DP1 2020-2021

Documento de Diseño del Sistema

Proyecto codeUS

https://github.com/gii-is-DP1/dp1-2020-g2-06

Miembros:

- Aparicio Ortiz, Jesús
- Barranco Ledesma, Alejandro
- Brincau Cano, David
- Granero Gil, Victor Javier
- Ostos Rubio, Juan Ramón

Tutor: Irene Bedilia Estrada Torres

GRUPO G2-06

Versión 2

10/01/2121

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
07/01/2021	V1	• Creación del documento	2

10/01/2021	V2	• Añadido diagrama de dominio/diseño • Explicación de la aplicación del patrón caché	3

Contents

Historial de versiones 1
Introducción 4
Diagrama(s) UML: 5
Diagrama de Dominio/Diseño 5
Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios) 6
Patrones de diseño y arquitectónicos aplicados 6
Decisiones de diseño 8
Decisión X 6
Descripción del problema: 6
Alternativas de solución evaluadas: 6
Justificación de la solución adoptada 6

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.

CodeUS es un portal de programación competitiva con objeto de animar e introducir a los alumnos de la Universidad de Sevilla en este campo. La actividad principal de la página es resolver problemas de programación competitiva para conseguir puntos y así subir puntos en el ranking.

El alumnado encontrará en CodeUS problemas de programación de todo tipo (numéricos, grafos, recursión...) para poder enviar soluciones y comprobar su validez con el juez online. Cada problema tiene asociados unos puntos, los cuales serán sumados a los puntos del alumno cuando este realice un envío que el juez considere válido. De esta forma se anima al alumnado a mejorar sus habilidades de programación compitiendo entre ellos. También prepara a los alumnos para competiciones oficiales del ámbito, como Ada Byron, Las12Uvas, Google Code Jam...

El comportamiento de la página es similar a http://aceptaelreto.com pero añadiéndole un carácter más competitivo, tal y como hacen los videojuegos de moda actuales (rankings, puntos, rangos…).

También le añadimos valor educativo al incluir una figura de tutor, que ayuda y guía en la resolución de problemas. Además, publica noticias y artículos resolviendo problemas retirados para la consulta de los alumnos.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

