [En línea]. Available: https://www.forrester.com/report/TechRadar+Modern+Software+Requirements+Management+Tools+Q2+2016/-/E-RES131008.

[13] Seilevel, «Requirements Management Tools Evaluation,» [En línea]. Available: https://www.seilevel.com/business-analyst-resources/requirements-tools-reviews/. [Último acceso: 20 06 2018].

[14] IBM, «Rational DOORS,» [En línea]. Available: https://www.ibm.com/es-es/marketplace/requirements-management. [Último acceso: 20 06 2018].

[15] Modern Requirements, «Modern Requirements4TFS,» [En línea]. Available: https://www.modernrequirements.com/modern-requirements4tfs/. [Último acceso: 20 06 2018].

[16] JAMA, [En línea]. Available: https://www.jamasoftware.com/. [Último acceso: 20 06 2018].

[17] Wikipedia, «Desarrollo ágil de software,» [En línea]. Available: https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software. [Último acceso: 20 06 2018].

[18] Atlasian, «Bitbucket,» [En línea]. Available: https://bitbucket.org. [Último acceso: 20 06 2018].

[19] S. Chacon, «Git,» [En línea]. Available: https://git-scm.com/. [Último acceso: 20 06 2018].

[20] Atlassian, «Trello,» [En línea]. Available: https://trello.com/. [Último acceso: 20 06 2018].

[21] Wikipedia, «Servicio web,» [En línea]. Available: https://es.wikipedia.org/wiki/Servicio_web. [Último acceso: 20 06 2018].

[22] FIB, «Treball de FI de Grau,» [En línea]. Available: https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/treball-de-fi-de-grau. [Último acceso: 20 06 2018].

[23] A. Rambal, «Desarrollo y mejora de servicios web de acceso a patrones de requisitos,» [En línea]. Available: https://upcommons.upc.edu/bitstream/handle/2117/114148/126696.pdf?sequence=1&isAllowed=y. [Último acceso: 20 06 2018].

[24] F. M. Bernabé, «Development of web services for the PABRE system,» [En línea]. Available: https://upcommons.upc.edu/bitstream/handle/2099.1/19080/90888.pdf?sequence=1&isAllowed=y. [Último acceso: 20 06 2018].

[25] Oracle, «Java Code conventions,» [En línea]. Available: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html. [Último acceso: 20 06 2018].

[26] Apiary, «Apiary,» [En línea]. Available: https://apiary.io/. [Último acceso: 20 06 2018].

[27] Postdot Technologies Inc., «Postman,» [En línea]. Available: https://www.getpostman.com/. [Último acceso: 20 06 2018].

[28] J. Haleby, «Rest Assured,» [En línea]. Available: http://rest-assured.io/. [Último acceso: 20 06 2018].

[29] Govern de les illes Balears, «Adjudicación govern balear,» [En línea]. Available: http://www.plataformadecontractacio.caib.es/DocumentoAdjuntoView?idLicitacion=38755&idTipoDocumento=30.

[30] FIB, «Software Lliure i desenvolupament Social,» [En línea]. Available: https://www.fib.upc.edu/ca/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/SLDS. [Último acceso: 20 06 2018].

[31] FIB, «Sistemes Operatius per a Aplicacions distribuides,» [En línea]. Available: https://www.fib.upc.edu/estudis/graus/grau-en-enginyeria-informatica/pla-destudis/assignatures/SOAD. [Último acceso: 20 06 2018].

[32] Wikipedia, «Protocolo de transferencia de hipertexto,» [En línea]. Available: https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto. [Último acceso: 20 06 2018].

[33] W3schools, «JSON - Introduction,» [En línea]. Available: https://www.w3schools.com/js/js_json_intro.asp.

[34] Oracle, «Java,» [En línea]. Available: https://www.oracle.com/java/index.html. [Último acceso: 20 06 2018].

[35] Apache software Foundation, «Apache Tomcat,» [En línea]. Available: http://tomcat.apache.org/. [Último acceso: 20 06 2018].

