

## PRÁCTICA 3 APLICACIÓN PATRÓN DE CADENA DE MANDO A JSON.

### Contenido

Introducción.....	1
Esquema inicial del proyecto.....	2
Aplicación patrón cadena de mando.....	3
Aplicación patrón plantilla.....	7
Conclusión.....	10

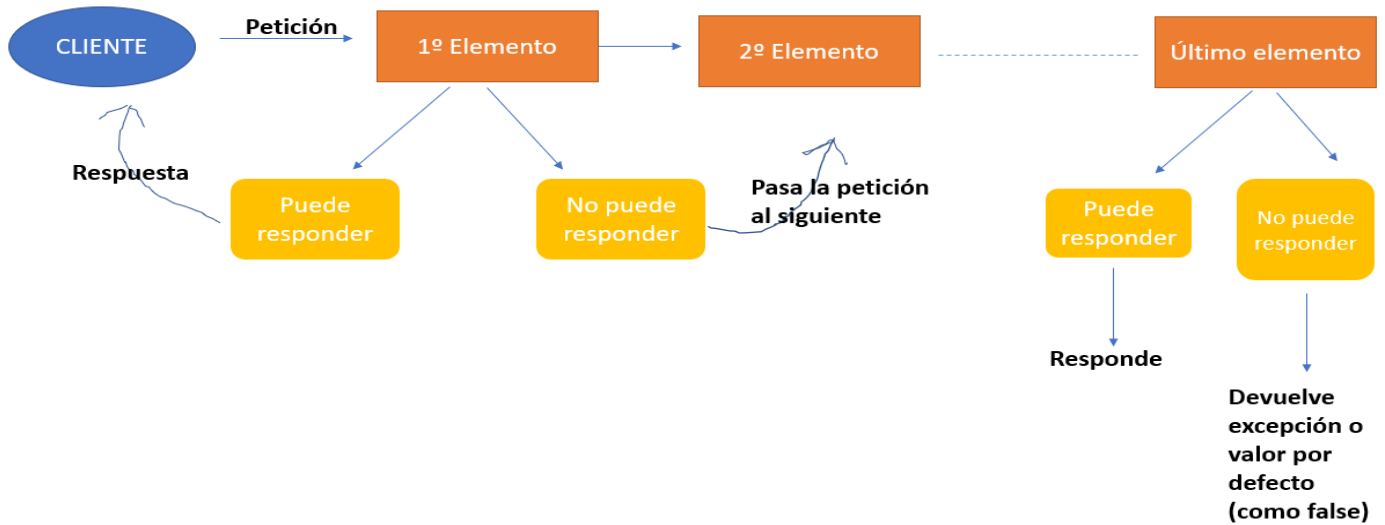
### Introducción.

En esta práctica el objetivo es aplicar el patrón cadena de mando sobre un archivo Json con información sobre medicamentos, ingredientes activos, referencias de inhaladores, etc.

El motivo por el que se aplica este diseño es porque si este archivo se emplea para construir una aplicación y esta misma está en constante cambio, al irse añadiendo etiquetas al archivo, la funcionalidad de la clase DatabaseJsonReader se amplía violando el principio abierto/cerrado. Por ello, queremos aplicar un patrón que nos permita modificar los aparatados que se van a leer en el JSon sin tener que modificar código que funciona perfectamente. Por ello, se aplicará el patrón cadena de mando, pero antes, voy a explicar por encima en qué consiste.

- Patrón cadena de mando:

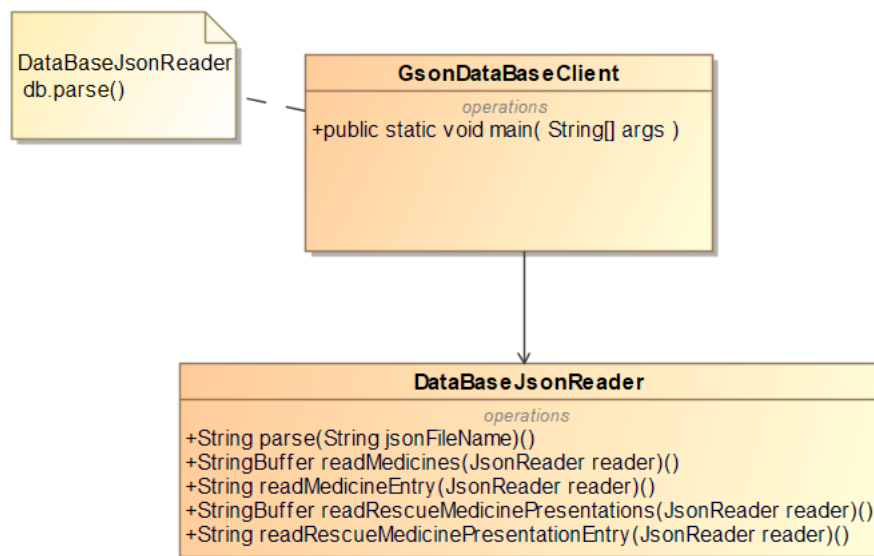
Es un patrón que tiene como objetivo evitar que más de un objeto pueda responder a una petición del cliente. Por este motivo, se encadenan los elementos receptores y las peticiones se pasarán de unos a otros hasta que uno de ellos esté autorizado para hacerlo, es decir, que tenga información suficiente para poder elaborar una respuesta. El esquema sería algo parecido al siguiente:



Por tanto, el modelo más común suele ser un cliente, una clase elementoCadenadeMando que puede ser abstracta o no, y los elementos de la cadena que heredan de esta última clase.

## Esquema inicial del proyecto.

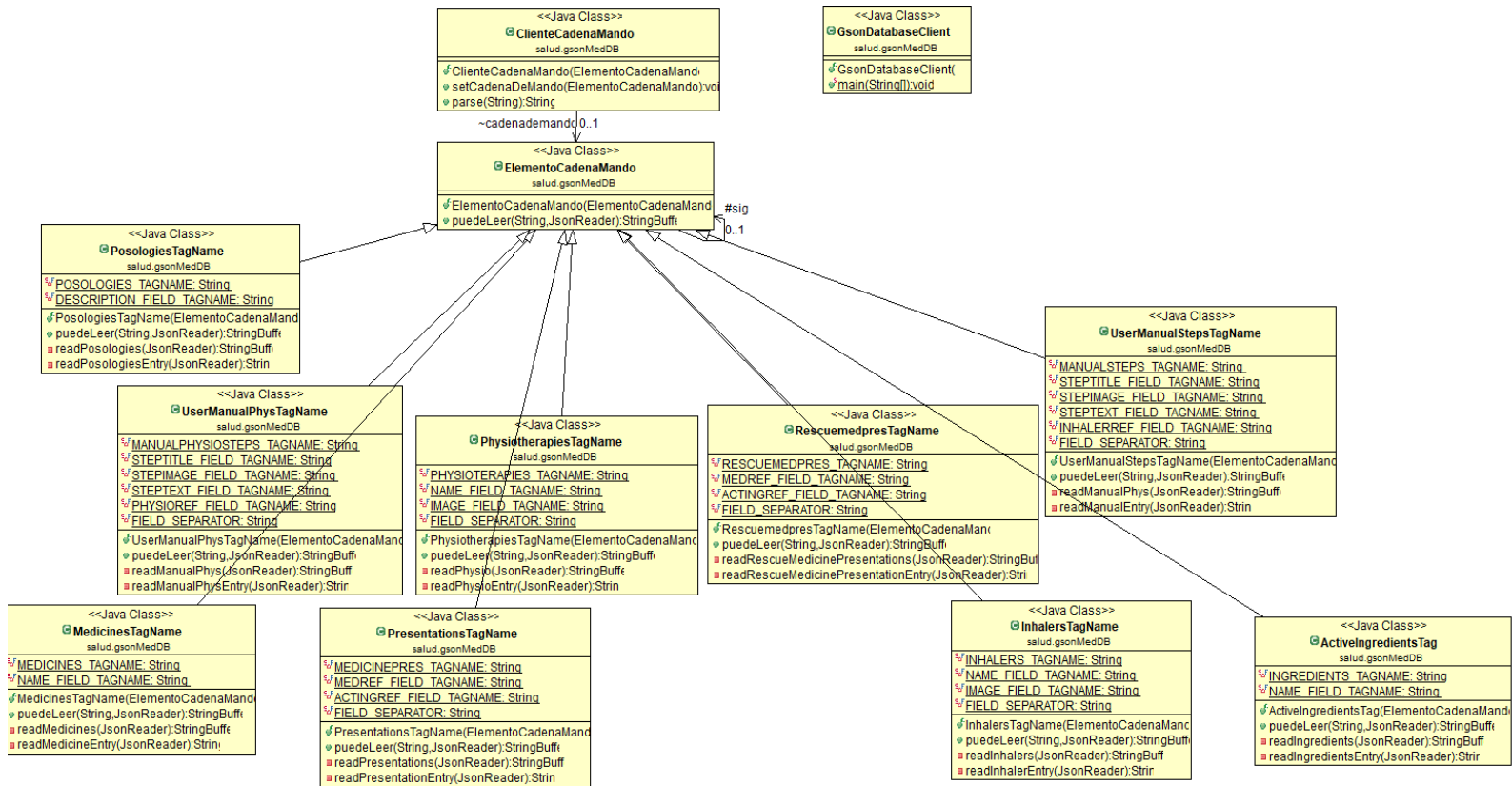
El proyecto proporcionado en el campus virtual cuenta con la clase DataBaseJsonReader que se encarga de abrir el archivo Json y leer sus distintos apartados, y la clase GsonDatabaseClient que se encarga de “enviar” a la clase anterior el archivo que se quiere abrir.



