DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Golden5

https://github.com/gii-is-DP1/dp1-2020-g2-13.git

Miembros:

* Arrans Vega, Isabel
* Beltrán Rabadán, Francisco Javier
* Bwye Lera, Matthew
* Colmenero Capote, Pablo
* López Rosado, Guillermo

Tutor: Müller Cejas, Carlos Guillermo

GRUPO G2-13

Versión 1

10 de enero de 2021

# Historial de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción de los cambios** | **Sprint** |
| 04/1/2021 | V1 | * Creación del documento | 3 |
| 10/1/2021 | V2 | * Añadido todo el contenido | 3 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Historial de versiones 2](#_Toc58783389)

[Introducción 4](#_Toc58783390)

[Diagrama(s) UML: 4](#_Toc58783391)

[Diagrama de Dominio/Diseño 4](#_Toc58783392)

[Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios) 5](#_Toc58783393)

[Patrones de diseño y arquitectónicos aplicados 5](#_Toc58783394)

[Decisiones de diseño 9](#_Toc58783395)

[Decisión X 13](#_Toc58783396)

[Descripción del problema: 13](#_Toc58783397)

[Alternativas de solución evaluadas: 13](#_Toc58783398)

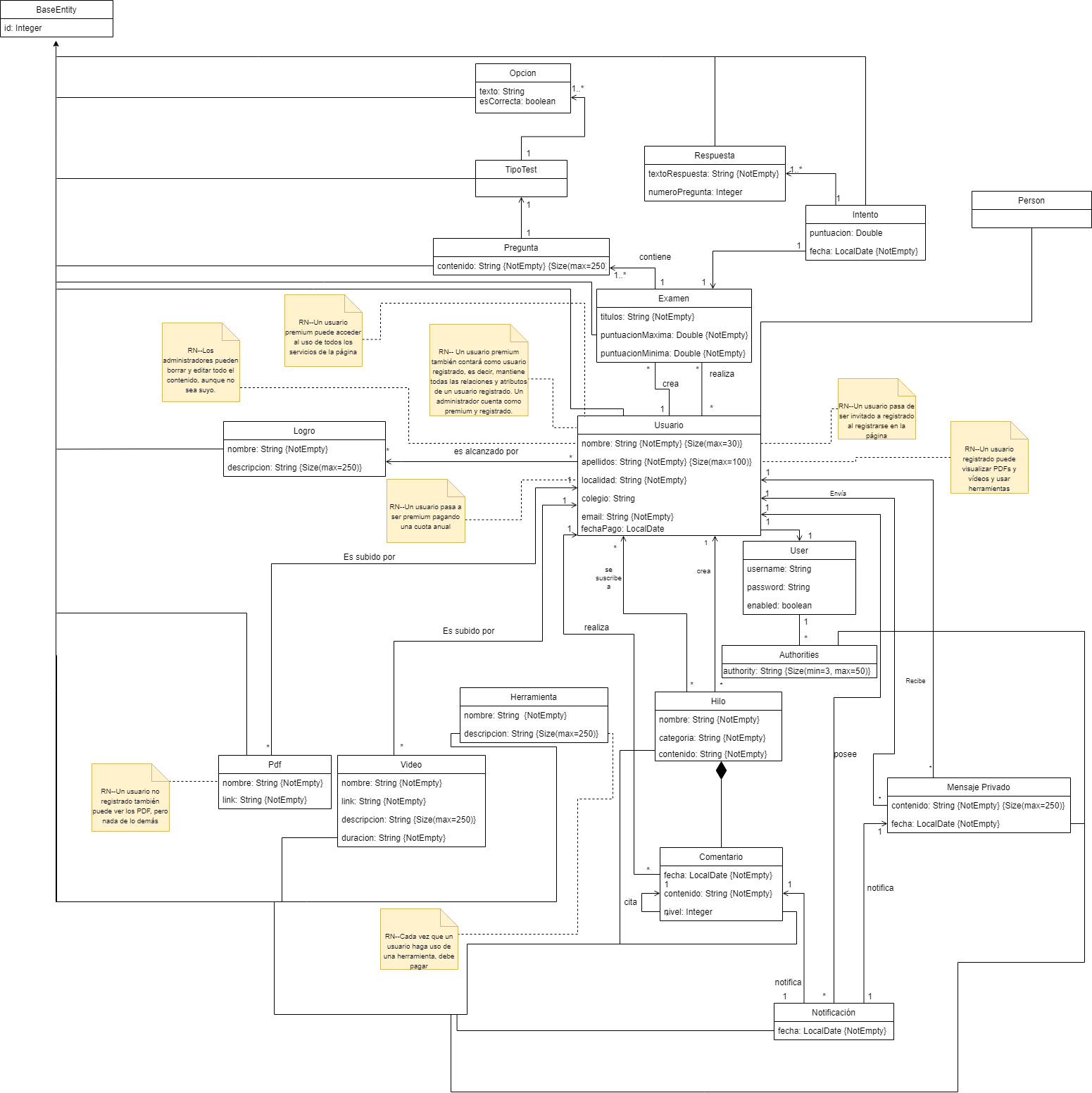
[Justificación de la solución adoptada 13](#_Toc58783399)