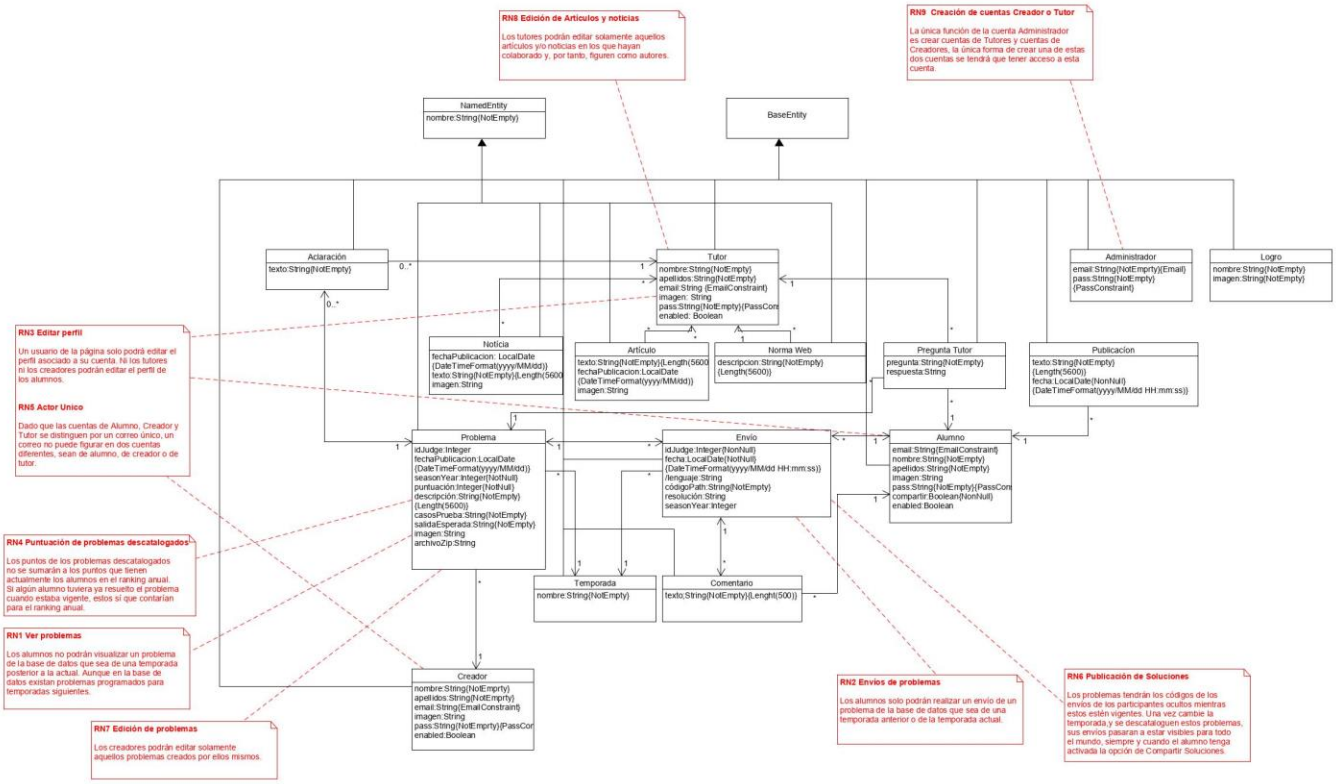
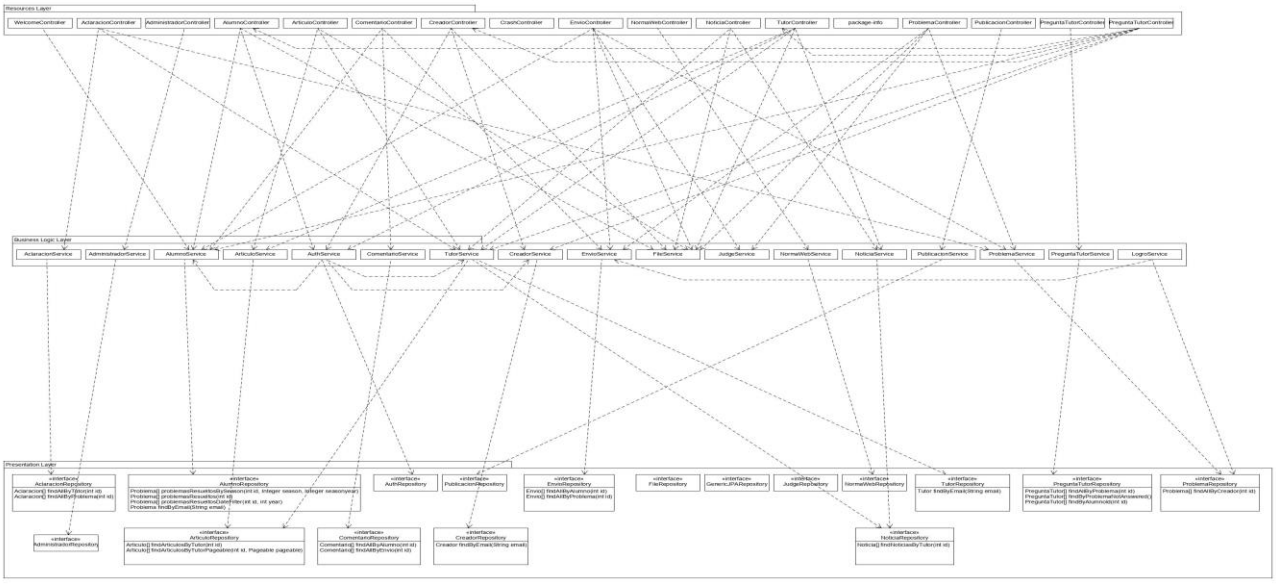


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: Arquitectónico

Contexto de Aplicación

Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

El patrón se ha aplicado en toda la aplicación a excepción del tema de seguridad que se gestiona aparte.

Clases o paquetes creados

Para el modelo se crean los distintos modelos, servicios y repositorios en los respectivos paquetes:

- org.springframework.samples.petclinic.model
- org.springframework.samples.petclinic.service
- org.springframework.samples.constraint
- org.springframework.samples.constraint.validators
- org.springframework.samples.petclinic.repository
- org.springframework.samples.petclinic.domjudge
- org.springframework.samples.petclinic.utils

*El paquete “domjudge” contiene todo lo relacionado con el modelo y funcionalidades del juez que valida los envíos de los problemas. La clase “Utils.java” dentro del paquete “utils” contiene

funcionalidades. Los paquetes “constraint” y “constraint.validators” contienen validaciones para los modelos.

Para el controlador se crean los distintos controladores dentro del paquete:

- org.springframework.samples.petclinic.web

Para la vista se crean los distintos “.jsp” dentro de la carpeta “jsp”, además de los “.less” para dar estilo a la página.

