UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Ingeniería Informática

Trabajo Fin de Grado
Automatización del análisis de accesibilidad de páginas y documentos en sitios web

Autor: Marcel Andrei Voicu

Tutor/es: Luis Fernández Sanz

TRIBUNAL:

Presidente: Javier Albert Seguí

Vocal 1º: Vera Pospelova

Vocal 2º: Luis Fernández Sanz

FECHA: 22 de septiembre de 2022

Índice

Índice de figuras

Resumen

La accesibilidad del contenido web se está volviendo una obligación, a través de nuevas leyes y directivas instauradas por nuestro Estado y la Unión Europea. Así, nace la necesidad de comprobar el grado de accesibilidad que poseen los sitios web actualmente, incluidos los ficheros que alojan. El objetivo de nuestro proyecto es agilizar este proceso, centrándonos exclusivamente en el análisis de accesibilidad de los documentos en formato PDF alojados en sitios web, automatizando su búsqueda y el almacenamiento de los resultados de análisis.

Abstract

The accessibility of web content is becoming an obligation, through new laws and directives introduced by our State and the European Union. Thus, the need to check the degree of accessibility of current websites, including the files they host, has arisen. The aim of our project is to speed up this process, focusing exclusively on the accessibility analysis of PDF documents hosted on websites, automating their search and the storage of the analysis results.

Palabras clave

Accesibilidad, web, PDF, análisis, software.

Agradecimientos y dedicatoria

Agradezco a Vera Pospelova y a Ana Castillo por su ayuda ante algunos problemas enfrentados durante la realización de este proyecto, así como sus consejos. Y, por supuesto, agradezco a mi tutor, Luis Fernández, por todo su apoyo.

1 Introducción

1.1 Presentación del proyecto

Según el Real Decreto 1112/2018 [1], toda la información de los sitios web pertenecientes al sector público debe ser accesible, es decir, debe ser utilizable por el máximo número de personas sin que sus capacidades personales, conocimientos o características técnicas de su equipo resulten un impedimento. Además, la Directiva 2019/882 de la Unión Europea [2] pretende aplicar una medida similar también al sector privado. Esto implica que la accesibilidad web se está convirtiendo en un requerimiento que se debe cumplir.

Adaptar un sitio web para que sea lo más accesible posible puede llevar mucho tiempo, dependiendo de sus dimensiones. Además, sus archivos adjuntos, como por ejemplo aquellos en formato PDF, también deben ser accesibles. Esto requiere que conozcamos las características que hacen, a diferentes formatos de información, accesibles. Por suerte, ya existen estándares de accesibilidad como el de la World Wide Web Consortium (W3C)[1], que estipula pautas para que los sitios web sean accesibles.

Para ayudar en la tarea de mejora de la accesibilidad de los sitios web, hemos decidido centrarnos en el análisis de los documentos en formato PDF. Inicialmente, también pretendíamos analizar y guardar los resultados de los sitios web, como se puede ver reflejado en el anteproyecto, pero, debido al margen de tiempo disponible y las dificultades que fuimos enfrentando, optamos por limitarnos a los documentos PDF, si bien el proceso general de extracción del árbol de estructura de un sitio web sirve de base para una futura extensión de sitio web.

Para los archivos en formato PDF, dos de los estándares más extendidos son ISO 14289-1:2014 [3] y WCAG 2.0 [4], este último desarrollado por W3C, existe un servicio web gratuito que analiza algunas de las pautas de este estándar, llamado Tingtun Checker[2]. El analizador de referencia que podemos considerar es el que está integrado en Adobe Acrobat Reader, de la empresa Adobe, ya que el estándar que especifica el formato PDF (ISO 32000 [5]) fue desarrollado por ellos. El único inconveniente del analizador de Adobe es que es de pago.

Por otra parte, además de los softwares de análisis mencionados anteriormente, conocemos un tercero llamado PAVE[3] que realiza un análisis más centrado en los metadatos de los documentos y en la estructura de sus etiquetas, sin seguir un estándar en concreto.

Dicho esto, conocemos los estándares de accesibilidad, incluso tenemos acceso a un par de herramientas gratuitas que nos ayudan a comprobar automáticamente algunas propiedades, pero el proceso de analizar todos los documentos de un sitio web puede seguir siendo largo y tedioso. Un sitio web puede contener una gran cantidad de documentos, por lo que la tarea de analizarlos todos requeriría de, primero, obtener de algún modo todos y cada uno de los documentos alojados en el sitio web y, segundo, ir insertándolos uno a uno en los softwares de análisis mencionados, guardando sus resultados.

[1] Sitio web oficial de W3C: https://www.w3.org/
[2] Sitio web del analizador de PDF de Tingtun: https://checkers.eiii.eu/en/pdfcheck/
[3] Sitio web oficial de PAVE: https://pave-pdf.org/index.html

Por todo esto, nuestro objetivo es automatizar este proceso. Pretendemos desarrollar un software capaz de encontrar los documentos alojados en un sitio web dado, analizarlos y guardar sus resultados en una base de datos.

1.2 Objetivos

Los objetivos del proyecto, por lo tanto, serán los siguientes:

1. Automatizar la búsqueda de documentos PDF alojados en un sitio web, a través de la generación de su árbol de estructura de navegación.
2. Automatizar la obtención de los resultados de los análisis de accesibilidad de esos documentos.
3. Automatizar el guardado de los resultados en una base de datos de PostgreSQL.
4. Integrar los objetivos anteriores en un solo programa, utilizando lenguaje Java y librerías de terceros.
5. Crear una base de datos en PostgreSQL, donde guardar los resultados.

Bajo ningún concepto pretendemos realizar lo siguiente:

1. Realizar análisis de accesibilidad sobre los sitios web. Como ya explicamos, acabamos centrándonos únicamente en los documentos PDF.
2. Programar un software de búsqueda de documentos PDF alojados en un sitio web. Para ello usaremos una herramienta ya desarrollada.
3. Programar un software de análisis de accesibilidad de documentos PDF. Para ello usaremos las herramientas Tingtun y PAVE.
4. Obtener los resultados de todas las propiedades especificadas por el estándar WCAG 2.0, por ISO 32000 o por cualquier otro.

