DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Disan Limpiezas

< https://github.com/gii-is-DP1/dp1-2020-g1-02 >

Miembros:

- José Manuel Bejarano Pozo
- Pablo González Moncalvillo
- José Carlos Morales Borreguero
- José Carlos Romero Pozo
- Fernando Valdés Navarro
- Carlos Jesús Villadiego García

Tutor: Manuel Resinas Arias de Reyna

GRUPO G1-02

Versión 1.0

19/12/2020

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint

13/12/2020	V1.0	• Creación del documento • Diagrama de Diseño • Diagrama de Capas • Patrones de diseño y arquitectónicos • Decisiones de diseño	3

Contents

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

En esta sección debe proporcionar un diagrama UML de clases que describa el modelo de dominio, recuerda que debe estar basado en el diagrama conceptual del documento de análisis de requisitos del sistema pero que debe:

- *Especificar la direccionalidad de las relaciones (a no ser que sean bidireccionales)*
- *Especificar la cardinalidad de las relaciones*
- *Especificar el tipo de los atributos*
- *Especificar las restricciones simples aplicadas a cada atributo de cada clase de domino*
- *Incluir las clases específicas de la tecnología usada, como por ejemplo BaseEntity, NamedEntity, etc.*
- *Incluir los validadores específicos creados para las distintas clases de dominio (indicando en su caso una relación de uso con el estereotipo <<validates>>.*

Un ejemplo de diagrama para los ejercicios planteados en los boletines de laboratorio sería (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama):

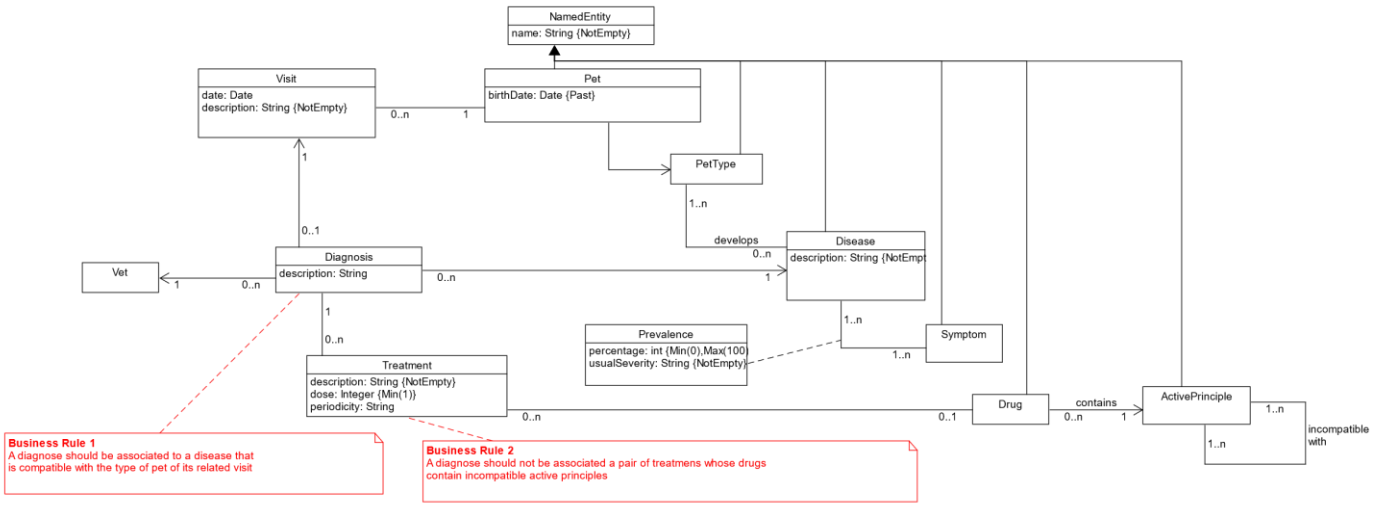
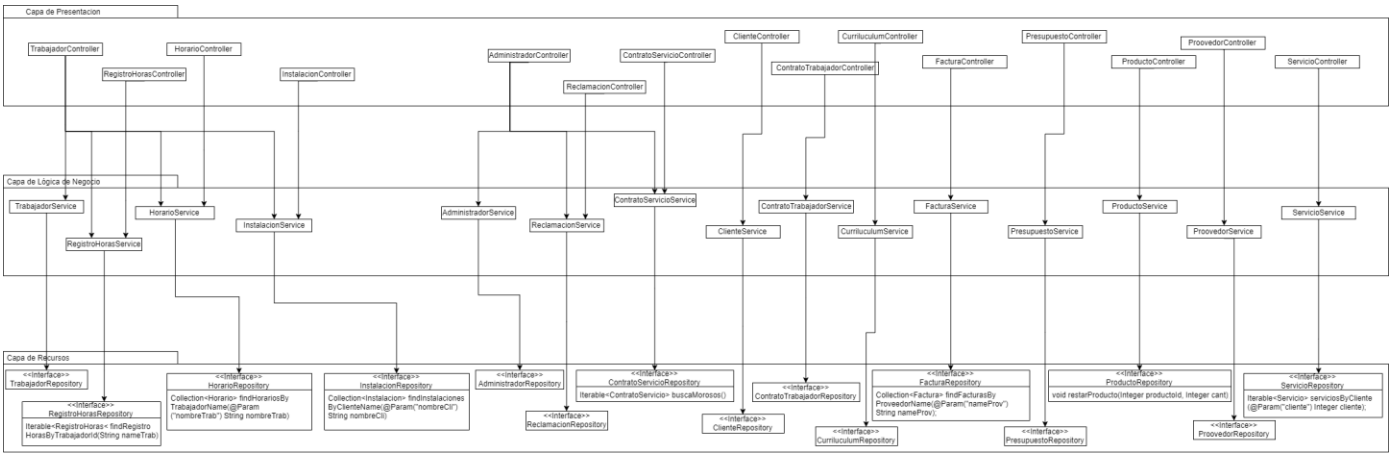


