IDAÑEZ, LUCIA MARÍA

Map => diccionario que tiene forma clave-valor

19/04 Ejercicio Resuelto en Clase

EJERCICIO MÓDULO EXPORTADOR **- PATRÓN STRATEGY Y ADAPTER -**

Módulo Exportador

Nos han solicitado el diseño y desarrollo de un módulo exportador de documentos. Los documentos contienen datos que estarán en formato de tabla, más específicamente en un Map, y se deberá permitir su exportación a Excel o PDF, según el usuario elija en momento de ejecución.

Para la exportación a Excel se utilizará *Apache POI*, ya que es una biblioteca sumamente confiable para estos usos. Es probable que se requiera actualizar su versión en un futuro, pero no cambiar a otra biblioteca distinta.

En cambio, para la exportación a PDF tendremos como primera opción a *Apache PDFBox*; pero, como por experiencias anteriores, estas herramientas siempre traen problemas, se deberá dejar abierta la posibilidad de reemplazarla por otra biblioteca.

Tendremos como referencia el repositorio global de Maven, donde podremos encontrar las distintas bibliotecas nombradas.

Si yo detecto que el comportamiento de la interfaz en las clases que la van a implementar es Stateless, es decir que no necesito guardar estado, usamos **interfaz**; en cambio, si detecto que tengo comportamiento que voy a necesitar guardar su estado, uso una **clase abstracta**.

En este ejercicio nos habla del "diseño y desarrollo de un módulo exportador de documentos", esto quiere decir que al ser **un módulo** nos hace referencia a que esto es una parte de un sistema más grande y que debe ser reutilizable (porque justamente es un módulo). Este módulo debe permitirnos exportar documentos y estos documentos contienen datos que van a estar en forma de tabla, particularmente en un tipo de dato de Java **Map**; a su vez, la exportación va a ser elegida por un tercero, esta puede ser en Excel o PDF.
Entonces, los inputs son los datos tabulares en un map y la salida va a ser el archivo físico generado, pero de nuestro modulo no va a salir el archivo físico, lo que puede llegar a salir de nuestro módulo es la ruta donde se guardó ese archivo, esto quiere decir que la salida es un String que hace referencia a dónde se guardó el archivo.

Si pensamos un segundo, este módulo es algo chiquito que forma parte de algo más grande, entonces ¿qué estamos queriendo generar? ¡UNA BIBLIOTECA! Así es, alguien va a utilizar esta biblioteca, pero no sabemos cómo.

Siguiendo con el enunciado, vemos que la tarea de generar un Excel y un PDF ya va a estar resuelta, para generar un Excel tenemos ApachePOI, pero nos dice que puede suceder que esa biblioteca requiera actualizarse, en cambio, para generar PDF, utilizaremos ApachePDFBox y nos dice que puede ser que esta biblioteca sí cambie (PROBLEMA). Con esto dicho, debemos tener en cuenta que no tenemos que tirar todo a la basura solo porque cambie la biblioteca, la idea sería cambiar la biblioteca minimizando el impacto del cambio y que el resto no se dé cuenta del cambio.

Pensando en cómo nos van a utilizar los demás, comenzamos planteando una clase principal **Exportador**, esta clase va a ser la que todos van a usar, y le vamos a agregar su método principal que vamos a llamar **exportar()** que, por ahora, va a tener dos parámetros "datos String" y "formatoExportar": **exportar(datos String, formatoExportar)** y devuelve una ruta (la ruta donde está guardado el archivo físico), en este caso String. A partir de esto, nuestro problema se divide en dos: cómo modelo los datos a exportar (datos de entrada) y cómo modelo el formato.

Empezamos con "cómo modelar los datos de entrada", sabemos que estos datos se quieren/deben ser en un **map** pero para tener todo más controlado y para que sea más expresivo modelamos una interfaz "Exportable" y le agregamos un método **getDatos()** que nos devuelve el **map** donde las claves van a ser Strings y los valores van a ser un List<String>, es decir **Map<String, List<String>>**.

->Con esto ya podemos ir modificando la firma del método que tenemos en **Exportador**, dando como resultado: **exportar(Exportable datos, formatoExportar)**

Para que una clase sea válida debe implementar la interfaz **Exportable**, ya que los datos a exportar por el Exportador deben estar sí o sí en el formato Exportable.

->Si queremos una clase Documento, esta clase debe implementar la interfaz Exportable.

Seguimos con el otro problema, modelar el formato; lo que vamos a hacer es separar el comportamiento de generación de PDF y generación de Excel en dos clases separadas, para ello, primero creamos una interfaz **EstrategiaDeExportación**.

->Con esto ya podemos ir modificando la firma del método que tenemos en **Exportador**, dando como resultado: **exportar(Exportable datos, EstrategiasDeExportacion estrategia)**

IDAÑEZ, LUCIA MARÍA

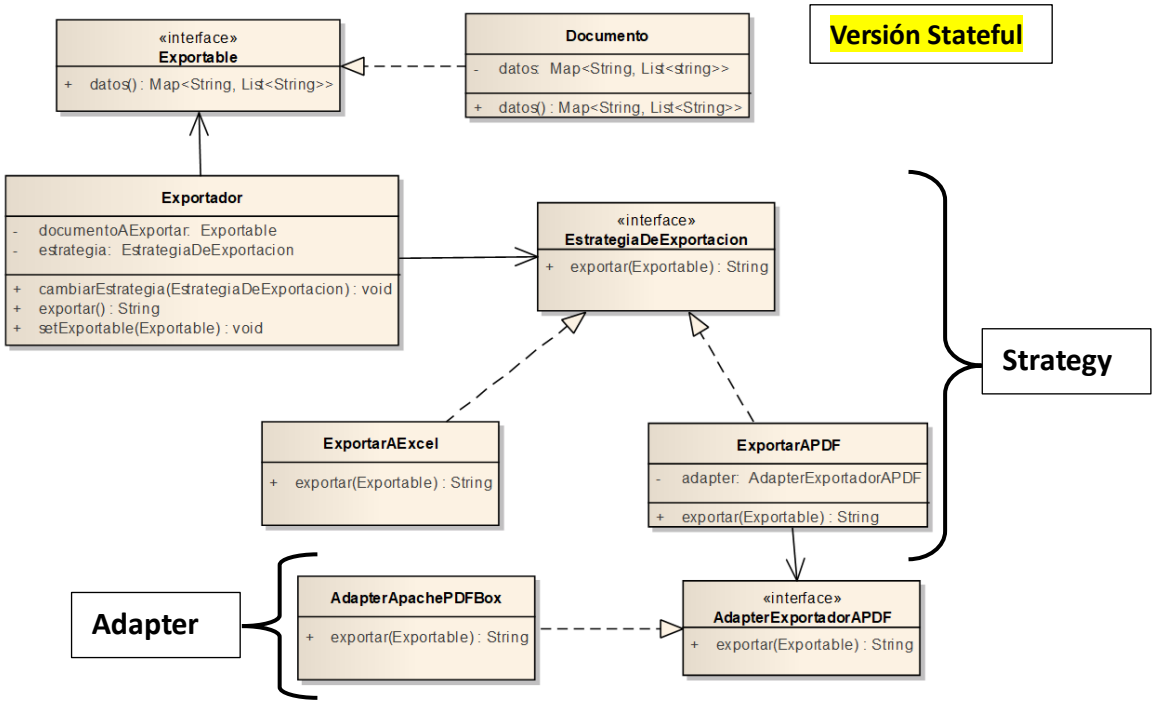
Una vez definida la interfaz **EstrategiasDeExportacion** sabemos que a partir de ella vamos a tener sí o sí, dos clases que la van a implementar, la primera de las clases va a ser **ExportarAExcel** y la otra **ExportarAPDF** y ambas van a tener un método llamado **exportar(Exportable): String** el cual va a recibir esos datos de tipo **Exportable** y nos va a dar como resultado la ruta de dónde está ese archivo físico generado.
Hasta acá logramos que la clase **Exportador** utilice de forma polimórfica a las clases **ExportarAExcel** y **ExportarAPDF**.

->Esto mejora la mantenibilidad, ya que, si se "rompe" o tenemos problemas con la clase ExportarAPDF, no afecta en nada a la clase ExportarAExcel, dado que están en clases separadas y no tendrían porqué conocerse.

Hasta ahora el diseño está bien, sin embargo, debemos tener en cuenta otras cosas que plantea el enunciado como el cambio de biblioteca para los PDF y la actualización de la biblioteca de Excel.