Algunas condiciones que tendremos en cuenta para la realización del proyecto son:

1. El programa carecerá de interfaz gráfica de usuario, pasando como argumentos de entrada aquellos parámetros que sean necesarios, de modo que su uso sea rápido y sencillo.
2. Las herramientas de terceros que se utilicen para el desarrollo del sistema serán gratuitas. El motivo es eliminar el impedimento económico como uno de los factores por los cuales no se utilice este programa.
3. Adaptar el código para facilitar, en el futuro, la adición de nuevos analizadores de accesibilidad (ya sea para utilizar otras herramientas web diferentes a Tingtun y PAVE o directamente para implementar código que se encargue de ello)

1.3 Estructura de los capítulos

Capítulo 2: se detalla la solución adoptada para obtener los documentos PDF.

Capítulo 3: breve explicación del formato PDF y de las herramientas de análisis Tingtun y PAVE.

Capítulo 4: diseño de la arquitectura del programa, desde su primer planteamiento hasta su estado final. Se explican las decisiones de diseño más importantes y algunos problemas tratados.

Capítulo 5: explicación concreta del código implementado para ciertos aspectos relevantes del programa.

Capítulo 6: muestra de las pruebas realizadas con el programa.

Capítulo 7: conclusiones obtenidas por la realización de este proyecto, y propuesta de trabajos que se pueden realizar partiendo de este.

1.4 Glosario de acrónimos y abreviaturas

- **Scraper:** expresión inglesa, referida, en un contexto informático, a la acción de extraer datos contenidos en páginas web accesibles a través de Internet.
- **HashMap:** estructura de datos proporcionada por el lenguaje de programación Java. Consiste en una lista de parejas de datos, donde un elemento actúa como clave para poder acceder al otro.
- **Commit:** expresión inglesa, referida, en el contexto de las bases de datos, a la acción de finalizar una transacción de datos y almacenarlos permanentemente.

2 Obtención de los documentos PDF de un sitio web

Comenzamos por recolectar todos los documentos PDF que se encuentren referenciados dentro de un sitio web. La primera solución que se nos puede ocurrir para este problema es utilizar una librería, o algún otro tipo de herramienta, que nos facilite leer el contenido de las páginas web, para así encontrar las URL que nos redirijan a otras páginas pertenecientes al mismo dominio e ir almacenando aquellas que referencien documentos PDF. Las librerías que nos ayudan a leer y analizar páginas web se denominan *scrapers*. Aunque esta solución sea totalmente válida, programar un software suficientemente robusto como para asegurarnos de que encontrará todos los documentos PDF de un sitio web, sin dejarse ninguno, no es tarea fácil.

La alternativa a esto es aprovecharnos de un fichero que, hoy en día, prácticamente todos los sitios web poseen, denominado *sitemap*.

