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

Este proyecto está diseñado con la finalidad de aportar una interfaz que posibilite la interacción en los procesos que se realizan en la empresa Disan. Para ello, se ha desarrollado una página web en la que dependiendo de la autoridad del usuario que la use (cliente, administrador, trabajador o proveedor) puede realizar distintas funcionalidades que faciliten y agilicen todo tipo de necesidades dentro del negocio, como por ejemplo la visualización de los horarios de los trabajadores o la oferta de productos por parte de los proveedores para poder ser evaluadas por los administradores.

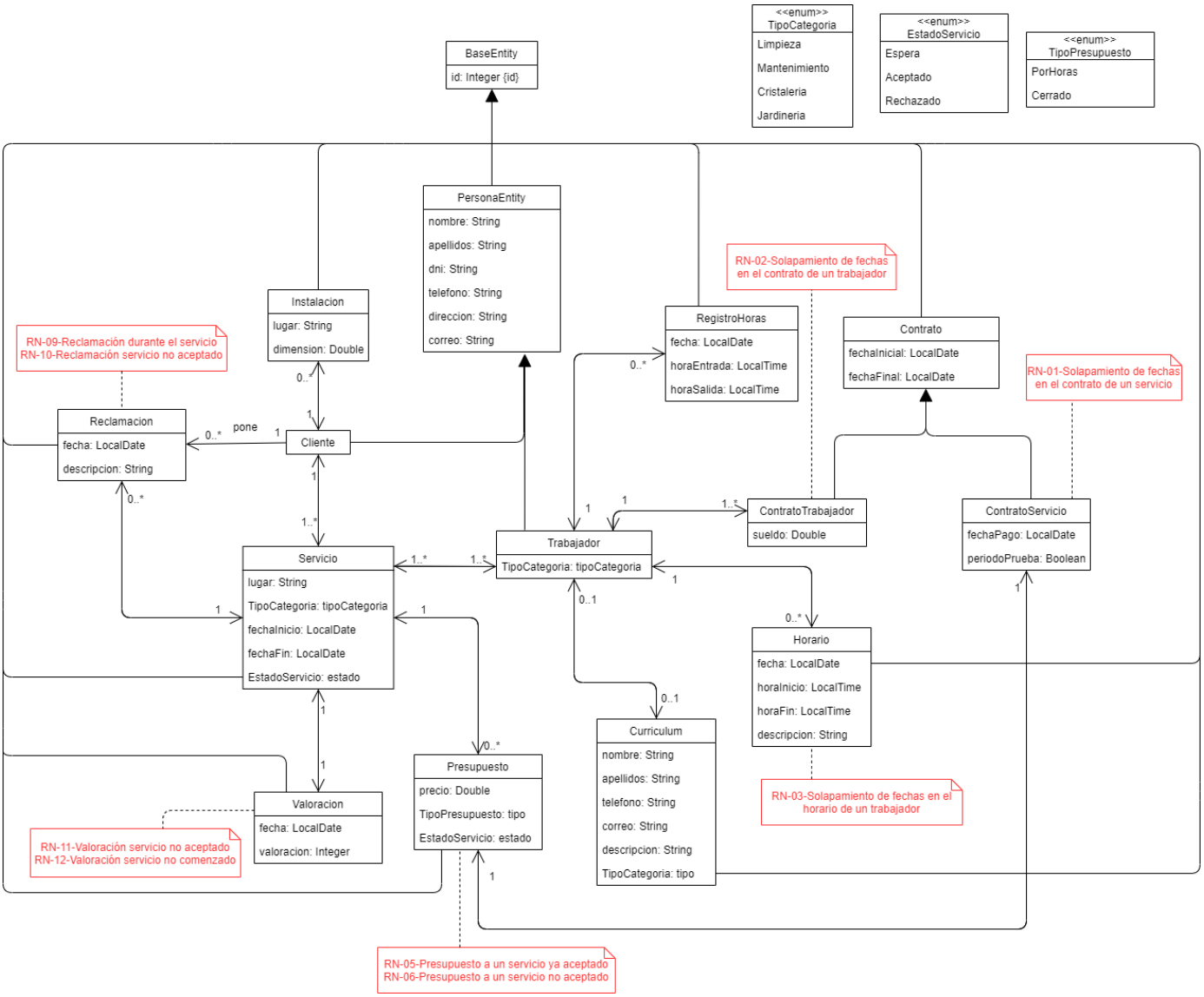
Diagrama(s) UML:

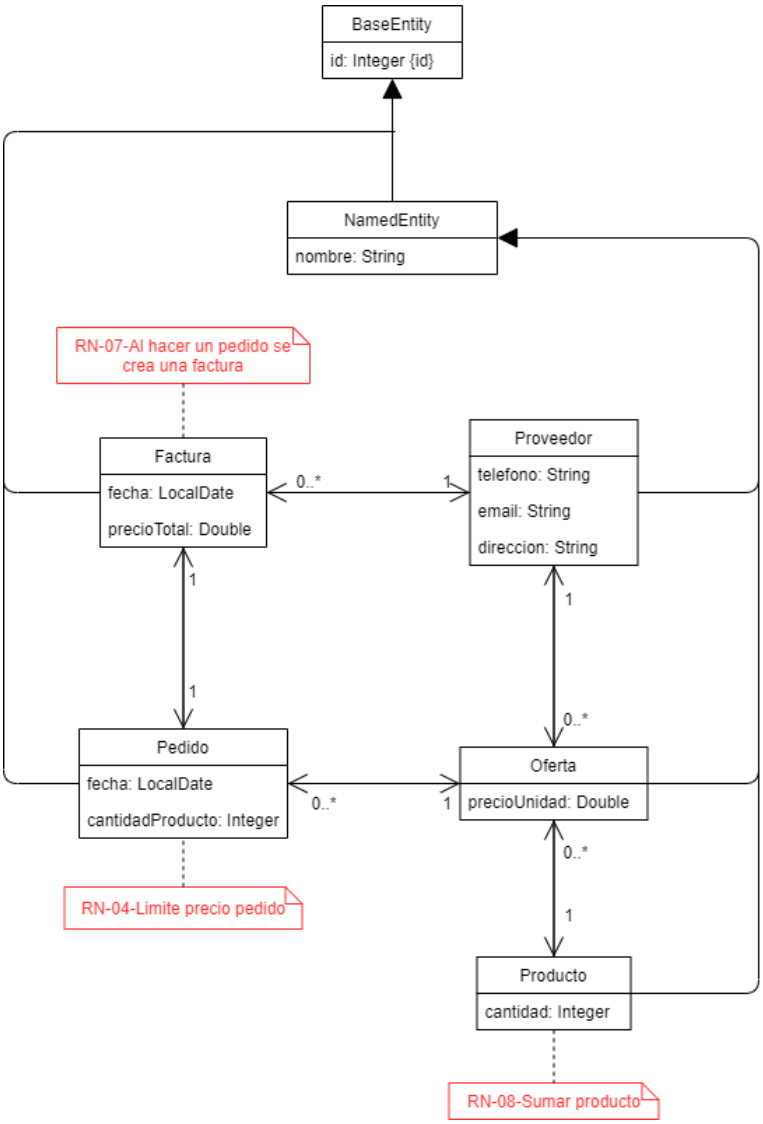
Diagrama de Dominio/Diseño

La mayoría de los atributos de cada clase tienen la restricción NotEmpty. Con el fin de facilitar la comprensión de los diagramas, se ha asignado la restricción Empty a aquellos atributos que sí puedan estar vacíos en la base de datos.