->En el caso del Excel, tenemos otra situación, ya que es solo una actualización, por ende, es poco probable que me rompan las firmas, los objetos son los mismos con los que trabajamos, las firmas siguen siendo las mismas, puede suceder que se sumen cosas, pero lo que ya está no se va a romper. Entonces, la parte del diseño ExpotarAExcel solo llega hasta ahí.

Por el lado de ExportarAPDF, es más complicado ya que el enunciado nos dice que la biblioteca pued cambiar, esto quiere decir que es TODO nuevo, por ende, vamos a utilizar un Adapter. Para esto, en la clase **ExportarAPDF** (previamente creada), le agregamos un atributo **adapter** que contenga un valor de tipo **AdapterExportadorAPDF**, esto es una interface que vamos a crear.
Cuando creamos esta interface **AdapterExportadorAPDF** le definimos el método **exportar(Exportable): String** y, ahora, es momento de crear una clase concreta **AdapterApachePDFBox** que IMPLEMENTE la interface AdapterExportadorAPDF. Esta clase concreta **AdapterApachePDFBox** es la que se va a integrar con la biblioteca y, si queremos cambiar la biblioteca, vamos a crear otra clase que se integre a esa biblioteca y solo deberemos modificar el valor del atributo **adapter** en la clase **ExportarAPDF**
La clase **ExportarAPDF** va a usar de forma polimórfica a todas las clases que implementen la interface **AdapterExportadorAPDF**, ya que las clases que implementen esa interface son las que se van a acoplar/integrar a las bibliotecas.



La versión que escribimos es Stateless, ya que también se pueden pasar por parámetro al método **exportar(_,_)** los parámetros Exportable y EstrategiaDeExportacion.

IDAÑEZ, LUCIA MARÍA

FINAL CUIDÁNDONOS – PATRÓN ADAPTER, STATE Y STRATEGY –

Cuidándonos

Contexto general

El Gobierno de la Ciudad nos ha contratado para la realización de un sistema de seguridad personal, el cual debe asegurar que los vecinos de todas las comunas puedan caminar de un destino a otro sin peligro. Será una aplicación Mobile en donde una persona, el ***transeúnte***, podrá escoger quienes serán sus ***cuidadores*** personales durante todo el trayecto que recorra.

Siendo este el dominio general, se deberán tener en cuenta las siguientes especificaciones:

1. De las personas nos interesa saber: nombre y apellido, dirección, edad y sexo.
2. Para que una persona sea cuidadora de otra deberá tener instalada la aplicación; o sea que al menos deberá ser un usuario pasivo. Se considera usuario activo a aquel que solicita los acompañamientos.
3. Cada vez que un usuario quiera ir hacia un ***destino***, deberá especificar la dirección exacta donde se encuentra actualmente y la del destino final; además de escoger quiénes serán sus cuidadores (puede haber un solo cuidador). Una vez especificados estos datos, se deberá presionar el botón de *confirmar cuidadores*. Los cuidadores seleccionados por el transeúnte serán notificados y deberán aceptar o rechazar el cuidado.
4. Si al menos un cuidador acepta la responsabilidad durante el trayecto, al transeúnte se le habilitará el botón de "*comenzar*". Al ser presionado este botón, el sistema deberá calcular el *tiempo de demora* aproximado y volverle a notificar a sus cuidadores. La distancia (en metros) entre dos direcciones será calculada por "Distance Matrix API" de Google, cuyo sistema nos brinda una interface REST.
5. Durante todo el recorrido, el sistema no deberá enviar notificaciones al transeúnte (por motivos de seguridad), ya que el mismo estará en movimiento.
6. Una vez que el transeúnte llegue a su destino, deberá presionar el botón "*llegué bien!*". El sistema deberá volver a habilitar las notificaciones, ya que se considera que no hay peligro alguno, y se deberá volver a notificar a sus cuidadores con esta situación.
7. Si algo malo sucede, el sistema deberá darse cuenta de esta situación por el tiempo aproximado que calculó. El mismo va a reaccionar frente a este incidente según lo que haya configurado el usuario:
   - Enviar un mensaje de alerta a sus cuidadores
   - Realizar una llamada automática a la policía
   - Realizar una llamada al celular del usuario
   - Esperar N minutos para ver si es una falsa alarma (los minutos deben ser parametrizables)

   Se debe considerar que pueden surgir nuevas formas de reaccionar frente a un incidente y que el usuario puede cambiar esta configuración cuantas veces quiera.

COSAS QUE SÍ O SÍ DEBEN ESTAR EN NUESTRO SISTEMA

→Persona (pueden ser transeúntes o cuidadores)
- ->Nombre
- ->Apellido
- ->Edad
- ->Sexo

→Trayecto | Tramo | Viaje | Recorrido
- ->Dónde estoy | Punto de Partida
- ->A dónde voy | Punto de Llegada
- ->Quiénes van a ser mis cuidadores? | Cuidadores deseados
- ->Quiénes son los cuidadores? | Cuidadores que aceptaron mi viaje
- ->Quién está realizando el viaje? | Transeúnte
- ->Tiempo de demora aproximado - -> se debe saber la unidad de tiempo
- ->Fecha y hora inicial
- ->En qué estado está mi viaje? | En Proceso, Realizado, Cancelado (¿hay más estados?)

→Detalle de estados del Viaje

A realizar
- El viaje todavía no comenzó, porque quizás estamos esperando que alguien lo acepte

En proceso
- Sabemos que al menos hay 1 cuidador
- No debemos enviarle notificaciones al transeúnte

Realizado
- Entra a este estado si el transeúnte presionó el botón de "llegué bien!
- El transeúnte puede recibir notificaciones.
- El Sistema debe avisarle a los cuidadores que el transeúnte llegó bien

En peligro
- Llegamos a este estado por haber notado una diferencia en el tiempo que debía llegar
- El Sistema debe reaccionar con lo que configuró, previamente, el transeúnte

IDAÑEZ, LUCIA MARÍA

→Formas de reaccionar ante el peligro

->Enviar mensaje de alerta a sus cuidadores

->Realizar una llamada automática a la policía

->Realizar una llamada al celular del usuario

->Esperar N minutos para ver si es una falsa alarma

→Algo que nos puede ayudar a saber si está Demorado el transeúnte es el CronTask y, si está demorado, es que está en peligro.

PATRONES QUE PODRÍAN AYUDARNOS

Para el requerimiento n°4:

"(…) el sistema deberá calcular el tiempo de demora aproximado (…). La distancia (en metros) entre dos direcciones será calculada por "Distance Matrix API" de Google, cuyo sistema nos brinda una interface REST."

En este caso tenemos una biblioteca externa que nos soluciona el cálculo de las distancias, sin embargo, no sabemos cómo funciona, pero sí su responsabilidad, entonces el patrón **Adapter** nos puede ayudar a desacoplar nuestro Sistema de la API de Google, además de permitirnos seguir adelante con el diseño e implementación sin integrar el componente todavía.

Para el requerimiento n°5:

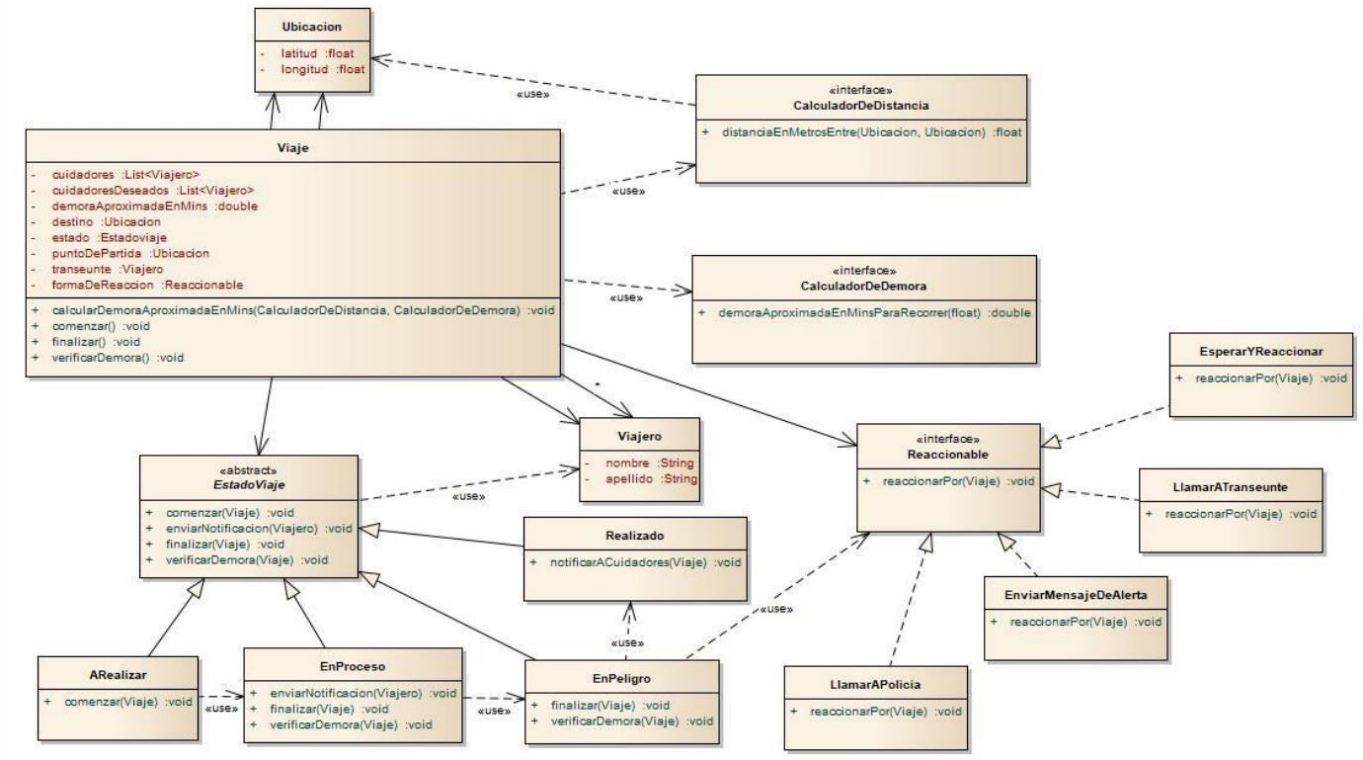
"Durante todo el recorrido, el sistema no deberá enviar notificaciones al transeúnte (por motivos de seguridad), ya que el mismo estará en movimiento."

Podríamos considerar la utilización del Patrón **State**, teniendo en cuenta que en un hipotético método "enviarNotificación" podría comportarse de forma diferente dependiendo el estado en el que esté el recorrido.

Para el requerimiento n°7:

"(…) Si algo malo sucede, el sistema deberá darse cuenta de esta situación por el tiempo aproximado que calculó. El mismo va a reaccionar frente a este incidente según lo que haya configurado el usuario (…)"

Para este caso tenemos una misma acción que puede tener diferentes comportamientos, esto nos indica que el patrón **Strategy** nos puede ayudar a modelar cada una de las distintas formas de reaccionar que existen para lograr una alta cohesión en esos componentes.



IDAÑEZ, LUCIA MARÍA

Clase 03/04 – Ejercicio control de stock y precio

Un **módulo** es un componente de software (un conjunto de clases) genérico que se podrá llevar de un sistema a otro, sin que quede atado a algo en particular.

Módulo de manejo de stock y precios

Nos han solicitado el diseño y desarrollo de un módulo que se encargue de calcular el stock de productos de cualquier local comercial, así como también el cálculo de precios de los mismos.

Se debe tener en cuenta que no solamente existen y se venden productos simples, sino que también existen combos. Un combo está formado por productos que se venden juntos. También pueden existir combos de combos.

Por ejemplo, un local de motos vende motos, cascos, guantes, chalecos, pilotos, entre otros productos; pero también vende algunos combos como por ejemplo guantes + casco + chaleco.

Cada producto tiene un precio en particular. El precio del combo es el resultado de la suma de los productos que contiene.

Además, se debe permitir aplicar descuentos a los distintos productos/combos, los cuales podrían ser acumulables.

Por último, nos han avisado que los productos/combos podrían ser vendidos con distintos packagings, cada uno de los cuales tiene un precio particular que se debería sumar al precio final del producto.

Empezamos considerando que no nos piden un sistema completo sino un MÓDULO; algo genérico que pueda ser utilizado por todos.

En la consigna aparecen algunas palabras clave: **productos**, **stock**, **precios**.

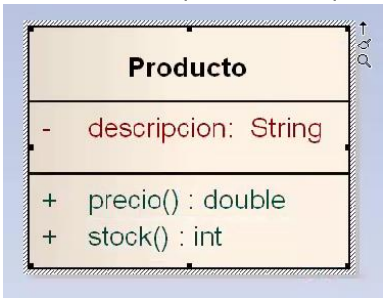
Otras palabras claves que aparecen son: **productos simples**, **combos**, **combos de combos**.

Luego, aparecen otras: **descuentos**, **packagings**.

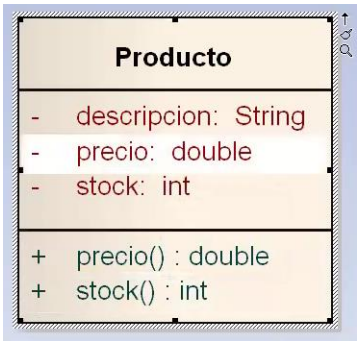
¿Por dónde arrancamos a modelar?

Primero, dividimos el problema en dos partes, por un lado, tenemos el modelado de los productos con el stock y precio, productos simples y combos y, por otro lado, tenemos los descuentos y packagings.

Entonces, arrancamos con una clase **Producto**, esta clase debe tener un atributo **descripción**; debemos tener en cuenta que UNA INSTANCIA de la clase Producto va a representar a todos los productos que hagan alusión a esa instancia, es decir si instanciamos un Producto Coca_Cola_600ml, esa instancia va a representar a todas las Coca Cola de 600ml. De los productos necesito el precio y stock, por ende voy a definir los métodos **precio()** y **stock()** -en nuestro caso, consideramos al **stock** por UNIDADES, esto quiere decir que este módulo no contempla kilos, gr u otro-.



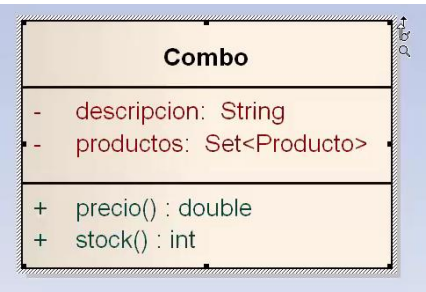
Pero nos faltan los atributos **precio** y **stock**, entonces si los colocamos, nuestros métodos quedarían como simples getters de nuestros atributos.



IDAÑEZ, LUCIA MARÍA 😸

Pero en nuestro enunciado nos dice que no solamente se venden productos simples sino también combos que están formados por productos, este modelo no permite y tampoco representa los combos, ¿qué hacemos?

Creamos una clase **Combo**, la cual podría tener una **descripción** como atributo y del **Combo** también nos interesa el **precio** y **stock**, por lo tanto agregamos los métodos **precio()** y **stock()**. También sabemos que el **Combo** está formado por productos, entonces le agregamos un atributo que represente eso, tal como **productos: Set<Producto>** en este caso acotamos un poco nuestro diseño, ya que el poner un **Set** estamos diciendo que los productos que estén en el combo no van a repetirse y por cada producto vamos a tener UNA unidad.



¿Cómo calculamos el precio y stock para los combos?

Por enunciado sabemos que el precio de un combo se calcula, es la sumatoria de los productos que conforman al combo, por esta razón no es necesario poner un atributo "precio" en la clase **Combo**. Entonces, para la clase **Producto**, **precio()** es un simple getter y en la clase **Combo**, **precio()** es un cálculo.

Para el caso del stock, debemos recordar que el Combo tiene UNA UNIDAD de cada producto, por ende, el **stock** de un **Combo** se calcula como el MÍNIMO DE TODOS LOS STOCK DE LOS PRODUCTOS QUE ESTÉN EN LA LISTA/COLECCIÓN DE PRODUCTOS DEL COMBO.

->Ejemplo: tenemos un Combo de 1 Coca y 1 alfajor, y tenemos 5 Coca y 8 alfajores, ¿cuántos combos podemos armar? Podemos armar 5 combos, ya que 5 es el mínimo stock entre los dos productos que conforman al combo.

→**Esto aplica para este caso, ya que tenemos UNA UNIDAD de cada producto en el Combo**

¿Pero cómo armamos los combos de combos?

Para nuestra solución actual, no podemos armar combos de combos, ya que **Combo** solo tiene una colección de **Producto**. Para combatir este problema estructural, realizamos unos cambios:

->A nuestra clase **Producto** le cambiamos el nombre y la llamamos **ProductoSimple**

->Creamos una clase abstracta que llamaremos **Producto**

->A la clase **Producto** le agregamos el atributo **descripción**, ya que es el atributo que, hasta ahora, se repetía en nuestras **ProductoSimple** y **Combo**.