2.1 Los ficheros *sitemap*

Un *sitemap*[4] es un archivo que recoge información de un sitio web y las páginas pertenecientes a su mismo dominio, aportando información como, por ejemplo, cuándo fue la última vez que se actualizó cada URL, o cómo de importante es cada una. Estos archivos se crearon para que los buscadores de Internet pudieran rastrear, con facilidad, páginas pertenecientes a un mismo dominio web, así como para descubrir otros nuevos sitios web. Los ficheros *sitemap* siempre se encuentran en formato XML. En la Figura 1 podemos ver un ejemplo de este archivo, perteneciente al sitio web de Gmail, el servicio de correo electrónico de Google.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>https://www.google.com/intl/am/gmail/about/</loc>
    <xhtml:link href="https://www.google.com/gmail/about/" hreflang="x-default" rel="alternate" />
    <!-- ... hay más URL por aquí, una por cada idioma al que se ha traducido la msima página  -->
    <xhtml:link href="https://www.google.com/intl/th/gmail/about/" hreflang="th" rel="alternate" />
  </url>
  <url>
    <loc>https://www.google.com/intl/am/gmail/about/for-work/</loc>
    <xhtml:link href="https://www.google.com/gmail/about/for-work/" hreflang="x-default" rel="alternate" />
    <!-- ... hay más URL por aquí, una por cada idioma al que se ha traducido la msima página  -->
    <xhtml:link href="https://www.google.com/intl/th/gmail/about/for-work/" hreflang="th" rel="alternate" />
  </url>
  <url>
    <loc>https://www.google.com/intl/am/gmail/about/policy/</loc>
    <xhtml:link href="https://www.google.com/gmail/about/policy/" hreflang="x-default" rel="alternate" />
    <!-- ... hay más URL por aquí, una por cada idioma al que se ha traducido la msima página  -->
    <xhtml:link href="https://www.google.com/intl/th/gmail/about/policy/" hreflang="th" rel="alternate" />
  </url>
  <url>
    <loc>https://www.google.com/intl/ar/gmail/about/</loc>
```

Figura 1. Fichero sitemap de la página web de Gmail

Dado que el *sitemap* ya nos proporciona la lista con todas las subpáginas de un sitio web, así como los documentos alojados, tan solo tendremos que leerlo y recoger las URL que nos interesen. Esto es sencillo de automatizar, dado que las URL se rodean por las etiquetas "`<loc>`" y "`</loc>`", por lo que tan solo tendríamos que detectarlas y obtener las subcadenas correspondientes a las URL, sin necesidad siquiera de usar alguna librería que nos analice los ficheros XML.

[4] Sitio web oficial donde se ofrece información sobre los *sitemap*: https://www.sitemaps.org/

2.2 Obtención de los ficheros *sitemap*

Para poder hacer uso de los ficheros *sitemap*, antes debemos obtenerlos de algún modo. La solución que terminamos adoptando es hacer uso de un programa desarrollado por un tercero, como detallaremos en la sección 2.2.4. A continuación, analizaremos las posibles soluciones:

2.2.1 Obtener el *sitemap* a partir del propio sitio web

Existe la posibilidad de obtenerlos directamente desde el sitio web al que pertenece, pues estos archivos deben estar alojados en el sitio web de origen para que puedan ser encontrados por los rastreadores. La forma más sencilla de conseguirlo es añadiendo, al final de la URL de la página principal, el texto "sitemap.xml" o "sitemap_index.xml". Pero existen dos problemas, el primero es que no siempre se encuentran en la ruta principal (ya que el *sitemap* puede alojarse en cualquiera de las rutas del servidor web), por lo que la segunda opción es añadir, al final de la URL, el texto "robots.txt", pues en ese fichero suele indicarse la ruta exacta en la que se aloja el *sitemap*. El segundo problema es que, aunque se consiga acceder al *sitemap*, a veces su contenido no posee directamente las URL a las diferentes páginas y documentos, sino que reúne las URL a otros *sitemap* y estos, a su vez, también pueden redirigir a otros *sitemap* diferentes. Es decir, que tendríamos que utilizar un *scraper* para ir profundizando en esta especie de árbol de *sitemaps* y, así, poder obtener todas las URL que nos interesan. Es una opción plausible, pero veremos que existe una alternativa más fácil de implementar.