# Introducción

*En esta sección debes describir de manera general cual es la funcionalidad del proyecto a rasgos generales (puedes copiar el contenido del documento de análisis del sistema). Además puedes indicar las funcionalidades del sistema (a nivel de módulos o historias de usuario) que consideras más interesantes desde el punto de vista del diseño realizado.*

# Diagrama(s) UML:

## Diagrama de Dominio/Diseño



## Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

# Patrones de diseño y arquitectónicos aplicados

En esta sección de especificar el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

## Patrón: Modelo vista controlador

### Tipo: Arquitectónico

### Contexto de Aplicación

Nuestra aplicación maneja datos, los procesa y los muestra al usuario. Con este patrón, nuestra intención es distinguir estas actividades, y dividir el trabajo a realizar entre ellas (las actividades se dividen en 3 capas, en el modelo (procesamiento y almacenamiento de datos), en la vista (consumo de datos) y en el controlador (interacción entre el consumo de datos y las operaciones con esos datos)).

### Clases o paquetes creados

Paquetes model, repository, service y web. En el paquete de model se incluyen todas las clases que describen a los distintos tipos de objetos que almacenaremos en nuestra base de datos. En repository se incluyen las interfaces que luego implementarán las clases de service; en estas clases se definen las funciones que trabajan el acceso a la base de datos, es decir, donde se aplican las funciones CRUD. En web tenemos todas las clases controladoras, redirigen los datos a los distintos archivos jsp que conforman la vista de nuestra página.

### Ventajas alcanzadas al aplicar el patrón

Permite manejar fácilmente una gran cantidad de datos, operar con ellos, y mostrarlos al usuario. También es fácil cambiar aspectos de las distintas clases sin que afecten a otras partes de la aplicación (principio de modularidad), lo que facilita la mantenibilidad del sistema.

## Patrón: Front controller

### Tipo: Arquitectónico

### Contexto de Aplicación

Los datos a procesar por nuestra aplicación no reciben un tratamiento trivial, hay multitud de operaciones que son ejecutadas. Con este patrón pretendemos que las operaciones que se encuentran entre las operaciones del usuario y los datos almacenados en la base de datos estén mejor organizados y se simplifiquen en un menor número de clases, que contienen todos los métodos relacionados con cierto tipo de objetos.

### Clases o paquetes creados

Creamos el paquete web, que contiene las clases controladoras de la aplicación. Estas clases hacen uso de ciertas anotaciones de Spring que permiten hacer uso de este patrón con facilidad, simplificando la especificación de las urls que harán las llamadas a las funciones que se ejecutarán en nuestra aplicación.

### Ventajas alcanzadas al aplicar el patrón

Terminamos creando muchas menos clases y la complejidad de la definición de las funciones de los controladores se reduce, ya que se reutiliza gran parte del código de los distintos métodos que conforman el controlador de cierto tipo de objeto de la base de datos.

## Patrón: Proxy

### Tipo: Diseño

### Contexto de Aplicación

Este patrón de diseño es aplicado por el propio Spring de manera transparente mediante algunas notaciones como @Transactional.