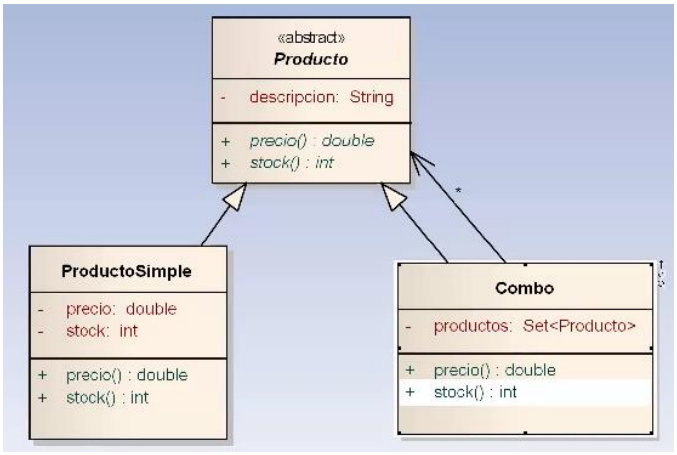
También, en nuestra clase **Producto** vamos a agregar los métodos abstractos **precio()** y **stock()** ya que nos interesa que todos productos nos puedan decir su **precio** y **stock**.

->Ahora vamos a establecer una relación de herencia entre **ProductoSimple** y **Producto**, lo mismo hacemos con **Combo**.

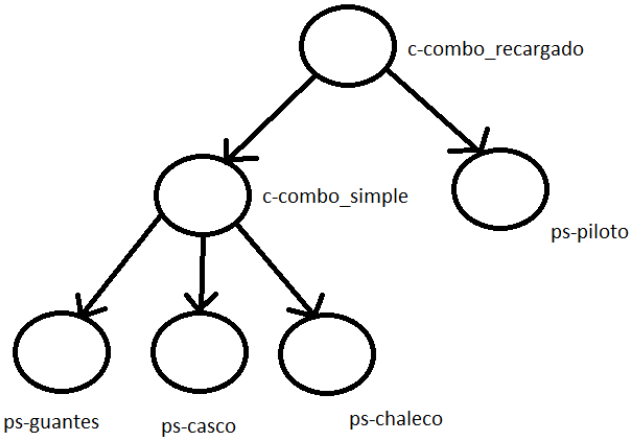
->Como paso final para la generación de combos de combos, recordemos que **Combo** tenía una colección de **Producto** que hacía referencia a la clase que ahora se llama **ProductoSimple**, entonces:

→¿quién es ahora **Producto**? La clase abstracta **Producto** que creamos, por ende, esa colección que está en la clase **Combo** hace referencia a la clase abstracta **Producto**, por esta razón dentro de los **Combo** vamos a poder tener tanto **ProductoSimple** como **Combo** ya que ambas clases comparten a un mismo padre.

IDAÑEZ, LUCIA MARÍA



Veamos un ejemplo para que sea más visible:



Veamos la estructura que tiene, este modelo tiene la estructura de un árbol donde las **HOJAS** son nuestros PRODUCTOS SIMPLES y los **NODOS INTERIORES** son los COMBOS. También notemos que los PRODUCTOS SIMPLES que no tienen aristas salientes, pero sí tienen o pueden tener una o más aristas entrantes; los COMBOS tienen aristas salientes y aristas entrantes.

Veamos un poco la implementación:

```java
public abstract class Producto {
    2 usages
    private String descripcion;

    no usages
    public Producto(String descripcion) {
        this.descripcion = descripcion;
    }

    no usages
    public String getDescripcion() {
        return descripcion;
    }

    no usages
    public abstract int stock();

    no usages
    public abstract double precio();
}
```

```java
public class ProductoSimple extends  Producto {
    2 usages
    private int stock;
    2 usages
    private double precio;

    no usages
    public ProductoSimple(String descripcion, double precio, int stock) {
        super(descripcion);
        this.stock = stock;
        this.precio = precio;
    }

    no usages
    @Override
    public int stock() {
        return this.stock;
    }

    no usages
    @Override
    public double precio() {
        return this.precio;
    }
}
```

```java
public class Combo extends Producto {
    3 usages
    private Set<Producto> productos;

    no usages
    public Combo(String descripcion) {
        super(descripcion);
        this.productos = new HashSet<>();
    }

    2 usages
    @Override
```

IDAÑEZ, LUCIA MARÍA

EL **PATRÓN COMPOSITE** NOS PERMITIÓ RESOLVER ESTO

IDAÑEZ, LUCIA MARÍA

El ejercicio aún no termina. El enunciado nos habla de:

->DESCUENTOS que se deben aplicar a los combos/productos. Los descuentos pueden ser acumulables;

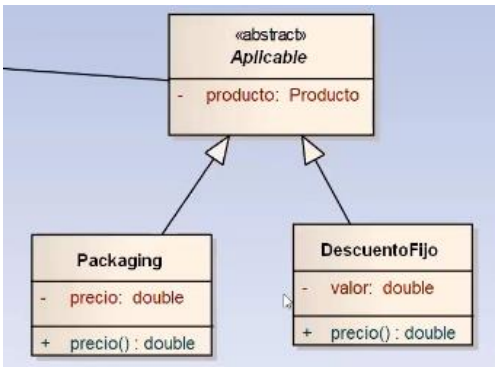
->PACKAGINGS, cada uno tiene un precio que se le debe sumar al precio total del producto.

Estas dos cosas tienen en común que ambos afectan al precio del Producto/Combo, ya sea sumando o restando.

¿Cómo modelo esto?

Generamos dos clases diferentes, primero la clase **Packaging** de la cual conocemos su **precio** que será un atributo y tendrá su getter, por otro lado, tenemos un **DescuentoFijo** y, considerando que es fijo el valor de descuento, nos interesa su **valor** y tendrá su getter.

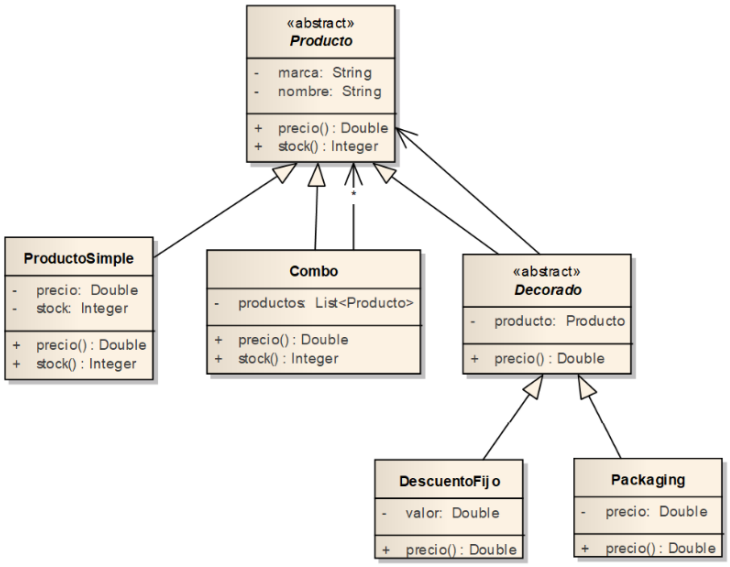
Para poder trabajar con estas dos clases que sabemos afectan al precio de un producto, vamos a generar una clase abstracta en común llamada **Aplicable**, de **Aplicable** van a heredar **Packaging** y **DescuentoFijo** y como sabemos que estas dos clases afectan al precio de un **Producto**, este debe estar como atributo en la clase **Aplicable**, es decir un atributo **producto: Producto**.



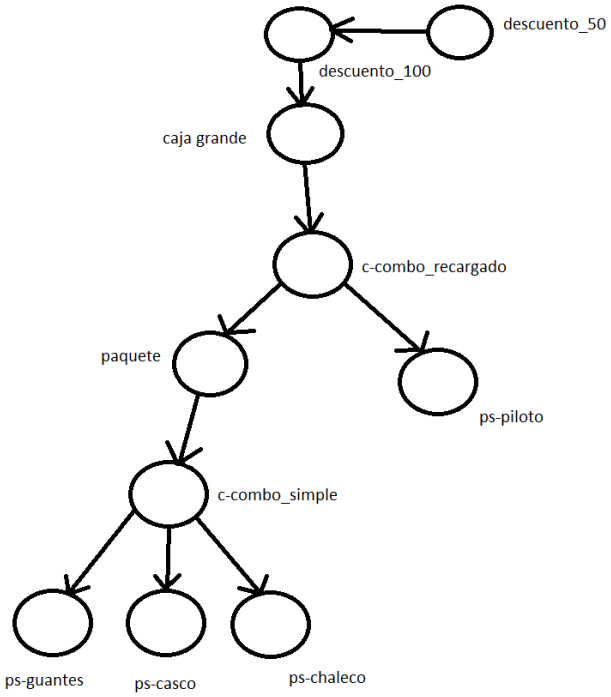
El precio final de un Producto se lo preguntamos al Packaging o al DescuentoFijo.