2.2.2 Generar el *sitemap* mediante herramientas web

Otra opción es utilizar uno de los tantos sitios web que te generan un nuevo *sitemap* con tan solo introducirle una URL, como por ejemplo xml-sitemaps.com, el cual era la solución que consideramos en el anteproyecto. Automatizar este proceso sería relativamente sencillo mediante las herramientas y librerías de Selenium, aunque presenta el inconveniente de tener que rehacer el código cada vez que la interfaz de la herramienta web cambie o se quiera utilizar otra diferente. Y más importante aún, dependeríamos de la disponibilidad de la herramienta web que utilizáramos, por lo que, si en algún momento dejara de funcionar por el motivo que fuera, nuestra herramienta se quedaría también inutilizable.

2.2.3 Generar el *sitemap* mediante una librería

La opción más razonable, por su inmediatez y sencillez, sería utilizar alguna librería que sirviera para generar automáticamente el *sitemap*. Por desgracia, no conseguimos encontrar una librería semejante. Las únicas librerías relacionadas que hemos encontrado servían para generar un *sitemap* programando, explícitamente, los elementos que quisiéramos que contuviera. Es decir, que no los generaba de forma automática.

2.2.4 Generar el *sitemap* mediante un programa ejecutable

Por suerte, encontramos un software, Sitemap Generator[5], capaz de realizar lo que queremos, en lenguaje JavaScript. Se debe ejecutar mediante Node JS[6], un intérprete de JavaScript.

Para poder construir nuestro ejecutable en formato JAR, añadiendo código de JavaScript, existe la posibilidad de utilizar un JDK especial, llamado GraalVM[7], que viene preparado para esta función. Pero al probar esta solución descubrimos que, como Sitemap Generator estaba

[5] Repositorio del programa: https://github.com/lgraubner/sitemap-generator
[6] Información oficial acerca de Node JS: https://nodejs.org/es/about/
[7] Sitio web oficial de GraalVM: https://www.graalvm.org/

programado para ser ejecutado con Node JS, GraalVM no era capaz de compilarlo, pues solamente puede compilar código original de JavaScript.

La única alternativa que nos queda es, desde nuestro código Java, hacer una llamada a la terminal de comandos del sistema operativo y ejecutar el generador del *sitemap* desde ahí. De esta forma, basta con tener el generador almacenado en el disco duro y conocer su ruta. Por supuesto, lo más adecuado será almacenarlo en una carpeta adyacente a nuestro programa.

3 Introducción al análisis de documentos PDF

Antes de tratar algunos detalles del análisis de accesibilidad de los documentos PDF, vamos a explicar brevemente cómo se estructura un documento PDF.

El formato PDF tiene como objetivo ser utilizable en cualquier sistema operativo y hardware. Para ello, toda la información necesaria debe estar contenida en el propio fichero, incluso aspectos como la tipografía del texto. Internamente, un documento PDF se compone de cuatro partes:

- Cabecera: detalles sobre el estándar seguido, como por ejemplo la versión.
- Cuerpo: contiene los elementos que se presentan en el documento, como por ejemplo imágenes, texto, etc.
- Tabla de referencias cruzadas: contiene la información necesaria para que ciertos elementos del documento referencien a otros.
