WEB SCRAPING

PRÁCTICA MODELOS DE COMPUTACIÓN

Gonzalo Álvarez Moreno

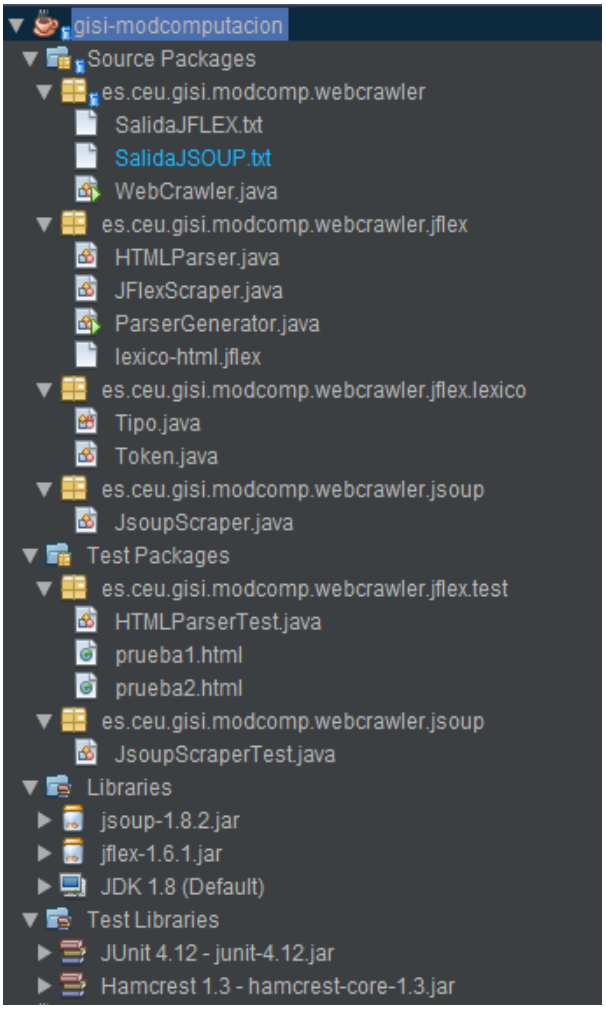
Práctica Web-Scraping

1. ANÁLISIS Y DESCRIPCIÓN DEL PROYECTO

INTRODUCCIÓN A LA PRACTICA

Está práctica consta de diferentes partes, por una parte, la programación de la clase JflexScrapper para poder analizar cualquiera de los ficheros de prueba del paquete test. Por otra parte, la elaboración de la clase JsoupScrapper que recibe una web elegida, en nuestro caso la web del periódico "El País".

Antes de analizar cada una de las clases, comentar rasgos generales del proyecto



Este sería el árbol del proyecto. Cuenta con 2 paquetes, uno llamado Source Packages, dedicado a las clases principales como Jsoup, Jflex o la clase con el método main, WebCrawler.

Además, también encontramos en el paquete jflex.lexico las clases en las que se declaran los tipos y valores de los tokens para la realización del autómata.

Los ficheros de SalidaJFLEX.txt y SalidaJSOUP.txt se crear con la ejecución del programa, llevan la salida a esos ficheros.

Lo mismo sucede con la clase HTMLParser que como ya sabemos se crea con la ejecución de la clase ParserGenerator.

En el paquete TEST se encuentran los test necesarios tanto para el analizador del HTML de JFLEX como la clase correspondiente para los test de necesarios de JSOUP.

Encontramos los ficheros de prueba1 y prueba2 que contienen código HTML apara analizar.

Finalmente, hay que comentar que para la correcta ejecución es necesario el uso de librerías JSOUP, JFLEX y el JDK, además también librerías para test como JUNIT y HAMCREST.

Por lo tanto, es necesario dividir la memoria en analizar por una parte el Análisis de JFLEX y por otra parte el de JSOUP. Para ello veremos el contenido de todas las clases que se ven involucradas en el proceso para analizar los diferentes códigos HTML.

JFLEX

Jflex en este programa es una clase que crea un autómata finito no determinista que utiliza los tipos de tokens para realizar las transiciones. Además, se debe saber si el documento está balanceado, para ello inicializaremos una pila que irá introduciendo valores a medida que se abren etiquetas, y quitándolos a medida que se cierran.