### Clases o paquetes creados

De manera transparente al desarrollador, Spring creará una interfaz que implementara el sujeto y una clase proxy que hace referencia a este. () Duda -> poner que no hacemos nada o lo que haría spring por detrás

### Ventajas alcanzadas al aplicar el patrón

1. Evita que se dupliquen objetos, mejorando el rendimiento de la aplicación.

Mayor seguridad

## Patrón: Dependency injection

### Tipo: Diseño

### Contexto de Aplicación

Este patrón de diseño es aplicado dentro de nuestro proyecto en cada inyección de dependencias, es decir, en cada etiqueta @Autowired. Ha sido empleada con frecuencia en servicios, controladores y las pruebas unitarias. En Spring, Application Context es el assembler requerido por dicho patrón que gestiona la inversión de control, sobre clases con anotaciones como @Component, @Repository, @Controller, @Service o @Configuration.

### Clases o paquetes creados

No se requiere de la creación de ninguna clase o paquete adicional.

### Ventajas alcanzadas al aplicar el patrón

1. Abolimos la repetición de creaciones de clases innecesarias.
2. Nos despreocupamos sobre la instanciación de clases.
3. Disminuye el acoplamiento.

## Patrón: Service Layer

### Tipo: Diseño

### Contexto de Aplicación

En este patrón, tratamos de añadir una capa intermedia entre la capa de lógica de negocio y la capa de recursos, estableciendo una comunicación entre ellas.

### Clases o paquetes creados

Todos los servicios de la aplicación, siendo el paquete concreto: org.springframework.samples.petclinic.service.

### Ventajas alcanzadas al aplicar el patrón

1. Aumenta en gran medida la cohesión, pudiendo variar las capas externas.

## Patrón: Domain Model

### Tipo: Diseño

### Contexto de Aplicación

En este patrón tratamos de, usando objetos, incorporar tanto lógica de comportamiento como datos en sí.

### Clases o paquetes creados

Todos las clases que conforman parte del modelo de la aplicación, siendo el paquete concreto: org.springframework.samples.petclinic.model.

### Ventajas alcanzadas al aplicar el patrón

1. Permite implementar lógicas de negocio más complejas.
2. Soportado por la mayoría de frameworks.

## Patrón: Repository Pattern

### Tipo: Diseño

### Contexto de Aplicación

En este patrón, tratamos de encapsular la lógica necesaria para tener acceso a la base de datos. Centralizan la funcionalidad de acceso a datos comunes. Un repositorio realiza las tareas de un intermediario entre los niveles de modelo de dominio y asignación de datos.

### Clases o paquetes creados

Todos los repositorios de la aplicación, siendo el paquete concreto: org.springframework.samples.petclinic.repository.

### Ventajas alcanzadas al aplicar el patrón

1. Proporciona un mejor mantenimiento.
2. Desacopla la infraestructura o tecnología que se usa para acceder a bases de datos desde el nivel de modelo de dominio.

## Patrón: Template View

### Tipo: de Diseño

### Contexto de Aplicación

Hemos utilizado este patrón de cara a la creación de las vistas en la carpeta jsp, ya que Spring usa jsp, que requiere este patrón de diseño.

### Clases o paquetes creados

Clases jsp creadas en la ruta src/main/webapp/WEB-INF/jsp

### Ventajas alcanzadas al aplicar el patrón

Es el patrón de vista que se integra mejor con Spring. Al ser más parecido a html, es más sencillo de modificar una parte de la vista sin tener que cambiar demasiado código.

## Patrón: Data Mapper

### Tipo: de Diseño

### Contexto de Aplicación

Hemos utilizado este patrón de cara a la persistencia de datos, en el mapeo de entidades.

Una de las etiquetas en las que se refleja es @Entity

### Clases o paquetes creados

Clases jsp creadas en la ruta src/main/webapp/WEB-INF/jsp

### Ventajas alcanzadas al aplicar el patrón

Permite la independencia entre los objetos y la base de datos, es decir, aumenta la cohesión.

# Decisiones de diseño

En esta sección describiremos las decisiones de diseño que se han tomado a lo largo del desarrollo de la aplicación que vayan más allá de la mera aplicación de patrones de diseño o arquitectónicos.