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

La foto del diagrama está en la carpeta documentación en el proyecto en GitHub



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: <The Model-View-Controller (MVC) pattern in Spring>

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 controlador responde a eventos en la interfaz de usuario (solicitudes HTTP en nuestro caso), invoca cambios en el modelo y, probablemente, en la vista.

Hemos hecho uso del controlador en el paquete org.springframework.samples.petclinic.web en todas las clases que se incluyen dentro de esto paquete el cuál se encuentra dentro de la carpeta src/main/java.

El modelo es la representación de la información. Incluye tanto los datos como la lógica de negocio necesaria para trabajar con ellos. No se debe colocar ninguna lógica de negocio fuera del modelo.

Hemos hecho uso del modelo en el paquete org.springframework.samples.petclinic.model en todas las clases que se incluyen dentro de esto paquete el cuál se encuentra dentro de la carpeta src/main/java.

La vista es la representación del modelo de manera que el usuario puede interactuar con él. Normalmente esto se hace por medio de una interfaz de usuario.

Hemos hecho uso del modelo en la carpeta jsp en todas las carpetas que se incluyen dentro de esta carpeta la cual se encuentra dentro de la carpeta src/main/webapp(WEB-INF/jsp.

Clases o paquetes creados

Para el modelo se ha creado el paquete org.springframework.samples.petclinic.model y dentro de este paquete se han creado las clases; Administrador, BaseEntity, Cliente, ContratoServicio, ContratoTrabajador, Curriculum, EstadoServicio, Factura, Horario, Instalacion, LineaPedido, NamedEntity, Oferta, package-info, Pedido, PersonaEntity, Presupuesto, Producto, Proveedor, Reclamación, Servicio, TipoCategoria, TipoPresupuesto, Trabajador.

Para el controlador se ha creado el paquete org.springframework.samples.petclinic.web y dentro de este paquete se han creado las clases; AdministradorController, ClienteController, ContratoServicioController, ContratoTrabajadorController, CrashController, CurriculumController, FacturaController, HorarioController, InstalaciónController, OfertaController, package-info, PresupuestoController, ProductoController, ProveedorController, ReclamacionController, ServicioController, TrabajadorController, WelcomeController.