El resultado obtenido debería ser balanceado, es decir, que todas las etiquetas que se han abierto en el fichero HTML han sido cerradas. Si la pila queda con algún elemento significara que no se ha cerrado o tenemos algún problema en la asignación de valores a la pila.

```
ArrayList<String> enlacesA = new ArrayList();
ArrayList<String> enlacesIMG = new ArrayList();
HTMLParser analizador;
Stack<String> etiquetasAbiertas = new Stack();
private boolean estaBalanceado = true;
private int estado = 0;
```

Por lo tanto. Las variables que he definido han sido la pila llamada “etiquetasAbiertas” utilizando la clase se Java Stack.

Además de utilizar el analizador, proporcionado por la clase HTMLParser, creamos los ArrayList que van a contener el valor de los enlaces extraídos por el autómata, una variable booleana que nos indique si e fichero esta balanceado y la declaración de la variable estados.

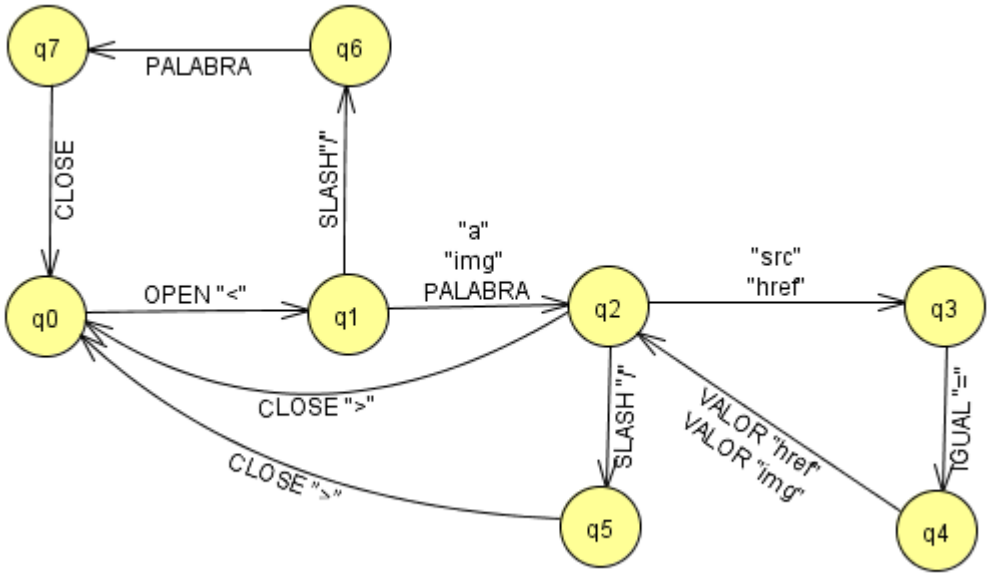
```
public JFlexScraper(File fichero) throws FileNotFoundException, IOException {
    Reader reader = new BufferedReader(new FileReader(fichero));
    analizador = new HTMLParser(reader);

    Token token;
    boolean etiquetaA = false;
    boolean etiquetaIMG = false;
    boolean valorHREF = false;
    boolean valorIMG = false;
    while ((token = analizador.nextToken()) != null) {
        System.out.print(token.getValor() + " ");
        switch (estado) {

            case 0:
                if (token.getTipo().equals(OPEN)) {
                    estado = 1;
                }
```

El método del autómata comienza con la apertura del fichero, por lo que utiliza BufferedReader, con sus correspondientes excepciones.

Declaramos el objeto token de la clase Token y las etiquetas de los hiperenlaces , todas boolean ya que solo queremos saber si su valor es verdadero o falso para almacenar o no sus valores en los arrays anteriormente mencionados.

AUTOMÁTA

A continuación, pasamos a analizar y describir el modelo de autómata usado para la clase **JFLEXScraper.**



- Para mostrar los casos en los que tengamos un **enlace de imagen** o un **hiperenlace**, transitamos a q1 abriendo la etiqueta. De q1 a q2 podemos recibir una palabra, pero debemos hacer la distinción si se trata de “img” o “a” ya que son las que deberemos diferenciar. Para ello utilizamos las variables booleanas definidas anteriormente para saber que tipo de etiqueta se está analizando.
  Si no hacemos esto nos encontraremos con un caso como “< a src…>” o “<img href=…>”
  De manera que cuento sea imagen pondremos la etiquetaA a false y viceversa.

```
case 1:
    if (token.getTipo().equals(PALABRA)) {
        estado = 2;
        etiquetasAbiertas.push(token.getValor().toLowerCase());

        if (token.getValor().toLowerCase().equals("a")) {
            etiquetaA = true;
            etiquetaIMG = false;
        } else if (token.getValor().toLowerCase().equals("img")) {
            etiquetaIMG = true;
            etiquetaA = false;

        } else {
            etiquetaIMG = false;
            etiquetaA = false;
```

  En este estado ya sabemos que estamos abriendo una etiqueta y no cerrándola por lo que debemos introducir en la pila la Palabra con un **push** y no sacarla hasta que se cierre la etiqueta.