## Decisión 1: Añadir preguntas a un examen

### Descripción del problema:

Nos planteamos la adición de preguntas a un examen en la página. El problema reside en que no podemos saber de antemano el número de preguntas que tendrá ni de qué tipo serán, ya que no hay una respuesta fija.

### Alternativas de solución evaluadas:

Alternativa 1.a: Añadir las preguntas en la página de creación o edición del examen.

En la página de creación/edición del examen se colocaría un botón de añadir pregunta. Mediante Javascript se irían añadiendo secciones al formulario y se pasarán a la base de datos desde un único formulario.

**Ventajas**:

* Es la forma más rápida si se quieren añadir varias preguntas.
* Se realizan menos accesos a la base de datos.

**Inconvenientes**:

* Encontramos, el procedimiento, complejo de implementar, ya que no tenemos todos los conocimientos necesarios de las tecnologías requeridas para llevarlo a cabo.
* Puede resultar confuso para la navegabilidad del usuario medio que va a utilizar la aplicación.

Alternativa 1.b: Crear una sección enfocada exclusivamente a las preguntas del examen, en las que se puedan añadir y editar las preguntas de una en una.

**Ventajas:**

* Para un usuario promedio, al estar dividido en secciones más específicas, puede ser más sencillo de comprender.
* La implementación es sencilla.
* Permite editar una pregunta sin tener que modificar el resto del examen.
* Se pueden añadir preguntas de distinto tipo a un examen más fácilmente.

**Inconvenientes:**

* Múltiples accesos a la base de datos.
* Puede ser lento si se desean añadir muchas preguntas.

### Justificación de la solución adoptada

Alternativa 1b: Creemos que gran parte del público estará conformada por personas que no están acostumbradas a trabajar con sistemas de información de uso avanzado. Por tanto, prima que el funcionamiento de nuestra aplicación sea sencillo. Además, debido al plazo del que disponemos, nos hemos decantado por la solución que podíamos conocer mejor a primera vista.

## Decisión 2: Añadir opciones a una pregunta

### Descripción del problema:

El problema es similar que surge al añadir preguntas a un examen. Queremos añadir opciones a una pregunta. El problema reside en que no tienen un número fijo de opciones.

### Alternativas de solución evaluadas:

Alternativa 1.a: Añadir las opciones en el momento en el que se crea o edita una pregunta.

En el formulario de creación de pregunta, habrá un botón “Añadir opción” que, cada vez que sea pulsado, añadirá de manera asíncrona un formulario para una nueva opción.

**Ventajas**:

* Es la forma más rápida si se quieren añadir varias opciones.
* Se realizan menos accesos a la base de datos.

**Inconvenientes**:

* Al igual que con la decisión de añadir preguntas a exámenes, encontramos, el procedimiento, complejo de implementar. Presenta dificultades similares a la anterior decisión.
* Puede resultar confuso para la navegabilidad del usuario medio que va a utilizar la aplicación.

Alternativa 1.b: En la página donde se listan las preguntas del examen, al lado de cada pregunta, se muestra el botón “Añadir opción”. Este botón llevará a un formulario exclusivo para esa opción.

**Ventajas:**

* Para un usuario promedio, puede ser un procedimiento fácil de entender.
* La implementación es sencilla.
* Permite editar una opción sin alterar el resto de preguntas, de manera que es menos probable modificar otras opciones o preguntas por error.
* Es una solución más coherente a la ya decidida para añadir preguntas al examen.

**Inconvenientes:**

* Múltiples accesos a la base de datos.
* Puede ser lento si se desean añadir muchas opciones.

### Justificación de la solución adoptada

Alternativa 1b: Creemos que gran parte del público estará conformada por personas que no están acostumbradas a trabajar con sistemas de información de uso avanzado. Por tanto, prima que el funcionamiento de nuestra aplicación sea sencillo. Además, debido al plazo del que disponemos, nos hemos decantado por la solución que podíamos conocer mejor a primera vista.

## Decisión 3: Realización de examen

### Descripción del problema:

Nos planeamos el problema de, si tenemos un examen con sus correspondientes preguntas, cómo se puede realizar este, de forma que sea cómodo para el usuario y quede constancia de lo que respondió.

### Alternativas de solución evaluadas:

Alternativa 1.a: Poner todas las preguntas en una misma página.

