



MEMORIA PRÁCTICA 2

Tecnologías Avanzadas de Programación

Desarrollo de una aplicación web de
gestión de ascensores de un edificio.

Daniel López, Pablo Novoa, Guzmán Oñate y Samuel Soria
Ingeniería Informática. Grupo A1

Índice

Índice	1
1. Introducción	2
2. Diseño de la aplicación	3
Principios de diseño	3
Patrones de diseño	4
Singleton	4
State	4
Observer	5
3. Evidencia de calidad	6
4. Diagrama de clases	8
5. Frontend	10
6. Bibliografía	13

1. Introducción

El objetivo de este documento es explicar el desarrollo del proyecto realizado para la asignatura Tecnologías Avanzadas de Programación, consistente en desarrollar una aplicación web de gestión de ascensores de un edificio, reflejando todos los contenidos aprendidos durante el curso.

Durante el curso hemos repasado los conceptos de la Programación Orientada a Objetos, hemos estudiado UML, los principios y patrones de diseño y el proceso de refactoring. Todo ello y ciertos contenidos aprendidos en la asignatura DIS del curso anterior se han puesto en práctica para el desarrollo de esta práctica.

El proyecto se ha desarrollado con Java (backend), y se ha empleado el framework para aplicaciones web Java Vaadin (frontend). El código se ha desarrollado en la plataforma software Eclipse, pero también se ha utilizado SonarQube para realizar un reporte del código fuente, Heroku para que la aplicación pueda ser accedida online y diversas consolas de comandos.

2. Diseño de la aplicación

Principios de diseño

En esta sección trataremos los principios de diseño que hemos empleado en el desarrollo de nuestra práctica.

Para empezar, podemos hablar del principio “Encapsular lo que varía”, pues la información que cambia de los ascensores cambia en cada objeto ascensor.

También hemos empleado el principio “Programar a interfaces no implementaciones”, pues cada tipo de estado depende de una interfaz, no de clases concretas.

Por otro lado, “Tell don't ask” se puede apreciar por ejemplo, en la función de cambiar de piso, donde en vez de estar preguntando por el piso al ascensor, directamente le ordenamos que cambie al piso deseado.

Hemos respetado el Principio de Nivel Único de Abstracción, pues en vez de programar métodos muy largos, hemos preferido crear muchos métodos distintos y concisos, donde todas las instrucciones del método operan en el mismo nivel de abstracción, realizando una única tarea (podemos verlo en la clase DisplayAscensor).

Respecto a los principios S.O.L.I.D., podemos observar el uso del Principio de responsabilidad única, pues hemos creado una clase DisplayControl que se encarga exclusivamente de observar los datos que la pantalla de control requiere, o la clase edificio que se encarga de inicializar los ascensores y los pisos).

Un ejemplo de Inversión de dependencias que hemos aplicado es el observer realizado, ya que hemos creado la interfaz observer (High level) y dos clases diferentes DisplayAscensor y DisplayControl (Low level) que implementan Observer. Al programar el front, dependiendo del objeto que inicializamos (DisplayAscensor o DisplayControl), los métodos realizarán una cosa u otra.

Patrones de diseño

Los patrones de diseño empleados para el desarrollo de nuestro proyecto han sido un Singleton, un Stat y un Observer.

Singleton

El singleton nos servirá para poder instanciar un objeto de una clase en concreto una única vez.

Para esta práctica, se ha utilizado para inicializar el objeto denominado edificio, compuesto por ascensores y pisos. De esta forma, en cualquier ventana del frontend de la aplicación en cuestión se hará referencia en todo momento al mismo objeto.

Otra ventaja que existe al inicializar este singleton es que nos ayuda a cumplir el principio de diseño de encapsular lo que varía por lo que si se decidiera añadir un nuevo ascensor o una nueva planta solo habría que añadir en el objeto ascensor más información

State

El patrón de diseño state es un patrón que nos permite poder cambiar de estados según determinados comportamientos. Así mismo poder realizar una máquina de estados a través de una interfaz compuesta por cada uno de los estados y los métodos que los componen serán las diferentes iteraciones hacia cada estado.