Seguimos **transitando a q3** haciendo el mismo recorrido que en anterior caso, si con la etiqueta A nos llega un “href” su valor pasa a true y el de la imagen a false, igual aplicamos para en el que teniendo una “img” nos llega un “src”, ponemos el valor de img a true y el de href a false.
Además, si no nos llega ninguna ponemos las 2 a false ya que en ese caso tendremos un atributo como la siguiente forma:

```
<P style ="Madrid"></P>
```

```
case 2:
    if (token.getTipo().equals(PALABRA)) {
        estado = 3;

        if (etiquetaA) {
            if (token.getValor().equalsIgnoreCase("href")) {

                valorHREF = true;
                valorIMG = false;
            }
        } else if (etiquetaIMG) {
            if (token.getValor().equalsIgnoreCase("src")) {

                valorIMG = true;
                valorHREF = false;
            }
        } else {
            valorHREF = false;
            valorIMG = false;
        }
    }
```

Solo **transitaremos a Q4** ya que es la única forma que prosigue a todas las etiquetas como podemos observar:

```
<IMG src="brushedsteel.jpg"/>
<A href="http://www.bbc.co.uk">link</A>
<P style ="Madrid"></P>
```

Para transitar a Q5 debemos recibir el valor de los enlaces tanto de las imágenes como de los hiperenlaces. Para ello utilizamos el token de tipo VALOR, que espera un enlace. Lo ponemos de tal forma que si el valor de **VALORHREF** es true es decir que ha leído “<a href =” por lo que el siguiente valor lo almacena en el ArrayList de “enlacesA” o hiperenlaces.
De la misma forma si hemos obtenido **que VALORIMG** es true ha leído “<img src=” por lo que almacenará el valor en el ArrayList de imágenes “enlacesIMG”.

```
case 4:

    if (token.getTipo() == Tipo.VALOR) {
        estado = 2;
        System.out.println("href " + valorHREF);
        if (valorHREF) {

            enlacesA.add(token.getValor());
        }
        System.out.println("img " + valorIMG);
        if (valorIMG) {
            enlacesIMG.add(token.getValor());
```

Transitaremos a Q2. Desde aquí quiero recordar que podemos cerrar la etiqueta e 2 formas:
Cerrando directamente después del valor con un SLASH y CLOSE

```
<IMG src="brushedsteel.jpg"/>
```

O cerrando la etiqueta de apertura, introducir algún texto y proceder con la etiqueta de cierre

```
<A href="http://www.bbc.co.uk">link</A>
```

HAY QUE RECORDAR que para cerrar debemos sacar los valore de la pila para que quede balanceado indicando que se ha cerrado con éxito.

Para la **FORMA 1** fijándonos en el autómata transitaríamos **a Q5** ya que recibiríamos un SLASH donde sacaríamos el valor de la pila con un pop de la siguiente manera:

```
if (token.getTipo().equals(SLASH)) {
    estado = 5;
    etiquetasAbiertas.pop(); //sacamos de la pila
}
```

Finalmente transitaríamos a Q0 con un CLOSE para cerrar la etiqueta

Para **la FORMA 2**, situándonos en Q2 transitaríamos directamente a Q0 para cerrar la etiqueta de apertura, luego recorreríamos los estados como se muestra en el autómata **de Q1-Q6-Q7-Q0** es decir los estados necesarios para una etiqueta de CIERRE.
Como tenemos que sacar el valor de la pila, lo haremos cuando recibamos en **Q6** la palabra de cierre, diremos que si saca algo de la pila que no corresponde a ningún valor del token entonces **NO estará balanceado el archivo,** si corresponde con un valor de la pila haremos un pop sacando ese valor de la pila, de todas formas, esté balanceado o no transitaremos a **Q7** donde se cierra la etiqueta con **un CLOSE** y volvemos a **Q0**.

```
case 6:
    if (token.getTipo().equals(PALABRA)) {
        if (!token.getValor().equalsIgnoreCase(etiquetasAbiertas.pop())) {
            estaBalanceado = false;
        }
        estado = 7;
    }
```

- Para mostrar sentencias simples como **“ ”** simplemente recorrerá los estados de manera similar. En este caso no recorrerá los estados **Q3-Q4** ya que estos son exclusivamente para enlaces y atributos. Por lo que simplemente recorrerá observando el autómata para la etiqueta de **APERTURA Q0-Q1-Q2-Q0**, introduciendo el valor en la **pila en Q1** al recibir la palabra.
  Para la etiqueta de **CIERRE recorrerá: Q0-Q1-Q6-Q7-Q0**. De igual manera que los anteriores casos retiraremos el valor de la pila al introducir después del SLASH la palabra.