Las preguntas se dispondrían en una única página y a la base de datos, una vez pulsado un botón “enviar respuesta”, se le pasa una lista con las respuestas seleccionadas.

**Ventajas**:

* Se realizan menos accesos a la base de datos.

**Inconvenientes**:

* La dificultad, a nivel programático, aumentaría a la hora de gestionar las respuestas.
* Ante un error de conexión, como el envío no se realiza hasta que no se finaliza el examen, se podrían perder todas las respuestas que el usuario había dado hasta el momento.

Alternativa 1.b: Mostrar las preguntas de una en una, cada una en una página diferente. En la base de datos, las respuestas se irían añadiendo al intento de una en una.

**Ventajas:**

* La implementación es más sencilla que la anterior alternativa.
* El usuario es más consciente del número de pregunta que ya ha respondido y no tiene que hacer scroll para ver todas las preguntas.
* Ante un error de conexión, las preguntas que ya se habían contestado estarán almacenadas en la base de datos y, por tanto, no desaparecerán las respuestas.

**Inconvenientes:**

* Múltiples accesos a la base de datos.

### Justificación de la solución adoptada

Alternativa 1b: Hemos seleccionado esta opción principalmente porque nos facilita el tratamiento de datos y porque disminuye las posibilidades de pérdida de información. Además, creemos que, de esta manera, podemos hacer la vista del usuario más amigable que mediante la opción 1.a.

## Decisión 4: Tipos de pregunta ampliables

### Descripción del problema:

A pesar de que principalmente se plantean preguntas de tipo test o de redactar una corta respuesta, nuestra cliente pide que se puedan añadir otro tipo de preguntas en el futuro.

### Alternativas de solución evaluadas:

Alternativa 1.a: Poner un atributo a las preguntas de tipo enum, que podrá tomar los valores de los distintos tipos de preguntas existentes en la aplicación.

**Ventajas**:

* Se puede conocer de manera mucho más rápida el tipo de la pregunta tan sólo comprobando el valor del enum.

**Inconvenientes**:

* En la clase pregunta habría que poner numerosos atributos para que pudiera ajustarse a todos los tipos de pregunta, no usándose la mayoría y reservando más espacios de la cuenta que estarán a null.
* Para añadir nuevos tipos de pregunta, hay que añadir todos los atributos distintos de esta nueva pregunta a la clase, teniendo que modificar mayor cantidad de código.
* Hay que modificar la lógica que siguen todas las preguntas cada vez que se quiera añadir un nuevo tipo de pregunta.

Alternativa 1.b: Hacer una jerarquía de clases de tipos de pregunta con una única clase, “TipoTest”. De esta forma, será sencillo añadir otro tipo de pregunta. Por defecto, se entiende que, si la pregunta no tiene ningún tipo de la jerarquía asociado, es de redacción.

**Ventajas:**

* Para añadir un nuevo tipo de pregunta, tan sólo hay que añadir una nueva clase a la jerarquía de tipos de pregunta con los atributos exclusivos de su tipo.
* Es más eficiente, ya que la lógica que hay que implementar es más simple.

**Inconvenientes:**

* Hay más relaciones intermedias y esto puede ser más complejo la hora de implementar nuevas funcionalidades.

### Justificación de la solución adoptada

Alternativa 1b: Hemos seleccionado esta opción porque presenta más facilidades a la hora de ampliar los tipos de pregunta disponibles y resulta más ventajosa que la primera alternativa.

## Decisión 5: Recursividad en citas de comentarios

### Descripción del problema:

En nuestra aplicación, los comentarios de un hilo tienen la característica de citar o responder a otro. La idea era que el comentario que responde o cita al original apareciese justo debajo, con una marca que representase que es una cita o respuesta. La cuestión es que esa cita o respuesta es también un comentario, y puede ser citado o respondido de igual forma. Esto quiere decir que al mostrar la lista de comentarios, se debían mostrar los comentarios y sus respuestas de forma recursiva.

### Alternativas de solución evaluadas:

Alternativa 5.a: Importar librerías de jsp que implementasen la función recursiva.

**Ventajas**:

* Es simple y rápido (sólo requiere una importación y aplicar los métodos correspondientes).