DP1 2020/21
DIsan

Grupo: G1-02




BaseEntity
id: Integer {id}
NamedEntity
nombre: String
RN-07-Al hacer un pedido se crea una factura
Factura
fecha: LocalDate
precioTotal: Double
Proveedor
telefono: String
email: String
direccion: String
Pedido
fecha: LocalDate
cantidadProducto: Integer
Oferta
precioUnidad: Double
Producto
cantidad: Integer
RN-04-Limite precio pedido
RN-08-Sumar producto
0..*
1
1
1
1
0..*
0..*
1
1
0..*

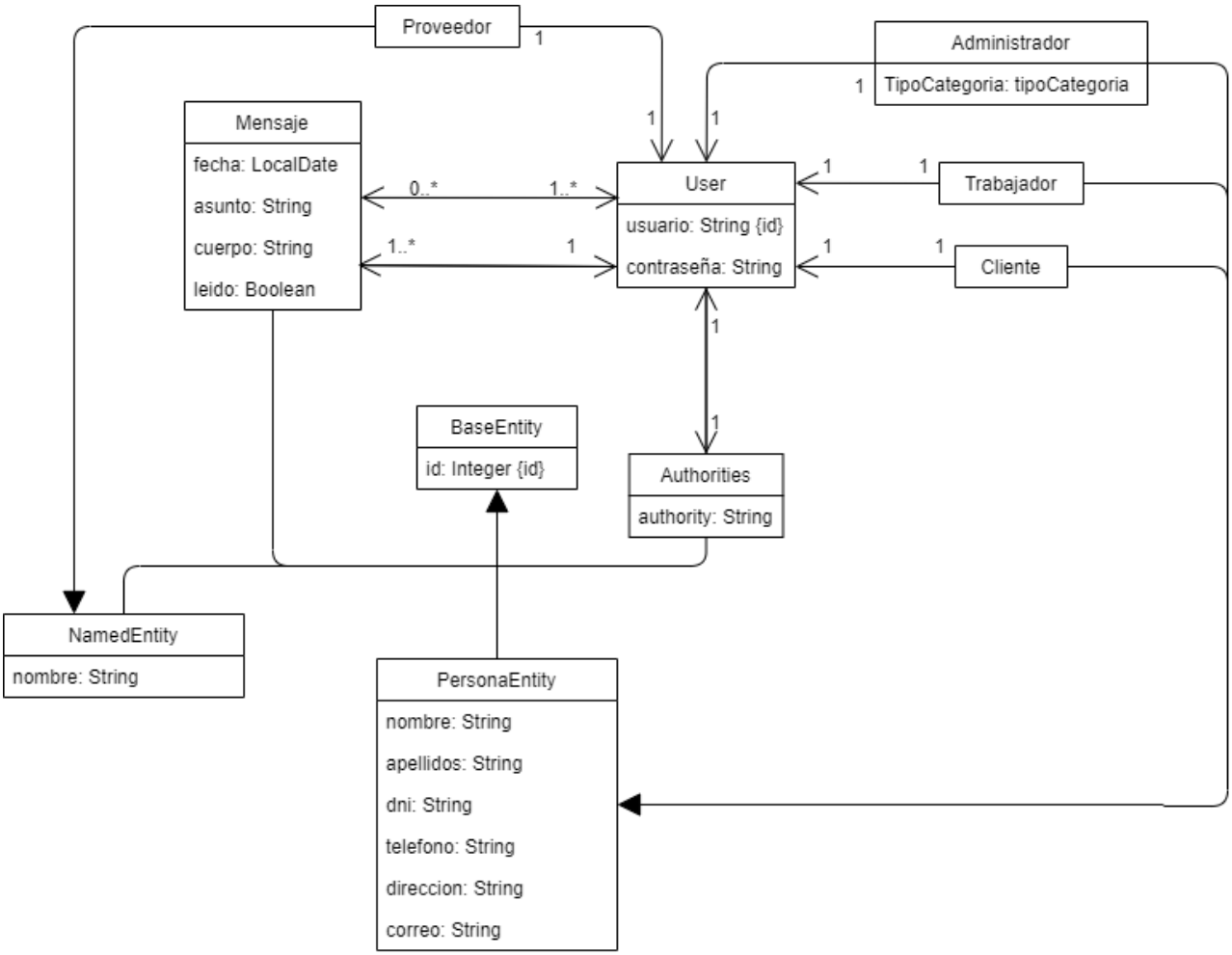
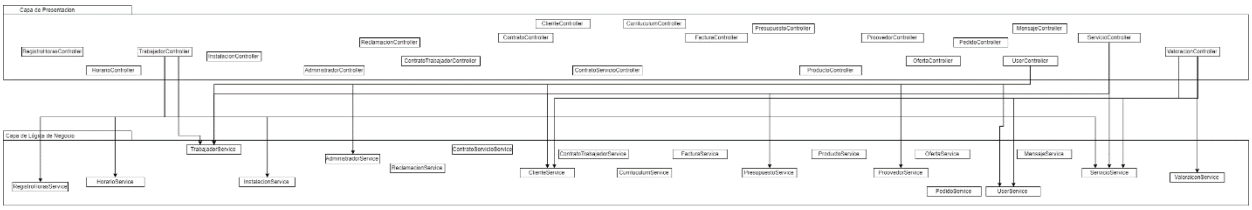
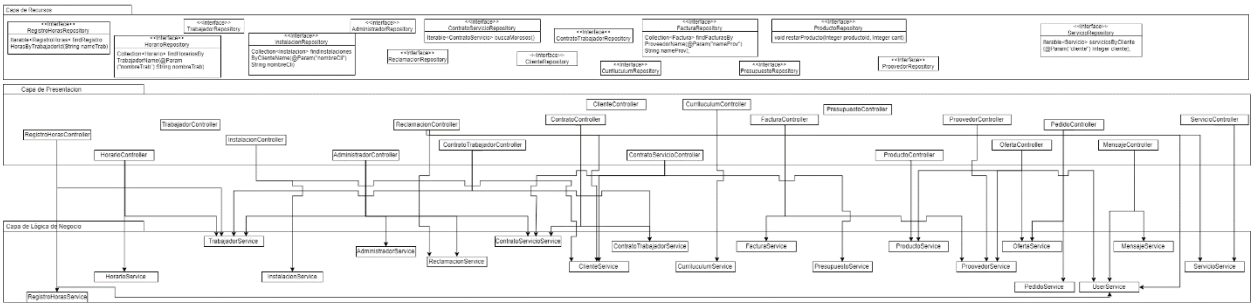