En este caso, el state lo usaremos para poder alternar entre los diferentes estados en los que se encuentren los ascensores. Los estados que tendremos para este ejemplo serán parado, cerrando puerta, abriendo puerta y movimiento. El estado inicial en el que comienzan los ascensores es parado, pudiendo acceder a diversos estados a través de acciones, que llevan siempre a su vez al estado parado, ya que no existe un estado final, si no que es cíclico el programa.

Observer

El patrón de diseño observer consiste en la creación de una clase que observe y otra clase observable, generando una relación entre éstas, que obligue a que siempre que la clase observable genere algún tipo de cambio, se vea obligada a notificar a la clase Observer.

Para esta práctica hemos decidido usar dos observers distintos. El motivo de realizar dos distintos es para que se usen en los distintos paneles de control del frontend y no tener que modificarlo en esa parte. Gracias a esto evitaremos modificar código en el frontend y tener un código más legible y sencillo de entender.

El primer observer mostrará el ascensor que es, con su estado actual y la planta en la que se encuentra. Este observer se mostrará a través de una tabla en el apartado del panel de control con el fin de observar el correcto funcionamiento de los ascensores. Y, el segundo y último observer mostrará únicamente el número del ascensor y el piso en el que se encuentra.

3. Evidencia de calidad

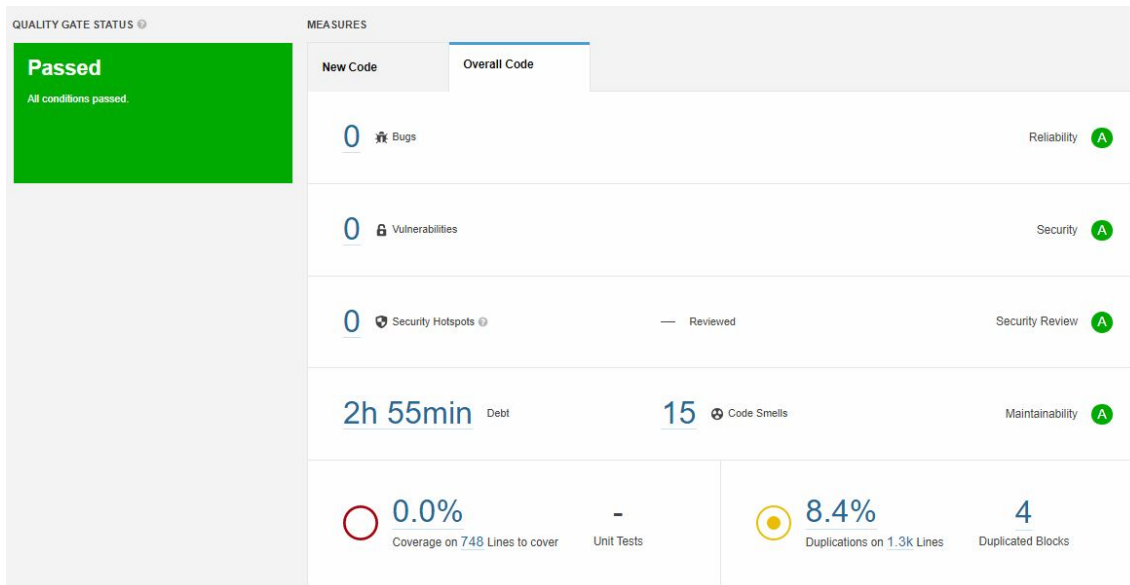
Tras terminar el desarrollo del proyecto, procedimos a refactorizar nuestro código, haciendo uso de la plataforma SonarQube. Dicha herramienta de software libre nos brinda la posibilidad de evaluar nuestro código fuente, ayudándonos a luchar contra la deuda técnica y mejorar la facilidad de comprensión del código.

Tras analizar nuestro proyecto, observamos que había un total de 115 smell codes. Tuvimos que corregir diferentes aspectos del código, como nombres de variables (pues estaban en un formato incorrecto), código muerto, etc.