**Inconvenientes**:

* Requeriría que nos familiarizáramos con una librería que no habíamos usado antes.
* Había escasa información de la librería en línea.

Alternativa 5.b: Hacer uso de Javascript para hacer llamadas recursivas.

**Ventajas**:

* Javascript tiende a dar muy buen manejo de muchas situaciones.

**Inconvenientes**:

* No había garantía de que esta solución fuese a funcionar.
* No habíamos visto esta alternativa entre las múltiples propuestas que hay en foros en internet, por lo que seguramente habría algún inconveniente más.

Alternativa 5.c: Hacer llamadas a un mismo archivo jsp cambiando los valores del model para distinguir las distintas llamadas, emulando un efecto de recursividad.

**Ventajas**:

* Es la solución que más se recomienda en internet, y ya había una plantilla de código en stack overflow y otras páginas.
* Es muy visual y fácil de comprender.

**Inconvenientes**:

* Algunas de las características que tendría una función recursiva en un lenguaje que admita esta funcionalidad, por ejemplo, Java, se pierden, dando lugar a menor manejo de la situación de recursividad.

### Justificación de la solución adoptada

Seleccionamos la alternativa 5.c, porque era la más recomendada en internet (de hecho, en internet con una rápida búsqueda se puede copiar la estructura básica de la recursividad siguiendo este método) y rápidamente entendimos cómo se realizaba la recursividad sin necesidad de adaptarnos a una nueva librería o a aventurarnos a usar más lenguajes de programación de forma innecesaria (javascript).

## Decisión 6: Mostrar número de notificaciones del usuario autenticado a tiempo real.

### Descripción del problema:

Nuestra aplicación da soporte a un sistema de notificaciones. Para que este sea útil, vimos necesario que un usuario registrado pudiera ver el número de notificaciones que tiene en cualquier momento de la página, y que este se actualizase de forma constante. Por ello, decidimos mostrar este número en el header de la página, cuyo código se encuentra en el archivo menu.tag. El problema vino al intentar cambiar este número a tiempo real.

### Alternativas de solución evaluadas:

Alternativa 6.a: Hacer uso de anotaciones de Spring que ejecutasen un método java cada cierto tiempo.

**Ventajas**:

* Es simple de implementar.

**Inconvenientes**:

* Como prácticamente cualquier funcionalidad de Spring, el principal inconveniente es que no es fácil qué está ocurriendo detrás de cada anotación, y tampoco se puede editar ese código para adaptarlo a nuestra situación.
* El código no se ejecutaba en el orden que teníamos pensado, y daba lugar a bastantes errores.
* Por algún motivo, hacer uso de esta anotación impedía que funcionasen los Junit tests.

Alternativa 6.b: Hacer uso de Javascript y Ajax para hacer llamadas a una función Java a tiempo real.

**Ventajas**:

* El uso de Javascript para cambios a tiempo real en la página sin necesidad de recargar la página es lo más recomendado.

**Inconvenientes**:

* No había garantía de que esta solución fuese a funcionar.
* No sabíamos si podría provocar conflictos con Spring.

Alternativa 6.c: Crear un atributo en la clase del usuario que fuese el número de notificaciones, y hacer una llamada a este atributo en el código de menu.tag, de igual forma que se coge el nombre de usuario del usuario autenticado.

**Ventajas**:

* Muy sencillo de implementar en jsp.

**Inconvenientes**:

* No habría forma de hacer que se actualizase a tiempo real a menos que el usuario recargase la página.
* Tendría que haber un intercambio de información entre las creaciones de las notificaciones y el número que representa el número de notificaciones del usuario, lo que complicaría otros códigos de la aplicación.

### Justificación de la solución adoptada

Probamos las tres soluciones, empezando por 6.a, y terminando por 6.b, que es la que finalmente elegimos. 6.a quedó rechazada al dar problemas con las pruebas unitarias. La alternativa 6.c tampoco nos valía porque no se podía hacer una llamada a cualquier atributo del usuario con tanta facilidad como pensábamos que se podría. La solución 6.b es la última que se nos ocurrió, y fue sencillo de implementar y no dio ningún problema.