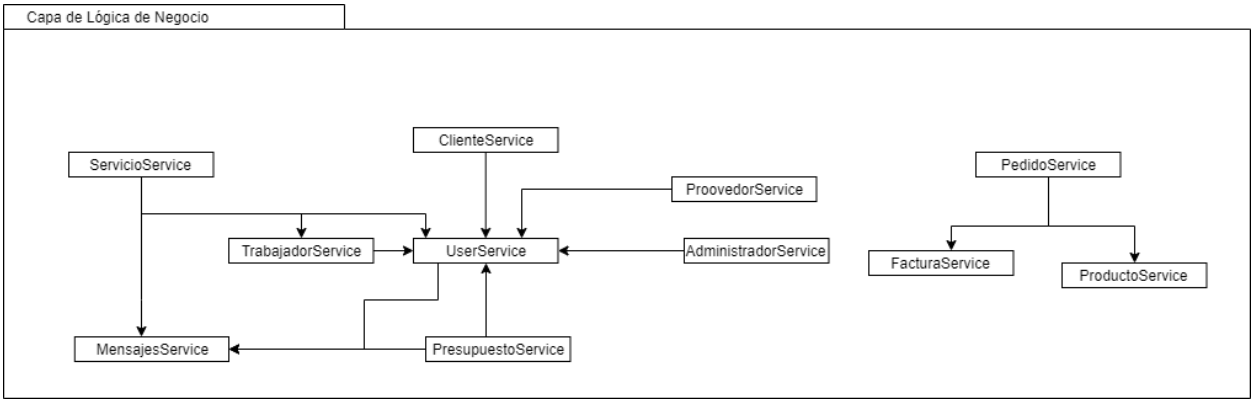
Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