Ventajas alcanzadas al aplicar el patrón

Este patrón era interesante ya que permite distribuir los datos, la metodología y la interfaz gráfica en distintos componentes. Cada uno de estos componentes se encarga específicamente de una tarea: los datos de una aplicación (Modelo), la interfaz del usuario (Vista) y la lógica de control (Controlador). Las principales ventajas de utilizar este patrón arquitectónico son la alta cohesión y el bajo acoplamiento que hay entre los distintos componentes del sistema, además de la escalabilidad del sistema al separarlo en distintos componentes. Por otro lado, los errores se pueden solucionar de una manera más sencilla, ya que al haber bajo acoplamiento se pueden solucionar dichos errores de manera más rápida sin necesidad de arreglarlos en todo el sistema, si no únicamente en el componente con dicho problema.

Patrón: Capas

Tipo: Diseño

Contexto de Aplicación

Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

Clases o paquetes creados

Para la capa de lógica se crean los distintos modelos, servicios y repositorios en los respectivos paquetes:

- org.springframework.samples.petclinic.model
- org.springframework.samples.petclinic.service
- org.springframework.samples.constraint
- org.springframework.samples.constraint.validators
- org.springframework.samples.petclinic.repository
- org.springframework.samples.petclinic.domjudge
- org.springframework.samples.petclinic.utils

*El paquete “domjudge” contiene todo lo relacionado con el modelo y funcionalidades del juez que valida los envíos de los problemas. La clase “Utils.java” dentro del paquete “utils” contiene

funcionalidades. Los paquetes “constraint” y “constraint.validators” contienen validaciones para los modelos.

Para la capa de lógica de negocios se crean los distintos controladores dentro del paquete:

- org.springframework.samples.petclinic.web

Para la capa de presentación se crean los distintos “.jsp” dentro de la carpeta “jsp”, además de los “.less” para dar estilo a la página.

Ventajas alcanzadas al aplicar el patrón

Era interesante aplicar el patrón de diseño por capas para desacoplar las distintas partes de nuestro proyecto, de esta manera se puede modularizar más y con ello agrupar en cada capa por una funcionalidad más específica. Además, aplicando este patrón, en caso de error, se identificaría más rápido dicho error, pues fallaría una capa y no el sistema entero.

Decisiones de diseño

Decisión 1: Importación de datos reales para demostración

Descripción del problema:

Como grupo nos gustaría poder hacer pruebas con un conjunto de datos reales suficientes, porque resulta más motivador. El problema es al incluir todos esos datos como parte del script de inicialización de la base de datos, el arranque del sistema para desarrollo y pruebas resulta muy tedioso.

Alternativas de solución evaluadas:

Alternativa 1.a: Incluir los datos en el propio script de inicialización de la BD (data.sql).

Ventajas:

- Simple, no require nada más que escribir el SQL que genere los datos.

Inconvenientes:

- Ralentiza todo el trabajo con el sistema para el desarrollo.
- Tenemos que buscar nosotros los datos reales

Alternativa 1.b: Crear un script con los datos adicionales a incluir (extra-data.sql) y un controlador que se encargue de leerlo y lanzar las consultas a petición cuando queramos tener más datos para mostrar.

Ventajas:

- Podemos reutilizar parte de los datos que ya tenemos especificados en (data.sql).
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

Inconvenientes:

- Puede suponer saltarnos hasta cierto punto la división en capas si no creamos un servicio de carga de datos.
- Tenemos que buscar nosotros los datos reales adicionales

Alternativa 1.c: Crear un controlador que llame a un servicio de importación de datos, que a su vez invoca a un cliente REST de la API de datos oficiales de XXXX para traerse los datos, procesarlos y poder grabarlos desde el servicio de importación.

Ventajas:

- No necesitamos inventarnos ni buscar nosotros lo datos.
- Cumple 100% con la división en capas de la aplicación.
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

Inconvenientes:

- Supone mucho más trabajo.
- Añade cierta complejidad al proyecto

Justificación de la solución adoptada

Como consideramos que la carga inicial de datos no es excesivamente grande y se puede manejar bastante bien manualmente, seleccionamos la alternativa de diseño 1.a.

Decisión 2: Almacenar fotos

Descripción del problema:

Como la aplicación será utilizada tanto por alumnos, tutores, como creadores, entonces necesitamos almacenar dichas fotos de alguna manera en el sistema. Además, necesitamos guardar otras fotos para distintas entidades como problema, articulo o noticia.