Como observamos es un mecanismo muy simple comparado con las transiciones de los enlaces.

- Finalmente podemos encontrar el caso de etiquetas como **
.** Son etiquetas que se abren y cierran a la vez, este es el caso del salto de línea y lo podríamos encontrar en situaciones como:

```
<h1>Titulo de la <BR/> pagina web</h1>
```

Los situamos en la parte del texto en la que queremos que haya un salto de línea.
El **recorrido de los estados** sería el siguiente:
Q0: Transita a Q1 con OPEN "<"
Q1: Transita a Q2 con PALABRA, introducimos en la pila
Q2: Transita a Q5 con SLASH, sacamos de pila
Q5: Transita a Q0 con CLOSE >.

Una vez definido el autómata definimos los **métodos** que nos devuelven las variables que necesitamos.

```
public List<String> getImagenes() {
    return this.enlacesIMG;
}
```

```
public boolean getBalance() {
    return this.estaBalanceado;
}
```

```
public List<String> getLinks() {
    return this.enlacesA;
}
```

```
public Stack getStack() {
    return this.etiquetasAbiertas;
}
```

Además de estos métodos debemos implementar **OTRO MÉTODO** para volcar la salida de los enlaces a un fichero, este método simplemente crea un fichero en la ruta que especifiquemos mediante la clase **FileWriter** y le pedimos que la salida que nosotros le indiquemos, en vez de mostrarla por pantalla la vuelque a ese fichero

tendría la siguiente forma:

```
public void VolcarFicheroJFLEX() {
    FileWriter ficherol = null;
    PrintWriter pr = null;
    try {
        ficherol = new FileWriter("./src/es/ceu/gisi/modcomp/webcrawler/SalidaJFLEX.txt");
        pr = new PrintWriter(ficherol);
        pr.println("ENLACES: " + "\n\t\t" + getImagenes() + "\n" + "ENLACES IMAGENES: \n\t\t" + getLinks() + "\n");
    } catch (IOException e) {
    } finally {
        try {
            if (null != ficherol);
            ficherol.close();
        } catch (IOException e2) {
        }
```

Finalmente, en **la Clase WebCrawler** la cual contiene el método **MAIN** del programa crearemos un objeto de la clase **Jflex**, en este caso le asignaremos el análisis del ficheroPrueba2.

La salida no debe mostrar los enlaces **de imágenes, los hiperenlaces, el valor de la pila** que si todo sale bien debe valer 0 ya que como ya comenté anteriormente significaría que las etiquetas se están abriendo y cerrando correctamente además del correcto funcionamiento del autómata y por tanto estaría **balanceado**

El código quedaría de la siguiente forma:

```
public static void main(String[] args) throws IOException {
    // Deberá inicializar JFlexScraper con el fichero HTML a analizar
    // Y creará un fichero con todos los hiperenlaces que encuentre.
    // También deberá indicar, mediante un mensaje en pantalla que
    // el fichero HTML que se ha pasado está bien balanceado.
    JFlexScraper a = new JFlexScraper(ficheroPrueba2);
    System.out.println("----------------PARTE DE JFLEX-----------------");
    System.out.println(
            "\nimg: : " + a.getImagenes());
    System.out.println(
            "links: " + a.getLinks());
    System.out.println(
            "la pila queda: " + a.getStack());
    System.out.println(
            "isBalanced?: " + a.getBalance());
    a.VolcarFicheroJFLEX();
```

ANÁLISIS DE TEST

Los test que se han implementado han sido:

- CompruebaEtiquetaInicioHTML
- CompruebaIncionYFinEtiquetaHTML
- CompruebaInicioYFinEtiquetaBR
- CompruebaFichero1
- CompruebaFichero2

Antes de comenzar con los test en el constructor abrimos los 2 archivos de prueba con BufferedReader e imprimimos el posible error.