Al final conseguimos reducir a 15, lo que implica una corrección de 100 code smells. Muchos de los 15 restantes aparecen por emplear `System.out.println` en vez de `logger.log`, pero decidimos que no era necesario crear un `log` en estos casos, pues sólo queríamos imprimir por consola cierta información para facilitarle la corrección al profesor.

Como en cualquier refactorización, nuestra tarea consistía en modificar el código fuente respetando el comportamiento inicial. La mayoría de la deuda técnica que acumulamos procedía de la definición de los mismos textos repetidas veces (para las botoneras o nombres de paneles) y el uso de un formato incorrecto para las variables.

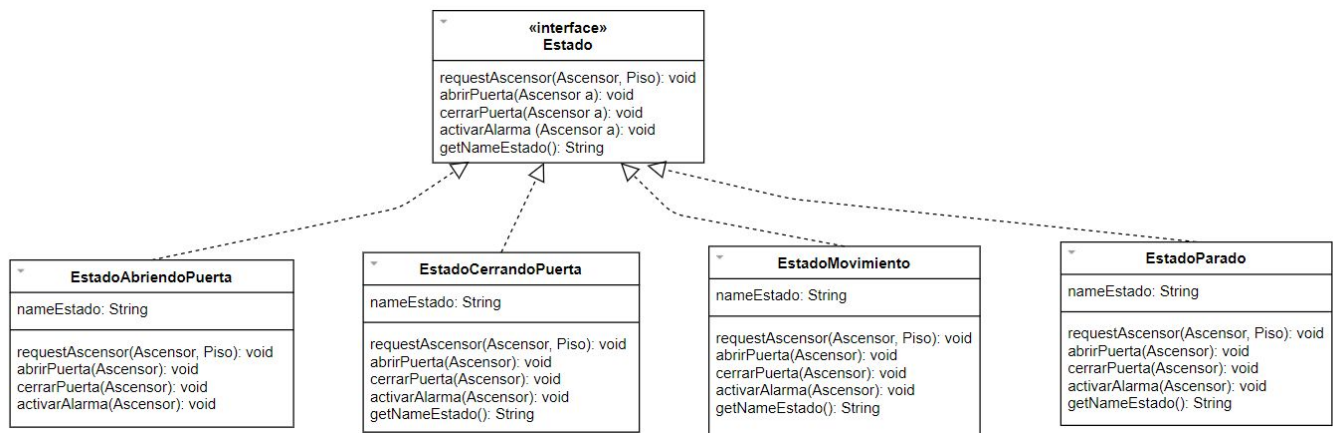


Como podemos ver, el análisis muestra que nuestro código fuente no posee ningún Bug, Vulnerabilidad o Security Hotspot, y un total de 15 code smells, lo que nos proporciona una puntuación de A en todo.