Este ejercicio todavía no está listo, ya que esta solución no permite que nuestros descuentos sean acumulables y tampoco permite que tengamos “paquetes adentro de paquetes”.

Para poder arreglar esto, vamos a hacer **Aplicable** herede de **Producto**, de esta manera vamos a permitir que se generen descuentos acumulables y que tengamos “paquetes dentro de paquetes” -se acumulen packagings- porque ahora dentro del atributo **producto: Producto** (de **Aplicable**) entra, además de un ProductoSimple o Combo, un Packaging y un DescuentoFijo.



IDAÑEZ, LUCIA MARÍA 😸
Un ejemplo gráfico:



Notemos que los **packaging** y los **descuentos** siempre tienen, como máximo, arista de saliente ya que pueden afectar a un producto y pueden tener más de una o ninguna arista entrante.

Veamos un poco la implementación:

```java
public abstract class Aplicable extends Producto {
    3 usages
    protected Producto producto;

    2 usages
    public Aplicable(String descripcion, Producto producto) {
        super(descripcion);
        this.producto = producto;
    }

    5 usages
    @Override
    public int stock() {
        return this.producto.stock();
    }
}
```

```java
public class DescuentoFijo extends Aplicable {
    2 usages
    private double valor;

    1 usage  1 related problem
    public DescuentoFijo(String descripcion, Producto producto, double valor) {
        super(descripcion, producto);
        this.valor = valor;
    }

    5 usages
    @Override
    public double precio() {
        return super.producto.precio() - this.valor;
    }
}
```

```java
public class Packaging extends Aplicable {
    2 usages
    private double precio;

    no usages
    public Packaging(String descripcion, Producto producto, double precio) {
        super(descripcion, producto);
        this.precio = precio;
    }

    5 usages
    @Override
    public double precio() {
        return super.producto.precio() + this.precio;
    }
}
```

EL **PATRÓN DECORATOR** NOS PERMITIÓ RESOLVER ESTO

IDAÑEZ, LUCIA MARÍA

Clase 31/05

Casa Segura

Dominio general

En esta oportunidad nos han contratado para desarrollar un sistema de seguridad integral para el hogar, el cual se llamará ISSD.
La finalidad de ISSD es cuidar al ***hogar*** para que el mismo esté protegido de las distintas amenazas o accidentes que pueden ocurrir durante el día (o la noche). Las amenazas y accidentes que se deberán tener en cuenta para esta primera etapa son:

- ***Robo***: se puede dar en cualquier momento del día. Los sensores de movimiento y las cámaras de seguridad deberán enviar sus datos recopilados cada cierto tiempo para que el sistema pueda detectar si se está produciendo un evento de este tipo. Ante la llegada de datos de cualquiera de estos dispositivos, los pasos para detectar si hay un robo son:
  1. *Decodificar los datos recibidos*: de ambos dispositivos solamente nos interesa el nivel de presencia humana y el grado de confiabilidad de este dato.
  2. *Analizar el grado de riesgo*: el riesgo puede ser bajo, medio o alto.
  3. *Verificar si existe un robo*: dependiendo el grado de riesgo aceptable configurado por el usuario, se decidirá si existe un evento de este tipo o no.
- ***Incendio***: también puede ocurrir en cualquier momento del día. En este caso, los sensores de humo enviarán los datos recopilados cada cierto tiempo al sistema para que éste detecte si los valores se encuentran dentro del rango normal esperado.
- ***Fuga de gas***: como no es tan probable que suceda, el detector de fuga de gas verificará que todo esté bien una vez por semana. El detector interactuará con el sistema para avisarle si hay o no una fuga de gas.
- ***Tensión baja de luz***: puede suceder en cualquier momento del día. La tensión será medida por un dispositivo que estará conectado a la red eléctrica entrante al hogar. Éste enviará el dato de la tensión entrante cada cierto tiempo para que el sistema evalúe si se encuentra dentro del rango esperado.

Ante la detección de alguna de estas amenazas o accidentes el sistema deberá reaccionar de una forma diferente:

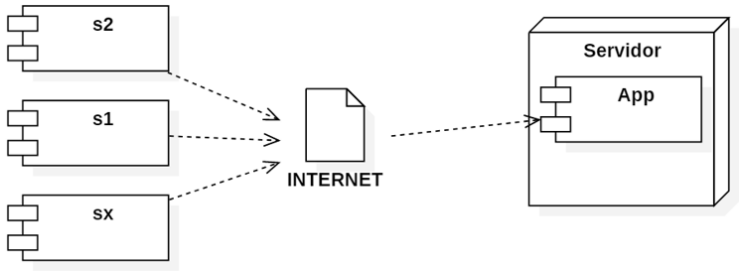
- Si se detecta un robo sonará la alarma y, además, se puede llamar a la policía y/o llamar a una persona confiable para dar aviso de esta situación. Estas acciones deberán poder ser configuradas por los propietarios del hogar.
- Si se detecta un incendio se deberá llamar a los bomberos y, opcionalmente y si el hogar lo dispone, se deberán abrir los rociadores de agua.
- Si se detecta una fuga de gas se deberá mostrar un aviso en la pantalla de control central que tendrá el hogar.
- Si se detecta baja tensión de luz se deberá mostrar un aviso en la pantalla central de control y, opcionalmente y si se dispone de este componente, encender el elevador de tensión.

Se debe tener en cuenta, además, que:

- Todos los eventos sucedidos deben quedar registrados para que puedan ser visualizados desde la pantalla central de control, junto con su fecha y hora.
- Del hogar nos interesa saber su dirección, su número de teléfono, su número de identificación en el sistema ISSD y su fecha de alta.
- Si el hogar está en peligro no se deberá mostrar su información por la pantalla central de control. Si no lo está, se podrá mostrar sin problemas. Se considera que un hogar está en peligro cuando sucede algún evento, y solo se puede volver a estar a salvo si el usuario así lo indica.

Para empezar con este ejercicio debemos entender qué parte del sistema estamos haciendo y dónde va a estar nuestro código.
Analicemos dos propuestas de Arquitectura:

1) Propuesta de Internet

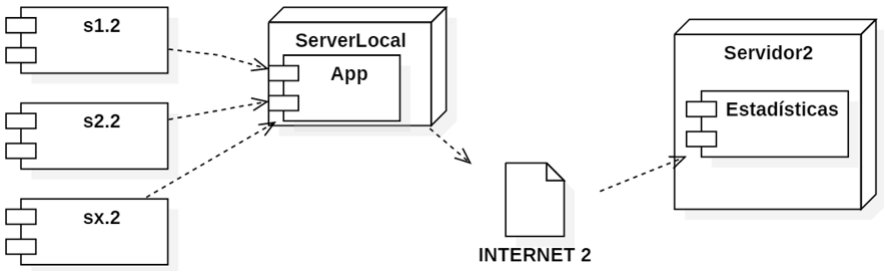


Análisis:

a) En base a **disponibilidad**, si me quedo sin internet, pierde propósito mi sistema ya que mi casa queda desprotegida y necesito asegurar la mayor disponibilidad posible.
b) Desde el punto de vista de **performance**, desde que los datos nos llegan, pasan a través de internet y se procesan en el servidor y verifican una respuesta, esa respuesta viaja a través de internet y vuelve a otros dispositivos, hay mucho tiempo y también se pierde parte del sentido de mi sistema.
    a. En cuanto a **procesamiento**, si tenemos mucho tráfico, sobrecargamos de procesamiento el servidor.
c) En cuanto a **seguridad**, puede ocurrir que nuestro servidor sea hackeado.