- **CompruebaEtiquetaInicioHTML ():**

Este test se encarga de asegurarse de que hay una etiqueta de apretura de HTML al comienzo del archivo de prueba1. Para ello utiliza los diferentes tokens, en concreto un OPEN-HTML-CLOSE, para filtrar la etiqueta de apertura de HTML.

Este test a pesar de que el siguiente hace también esta comprobación es necesario para establecer que el archivo tenga por obligación la etiqueta de apertura al menos ya que es una etiqueta obligatoria para cualquier archivo de HTML

```java
@Test
public void compruebaEtiquetaInicioHTML() {
    try {
        Token token1 = analizador.nextToken();
        Token token2 = analizador.nextToken();
        Token token3 = analizador.nextToken();
        assertEquals(token1.getTipo(), Tipo.OPEN);
        assertEquals(token2.getValor().toLowerCase(), "html");
        assertEquals(token3.getTipo(), Tipo.CLOSE);
    } catch (IOException ex) {
        Logger.getLogger(HTMLParserTest.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Como vemos hacemos un try-catch simple en el que definimos los 3 tokens, 2 de getTipo y un de getValor para filtrar la palabra HTML.

Finalmente encapsulamos el error con catch.

- **CompruebaIncionYFinEtiquetaHTML ():**
  Este test se encarga de comprobar que existe una etiqueta de Apertura de HTML y además otra de Cierre.
  Para ello se hace lo mismo que en el test anterior, pero se le añaden mas tokens para filtrar exactamente la palabra “</HTML>” por lo tanto la estructura es casi idéntica al anterior.

```java
@Test
public void compruebaInicioYFinEtiquetaHTML() {
    try {
        analizador.yyreset(reader1);
        //El inicio de una etiqueta HTML es: <NOMBREETIQUETA>
        Token token1 = analizador.nextToken();
        Token token2 = analizador.nextToken();
        Token token3 = analizador.nextToken();

        assertEquals(token1.getTipo(), Tipo.OPEN);
        assertEquals(token2.getValor().toLowerCase(), "html");
        assertEquals(token3.getTipo(), Tipo.CLOSE);

        //El final de una etiqueta HTML es: </NOMBREETIQUETA>
        Token token4 = analizador.nextToken();
        Token token5 = analizador.nextToken();
        Token token6 = analizador.nextToken();
        Token token7 = analizador.nextToken();
        assertEquals(token4.getTipo(), Tipo.OPEN);
        assertEquals(token5.getTipo(), Tipo.SLASH);
        assertEquals(token6.getValor().toLowerCase(), "html");
        assertEquals(token7.getTipo(), Tipo.CLOSE);
    } catch (IOException ex) {
        Logger.getLogger(HTMLParserTest.class.getName()).log(Level.SEVERE, null, ex);
        assertTrue(false);
    }
}
```

Utilizamos 4 tokens más estableciendo EL TOKEN 4,5,7 getTipo y el 6 getValor, es decir el mismo caso que el anterior, pero añadiendo un SLASH después del OPEN
Volvemos a encapsular el error.

- **CompruebaInicioYFinEtiquetaBR ():**

  Este test comprueba si el fichero analizado contiene una etiqueta de tipo
  Hay que saber que esta etiqueta se abre si se cierra por lo que no hay una etiqueta para la apertura y otra para el cierre. Por lo tanto, esta etiqueta hace ambas funciones.
  Para la programación del test se usa un **boolean "encontradoBR"** que será true si encuentra justo esa etiqueta.

  Por lo tanto, se utiliza un **while** que si después del open no le llega un **BR** se queda en false el valor de la variable.

  Si el token recibe el valor de **"br"** se establecen 2 tokens más para finalizar la etiqueta con el SLASH y el CLOSE.

  Finalmente encapsulamos el error.

código:

```
@Test
public void compruebaInicioYFinEtiquetaBR() {
    try {
        analizador.yyreset(reader2);
        //Una etiqueta BR tiene la forma: <BR /> (incluye inicio y fin de etiqueta)
        boolean encontradoBR = false;
        while (!encontradoBR) {
            Token token1 = analizador.nextToken();
            while (token1.getTipo() == Tipo.OPEN) {
                //Si encuentro un token OPEN puede ser el inicio de una etiqueta BR...
                Token token2 = analizador.nextToken();
                if (token2.getValor().toLowerCase().equals("br")) {
                    //Es una etiqueta BR:
                    encontradoBR = true;
                    Token token4 = analizador.nextToken();
                    Token token5 = analizador.nextToken();
                    assertEquals(token4.getTipo(), Tipo.SLASH);
                    assertEquals(token5.getTipo(), Tipo.CLOSE);
                    break;}}}