- Coda: indica dónde se encuentra la tabla de referencias cruzadas.

En la Figura 2[8] podemos ver un ejemplo del contenido de un documento PDF. El cuerpo es la parte más importante a revisar. Esta sección posee una estructura de árbol cuyos nodos son etiquetas, las cuales proporcionan información sobre los distintos elementos que aparecen en el documento, algo crucial para que los lectores de pantalla puedan leerlos correctamente. De hecho, el orden de aparición de los elementos en este árbol de etiquetas también debe ser cuidado.

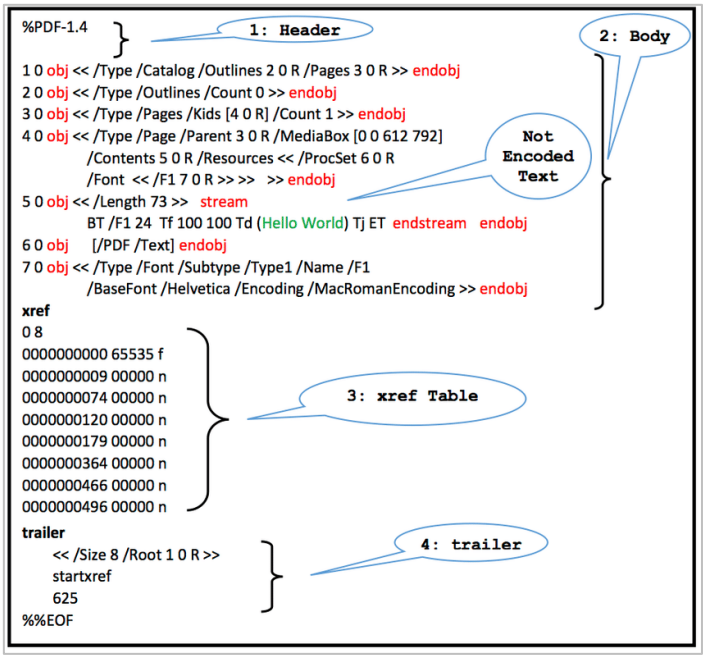

Figura 2. Ejemplo de la estructura de un documento PDF

[8] Fuente de la Figura 2: https://www.researchgate.net/figure/An-example-of-a-simple-PDF-file-structure-that-consists-of-one-page-that-contains-a_fig1_326102942

Dicho esto, se siga el estándar ISO 14289-1:2014 o el WCAG 2.0, analizar la accesibilidad de un documento PDF no es tarea fácil. De hecho, existen propiedades que todavía necesitan ser comprobadas manualmente por una persona, como por ejemplo el contraste de color entre el texto y el fondo sobre el que está escrito, o que los elementos del árbol de etiquetas sigan un orden lógico. Paralelamente a la realización de este proyecto, hicimos el intento de desarrollar un software propio de análisis de accesibilidad de los documentos PDF, tomando como referencia los análisis que realiza Adobe Acrobat Reader y utilizando la librería PDFBox[9], también en lenguaje Java. Verificar la existencia de los metadatos del documento fue algo sencillo de conseguir (título del documento, idioma del texto, etc.), pero, al intentar analizar otras propiedades más trabajosas, la dificultad aumentaba. Un ejemplo de esto es detectar aquellos elementos del documento que no estén etiquetados, pues era necesario determinar si el identificador de cada elemento se encontraba dentro del árbol de etiquetas, y para ello era necesario leer el código del documento, analizando su sintaxis para encontrar los elementos que nos interesan y así poder obtener sus identificadores.

Por lo tanto, la alternativa más factible para el desarrollo de este proyecto es utilizar herramientas de terceros. El analizador de Adobe Acrobat DC sería la solución más completa, pero es de pago y pretendemos desarrollar un software que funcione gratuitamente. Por ello, las alternativas gratuitas que utilizaremos serán Tingtun y PAVE. Ambas son herramientas alojadas en la nube, accesibles a través de un navegador web.

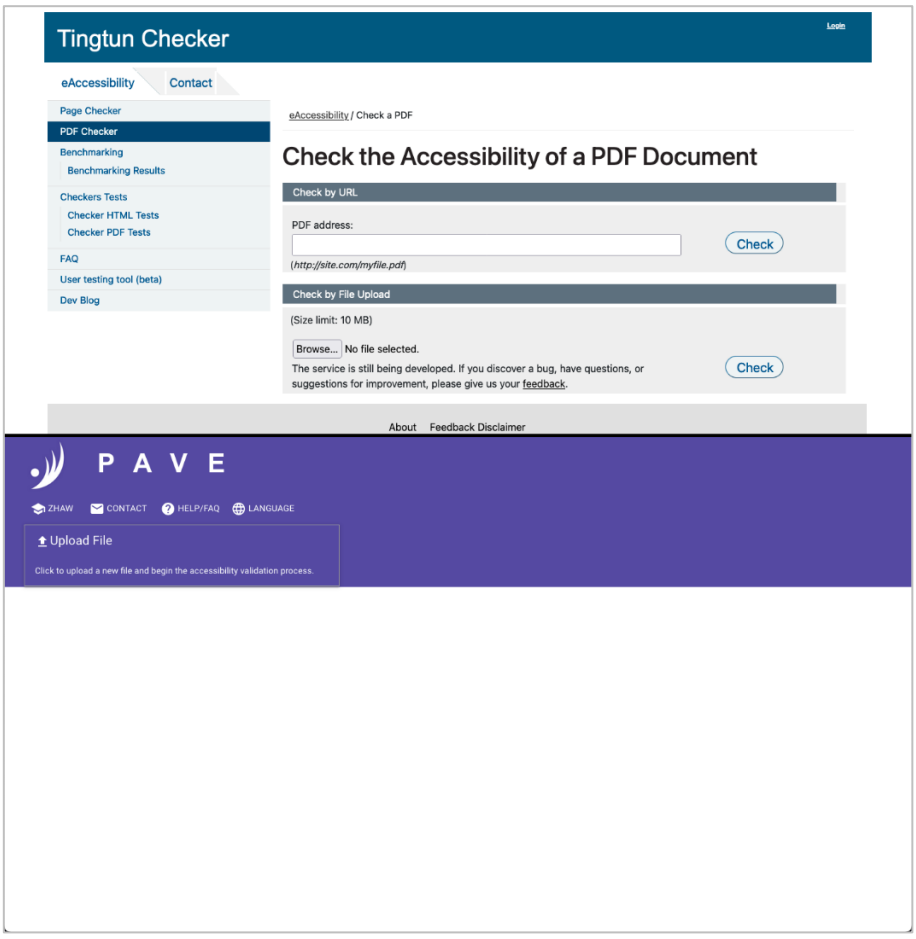

Figura 3. Páginas principales de las herramientas Tingtun y PAVE, respectivamente

[9] Sitio web oficial de la librería java PDFBox: https://pdfbox.apache.org/

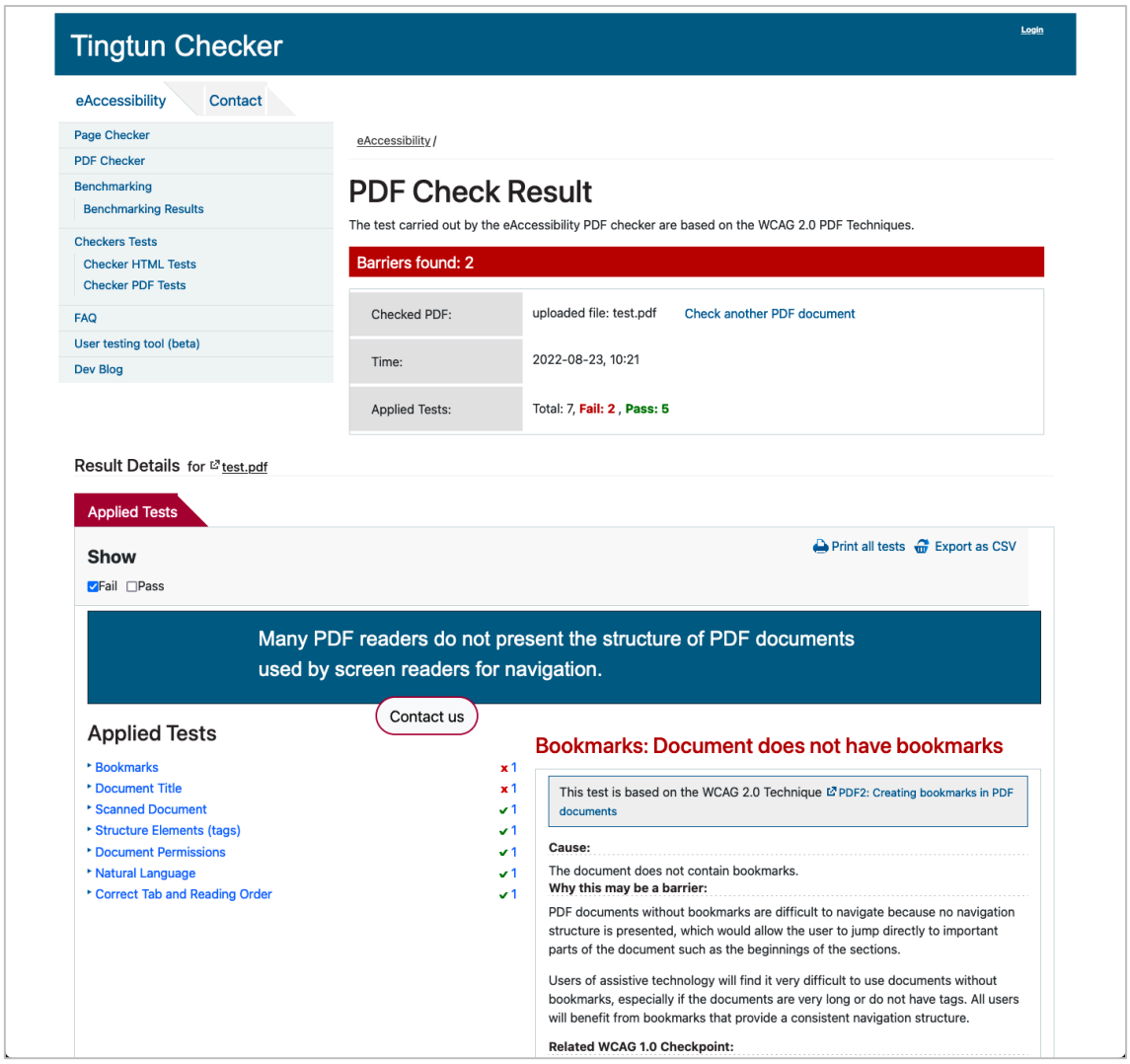

Figura 4. Resultados de Tingtun

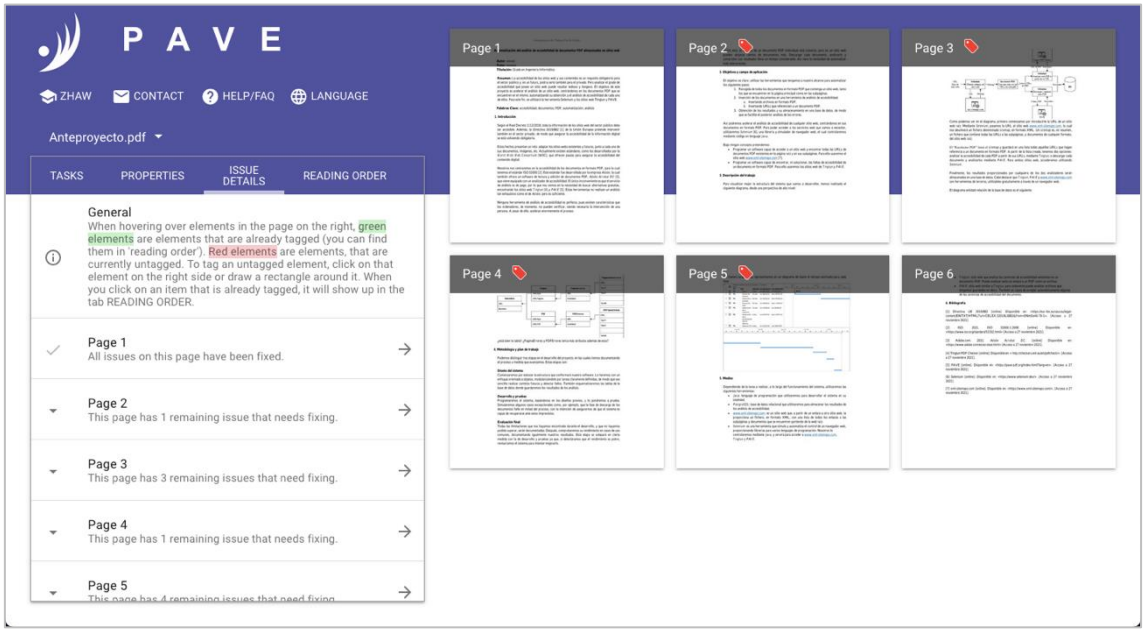

Figura 5. Resultados de PAVE

Como podemos ver en las figurasFigura 3,Figura 4 yFigura 5, para utilizar estas herramientas debemos interactuar con sus interfaces gráficas de usuario. Para automatizar esta tarea utilizaremos Selenium, herramienta que explicaremos en la sección 4.

En el caso de Tingtun, podemos introducir directamente la URL del documento que queremos analizar, o enviar el archivo que tengamos almacenado en nuestro disco duro. En nuestro caso, introduciremos directamente la URL, así no tendremos que descargar primero el documento para cargarlo después, lo que ralentizaría la ejecución. En el caso de PAVE no nos queda otra opción que cargar el documento desde nuestro disco duro.

A continuación, echaremos un vistazo a las propiedades de accesibilidad que analizan ambas herramientas.

3.1 Análisis de Tingtun