## Decisión 7: Niveles en los comentarios.

### Descripción del problema:

Relacionada con la decisión 5, este problema vuelve a tener que ver con comentarios y citas. En este caso, el problema estaba en cómo distinguir en qué nivel del árbol se encontraba cada comentario (donde la raíz del árbol sería el comentario que no citaba a ninguno, las ramas los comentarios que no han sido citados, y los demás nodos los comentarios que citan a otro comentario y que luego son citados por otro).

### Alternativas de solución evaluadas:

Alternativa 7.a: En la llamada recursiva, aumentar en uno el nivel de un comentario por cada llamada.

**Ventajas**:

* Alternativa simple y lógica. Prácticamente igual que como hemos programado siempre las funciones recursivas en Java.

**Inconvenientes**:

* Debido a que la alternativa escogida en la decisión 5 fue la de hacer llamadas recursivas a un mismo archivo jsp, muchas funcionalidades lógicas de una función recursiva no funcionaban como uno se esperaría, dificultando en gran medida esta alternativa.

Alternativa 7.b: Crear un atributo en comentario llamado nivel, que le es asignado en su creación en función de si cita o no a un comentario, y en función del nivel del comentario que esté citando, si es que cita a algún comentario.

**Ventajas**:

* Solucionaría el problema sin tener que tocar la función recursiva.

**Inconvenientes**:

* No es la respuesta más lógica al problema.
* Hay que añadir atributos que en principio no serían necesarios ya que son derivados de otros atributos de la clase (la clase comentario tiene un atributo que representa una relación de cita entre comentarios, a través del cual se podría deducir el nivel del comentario sin necesidad de hacer un atributo nuevo).

### Justificación de la solución adoptada

En primer lugar, elegimos aplicar la alternativa 7.a, pero el inconveniente de esa alternativa (el que la recursividad no funcionaba como en otros entornos de programación con los que hemos trabajado) no nos permitió conseguir los resultados que queríamos. La alternativa 7.b fue un poco más laboriosa de lo que hubiera sido la 7.a (si no hubiese sido por su inconveniente), pero funcionó bien, así que fue la que finalmente escogimos.

## Decisión 8: Implementación de los distintos tipos de usuario.

### Descripción del problema:

Nuestra aplicación hace uso de distintos roles para los usuarios. Estos roles permiten acceder a determinadas partes de la página e incluso a ciertas funcionalidades. La implementación de esta funcionalidad se podía hacer de distintas formas.

### Alternativas de solución evaluadas:

Alternativa 8.a: Crear un atributo en la clase del usuario que determinase su rol.

**Ventajas**:

* Alternativa sencilla de implementar.

**Inconvenientes**:

* No podrían aplicarse operaciones demasiado complejas a los distintos privilegios que otorga cada rol.

Alternativa 8.b: Crear una clase para cada atributo, y crear una relación entre la clase usuario y esas clases.

**Ventajas**:

* Permite hacer operaciones más complejas para el comportamiento de cada rol (restricciones más complejas, relaciones exclusivas de cada rol...).

**Inconvenientes**:

* La implementación no es trivial.

Alternativa 8.c: Hacer uso de la clase Authorities ya existente en el proyecto petclinic, que además ya tiene una relación con User (que a su vez tiene una relación con nuestra clase Usuario).

**Ventajas**:

* Solución más sencilla que la 6.b y no mucho más compleja que la 6.a.
* Permite hacer uso de ciertas funcionalidades que Spring tiene reservadas para esta clase, que está estrechamente ligada al sistema de seguridad de Spring.
* Hay soporte en línea para los distintos problemas que pudieran surgir.

**Inconvenientes**:

* Podría haber comportamiento inesperado si no supiéramos manejar bien las funcionalidades de Spring.
* No hay garantía de que pudiéramos implementar operaciones de mayor complejidad para los permisos de los distintos roles. Dependeríamos de las funciones internas de Spring.

### Justificación de la solución adoptada

La solución 8.c fue la que elegimos, porque no era demasiado difícil de implementar y podíamos buscar soluciones en línea para los posibles problemas que pudiese dar. Además, creíamos que podíamos hacer uso de ciertas funcionalidades de Spring para cambiar la vista con facilidad en función del rol del usuario, pero al final cuando intentamos usar estas funcionalidades no nos sirvieron, pero encontramos una forma alternativa de hacer lo mismo.