IDAÑEZ, LUCIA MARÍA

2) Propuesta Servidor Local



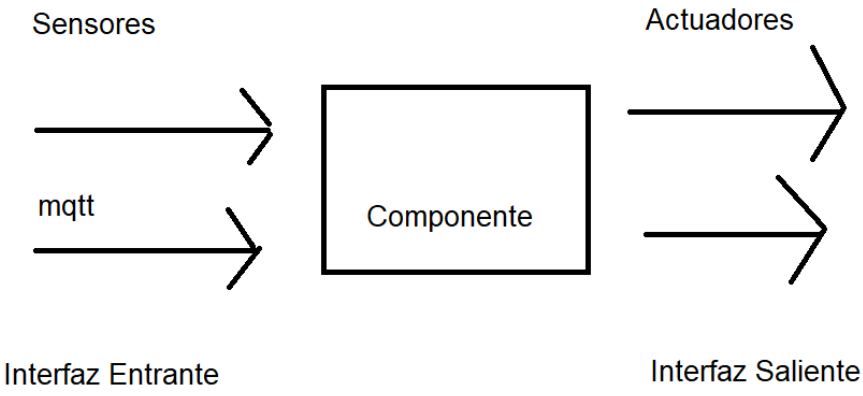
Nuestro aplicativo corre en nuestros hogares

Análisis:

a) En cuanto a **disponibilidad**, esta propuesta está mucho mejor ya que si se corta nuestro internet no sucede nada. Es menos probable un corte de luz que un corte de internet.
b) En cuanto a **performance**, el dato sale del aparato y ya nos llega a nosotros para ser procesado.
    a. El **procesamiento** tiene que ver con que cada hogar tendrá su propio servidor de procesamiento, en la propuesta anterior teníamos un solo servidor para todas las solicitudes de proceso.
c) Desde el punto de vista de la **seguridad**, es menos probable que ocurra un hackeo o algo por el estilo ya que nuestro servidor está aislado de todo el mundo, sin internet.
d) En cuanto a **actualizaciones**, presenta un inconveniente ya que si tenemos muchos clientes tendremos que realizar muchas actualizaciones casa por casa. En cambio, en la primera propuesta tenemos que realizar la actualización una vez (o la cantidad de servidores que tengamos).

<u>Preguntas que debemos hacernos: ¿Tiene sentido que mi sistema se rompa si ocurre algo?</u>
Tengo que preparar a mi servidor local para todo tipo de catástrofe.
->Visión de negocio: Si me sale muy caro blindar mi servidor local, pero, así como está, sin nada, me sale barato. Me conviene quedarme con la opción económica, ya que qué probabilidad hay que pase una catástrofe si justamente está diseñado para prevenirlas, además en caso de que ocurran y resulte dañado mi servidor, será económico recomponerlo.

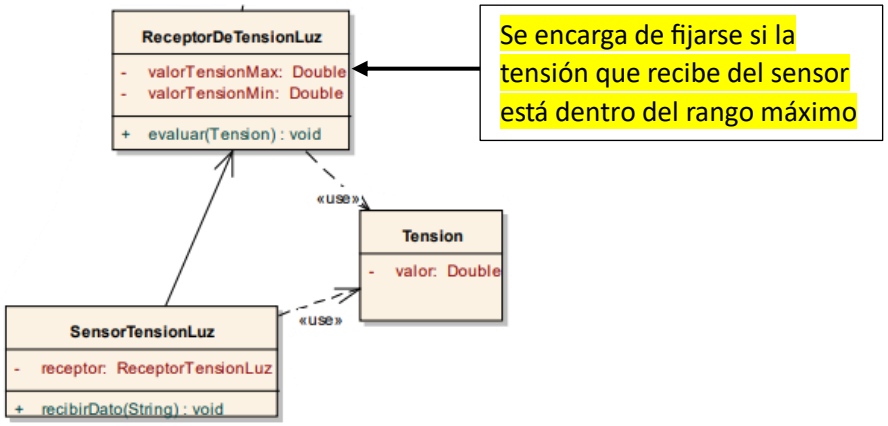
En nuestro diagrama de clase nunca debería haber un método buscarDato() porque nuestro sistema RECIBE LOS DATOS.



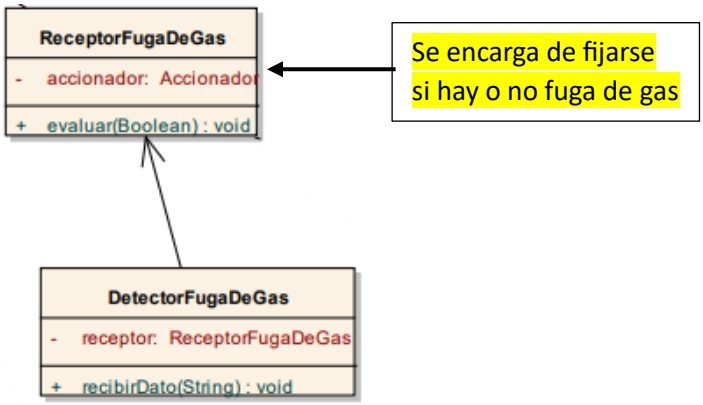
IDAÑEZ, LUCIA MARÍA

PARTE INTERFAZ ENTRANTE

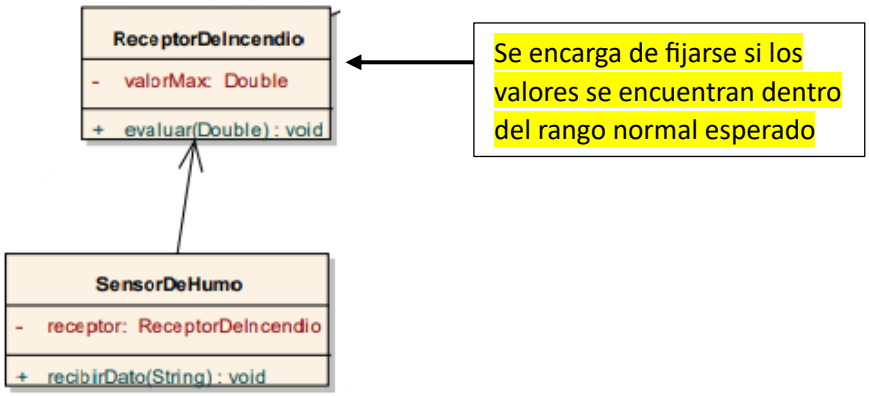
→TENSIÓN BAJA DE LUZ



→FUGA DE GAS



→INCENDIO

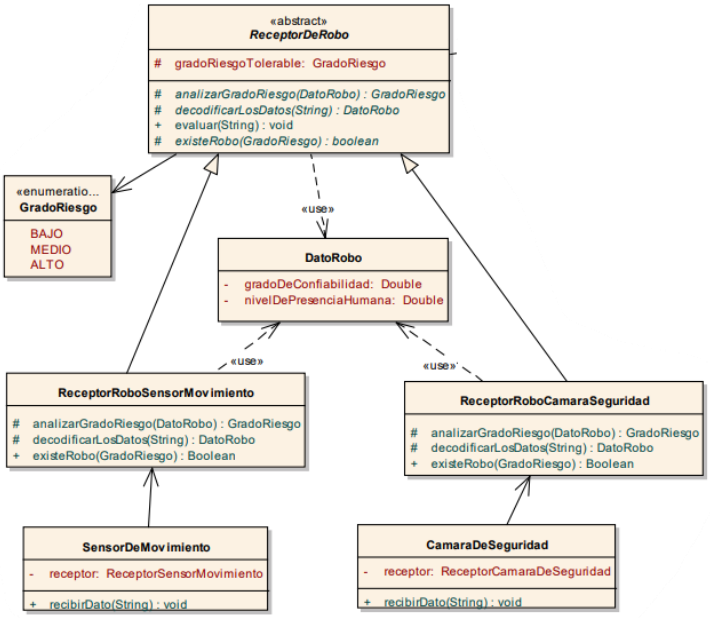


→ROBO -> Sensores de movimiento y cámara de seguridad

1) Decodificar: en los sensores y en la cámara de seguridad vienen los datos de “nivel de presencia humana” y “grado de confiabilidad”
    a. El parseo de los datos de la cámara de seguridad y el sensor son distintos
2) Analizar: cómo analizamos el grado de riesgo, se categoriza en BAJO, MEDIO y ALTO
3) Verificar: saber si existe o no un robo

Patrón Template Method, como la decodificación, el análisis y la verificación son diferentes para la cámara y el sensor, pero los pasos son los mismos se puede aplicar el patrón template method

IDAÑEZ, LUCIA MARÍA



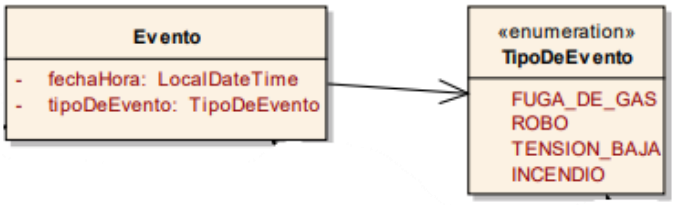
->Tenemos **n** instancias de sensores, cámaras, etc. siendo el **n** el número de dispositivos.
->Los receptores son los encargados de verificar si existe un evento.