## Aplicación patrón cadena de mando.

Para aplicar el patrón voy a seguir el ejemplo que se propuso en clase de la autorización para acceder a distintas redes sociales o páginas web.

El diagrama de clases con el que me voy a guiar es el siguiente:



He eliminado la clase DataBaseJson pues su contenido lo he pasado a la clase **clienteCadenaMando** que se encarga de abrir el archivo y leer la primera etiqueta que corresponde a medicines, physiotherapies, inhalers, etc.

```
package salud.gsonMedDB;
import java.io.File;

public class ClienteCadenaMando {
    ElementoCadenaMando cadenademando;
    public ClienteCadenaMando(ElementoCadenaMando cm) {
        cadenademando = cm;
    }
    public void setCadenaDeMando(ElementoCadenaMando ncm) {
        cadenademando = ncm;
    }
    public String parse(String jsonFileName) throws IOException {
        InputStream usersIS = new FileInputStream(new File(jsonFileName));
        JsonReader reader = new JsonReader(new InputStreamReader(usersIS, "UTF-8"));
        reader.beginObject();
        StringBuffer readData = new StringBuffer();
        while (reader.hasNext()) {
            String name = reader.nextName();
            readData.append(cadenademando.puedeLeer(name, reader));
        }
        reader.endObject();
        reader.close();
        usersIS.close();
        return new String(readData);
    }
}
```

Por su parte, la clase **elementoCadenaMando** crea la cadena de elementos y llama a la función **PuedeLeer()** de cada elemento que forma parte de la cadena.

```
package salud.gsonMedDB;
import java.io.IOException;
import com.google.gson.stream.JsonReader;

public class ElementoCadenaMando {
    protected ElementoCadenaMando sig;
    public ElementoCadenaMando(ElementoCadenaMando s) {
        sig = s;
    }
    public StringBuffer puedeLeer(String name, JsonReader reader) throws IOException {
        return sig.puedeLeer(name, reader);
    }
}
```

El método **puedeLeer** tiene como parámetros de entrada un **String** con el nombre de la etiqueta principal y un **JsonReader**, desde el que se leen las distintas entradas para cada apartado.

Por otra parte, la arquitectura de los elementos que forman la lista enlazada es muy similar en todos ellos.

Todos heredan de la clase **elementoCadenaMando** y poseen un método para leer el **JsonReader** en el caso de que el **String** coincida con el apartado al que tienen acceso. Si no es así, pasan los atributos de entrada a la cadena hasta que un elemento lo pueda leer o cuando no haya sido leído, cosa que se muestra devolviendo un mensaje.

Los métodos para leer las entradas del apartado también son muy parecidos para todos los componentes, pues varían solo en el nombre de las entradas.