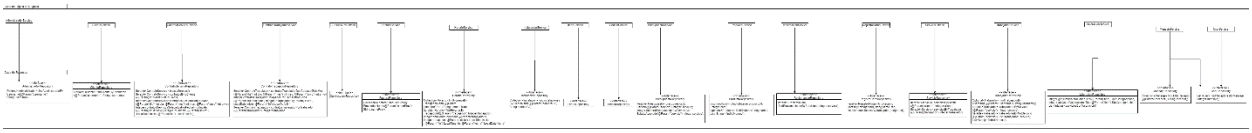
Hemos tenido que dividir el diagrama en varios ya que no se podían entender bien

Las fotos de los diagrama están 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: MVC

Tipo: Arquitectónico

Contexto de Aplicación

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 este 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 las siguientes entidades; Administrador, Authorities, BaseEntity, Cliente, Contrato, ContratoServicio, ContratoTrabajador, Curriculum, EstadoServicio, Factura, Horario, Instalacion, Mensaje, NamedEntity, NivelSatisfaccion, Oferta, Pedido, PersonaEntity, Presupuesto, Producto, Proveedor, Reclamación, RegistroHoras, Servicio, TipoCategoria, TipoPresupuesto, Trabajador, User, Valoración.

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, ContratoController, ContratoServicioController, ContratoTrabajadorController, CrashController, CurriculumController, DissanErrorController, FacturaController, HorarioController, InstalaciónController, MensajesController, OfertaController, PedidoController, ProductoController, ProveedorController, ReclamacionController, RegistroHorasController, ServicioController, TrabajadorController, UserController, ValoracionController, WelcomeController.

Para la vista se ha creado la carpeta jsp donde nos encontramos con carpetas de las vistas de; administradores, clientes, contratos, contratosServicios, contratosTrabajadores, curriculums, facturas, horarios, instalaciones, ofertas, presupuestos, productos, proveedores, reclamaciones, registroHoras, servicios, trabajadores, users y valoraciones. También tenemos las vistas de welcome, succesful

Ventajas alcanzadas al 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

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 pueden incluir las ventajas 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: Modelo de dominio y Capa de servicios

Descripción del problema:

Para poder alcanzar el máximo potencial en nuestra aplicación necesitamos un diseño de modelo de datos que cumpliesen con las expectativas del proyecto.
En la decisión de dicho diseño se contemplaron opciones que dejaban que desear como a continuación se describe. Y problemas como un lógica compleja y transacciones difíciles de implementar, así como evitar la duplicación de código y obtener todo el poder de los objetos nos encarrilaron a tomar la opción correcta.

Alternativas de solución evaluadas:

Alternativa 2.a: Script de transacción: consiste en organizar la lógica de negocio por procedimientos, donde cada procedimiento maneja una sola solicitud de la presentación.

Ventajas:

- Lógica simple y transacciones fáciles de implementar.

Inconvenientes:

- Tiende a duplicar código.
- Difícil de mantener si la lógica es compleja.

Alternativa 2.b: Módulo de tabla: una sola instancia que maneja la lógica de negocio de todas las filas de una tabla de base de datos

Ventajas:

- Fácil asignación con la estructura de la base de datos.

Inconvenientes:

- No proporciona todo el poder de los objetos(Sin relaciones ni herencias).

Alternativa 2.c: Modelo de dominio: un modelo de objeto del dominio que incorpora comportamientos y datos. Permite implementar una lógica de negocio compleja.

Ventajas:

- Apoyado por la mayoría de frameworks.

Inconvenientes:

- Asignación con la base de datos de mayor complejidad.

Alternativa 2.d: Capa de servicio: establece un conjunto de operaciones disponibles y coordina la respuesta de la aplicación en cada operación. En este patrón la capa de presentación interactúa con el dominio a través de la Capa de Servicio.

Justificación de la solución adoptada

Debido a que en nuestro caso tenemos varios tipos de clientes y muchos casos de uso que involucran diferentes entidades hemos decidido elegir un patrón de Modelo de Dominio, así como agregar una capa de servicio.