## Decisión 9: Borrado de usuarios

### Descripción del problema:

Queremos que un usuario pueda cancelar su cuenta en la página cuando lo considere necesario sin que esto cause problemas al resto de las entidades.

### Alternativas de solución evaluadas:

Alternativa 1.a: Eliminar el usuario de la base de datos

**Ventajas**:

* Se ahorra espacio en la base de datos, de manera que no hay sobrecarga de información innecesaria.
* Es fácil de implementar, ya que no sigue ninguna lógica de negocio compleja.

**Inconvenientes**:

* Un usuario en la base de datos tiene muchas dependencias, lo cual puede causar muchos problemas durante el borrado.
* Borrar un usuario implica que se pueda borrar información útil de otras entidades.
* Hay que gestionar el flujo de eliminación para que no se alteren otras partes de la aplicación.

Alternativa 1.b: Añadir un atributo al usuario que indique si tiene la cuenta habilitada para su uso.

**Ventajas:**

* Si un usuario decide eliminar su cuenta, podría retomarla en un futuro en el mismo estado en el que la deshabilitó.
* Permite mantener coherencia con las dependencias del usuario, de forma que no se pierde información.
* Aunque se deshabilite la cuenta, se puede aprovechar la información para aplicar algoritmos estadísticos para sacar conclusiones que ayuden a tomar decisiones a la organización, por ejemplo, el uso de big data.
* Puede ayudar, aparte de a hacer cuentas invisibles, a penalizar a usuarios por comportamiento indebido.

**Inconvenientes:**

* Mayor carga de información en la base de datos.
* Mayor complejidad de implementación debido a la lógica de negocio compleja.

### Justificación de la solución adoptada

Alternativa 1b: Hemos seleccionado esta opción porque hay una gran diferencia entre la cantidad de ventajas e inconvenientes que ofrece con respecto a la otra alternativa.

Decisión 10: Validaciones

### Descripción del problema:

Queremos validar que los datos de los formularios contienen campos correctos para la creación o edición de nuevas entidades. Teniendo en cuenta que existen varias estrategias para hacerlo, el problema reside en cuál de ellas escoger.

### Alternativas de solución evaluadas:

Alternativa 1.a: Únicamente a través de anotaciones definidas por Spring.

**Ventajas**:

* Soporta validaciones sintácticas
* Requiere de poca implementación, ya que vienen preconfiguradas por Spring.

**Inconvenientes**:

* No soportan validaciones semánticas, es decir, no podríamos implementar reglas de negocio complejas.

Alternativa 1.b: Crear una clase que realice la validación de los distintos campos y, además, utilizar las anotaciones predefinidas de Spring.

**Ventajas:**

* Soporta validaciones tanto semánticas como sintácticas.
* Es fácil de implementar con Spring.
* Podemos utilizar los métodos del BindingResult
* Favorece la cohesión de la aplicación

**Inconvenientes:**

* Un controller sólo puede tener asociado una clase validación, por tanto habría que planear previamente cómo validar todos los campos sin se utilizan en varias entidades a la vez.
* Hay que incluir de manera manual la clase validación de una entidad en cada una de las clases en las que se use esta.

Alternativa 1.c: Validación a través de excepciones personalizadas.

**Ventajas:**

* Soporta validaciones tanto semánticas como sintácticas.
* Lanzar excepciones en lenguajes orientados a objetos, en este caso Java, es fácil y común.
* Facilita el poder proporcionarle al usuario el motivo detallado de los fallos de las validaciones.

**Inconvenientes:**

* Puede complicar los códigos, ya que el tratamiento de las excepciones se debe intercalar en el código donde requiere utilizar la validación.
* No puede haber excepciones simultáneas, por lo que podrían quedarse campos sin validar.
* Ejecutar diferentes métodos sin comprobar excepciones puede dar lugar a inconsistencias en la base de datos.

### Justificación de la solución adoptada

Alternativa 1b: En general, hemos seleccionado esta opción a la hora de validar porque hemos decidido dar peso a la cohesión en nuestra aplicación y porque es sencilla de implementar.

Sin embargo, debido a la poca complejidad de algunas clases (como PDF), en ocasiones hemos usado la primera opción.