DP1 2020-2021

Documento de Diseño del Sistema

Proyecto <Nombre de Proyecto>

<URL del Repositorio de GitHub>

Miembros <en orden alfabético por apellidos>:

* <Eiji Aonuma>
* <Hidemaro Fujibayashi>
* <Koji Igarashi>
* <Satoru Iwata>
* <Hideo Kojima>
* <Masahiro Sakurai>

Tutor: <Shigeru Miyamoto>

GRUPO G<X>-<YY>

Versión <Z>

<Fecha>

# Historial de versiones

*Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto*

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción de los cambios** | **Sprint** |
| 13/12/2020 | V1 | * Creación del documento | 2 |
| 13/12/2020 | V2 | * Añadido diagrama de dominio/diseño * Explicación de la aplicación del patrón caché | 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 5](#_Toc58783395)

[Decisión X 6](#_Toc58783396)

[Descripción del problema: 6](#_Toc58783397)

[Alternativas de solución evaluadas: 6](#_Toc58783398)

[Justificación de la solución adoptada 6](#_Toc58783399)

Esta es una plantilla que sirve como guía para realizar este entregable. Por favor, mantén las mismas secciones y los contenidos que se indican para poder hacer su revisión más ágil.

# Introducción

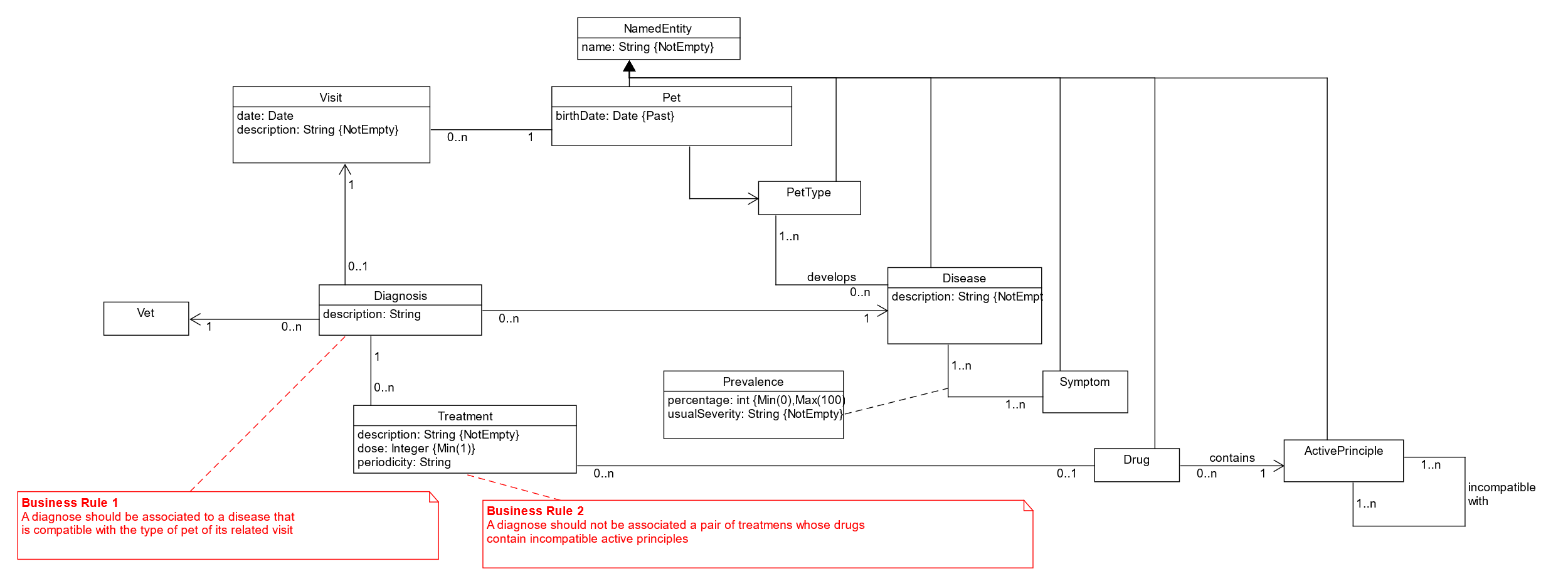
*En esta sección debes describir de manera general cual es la funcionalidad del proyecto a rasgos generales (puedes copiar el contenido del documento de análisis del sistema). Además puedes indicar las funcionalidades del sistema (a nivel de módulos o historias de usuario) que consideras más interesantes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

En esta sección debe proporcionar un diagrama UML de clases que describa el conjunto de controladores, servicios, y repositorios implementados, incluya la división en capas del sistema como paquetes horizontales tal y como se muestra en el siguiente ejemplo:

Diagram

Description automatically generated

El diagrama debe especificar además las relaciones de uso entre controladores y servicios, entre servicios y servicios, y entre servicios y repositorios.

