Justificación del Diagrama de Clases UML - Proyecto SmartHome Solutions

Introducción

En este documento queremos explicar por qué diseñamos nuestro diagrama de clases UML de esta manera para el proyecto *SmartHome Solutions*. Nuestro objetivo es crear un sistema que permita manejar dispositivos inteligentes en una casa, y para eso usamos los conceptos de Programación Orientada a Objetos que hemos estado viendo en clase.

Tratamos de aplicar los principios que estudiamos: abstracción, encapsulamiento, herencia, y las diferentes relaciones entre clases como composición, agregación y asociación. A continuación explicamos cómo los implementamos en nuestro diseño.

Desarrollo

Abstracción

Lo que hicimos fue identificar las cosas más importantes de nuestro sistema y crear clases para representarlas. Por ejemplo, para la clase **Usuario** pusimos solo lo básico que necesitábamos: nombre, email y el rol que tiene en el sistema. Para los **Dispositivos** incluimos el estado (si está encendido o apagado) y las operaciones principales como encender() y apagar().

No nos metimos en detalles técnicos complicados como qué tipo de procesador tiene cada dispositivo o cómo se comunican por WiFi, porque eso no es relevante para lo que queremos hacer con nuestro sistema.

Encapsulamiento

Aquí tratamos de proteger los datos de cada clase para que no se puedan modificar desde cualquier lado. Los atributos de cada clase están "encerrados" y solo se pueden cambiar usando los métodos que definimos.

Por ejemplo, el estado de un dispositivo no se puede cambiar directamente, sino que hay que usar los métodos encender() o apagar(). Esto nos ayuda a evitar errores y mantener todo más ordenado. Si alguien quiere saber la temperatura de un termostato, tiene que usar el método correspondiente, no puede acceder directamente al valor.

Herencia

Esta parte nos costó un poco al principio, pero creemos que la entendimos bien. Hicimos una clase general **Dispositivo** que tiene todo lo común a todos los dispositivos (estado, métodos básicos), y después creamos clases más específicas como **Luz**, **Cámara** y **Electrodoméstico** que heredan de esta clase padre.

Cada clase hija puede tener sus propios métodos especiales. Por ejemplo, la **Luz** tiene ajustarBrillo() y la **Cámara** tiene iniciarGrabación(). Así no tenemos que repetir código y es más fácil agregar nuevos tipos de dispositivos después.

Composición

En nuestro diseño, las **Automatizaciones** necesitan sí o sí de los dispositivos para funcionar. Si no hay dispositivos, la automatización no tiene sentido. Por eso usamos composición, que es una relación muy fuerte donde una parte no puede existir sin el todo.

En el diagrama esto se ve con el diamante negro. Significa que si eliminamos una automatización, también se eliminan las relaciones con sus dispositivos. Nos pareció lógico porque una automatización como "encender todas las luces a las 6 PM" no sirve si no tenemos luces que controlar.

Agregación

Para la relación entre **Usuario** y **Dispositivo** usamos agregación porque es más flexible. Un usuario puede tener varios dispositivos, pero si se va de la casa o cambia de sistema, los dispositivos pueden seguir existiendo y asignarse a otro usuario.

Es como cuando te mudas y dejas algunos electrodomésticos para el próximo inquilino. Los dispositivos tienen vida propia, no dependen completamente del usuario. En UML lo marcamos con el diamante blanco.

Asociación

También tenemos relaciones más simples, como entre **Usuario** y **Automatización**. Un usuario puede crear y manejar automatizaciones, pero ninguno de los dos depende del otro para existir. Pueden colaborar pero mantienen su independencia.

Esto nos pareció importante porque queremos que el sistema sea flexible y no esté todo muy acoplado entre sí.

Modularidad

Tratamos de que cada clase tenga una responsabilidad clara y bien definida. La clase **Usuario** se encarga de lo relacionado con los usuarios, **Dispositivo** maneja todo lo de los dispositivos, etc.

Esto hace que sea más fácil entender el código y también nos permite trabajar en diferentes partes del sistema sin pisarnos entre nosotros del grupo. Si después queremos agregar un nuevo tipo de dispositivo, podemos hacerlo sin tocar el resto del código.

Conclusión

Creemos que nuestro diagrama aplica bien los conceptos de POO que hemos visto en clase. Aunque al principio nos costó entender algunos conceptos como la diferencia entre composición y agregación, pensamos que el resultado final representa bien el problema que queremos resolver.

El diseño nos permite tener un sistema ordenado donde cada parte tiene su función específica, se puede reutilizar código y es fácil hacer cambios o agregar nuevas funcionalidades. Esperamos haber aplicado correctamente los principios de la programación orientada a objetos que estudiamos durante el cuatrimestre.

Sabemos que seguramente hay cosas que se pueden mejorar, pero estamos conformes con el trabajo realizado y creemos que es una buena base para empezar a programar el sistema.