Si consultamos su página web, encontraremos una sección titulada “Checker PDF Tests”, donde nos proporcionan una lista de todas las propiedades de accesibilidad que revisa Tingtun. Vemos también que Tingtun sigue el estándar WCAG 2.0 para llevar a cabo su análisis, aunque no revisa todas las propiedades.

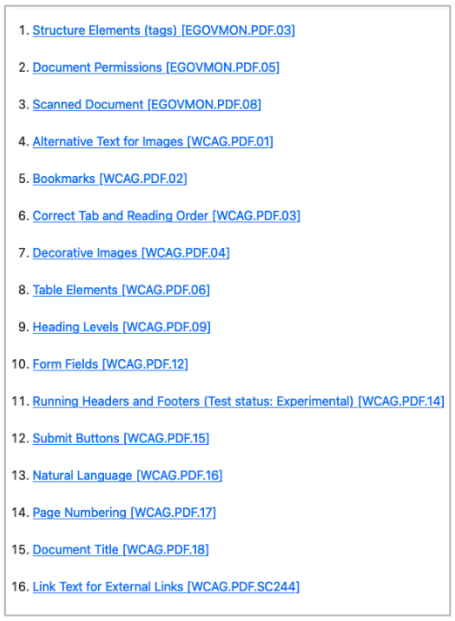

Figura 6. Lista de propiedades que analiza Tingtun

3.2 Análisis de PAVE

En su página web podemos encontrar, en la sección “FAQ”, la siguiente pregunta frecuente: “¿Qué necesitamos para que un documento PDF sea accesible?” (*What do we need to make a PDF document accessible?*). En la respuesta a esta pregunta nos explican las propiedades que analiza PAVE:

- Los metadatos del documento deben estar completados correctamente
- El documento debe tener todos sus elementos etiquetados

Los metadatos del documento son propiedades como, por ejemplo, el idioma en el que está escrito el texto, el título del documento, etc. En cuanto a los elementos etiquetados, se refiere

a introducir todos los elementos en el árbol de etiquetas explicado anteriormente. Por ejemplo, varias letras se pueden agrupar bajo una etiqueta de tipo párrafo, de tipo título etc.; o una figura se puede etiquetar como de tipo tabla. Esta estructura de etiquetas ayuda a los lectores de pantalla a leer el documento correctamente.

Aunque pueda parecer poca cosa, sigue resultando muy útil la información que nos proporciona. Que los elementos estén debidamente etiquetados es una característica muy importante para garantizar la accesibilidad del documento. Además, PAVE nos indica la página exacta donde se encuentra el error, y detalla mínimamente el motivo del error.

4 Arquitectura

Una vez resueltos los problemas de búsqueda de los documentos PDF alojados en un sitio web, y la obtención de los resultados de análisis de accesibilidad, podemos proceder al diseño del sistema.

4.1 Herramientas utilizadas

Para la generación del *sitemap* utilizamos el programa Sitemap Generator, explicado en la sección 2.2.4.

Para los análisis de accesibilidad de los documentos PDF, las herramientas web Tingtun y PAVE, como explicamos anteriormente.

Para automatizar la navegación por Tingtun y PAVE, y la posterior extracción de los resultados de análisis, utilizamos una herramienta llamada Selenium, la cual nos proporciona librerías para varios lenguajes de programación, así como los controladores de diferentes navegadores web. Esta herramienta fue concebida principalmente para realizar pruebas automáticas a los sitios web, con el propósito de comprobar su calidad, pero también sirve para automatizar tareas tediosas, como pretendemos hacer nosotros. Los controladores web funcionan igual que un navegador web, de hecho, Selenium ofrece varios controladores utilizables, como Chrome o Firefox. Estos controladores vienen preparados para que, desde código Java, podamos configurar diferentes opciones del navegador como, por ejemplo, que el controlador sea *headless*, es decir, que no abra una ventana donde nos muestre por pantalla su ejecución.

El lenguaje de programación que utilizamos es Java, debido a que es un lenguaje conocido y equipado con funciones que nos facilita la gestión de tareas concurrentes, además de que Selenium ofrece soporte para este lenguaje. Utilizaremos el JDK 17, debido a que es la última versión LTS (*Long Term Support*), es decir, la última a la cual seguirán dando soporte por un largo periodo de tiempo.

El sistema gestor de bases de datos utilizado es PostgreSQL, el cual sigue un modelo relacional, también por ser un sistema conocido y por ofrecer las capacidades necesarias para comunicarnos con este a través de Java, así como facilitar ciertos aspectos de la transacción de datos (principio ACID, como detallaremos más adelante). Estos últimos motivos son actualmente comunes en la mayoría de gestores conocidos.