Mostraré la estructura de la clase ActiveIngredientsTag, por ejemplo:

```
public class ActiveIngredientsTag extends ElementoCadenaMando {
    private static final String INGREDIENTS_TAGNAME = "activeIngredients";
    private static final String NAME_FIELD_TAGNAME = "name";
    public ActiveIngredientsTag(ElementoCadenaMando s) {
        super(s);
    }
    public StringBuffer puedeLeer(String name, JsonReader reader) throws IOException {
        StringBuffer readIngredients = new StringBuffer();
        if (name.equals(INGREDIENTS_TAGNAME)) {
            readIngredients.append(INGREDIENTS_TAGNAME+":");
            readIngredients.append("\n");
            readIngredients.append(readIngredients(reader)).append("\n");
        }
        else {
            if (sig != null) {
                return super.puedeLeer(name, reader);
            }
            else {
                reader.skipValue();
                System.err.println("Category " + name + " not processed.");
            }
        }
        return readIngredients;
    }
    private StringBuffer readIngredients(JsonReader reader) throws IOException {
        StringBuffer ingredientsData = new StringBuffer();
        reader.beginArray();
        while (reader.hasNext()) {
            reader.beginObject();
            ingredientsData.append(readIngredientsEntry(reader)).append("\n");
            reader.endObject();
        }
        ingredientsData.append("\n");
        reader.endArray();
        return ingredientsData;
    }
    private String readIngredientsEntry(JsonReader reader) throws IOException {
        String actName = null;
        while(reader.hasNext()){
            String name = reader.nextName();
            if (name.equals(NAME_FIELD_TAGNAME))
                actName = reader.nextString();
            else {
                reader.skipValue();
            }
        }
        return actName;
    }
}
```

Como ya he comentado, el resto de elementos presentan una arquitectura muy similar, por ello más adelante aplico el patrón plantilla para eliminar la repetición de código.

Por último, tenemos la clase GsonDatabaseClient que el único cambio que presenta con respecto al proyecto inicial es que antes de poder pasar el archivo a la clase ClienteCadenaMando, hay que crear la cadena de mando. Como ya he comentado, es lo mismo que una lista enlazada.



Hay que destacar que los elementos se añaden desde el último al primero en ese sentido.

```
public class GsonDatabaseClient {

    public static void main(String[] args) {
        try{
            RescuemedpresTagName rescMedPres=new RescuemedpresTagName(null);
            ActiveIngredientsTag ingredients= new ActiveIngredientsTag(rescMedPres);
            MedicinesTagName medicines= new MedicinesTagName(ingredients);
            PhysiotherapiesTagName physio= new PhysiotherapiesTagName(medicines);
            InhalersTagName inhalers= new InhalersTagName (physio);
            PosologiesTagName posologies = new PosologiesTagName(inhalers);
            PresentationsTagName presentations = new PresentationsTagName(posologies);
            UserManualPhysTagName manualPhys= new UserManualPhysTagName(presentations);
            UserManualStepsTagName manualSteps= new UserManualStepsTagName(manualPhys);
            ClienteCadenaMando ccm = new ClienteCadenaMando(manualSteps);
            try {
                System.out.println(ccm.parse("./datos.json"));
            } finally {
            }
        } catch (FileNotFoundException e){
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Como es de esperar, el resultado de la ejecución de esta última clase es el siguiente:

```
medicines:
bretarisName
ekliraName
brimicaName
duaklirName
enurdevName
seebriName
tovanorName
ulunarName
ultibroName
xoternaName
hirobriezName
ponbrezName
spirivaName
striverdiName
incruseName
anoroName
relvarName
pulmicortName
symbicortName
symbicortForteName
rilastName
rilasFortetName
oxistName
flivotidoName
```

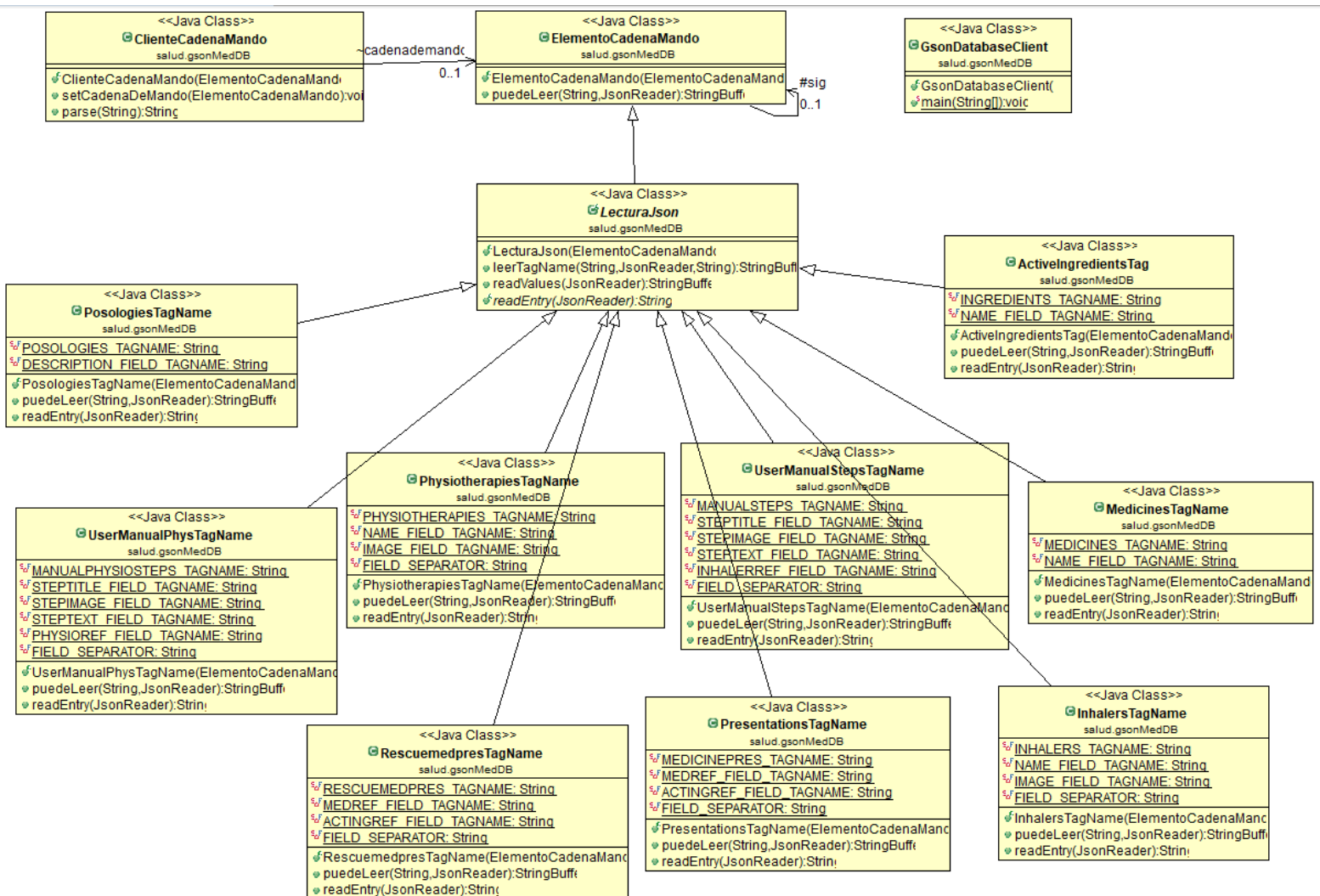
## Aplicación patrón plantilla.

Como he comentado antes, el programa funciona correctamente pero el diseño no acaba de ser óptimo pues hay mucha repetición de código que podría simplificarse.

Poe eso, he creado un nuevo proyecto en el que voy a refactorizar el código para que sea más bonito estilísticamente hablando.

Utilizaré el patrón plantilla, ya que su función es subir a una clase madre todos los métodos (o partes comunes) que coinciden en las clases que van a heredar de ella.

El diagrama de clases de mi proyecto es el siguiente:



Como se puede observar, la principal diferencia es que ahora hay una clase nueva, LecturaJson, que hereda de ElementoCadenaMando y los elementos de la cadena heredan de LecturaJson.

Esta contiene los métodos que se repetían, puedeLeer y readValues, y un método abstracto para la lectura de las entradas que es distinto en todas las clases hijas.

La estructura de la clase LecturaJson es la siguiente:

```
public abstract class LecturaJson extends ElementoCadenaMando {
    public LecturaJson(ElementoCadenaMando s) {
        super(s);
    }
    //common method for reading the Json File
    public StringBuffer leerTagName(String name, JsonReader reader, String tagName) throws IOException {
        StringBuffer readTag = new StringBuffer();
        if (name.equals(tagName)) {
            readTag.append(tagName+":");
            readTag.append("\n");
            readTag.append(readValues(reader)).append("\n");
        }
        else {
            if (sig != null) {
                return super.puedeLeer(name, reader);
            } else {
                reader.skipValue();
                System.err.println("Category " + name + " not processed.");
            }
        }
        return readTag;
    }
    //common method for reading the values of a Tag
    public StringBuffer readValues(JsonReader reader) throws IOException {
        StringBuffer tagnameData = new StringBuffer();
        reader.beginArray();
        while (reader.hasNext()) {
            reader.beginObject();
            tagnameData.append(readEntry(reader)).append("\n");
            reader.endObject();
        }
        tagnameData.append("\n");
        reader.endArray();
        return tagnameData;
    }
    //abstract method for reading the different entries of a Tag, which is different depending on the class.
    public abstract String readEntry(JsonReader reader) throws IOException;
}
```

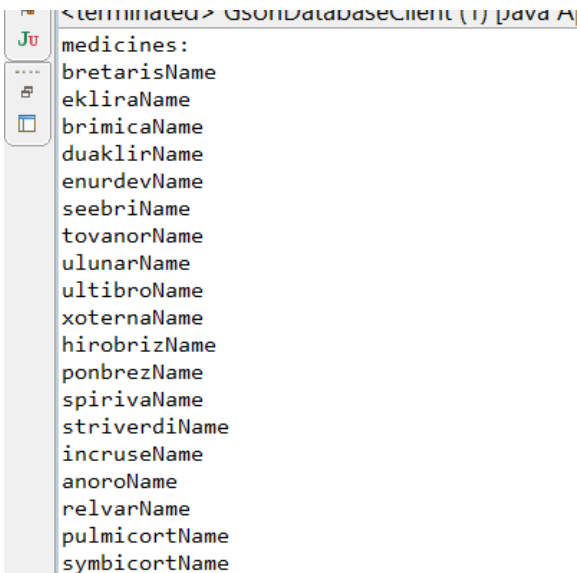
Como se puede apreciar, al método leerTagName se le pasa el 'name' que se ha leído, el JsonReader y el 'tagName' del elemento de la cadena en el que se está intentado leer el archivo. Se comprueba que esa clase puede leer esa etiqueta, en caso afirmativo se leen los valores, y en caso negativo se pasa la llamada a la cadena.

Vamos a ver más detalladamente como se estructura un elemento de la cadena, en este caso ActiveIngredientsTag para poder comparar la estructura sin aplicar el método plantilla.



```
public class ActiveIngredientsTag extends LecturaJson {
    private static final String INGREDIENTS_TAGNAME = "activeIngredients";
    private static final String NAME_FIELD_TAGNAME = "name";
    public ActiveIngredientsTag(ElementoCadenaMando s) {
        super(s);
    }
    public StringBuffer puedeLeer(String name, JsonReader reader) throws IOException {
        //Calls to mother's method leerTagName
        return super.leerTagName(name, reader, INGREDIENTS_TAGNAME);
    }
    //We don't need anymore readValues because it is the same for all element so it is in the super class
    //readEntry remains as before, because it is different for all classes
    public String readEntry(JsonReader reader) throws IOException {
        String actName = null;
        while(reader.hasNext()){
            String name = reader.nextName();
            if (name.equals(NAME_FIELD_TAGNAME))
                actName = reader.nextString();
            else {
                reader.skipValue();
            }
        }
        return actName;
    }
}
```

Vamos a comprobar que el resultado es el mismo al que hemos obtenido antes:



```
Terminated> GsonDatabaseClient (1) java A
medicines:
bretarisName
ekliraName
brimicaName
duaklirName
enurdevName
seebriName
tovanorName
ulunarName
ultibroName
xoternaName
hirobrizName
ponbrezName
spirivaName
striverdiName
incruiseName
anoroName
relvarName
pulmicortName
symbicortName
```

Podemos afirmar que el resultado es el mismo al anterior.

## Conclusión.

Finalmente, quiero hacer una pequeña valoración sobre lo aprendido en el tema de patrones de diseño.

Me ha parecido muy interesante la asignatura en general, pero concretamente este tema me ha gustado mucho pues estos patrones son muy útiles para el trabajo de un programador. No solo proporciona técnicas de resolución de problemas, sino que también permiten mejorar el diseño del programa y hacer que el código sea entendible por alguien que sabe poco sobre el tema, el cual podríamos decir que es uno de los objetivos más buscados, y a su vez difíciles de alcanzar, por un diseñador de software.

En esta sección de la asignatura hemos aprendido múltiples patrones que nos pueden resultar útiles en un futuro, algunos más que otros, pero lo más importante es que también los hemos puesto en práctica.

Al principio, debido a las clases a distancia o a la novedad del temario, me pareció un poco difícil aplicar los patrones, pero poco a poco con la práctica me fue resultando más fácil e intuitivo.