[36] Oracle, «Jersey RESTful Web Services in Java,» [En línea]. Available: https://jersey.github.io/. [Último acceso: 20 06 2018].

[37] Redhat foundation, «Hibernate,» [En línea]. Available: http://hibernate.org/. [Último acceso: 20 06 2018].

[38] Eclipse Foundation, «Eclipse,» [En línea]. Available: https://www.eclipse.org/. [Último acceso: 20 06 20128].

[39] G. W. R. M. a. o. Colin Bell, «Squirrel SQL,» [En línea]. Available: http://squirrel-sql.sourceforge.net/. [Último acceso: 20 06 2018].

[40] Apache Software FOundation, «Apache Maven Project,» [En línea]. Available: https://maven.apache.org/. [Último acceso: 20 06 2018].

[41] Sistemas Master magazina, «Definición de DBMS,» [En línea]. Available: https://sistemas.com/dbms.php. [Último acceso: 20 06 2018].

[42] Apache Software Foundation, «Apache Software Foundation,» [En línea]. Available: https://www.apache.org/. [Último acceso: 20 06 2018].

[43] Wikipedia, «Código abierto,» [En línea]. Available: https://es.wikipedia.org/wiki/C%C3%B3digo_abierto. [Último acceso: 20 06 2018].

[44] Apache Software Foundation, «Apache Licence 2.0,» [En línea]. Available: https://www.apache.org/licenses/LICENSE-2.0. [Último acceso: 20 06 2018].

[45] Oracle, «Oracle,» [En línea]. Available: https://www.oracle.com/es/index.html. [Último acceso: 20 06 2018].

[46] Vogella, «java debugging with eclipse,» [En línea]. Available: http://www.vogella.com/tutorials/EclipseDebugging/article.html. [Último acceso: 20 06 2018].

[47] DNU Foundation, «GNU Lesser General Public License 2.1,» [En línea]. Available: https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html. [Último acceso: 20 06 2018].

[48] Asociación Española de Programdores Informáticos, «Programción Orientada a Objetos en Java POO,» [En línea]. Available: https://www.asociacionaepi.es/programacion-orientada-a-objetos-en-java/. [Último acceso: 20 06 2018].

[49] MundoLinux, «XML ¿Qué es?,» [En línea]. Available: http://www.mundolinux.info/que-es-xml.htm. [Último acceso: 20 06 2018].

[50] Pluralsight, «Javascript,» [En línea]. Available: https://www.javascript.com/. [Último acceso: 20 06 2018].

[51] Copyright, «Copyright,» [En línea]. Available: http://www.copyright.es/. [Último acceso: 20 06 2018].

[52] Open-source community, «DBeaver,» [En línea]. Available: https://dbeaver.io/. [Último acceso: 20 06 2018].

[53] ProgramarYa, «Modificadores de acceso,» [En línea]. Available: https://www.programarya.com/Cursos/Java/Modificadores-de-Acceso. [Último acceso: 20 06 2018].

[54] Aprender a programar, «Conceptos de objetos y clases de Java, Definición de instancia,» [En línea]. Available: https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=411:conceptos-de-objetos-y-clases-en-java-definicion-de-instancia-ejemplos-basicos-y-practicos-cu00619b&catid=68&Itemid=188. [Último acceso: 20 06 2018].

[55] Wikipedia, «Singleton,» [En línea]. Available: https://es.wikipedia.org/wiki/Singleton. [Último acceso: 20 06 2018].

[56] Edu Java, «Sobreescribir un método,» [En línea]. Available: http://www.edu4java.com/es/progbasica/progbasica15.html. [Último acceso: 20 06 2018].

[57] Sparx Systems, «Diagrama de secuencia UML,» [En línea]. Available: http://www.sparxsystems.com.ar/resources/tutorial/uml2_sequencediagram.html. [Último acceso: 20 06 2018].