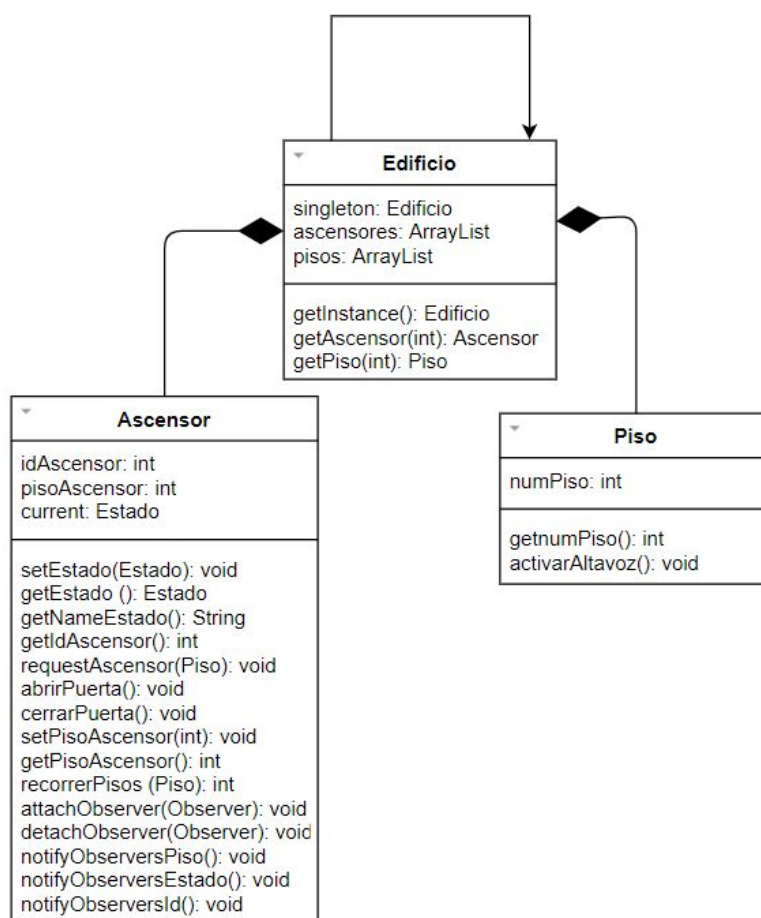
4. Diagrama de clases

UML de:

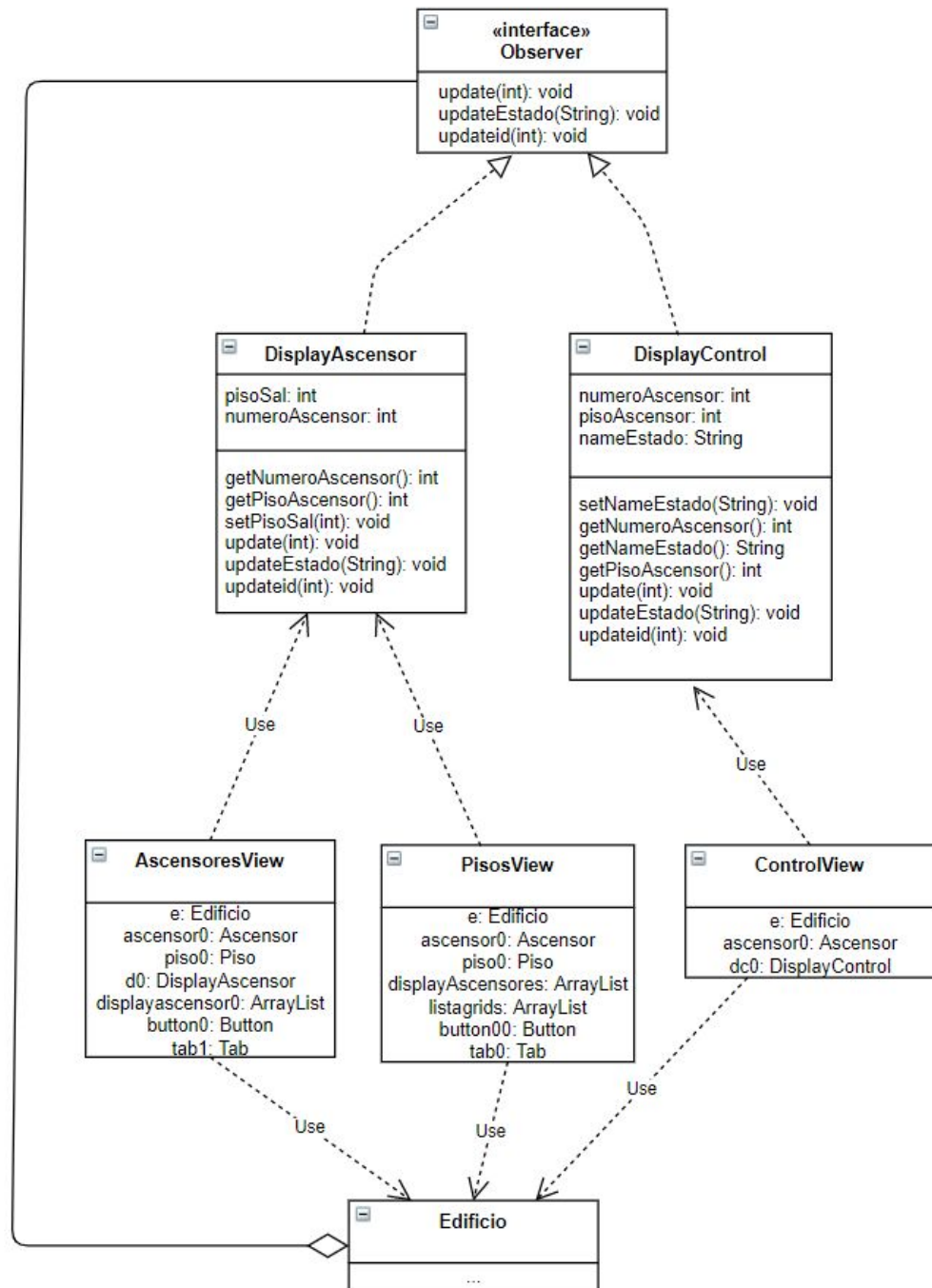
-State



-Singleton



-Observer (clases Observer, DisplayAscensor y DisplayControl) y Frontend (clases ControlView, AscensoresView y PisosView).

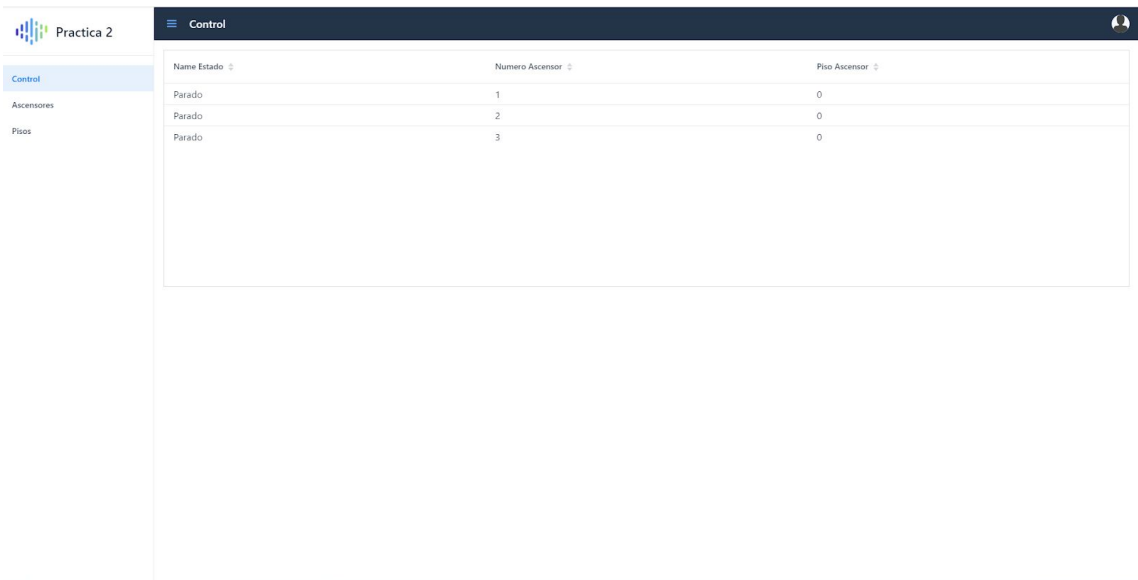


5. Frontend

En esta sección, explicaremos el interfaz que se puede encontrar en la aplicación web.