Tal y como se muestra en el diagrama de ejemplo, para el caso de los repositorios se deben especificar las consultas personalizadas creadas (usando la signatura de su método asociado).

# 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: <Nombre del patrón>

### Tipo: Arquitectónico | de 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

Indicar las clases o paquetes creados como resultado de la aplicación del patrón.

### Ventajas alcanzadas al aplicar el patrón

Describir porqué era interesante aplicar el patrón.

# 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 X

### 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.

### 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

### 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.

Ejemplo:

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

## Decision 2: Tests in controlers and services

### Problem description:

The aim when doing tests is to check the correct functioning of the different components from the project without needing to execute the application or doing tests over the running application.

Solution 2.a: Make tests by doing @Autowire of services and controllers.

**Advantages:**

* No need for mock-ups.
* Code is simpler.
* The data is already prepared (they are the same that are used when the app runs).

**Disadvantages:**

* Is no different to doing a test directly on the app.
* Is possible to forget doing some tests due to working with already existing data.
* If there are components that are used by controlers/services that should not be using them, we would not be able to notice the error.

Solution 2.b: Make the service tests by mocking the needed repository and make the controler tests by mocking the needed service. Validators should not be mocked.

**Advantages:**

* Total independence between tests and actual application.
* All necessary tests for the app’s correct functioning are carried away, if any of them is missing, the other existing tests would result in failure.
* Allows to make tests over custom data (simpler data if desired).
* Allows to test the modularity between project components.

**Disadvantages:**

* More code.
* The data must be prepared prior to the tests

### Justification of chosen solution

We have chosen the solution 2.b given that it was very important for the project a total independency between the tests and the application. Also some errors where discovered when using some specific components (mostly formatters) that would not have been noticed if tests would not check the modularity between different components.

## Decision 4: Validation subsequent to @Valid in

### Problem description:

When doing an object validation, this object does not have an id yest, so when wanting to check additional restrictions (specially those related with checking if the same elements already exists in the database), it was not possible to do so due to that id not existing. Making an additional validation needed.

Solution 4.a: Make the additional validation only when creating the object and not in modification, where the id is needed.

**Advantages:**

* Only valid function would be necessary.

**Disadvantages:**

* In case the rule is broken, an error would appear in the database which would cause the app to stop the process.
* The form validation would be incomplete.
* Different validation when creating and updating.

Solution 4.b: Have an automatic and not modifiable field corresponding to the object’s id.

**Advantages:**

* Does the additional validation in the valid function.

**Disadvantages:**

* Requires that more elements are added to the form.
* In creation, the id field would be an empty field or an id should be created prior to the form (without letting spring manage it).
* Spring must use the id that is given by us, and not the one it creates.

Solution 4.c: Use the setId that is already on the updates and make the validation afterwards, when the object has an id, by making a call to the function in the validator.

**Advantages:**

* No elements are added to the form.
* No modification of valid function.
* Error is captured and a user-friendly message is shown.
* It allows more Independence between validations.

**Disadvantages:**

* More code.
* Updates have more conditions in them.

### Justification of chosen solution

We have chosen the solution 4.c because it was easy to implement and did not interfere with the valid function. Also, the additional validations are done the same way in creation and updates, even when it could be done only with the valid function while creating an object. This was done so post functions of creation and update are more alike and to keep the valid function the same for both of them.

## Decision 6: Showing incomes from slot machines.

### Problem description:

Each slot machine has a different set of gains with each day they are in use. In other words, each day a slot machine earns a different amount of money that is saved by the system. The idea is to be able to show the gain of one slot machine from one specific day.

Solution 6.a: Make an additional view for each slot machine were all its gains are listed, or with a selector to pick the date.

**Advantages:**

* Easy to code.
* Is similar to other code that is already in the project.

**Disadvantages:**

* One must change views to view the gains of a slot.
* Is not original, just another list of numbers.
* Uncomfortable if someone wants to check different amounts from different slots

Solution 6.b: Add a selector with dates to each slot machine in the slotMachineList view, so when selecting the date of a gain of an specific slot, the amount that it earned that day is shown in the next column.

**Advantages:**

* Easy to use.
* Comfortable for the user.
* Does not show unimportant information to the user.
* All the information is in one view.

**Disadvantages:**

* Very complex ajax coding (a selector for each slot with different options in each selector).
* Needs more functions in the controller/service to work.

### Justification of chosen solution

We have chosen the solution 6.b, although we knew it was quite a difficult task to perform, we faced it as a fun challenge and after quite a bit of research and not few mistakes we managed to get the most optimal solution for this problem working. Being the coding difficulty the only disadvantage, after we solved it, all that was left were advantages for the final user.