MODELAMOS UNA CLASE HOGAR

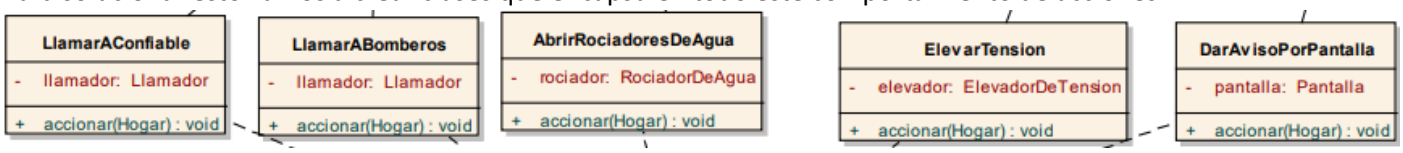
->Dependiendo de la arquitectura vamos a tener una o varias instancias de Hogar
->En el caso de la arquitectura de Internet vamos a tener varias instancias de hogar
->En el caso de la arquitectura de Server Local vamos a tener una única instancia y por lo tanto será una clase de tipo Singleton

¿QUÉ HACEMOS CUANDO OCURRE UN EVENTO?

Para dejar constancia de que ocurrió un Evento, vamos a dejarlo en un registro en la clase Hogar en un atributo **eventos:List<Evento>**
Entonces creamos la clase Evento:

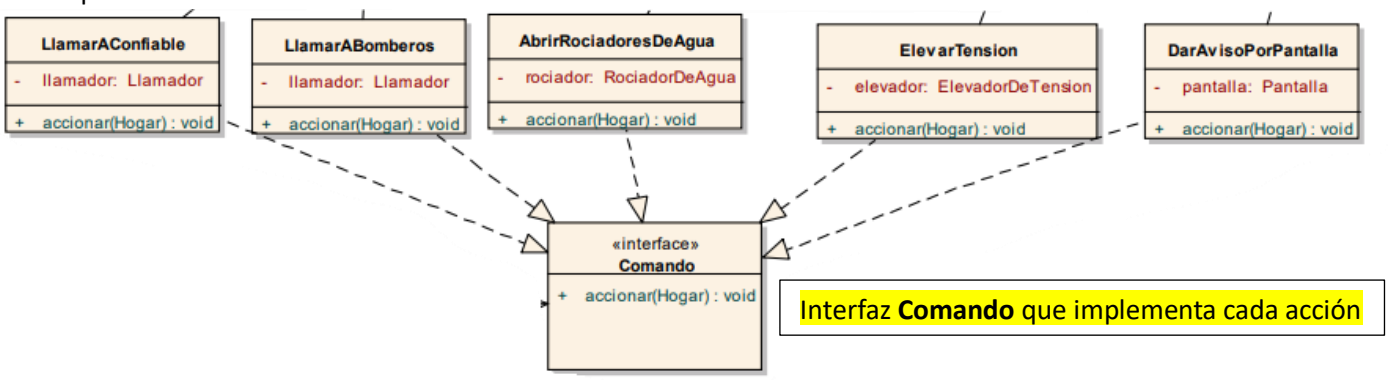


Cuando ocurre un Evento lo sabremos en cada uno de los receptores de cada Evento, es decir en el método **evaluar()** que tiene cada receptor. Por ende, allí haremos las acciones correspondientes a la respuesta de si está ocurriendo un Evento, entonces allí debemos registrar el Evento, pero si por cada Evento tenemos un Receptor entonces tendríamos **n** llamados a **registrarEvento(TipoEvento)** siendo **n** el número de receptores, esto nos indica código repetido que tendremos tanto en el registro de un evento como en las acciones que le corresponden.
Para solucionar esto vamos a crear clases que encapsulen todo este comportamiento de acciones:



->A cada una de las formas de reaccionar ante un Evento las encapsulo en una clase.

IDAÑEZ, LUCIA MARÍA 😸
->A su vez quiero que sea fácil crear nuevas formas de reaccionar ante un Evento, por lo tanto, creo una interfaz en común para todas ellas:



De esta manera podemos tratar polimórficamente a todas las acciones que implementen la interfaz Comando.
Para evitar el problema de la repetición de lógica creamos una clase ACCIONADOR que tendrá un método **sucedeEvento(TipoEvento)**, **registrarEvento(TipoEvento)** y **accionar()** y tendrá un atributo **comandos:List<Comando>**. Mi Accionador en su **sucedeEvento** tendrá:

```
sucedeEvento(tipoDeEvento){
        this.registrarEvento(tipoDeEvento);
        this.accionar();
}
```

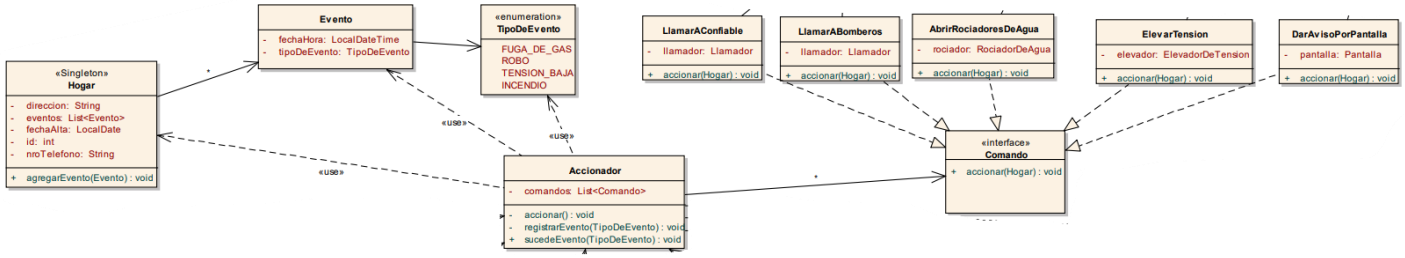
registrarEvento:

```
registrarEvento(tipoDeEvento){
        Evento evento = new Evento();
        evento.setTipo(tipoDeEvento);
        Hogar.getInstance().agregarEvento(evento);
}
```

Donde **accionar()** será:

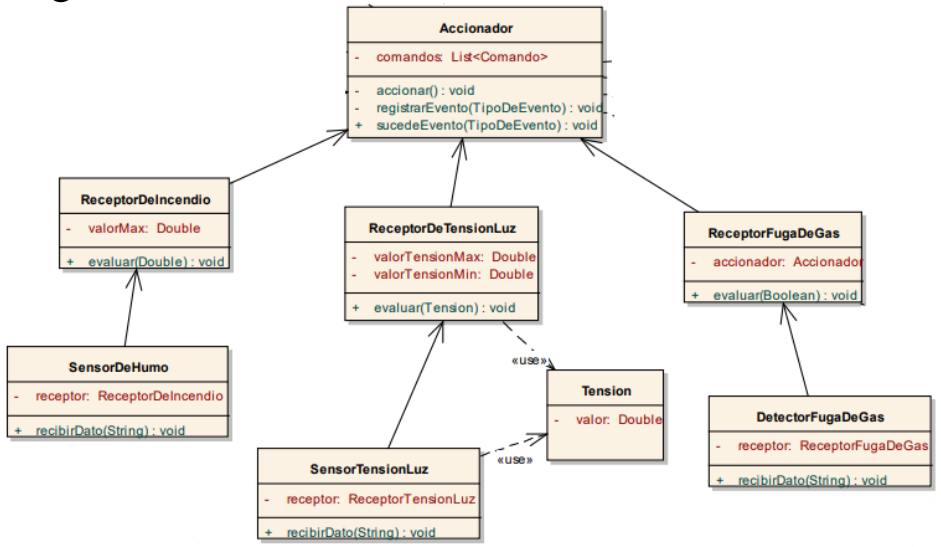
```
accionar() {
        this.comandos.forEach(c -> c.accionar(Hogar.getInstance()));
}
```

Es decir, que el método **sucedeEvento** registrará el Evento en el Hogar y aplicará todas las acciones de respuesta ante un Evento en el Hogar.