4.2 Diseño general

En el anteproyecto presentamos un diagrama, desde una perspectiva de alto nivel, de los componentes que deberían formar nuestro sistema y la ruta que trazarían los datos, desde la inserción de la URL del sitio web que queremos analizar, hasta el almacenamiento de los resultados en la base de datos. Este diagrama se puede ver a continuación:

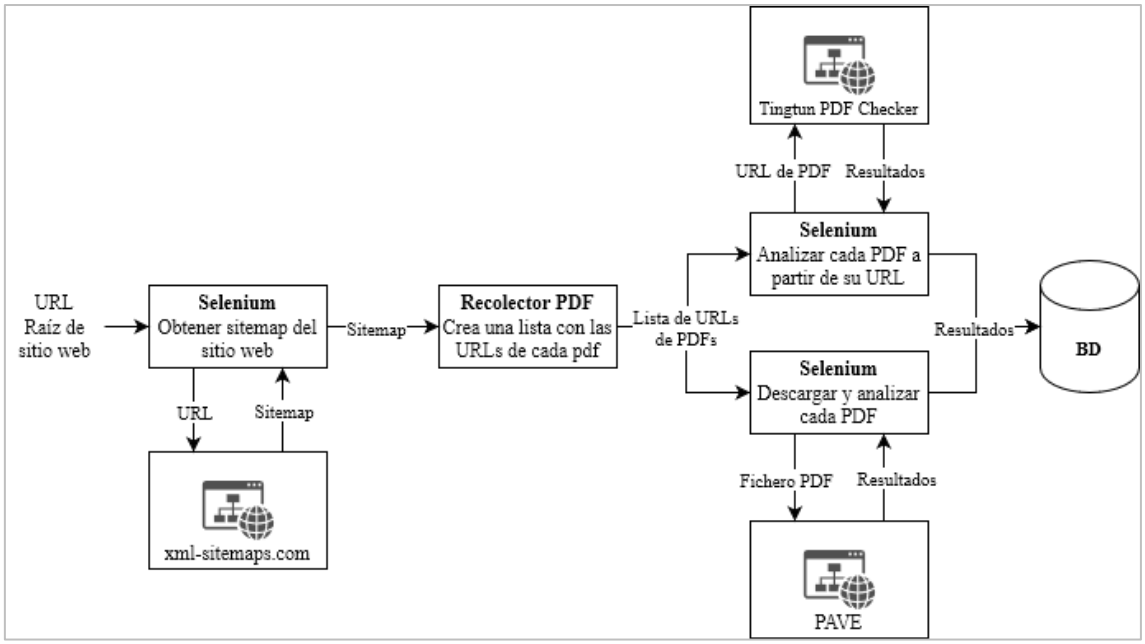

Figura 7. Diagrama general, presentado en el anteproyecto, del proceso que seguiría nuestro software

Pero este esquema general ha sufrido un ligero cambio, reemplazando el uso de la web xml-sitemaps.com por el Sitemap Generator, explicado en la sección 2.2.4, como podemos ver a continuación:

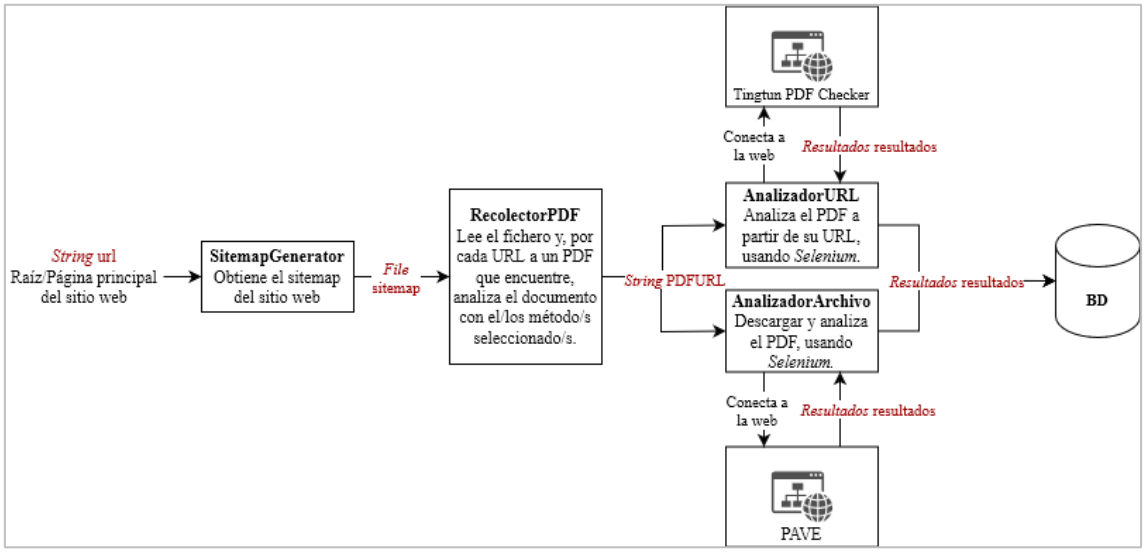

Figura 8. Diagrama general de nuestro software, después de incorporar el uso de Sitemap Generator

Cabe destacar que, aunque desde el inicio se haya pretendido hacer un esquema general de los elementos que compondrían nuestro sistema, la implementación se fue desarrollando bloque por bloque. Es decir, primero solucionábamos una funcionalidad concreta del sistema, antes de proceder con la siguiente. Concretamente, los bloques en los que dividimos el desarrollo del sistema son prácticamente los que podemos ver en el esquema anterior:

1. Generador del *sitemap*: aunque esta funcionalidad ya nos la proporcionaba el programa Sitemap Generator, tuvimos que crear una clase encargada de ejecutarla y gestionarla.
2. Recolector PDF: lectura del fichero *sitemap*, identificando los documentos PDF y lanzando el análisis de accesibilidad sobre ellos.

3. Análisis: obtención de los resultados de análisis de los documentos PDF, a través de la herramienta Selenium. Por cada herramienta de análisis era necesario programar acciones diferentes.
4. Almacenamiento: comunicación con la base de datos para guardar en ella los resultados de los análisis.

Por otra parte, el caso de uso es, prácticamente, único: lanzar el ejecutable JAR, insertándole los argumentos de entrada necesarios, y esperar a que finalice el proceso. Debido a esto, no se modelarán los casos de uso para este proyecto.

Los argumentos de entrada que esperará nuestro programa, por orden de aparición, son los siguientes:

1. El primer argumento debe ser la URL del sitio web que queremos analizar
2. El segundo argumento debe ser el nombre con el que queramos referirnos al *sitemap* del sitio web, de modo que pueda identificarse y reutilizarse un *sitemap* ya creado con anterioridad que esté presente en nuestro disco duro.
3. Todos los argumentos siguientes son opcionales. Aquí podremos especificar exactamente los tipos de analizadores que querremos utilizar, pudiendo utilizar solamente el analizador de Tingtun, solamente el de PAVE, ambos, o cualquier otra combinación que implique analizadores nuevos añadidos con posterioridad.

4.3 Proceso de diseño final

A continuación, comenzaremos por explicar el primer enfoque que le dimos al diseño del sistema. Después, explicaremos en detalle cada componente añadido, eliminado o modificado. De esta forma, pretendemos reflejar los problemas a los que nos hemos ido enfrentando, los detalles que hemos querido perfeccionar y, por lo tanto, el fundamento de las decisiones que hemos tomado durante el desarrollo. Aunque, para tener un mayor contexto del diseño que hemos implementado, y así comprender mejor las explicaciones venideras, comenzaremos primero por explicar la estructura de nuestra base de datos.

4.3.1 Estructura de la base de datos