Alternativas de solución evaluadas:

Alternativa 1.a: Guardar la foto como un tipo File en el modelo de cada entidad

Ventajas:

- Sería lo más rápido a implementar, además de lo más sencillo.

Inconvenientes:

- La base de datos no soporta objetos de tipo File.

Alternativa 1.b: Guardar la foto como un array de byte en el modelo de cada entidad

Ventajas:

- Sería una solución bastante rápida.

Inconvenientes:

- Se tendría que realizar la conversión de dicha imagen a bytes cada vez que se quiera guardar y volver a convertirla a imagen cuando se quiera mostrar en la vista.
- Tener un array de byte en vez de un path referenciando a la imagen es poco práctico.

Alternativa 1.c: Almacenar las imágenes por carpetas en función de las entidades dentro de la carpeta "static/resources/images" , de este modo las imágenes se guardarán de manera organizada y si hubiera un error se detectaría rápidamente en qué lugar surge ese error. Además, para referenciar las imágenes, se guardará el path donde están guardadas las imágenes en la entidad correspondiente, de este modo se podrá obtener dicho archivo a través del path.

Ventajas:

- Mayor organización
- Mayor Facilidad a la hora de mostrar las imágenes en la vista, ya que las imágenes están almacenadas en la carpeta "static"

Inconvenientes:

- Más difícil de gestionar
- Añade cierta complejidad al proyecto

Alternativa 1.d: Almacenar las imágenes con una url y no guardar ninguna imagen como archivo.

Ventajas:

- Sencillez a la hora de guardar las imágenes

Inconvenientes:

- No se puede colgar cualquier imagen

Justificación de la solución adoptada

Como trabajar de manera más organizada y tener las imágenes divididas en carpetas lo consideramos interesante para el futuro desarrollo de la aplicación ya que proporciona mayor escalabilidad al sistema, seleccionamos la alternativa de diseño 1.c.

Decisión 3: Nombre de las imágenes

Descripción del problema:

Las imágenes que se suban a la aplicación no pueden estar repetidas

Alternativas de solución evaluadas:

Alternativa 1.a: Dejar el nombre original del archivo

Ventajas:

- Simple, no requiere de cambios

Inconvenientes:

- Si dos personas suben una imagen con el mismo nombre salta una excepción

Alternativa 1.b: Poner como nombre de las imágenes el valor de un contador que va aumentando 1 cada vez que se suba una imagen

Ventajas:

- Sencillo, solo se requiere de un contador

Inconvenientes:

- Se pueden dar inconsistencias de datos

Alternativa 1.c: Utilizar de nombre de la imagen un identificador que se compone de: "AñoActual+MesActual+DíaActual+HoraActual+MinutoActual+SegundoActual+NanoSegundoActual"

Ventajas:

- Se genera un identificador de manera simple
- Prácticamente imposible de que dos imágenes tengan el mismo identificador

Inconvenientes:

- El identificador es bastante largo
- En un caso muy remoto puede llegar a saltar una excepción debido a que dos personas han hecho un envío al mismo tiempo exactamente

Justificación de la solución adoptada

Como consideramos que el hecho de que no existan inconsistencias en el sistema es algo fundamental, hemos considerado que la solución 1.c. es la mejor a implementar.

Decisión 4: Fecha Inicio y fin de competiciones

Descripción del problema:

No existe un tipo de input en html o jsp para obtener un objeto LocalDateTime, ya que se necesita fecha, hora y minuto.

Alternativas de solución evaluadas:

Alternativa 1.a: Utilizar librería Time

Ventajas:

- Justo contiene un objeto con la fecha, hora y minuto.

Inconvenientes:

- El tipo Time no se puede implementar para bindear automáticamente desde la vista al modelo.

Alternativa 1.b: Utilizar tipo LocalDate y LocalTime para obtener la fecha por un lado y la hora y minuto por otro

Ventajas:

- Se bindea todo sin problema

Inconvenientes:

- Hay que tener el doble de atributos para el mismo atributo

Justificación de la solución adoptada

Ya que la ventaja de poder bindear automáticamente es muy grande, hemos preferido dividir la fecha y su hora en fecha como un atributo y hora como otro atributo en el modelo, seleccionamos la alternativa de diseño 1.b.

Decisión 5: Ranking de putuación

Descripción del problema:

Como vamos a mostrar un ranking de los alumnos necesitamos obtener los puntos que lleva cada alumno tanto en la temporada actual como en el año actual.