Al **Accionador** no le importa la acción o Comando, es decir no le interesa si es LlamarAConfiable, ElevarTensión u otro, porque los trata a todos polimórficamente gracias a la interfaz Comando.
->El Accionador usa a la clase hogar para ejecutar los Comandos sobre ella
Pero ¿cómo relacionamos esto con los Receptores?
Cada receptor implementa la lógica de “hacer algo” si se detecta que está sucediendo un Evento, ejemplo: llamar a la policía si ocurre un robo, pero ahora esa lógica está en Accionador, el cual tiene una lista de Comandos que son las acciones en respuesta a un Evento, por lo tanto, cada Receptor debería tener un atributo Accionador.

IDAÑEZ, LUCIA MARÍA



Y, por ejemplo, en **ReceptorFugaGas** el método **evaluar(Boolean)** será:

```
evaluar(existeFugaDeGas){
        if(existeFugaDeGas)
                this.accionador.sucedeEvento(TipoDeEvento.FUGA_DE_GAS);
}
```

De esta manera cada Receptor tendrá su propia instancia de Accionador y si el Evento está ocurriendo llamará a **this.accionador.sucedeEvento(Tipo_Evento)**.
Esto se llama PATRÓN COMMAND.

<u>EJERCICIO MÓDULO EXPORTADOR:</u> MEJORADO - PATRÓN FACTORY -

Antiguamente, desde el punto de vista de un tercero que utiliza nuestro módulo, podemos detectar que conocemos muchas clases que pertenecen a nuestro componente, esto no estaría del todo bien, ya que quien usa nuestro módulo debe conocer la menor cantidad de componentes posibles de nuestro módulo con el fin de facilitarle el uso de este, que solamente sepa QUÉ hace nuestro módulo sin importar CÓMO lo hace. Además, perdemos el control en nuestro módulo sobre cómo lo usan los terceros.
En nuestro caso, un tercero conocía nuestra clase Exportador, ExportarAExcel, ExportarAPDF y AdapterApachePDFBox.

Para mejorar esto comenzamos creando una clase **FactoryFormatoDeExportacion** y que contenga un método de clase (static) **crear()** que devuelva una EstrategiaDeExportacion, a su vez **crear()** va a recibir un **formato** (como un String) y el **nombreDelArchivo** (un String).

```
public class FactoryFormatoDeExportacion {

    no usages
    public static EstrategiaDeExportacion crear(String formato, String nombreDelArchivo) {

    }
}
```

Dentro de este método vamos a concentrar la lógica de instanciación de las distintas estrategias de exportación. Para esto vamos a evaluar la condición, esta condición es el **formato**, esto quiere decir que por cada formato vamos a configurar una estrategia distinta.

```
public static EstrategiaDeExportacion crear(String formato, String nombreDelArchivo) {
    EstrategiaDeExportacion estrategia;
    switch (formato) {
        case "PDF": estrategia = new ExportarAPDF(new AdapterApachePDFBox(nombreDelArchivo)); break;
        case "EXCEL": estrategia = new ExportarAExcel(nombreDelArchivo);
        default: throw new NoExisteFormatoException();
    }
    return estrategia;
}
```

IDAÑEZ, LUCIA MARÍA 😸

Pero ¿cómo utilizamos esto ahora?
Anteriormente, lo hacíamos de esta manera:

```
this.exportador.exportar(this.documento, new ExportarAExcel( nombreDeArchivo: "datos.xlsx"));
```

Ahora nos quedaría así:

```
this.exportador.exportar(this.documento, FactoryFormatoDeExportacion.crear( formato: "EXCEL",  nombreDelArchivo: "datos.xlsx")));
```

Llamamos al **FactoryFormatoExportacion** y a su método **crear** el cual nos devolverá la estrategia de exportación que utiliza el **exportador**.
Sin embargo, podemos ocultar un poco más el detalle de nuestro módulo. Para esto, nos vamos a la clase **Exportador** y en el método **exportar**, en vez de recibir la estrategia, voy a recibir el **formato** y el **nombreDelArchivo**.

```
public class Exportador {

    2 usages
    public String exportar(Exportable exportable, String formato, String nombreDelArchivo){
        EstrategiaDeExportacion estrategia = FactoryFormatoDeExportacion.crear(formato, nombreDelArchivo);
        return estrategia.exportar(exportable);
    }
}
```

Quedando la utilización del módulo de la siguiente manera:

```
this.exportador.exportar(this.documento,  formato: "PDF",  nombreDelArchivo: "datos.pdf"));
```

Ya no conoce muchas clases de nuestro componente, solo la clase Exportador.

Clase 28/06
FRIVEGA **- PATRÓN BUILDER -**
→BUILDER => Configurar un objeto en pasos
Supongamos que en este ejercicio tenemos a los Productos, si los queremos utilizar, sí o sí le tengo que dar un **nombre**, un **precioBase**, un **Tipo** y, opcionalmente, una **descripcion**. A su vez, supongamos que quien maneja el sistema quiere configurar al objeto en pasos y quiero asegurar que el objeto tenga el nombre, el precioBase y el Tipo. Esto nos indica que queremos tener una instancia de un objeto bien configurada para que esa instancia sea consistente.
Para esto vamos a crear una clase que se llame **ProductoBuilder** que va a tener como atributo un **producto**, este producto va a ser el objeto que vamos a crear de forma consistente. Dentro de esta clase vamos a tener un método llamado **construir()** (o famosamente **build()**), el cual va a devolver la instancia del producto que pretendemos configurar, es decir que devuelve el atributo **producto** de nuestra clase; antes de devolver la instancia realiza otros procesos.

```
public class ProductoBuilder {
    1 usage
    private Producto producto;

    no usages
    public Producto construir() {
        //
        return this.producto;
    }
}
```

A su vez, tenemos otro método que va a devolver algo de tipo **PruductoBuilder**, este método lo vamos a llamar **conNombre(String nombre)**, el cual recibe el nombre del producto de tipo String, y lo que hace este método es configurar la instancia que teníamos de atributo, es decir que le setea a ese **producto** el nombre que recibe por parámetro y luego se devuelve a sí mismo.

IDAÑEZ, LUCIA MARÍA

```
public ProductoBuilder conNombre(String nombre){
    this.producto.setNombre(nombre);
    return this;
}
```

También realizamos otro método que se llame **conPrecioBase(double precioBase)** y configuramos el precioBase del **producto**, es decir que se setea el precioBase a ese **producto** y luego se devuelve a sí mismo.

```
public ProductoBuilder conPrecioBase(double precioBase) {
    this.producto.setPrecioBase(precioBase);
    return this;
}
```

Hacemos lo mismo con el **Tipo**:

```
public ProductoBuilder conTipo(TipoDeProducto tipoDeProducto) {
    this.producto.setTipo(tipoDeProducto);
    return this;
}
```

A ese **producto** que tiene como atributo nuestra clase **ProductoBuilder** lo vamos a instanciar en el constructor de nuestra clase:

```
public ProductoBuilder() {
    this.producto = new Producto();
}
```

Por ende, cada vez que yo instancio un ProductoBuilder, esta clase dentro suyo ya instancio el **Producto** pero no le mandó nada sino que define los métodos para configurar el **producto**.
Volviendo un poco al contexto, nuestra regla de negocio decía que todos nuestros productos deben tener nombre, precioBase y Tipo, pero con esto que realizamos hay un problema, podemos instanciar un objeto Producto sin alguna de esas tres cosas.

```
ProductoBuilder productoBuilder = new ProductoBuilder();

Producto tv50 = productoBuilder
        .conNombre("TV 50")
        .conPrecioBase(80.0)
        .construir();
```

Esto es posible pero no debería serlo

Por lo tanto, debemos tener validaciones para esto, esto quiere decir que ProductoBuilder debería chequear que el objeto tenga la configuración correcta para que se cumpla esa regla de negocio.
Las validaciones necesarias para nuestra regla de negocio las vamos a realizar en el método **construir()**.

```
public Producto construir() {
    if( this.producto.getNombre() == null ) {
        throw new ProductoSinNombreException();
    }
    if( this.producto.getPrecioBase() == null ) {
        throw new ProductoSinPrecioException();
    }
    if( this.producto.getTipo() == null ) {
        throw new ProductoSinTipoException();
    }
    return this.producto;
}
```

IDAÑEZ, LUCIA MARÍA 😸
En el método **construir()** estamos chequeando todas las cosas que sean reglas de negocio, es decir que no devolvemos una instancia a la cual no se le hayan configurado ciertas cosas.