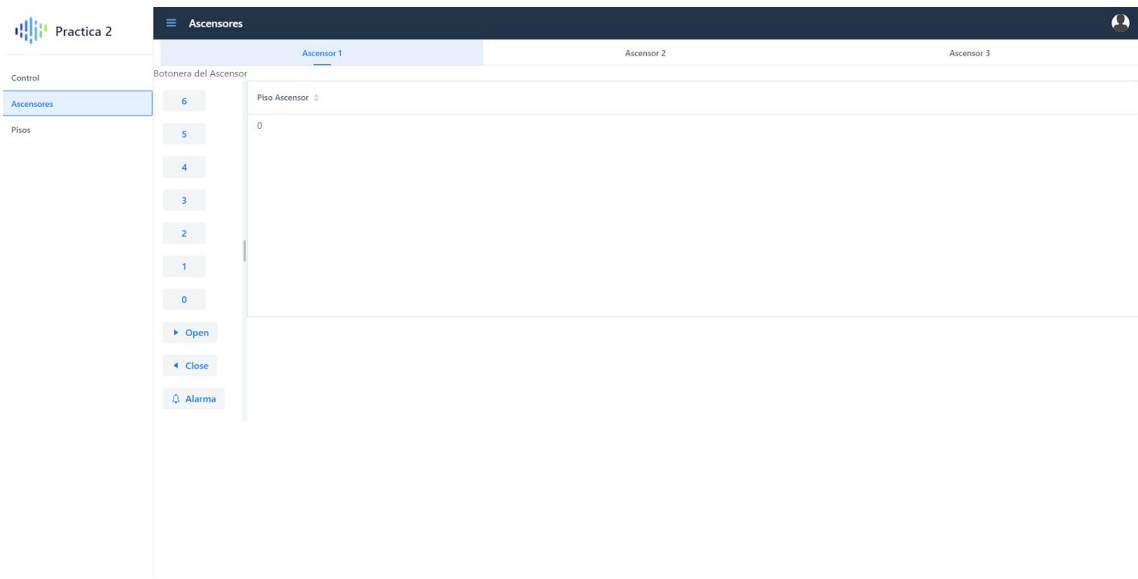
El frontend de la aplicación consta de las siguientes pantallas:

Pantalla de control: donde se permite visualizar la información total de todos los ascensores. El estado del ascensor, en función de cómo se encuentre (parado, en movimiento y abriendo o cerrando puertas); un identificador de cada uno y el piso en el que se encuentran.



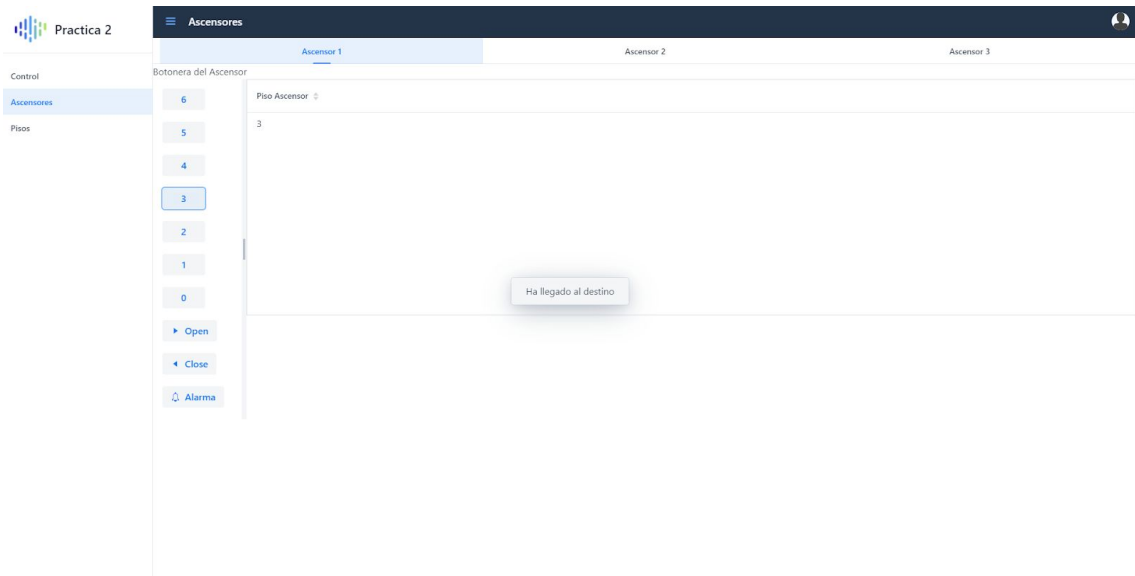
Name Estado	Numero Ascensor	Piso Ascensor
Parado	1	0
Parado	2	0
Parado	3	0

Pantalla de ascensores: donde se muestran los tres ascensores que tiene el edificio y los controles de estos. Se puede elegir el piso al que ir, abrir o cerrar las puertas y activar la alarma de emergencias.