    } catch (IOException ex) {
        Logger.getLogger(HTMLParserTest.class.getName()).log(Level.SEVERE, null, ex);
        assertTrue(false);
    }
}
```

- **CompruebaFichero1 ():**
  Este test se encargar de revisar si las imágenes y los enlaces almacenados por el autómata definido en la clase JFLEX se corresponden con los que debería obtener del **fichero 1.**
  Para ello utilizamos los métodos definidos en la clase JFLEX de getImagenes(), getLinks(), getBalance() para establecer con AssertEquals el valor que debemos obtener.
  Si no nos salen esos valores no pasaremos el test, entonces no estaremos analizando bien los enlaces de imágenes e hiperenlaces.

Si nosotros en el fichero de prueba tenemos 2 imágenes, ponemos en el test que el getImagenes(), size debe ser igual a 2 y si no nos pasa el test entonces el autómata no está funcionando bien.
El test tiene la siguiente forma:

```
@Test
public void compruebaFichero1() throws IOException, FileNotFoundException {

    JFlexScraper a;
    a = new JFlexScraper(ficheroPrueba1);
    assertEquals(a.getImagenes().size(), 0);
    assertEquals(a.getLinks().size(), 0);
    assertEquals(a.getBalance(), true);
    System.out.println("img: : " + a.getImagenes());
    System.out.println("links: " + a.getLinks());
    System.out.println("la pila queda: " + a.getStack());
    System.out.println("isBalanced?: " + a.getBalance());

}
```

Asignamos el autómata de la clase JFLEX al fichero de prueba1.html, establecemos los valores que nos tienen que devolver para pasar el test de las imágenes, hiperenlaces y que este Balaceado.

También mostramos la salida que obtenemos del autómata para comprobar los valores que nos deberían salir, pero esta parte no es necesaria.

- **CompruebaFichero2 ():**
  Este test hace exactamente lo mismo que test anterior, pero con el fichero de **prueba2.html.**
  Por lo tanto, lo único que cambiamos es el fichero sobre el que el autómata de la clase JFLEX va a analizar y luego los valores que debemos obtener del autómata para pasar el test de manera que nos quedaría algo como:

```
@Test
public void compruebaFichero2() throws IOException, FileNotFoundException {

    JFlexScraper a;
    a = new JFlexScraper(ficheroPrueba2);
    assertEquals(a.getImagenes().size(), 1);
    assertEquals(a.getLinks().size(), 2);
    assertEquals(a.getBalance(), true);
    System.out.println("img: : " + a.getImagenes());
    System.out.println("links: " + a.getLinks());
    System.out.println("la pila queda: " + a.getStack());
    System.out.println("isBalanced?: " + a.getBalance());

}
```

CONCLUSIÓN: En conclusión, estos test aseguran que los ficheros que vamos a analizar por una parte aseguran la etiqueta obligatoria de HTML tanto de apertura como de cierre, ambas necesarias para poder analizar un fichero HTML.

Por otra parte, nos aseguramos de que en el caso en el que encontremos etiquetas especiales como
 seamos capaz de analizarlo perfectamente, como el resto de las etiquetas usadas en HTML son de apertura y cierre no hacemos más test como este.

Finalmente es necesario que los valores de los enlaces y si están balanceados los archivo que analizamos con el autómata creado se corresponden con los que esperamos. De esta forma podemos comprobar que nuestro autómata esta filtrando bien las etiquetas que recibe y esta elaborado correctamente.

JSOUP

Jsoup se trata de un crawler que nos permite obtener información de un sitio web de manera ordenada.

En nuestro caso lo usaremos para obtener los enlaces de las imágenes, hiperenlaces y las estadísticas de uso varias de varias etiquetas HTML.

Definimos los métodos en la clase JsoupScraper.java para en la clase WebCrawler.

Antes de seguir debemos conocer para que sirven estas clases para entender el correcto funcionamiento de JSOUP

```
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;
```

- **Jsoup.Nodes.Document :** Esta clase representa la carga de documentos HTML a través de la biblioteca Jsoup. Puede usar esta clase para realizar operaciones que se aplican a todo el documento HTML.
- **Jsoup.Nodes.Element:** Los elementos HTML están compuestos por nombres de etiquetas, atributos y nodos secundarios. Usando la clase Element, puede extraer datos, atravesar nodos y manipular HTML.
- **Jsoup.Select:** Se trata de paquetes para admitir el selector de elementos en estilo CSS.

Inicializamos la variable **doc** de la clase **Documents** de la siguiente forma:

```
doc = Jsoup.connect(url.toString()).get();
```

De esta manera en la clase Webcrawler ya le podremos asignar a la clase una URL de un sitio web para analizar.

Definimos las variables que vamos a utilizar en los métodos. De forma que utilizamos un **String** llamado imagen para obtener el contenido de una única imagen ya que no necesitamos una lista para un solo elemento y dos **Listas, "enlaces"** para almacenar el valor de los enlaces de las imágenes y **"enlaces2**" de los hiperenlaces.

```
String imagen;
List<String> enlaces = new ArrayList<String>();
List<String> enlaces2 = new ArrayList<String>();
```

ANÁLISIS DE MÉTODOS

Los métodos que contiene esta clase son os siguientes:

- estadisticasEtiqueta ().
- ObtenerHiperEnlaces ().
- ObtenerHiperEnlacesImagenes().
- ObtenerContenidoImg()
- VolcarFichero().

- **estadisticasEtiqueta():**

Este método se encarga de devolverte la cantidad de etiquetas (la que seleccionemos) que se encuentran en un fichero analizado.

Para ello utilizamos la clase **Elements,** y creamos el objeto **"estadísticas"** que mediante la variable definida anteriormente "doc" podemos seleccionar la etiqueta.

El código nos quedaría de la siguiente manera.

```
public int estadisticasEtiqueta(String etiqueta) {

    Elements estadisticas = doc.select(etiqueta);
    return estadisticas.size();
}
```

- **ObtenerHiperEnlaces():**

Este método nos devuelve los hiperenlaces del fichero, para ello utilizamos otra vez la clase Elements que nos facilita junto con **"doc"** la búsqueda de elementos, en nuestro caso lo haremos con **getElementByTag**, filtrando por aquellas que empiecen por "a".

Si lo encuentra que le añada al array de enlaces el valor que representa la información relativa al atributo mediante **"attr",** con el valor de "**href".**

Finalmente, que nos devuelva el valor de la **lista.**

```
public List<String> obtenerHiperenlaces() {

    Elements links = doc.getElementsByTag("a");
    for (Element e : links) {
        enlaces.add(e.attr("href"));
    }
    return enlaces;
}
```

- **ObtenerHiperEnlacesImagenes()**

Para este método, como lo que queremos es que nos devuelva una **lista,** pero con los enlaces de imágenes entonces hacemos los mismo que en el método anterior, pero especificando que getElementByTag sea **"img"** y attr a **"src".**

Finalmente, que nos vuelque el contenido a la lista enlaces2.

```
public List<String> obtenerHiperenlacesImagenes() {
    Elements elementos = doc.getElementsByTag("IMG");
    for (Element e : elementos) {
        enlaces2.add(e.attr("src"));
    }
    return enlaces2;
}
```

- **ObtenerContenidoImg():**
  Este método Obtiene la imagen a la que hace referencia LA PRIMERA etiqueta IMG que encontramos.
  Para ello especificamos que al final de filtra por "IMG" en el **getElementByTag**, nos muestre solo la primera usando **.first(),** establecemos el **attr** como **"src"** y como solo nos va a mostrar un string usamos la variable imagen anteriormente declarada.
  Finalmente hacemos **return** de la variable **imagen.**

```
public String obtenerContenidoImg() {
    Element elemento = doc.getElementsByTag("IMG").first();
    imagen = elemento.attr("src");
    return imagen;
}
```

- **VolcarFichero():**

  Finalmente, le método de volcar fichero hace que se cree un archivo en el paquete **Webcrawler**, junto con la clase **WebCrawler** y la salida de **JFLEX**.
  Con la clase **PrintWriter** escribimos en el fichero la salida que deseemos. En este caso mostramos, los hiperenlaces, los enlaces de imágenes y el contenido de la imagen.
  Debemos observar en la salida un mayor numero de links en el apartado "Enlaces Imágenes" que en "Contenido Imagen" ya que el Contenido de imagen solo muestra la **primera etiqueta** con la que se encuentra de ahí que solo haya una y en los enlaces más.
  Encapsulamos los errores y el código nos quedaría algo así (muy similar al JFLEX):