Para la vista se ha creado la carpeta jsp donde nos encontramos con carpetas de las vistas de; administradores, clientes, contratosTrabajadores, curriculums, events, facturas, horarios, instalaciones, ofertas, owners, productos, proveedores, reclamaciones, servicios y trabajadores.

Ventajas alcanzadas al aplicar el patrón

Describir porqué era interesante aplicar el patrón.

Hemos utilizado este patrón porque es un patrón de diseño de software probado y se sabe que funciona. Con MVC la aplicación se puede desarrollar rápidamente, de forma modular y mantenible. Separar las funciones de la aplicación en modelos, vistas y controladores hace que la aplicación sea muy ligera. Estas características nuevas se añaden fácilmente y las antiguas toman automáticamente una forma nueva.

El diseño modular permite a los diseñadores y a los desarrolladores trabajar conjuntamente, así como realizar rápidamente el prototipado. Esta separación también permite hacer cambios en una parte de la aplicación sin que las demás se vean afectadas.

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: Realización de pruebas en los servicios

Descripción del problema:

Describir el problema de diseño que se detectó, o el porqué era necesario plantearse las posibilidades de diseño disponibles para implementar la funcionalidad asociada a esta decisión de diseño.

Como grupo nos gustaría realizar pruebas a todas las operaciones de consulta de cada uno de los servicios de nuestra aplicación. El problema es que al tener que hacer una prueba por cada operación de consulta de cada uno de los servicios resulta demasiado tedioso.

Alternativas de solución evaluadas:

Especificar las distintas alternativas que se evaluaron antes de seleccionar el diseño concreto implementado finalmente en el sistema. Si se considera oportuno se puden incluir las vestajas e inconvenientes de cada alternativa

Alternativa 1.a: Incluir todas las pruebas de todas las operaciones de consulta de un servicio en una misma clase (test).

Ventajas:

- No se mezclan las pruebas de las operaciones de consulta de los servicios de la aplicación.

Inconvenientes:

- Es necesario crear demasiadas clases de pruebas puesto que tenemos muchos servicios en la aplicación.

Alternativa 1.b: Crear una única clase (test) con todas las operaciones de consulta de todos los servicios de la aplicación.

Ventajas:

- Solo es necesario crear una única clase para todas las pruebas de las operaciones de consulta de los servicios de la aplicación.

Inconvenientes:

- Se mezclan todas las pruebas de las operaciones de consulta de los servicios de la aplicación.

Alternativa 1.c: Crear una clase (test) por cada una de las operaciones de consulta de ada uno de los servicios de la aplicación.

Ventajas:

- Muy buena separación entre las distintas pruebas de las operaciones de consulta de los servicios de la aplicación.

Inconvenientes:

- Es muy tedioso implementar las pruebas puesto que hay que crear una clase (test) por cada prueba de cada operación de consulta de los servicios de la aplicación.

Justificación de la solución adoptada

Describir porqué se escogió la solución adoptada. Si se considera oportuno puede hacerse en función de qué ventajas/inconvenientes de cada una de las soluciones consideramos más importantes.

La alternativa que hemos decidido escoger es la 1.a, ya que separa todas las operaciones de consulta por servicio de la aplicación y no resulta tan tedioso como la 1.c que hay que implementar cada operación de consulta de cada uno de los servicios de la aplicación en distintas clases.

Decisión 2: Importación de datos reales para la realización de las pruebas

Descripción del problema:

Como grupo nos gustaría poder hacer pruebas de cada una de las operaciones de consulta de los servicios de nuestra aplicación 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 a importar para la realización de las pruebas de cada una de las operaciones de consulta de los servicios de la aplicación en el propio script de inicialización de la BD (data.sql).

Ventajas:

- Simple, no requiere nada más que escribir el SQL que genere los datos para probarlos en las pruebas.

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 los 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 división en capas es fundamental y no queremos renunciar a un trabajo ágil durante el desarrollo de la aplicación, seleccionamos la alternativa de diseño 1.c.