Alternativas de solución evaluadas:

Alternativa 1.a: Añadir atributos de puntuación en la entidad alumno

Ventajas:

- Simplificación a la hora de realizar las consultas a la base de datos.

Inconvenientes:

- Almacenamiento de datos innecesarios.

Alternativa 1.b: Obtener los puntos a través de una consulta un tanto más complicada.

Ventajas:

- No se necesita almacenar nada más en la entidad alumno

Inconvenientes:

- Se necesita realizar una consulta a la base de datos muy compleja

Justificación de la solución adoptada

Ya que queremos realizar todo de la manera más correcta y siguiendo las buenas prácticas, preferimos no almacenar cosas innecesarias en la base de datos, ya que se pueden obtener dichos puntos a través de una consulta un tanto más compleja, por ellos hemos seleccionado la alternativa 1.b

Decisión 6: Declaración de validadores

Descripción del problema:

Como grupo nos gustaría poder hacer validaciones ha determinados atributos de algunas entidades, que no llega a cubrir las diferentes etiquetas como @NotEmpty, @NonNull, etc...

Alternativas de solución evaluadas:

Alternativa 1.a: Crear nuestros propios validadores.

Ventajas:

- Es un método que nos permite, fácilmente, implementar etiquetas en varios modelos.

Inconvenientes:

- Necesidad realizar consultas a la base de datos.
- Puede llegar a ser más complejo

Alternativa 1.b: Gestionarlo a través de excepciones.

Ventajas:

- No necesita consultar en ningún momento la base de datos

Inconvenientes:

- Es muy costoso, en términos de tiempo, tener que implementarlo en los diferentes controladores
- Hace que el código sea más difícil de leer

Justificación de la solución adoptada

Como las validaciones que queremos realizar son comunes para la mayoría de las entidades hemos decidido hacer nuestros propios validadores ya que nos será más útil a la hora de rápidamente aplicarlo a las diferentes entidades 1.a.

Decisión 8: Aplicación externa para juzgar los envíos

Descripción del problema:

Cada envío debe evaluarse para conseguir el veredicto y saber si ha sido aceptado o por el contrario presenta algún problema de límite de tiempo, compilación, error en tiempo de ejecución...

Alternativas de solución evaluadas:

Alternativa 1.a: Evaluar en nuestra propia aplicación cada script.

Ventajas:

- No se necesita usar aplicaciones externas ni desplegarlas.

Inconvenientes:

- El diseño de un juez es algo más complejo de lo que parece, sobre todo para evaluar correctamente los tiempos de ejecución.

Alternativa 1.b: Utilizar Mooshak como juez.

Ventajas:

- Desplegando un solo contenedor docker podemos usar este juez a través de su API.

Inconvenientes:

- Presenta problemas al interpretar código C (en la versión para docker).
- No cuenta con llamadas a la API para añadir nuevos problemas.

Alternativa 1.c: Utilizar DOMJudge como juez.

Ventajas:

- API muy completa, posibilita subir problemas. Diseñado de una forma modular (base de datos, server y jueces) de forma que es totalmente escalable si se necesitan más recursos en momentos de uso masivo.

Inconvenientes:

- Es necesario desplegar al menos 3 contenedores.

Justificación de la solución adoptada

Ya que necesitamos subir problemas y que el juez de Mooshak presenta problemas en algunos lenguajes decidimos usar DOMJudge. 1.c

Decisión 9: Tener estadisticas de nuestro problema(graficas)

Descripción del problema:

Al realizar un envió, pueden ocurrir varias cosas, tanto que el envió tiene una solución correcta, como que puede dar un Time error, como un Wrong Answer, por lo que vimos interesante poder ver unas estadísticas que reflejaran todas estas posibles respuestas.

Alternativas de solución evaluadas:

Alternativa 1.a: Utilizar la librería js “Morris.js”

Ventajas:

- Una librería simple y fácil de implementar.

Inconvenientes:

- Es una librería que, aunque es muy fácil de implementar es bastante escasa en cuanto a recursos
-

Alternativa 1.b: Utilizar la Api JFreeChart

Ventajas:

- API muy completa, con una gran variedad de gráficas, animaciones, etc...

Inconvenientes:

- Al contrario que la primera alternativa es mucho más compleja de implementar y poner en funcionamiento

Justificación de la solución adoptada

Ya que no necesitamos tener unas graficas potentes y que solo queremos gestionar los resultados de los envíos de un determinado problema, hemos decidido hacer uso de la librería js “Morris.js”, ya que nos permite hacer lo que estábamos buscando de una forma más sencilla y eficaz. 1.a