```
public void VolcarFichero() {
    FileWriter fichero;
    fichero = null;
    PrintWriter pw = null;
    try {
        fichero = new FileWriter("./src/es/ceu/gisi/modcomp/webcrawler/SalidaJSOUP.txt");
        pw = new PrintWriter(fichero);
        pw.println("ENLACES: " + "\n\t\t" + enlaces + "\n" + "ENLACES IMAGENES: \n\t\t" + enlaces2 +
                "\n" + "CONTENIDO IMAGEN: \n\t\t" + imagen + "\n");
    } catch (IOException e) {
    } finally {
        try {
            if (null != fichero);
            fichero.close();
        } catch (IOException e2) {
        }
    }
}
```

ANÁLISIS DE TEST

Para la elaboración de los diferentes test nos vamos a la clase JsoupScrapperTest.java ubicada en el paquete Test. En esta clase nos proporcionan un pequeño código HTML al que le haremos una seria de test para ver si podemos obtener la información que se no pide.

En estos test comprobaremos si mediante los métodos creados en la clase Jsoup podremos obtener los links que queremos

Los test que he elaborado son los siguientes:

```
@Test
public void recuperaNombrePrimeraImagen() {
    assertEquals(scraper.obtenerContenidoImg(), "brushedsteel.jpg");
}
List<String> hiperenlaces = new ArrayList<String>();


@Test
public void obtenerHiperenlaces() {
    this.hiperenlaces.add("http://www.bbc.co.uk");
    assertEquals(scraper.obtenerHiperenlaces(), hiperenlaces);
}
@Test
public void estadisticasEtiqueta(){
    assertEquals(scraper.estadisticasEtiqueta("a"),1);
}
```

- **Test recuperarNombrePrimeraImagen():**
  Para pasar el test es tan simple como , si con el método obtenerContenidoImg() de Jsoup obtienes “brushedsteel.jpg” entonces significa que esta funcionando el método ya que eso sería el valor que debería devolver el método.

- **Test obtenerHiperEnlaces():**

  Con este test lo que se pretende es saber si el método de obtención de Hiperenlaces de la clase Jsoup.java funciona correctamente, para ello añadimos el valor que debería obtener a la Lista “hiperenlaces”.

  Posteriormente comparamos si el resultado obtenido con el método es igual al valor almacenado en la lista.

  Si pasamos el test, significará que el método está funcionando correctamente.

- **Test estadísticasEtiquetas():**

  Para este test simplemente utilizamos un ejemplo, y si lo pasa entonces significa que las estadísticas de las etiquetas contabilizan adecuadamente.

  En este caso establecemos que si el numero de etiquetas que el método de la clase Jsoup.java es igual al que le introducimos nosotros, sabiendo que en el caso de la “a” solo hay una en todo el código, entonces pasará el test. Y esto lo podremos aplicar a cualquier etiqueta.

Conclusión TEST:

He decidido implementar estos test ya que los he considerado oportunos para el resultado que se exigen en la práctica. Por ello simplemente he visto conveniente realizar un test por cada método.

Sin embargo, no se incluye un test para el método de **ObtenerEnlacesImagenes()** ya que se entiende que, si funciona el obtener el de una, funcionará el de obtener todos.

Es necesario hacer varios test con el de las estadísticas para poder generalizarlo a cualquier etiqueta, así como el de la obtención de hiperenlaces.

CONCLUSIÓN GENERAL DEL PROYECTO

Me gustaría que si se desea crear al momento hay que hacer click derecho sobre el proyecto y darle a “generar javadoc”.

Sobre la practica me gustaría comentar que se me hizo bastante liosa al principio ya que intenté empezar muy prematuramente sin entender bien todos los conceptos.

Una vez veía que me costaba mucho avanzar me reviste todo el material y todos los conceptos necesarios y me empecé a organizar.

La parte que más fácil me ha resultado ha sido la del Jsoup, ,al no tener tanto lio con la programación del autómata. Sin embargo, a pesar de haber sido la mas costosa, la parte del JFLEX me ha resultado la más interesante, ya que he visto de forma directa como a medida que cambiaba el autómata iba obteniendo resultados.

El tiempo de la practica ha sido mas extenso del que me esperaba ya que hasta que no vi en la tutoría de la practica un modelo de autómata me costó empezar a completar las clases.

He seguido una buena línea de commits, sin embargo, como te comenté en un correo Sergio, resulta que estaba haciendo commits solo de una clase. Pero finalmente implementé todo el proyecto.

En conclusión, me ha gustado mucho esta práctica las 2 partes y el esfuerzo que he empleado en ella.