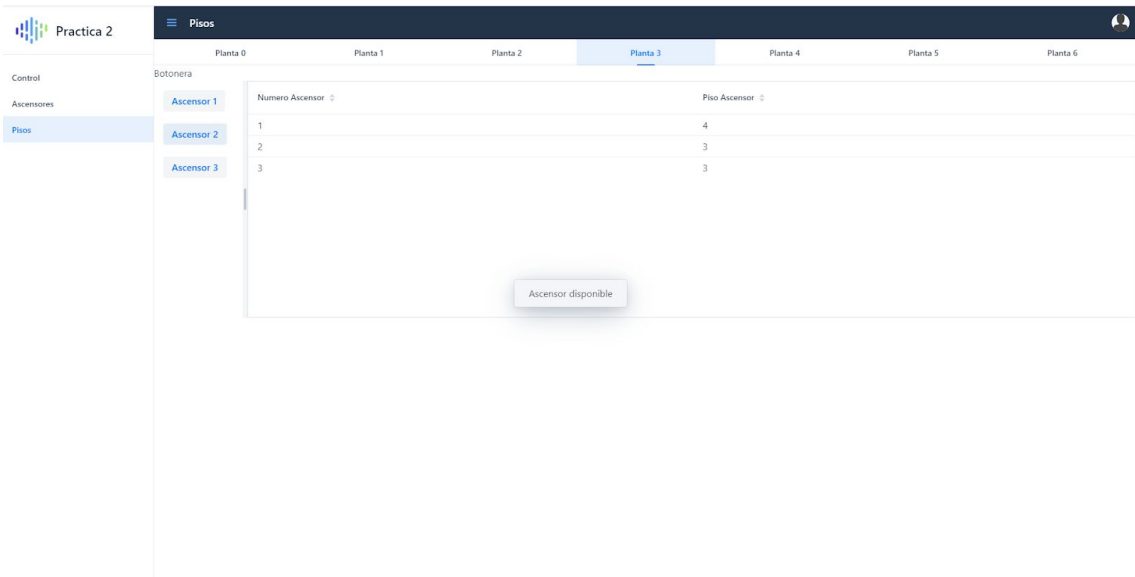


Ascensor 1	Ascensor 2	Ascensor 3
<p>Botonera del Ascensor</p> <p>6</p> <p>5</p> <p>4</p> <p>3</p> <p>2</p> <p>1</p> <p>0</p> <p>► Open</p> <p>◄ Close</p> <p>🚨 Alarma</p>		

Cada pulsación sobre un botón, está asociado a un evento que muestra un mensaje información de cada acción procesada por la aplicación. Se utiliza para simular las distintas acciones definidas el los requisitos.



Pantalla de pisos: donde se puede acceder a cada planta del edificio y ver el piso en el que se encuentra cada ascensor, así como llamarlos para poder utilizarlos. Cuando llegue el ascensor a la planta aparecerá un mensaje de aviso simulando un timbre.



La aplicación completa ha sido desplegada en Heroku. Se puede acceder desde el siguiente enlace: <https://tapprc2.herokuapp.com/control>

6. Bibliografía

- Vaadin, © 2021 . Disponible en:
<https://vaadin.com/learn/tutorials/cloud-deployment/heroku>
- SonarQube: , ©2021 .Disponible en: <https://docs.sonarqube.org/latest/>
- Universidad Francisco de Vitoria, ©2021 . Disponible en:
<https://ufv-es.instructure.com/courses/2104>