Como ya hemos dicho, el gestor que vamos a utilizar es PostgreSQL. Los datos que necesitaremos guardar son los siguientes:

- El sitio web que hemos analizado. Bastará con su URL y la fecha en que se realizó el análisis, de modo que así podamos diferenciar los análisis llevados a cabo hasta el momento y conocer el tiempo que ha pasado desde la última vez que se analizó.
- Cada documento PDF analizado. Guardaremos su URL y lo relacionaremos con la tabla destinada a los sitios webs. También lo tendremos que relacionar con la tabla que guarde los resultados de análisis.
- Los resultados de los análisis. Cada herramienta web ofrece unos resultados diferentes y, en el caso de Tingtun, podemos conocer exactamente qué propiedades analiza, a diferencia de PAVE. Por ello, la forma en que guardaremos los resultados de los análisis variará según la herramienta web que utilicemos, lo que implica la necesidad de crear tablas diferentes para cada herramienta.

Por lo tanto, un primer modelo lógico de la estructura de nuestra base de datos relacional puede ser el de la Figura 9.

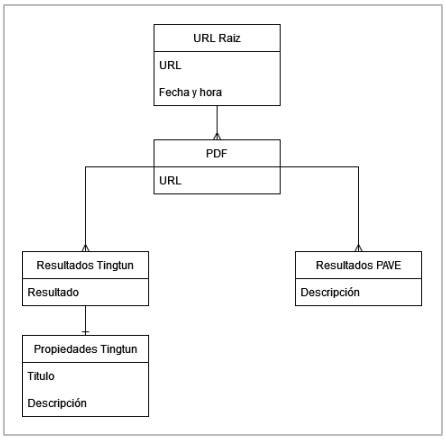

Figura 9. Boceto de modelo lógico para la base de datos

Algo que puede llamar la atención es que los resultados de PAVE, a diferencia de los de Tingtun, necesitan de una única tabla para almacenarse, en lugar de dos. Comencemos primero por tratar las tablas de Tingtun: como explicamos en la sección 3.1, tenemos acceso a la lista completa de las propiedades que, se supone, comprueba Tingtun. Teniendo esta información, lo que hacemos es crear una tabla que mantenga el título y la descripción de cada propiedad de esa lista. Así, en la tabla de resultados guardaremos una pequeña cadena de texto con el resultado recibido, y tan solo tendremos que añadirle una clave foránea de la tabla de propiedades. En cuanto a la tabla de resultados de PAVE, sus desarrolladores no nos proporcionan ninguna lista similar. La solución a esta incertidumbre es introducir los resultados en una única tabla, sea cuales sean los que obtengamos. Como PAVE solamente nos avisa de los errores, tampoco tendría sentido guardar el resultado de la propiedad analizada.

Dicho esto, procedemos a modelar la base de datos utilizando el programa pgModeler. El motivo de utilizar este software es que, una vez terminado el modelado, nos permite exportar la base de datos directamente en un servidor de PostgreSQL. También nos permite generar el código SQL necesario para su creación. El modelo final es el siguiente:

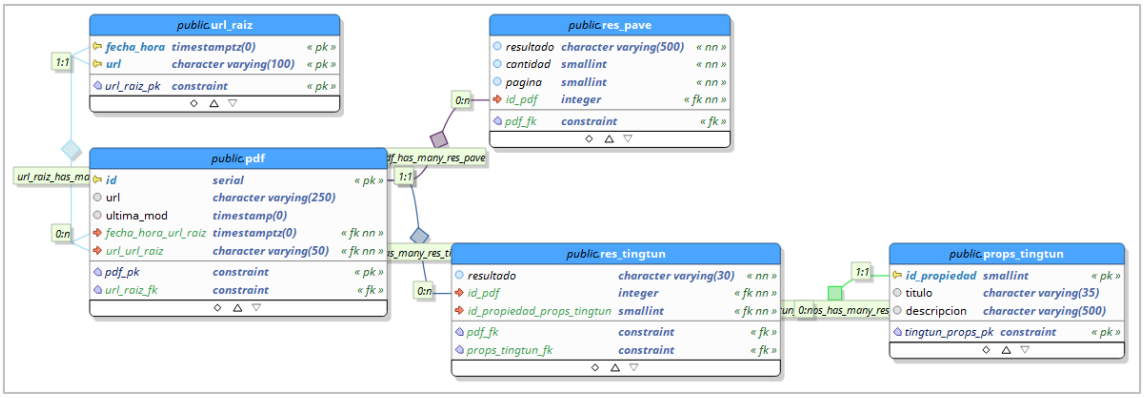

Figura 10. Modelo lógico de la base de datos

A continuación, explicaremos el diseño de cada tabla:

- url_raiz: se mantiene prácticamente igual que en el boceto. Ambas columnas se consideran como clave primaria, para así poder diferenciar los distintos análisis que se han realizado hasta el momento.
- pdf: su clave primaria es una nueva columna que posee un número identificador, asignado ordenadamente según se vayan introduciendo nuevos documentos. Se ha

optado por usar este identificador como clave primaria para que, al almacenarse los datos en el disco duro, la tablas “res_tingtun”, “res_pave” y cualquier otra posible tabla futura de resultados de análisis, ocupen menos espacio. Además, así no será necesario introducir en esas tablas la URL, la fecha y la hora de análisis como claves foráneas. Por último, se ha añadido la columna “ultima_mod”, que guarda la fecha y hora de la última vez que se modificó el documento. Gracias a este dato, podemos saber si un documento que no cumple con los requisitos de accesibilidad ha sido modificado posteriormente al Real Decreto, lo que implicaría que el documento debe ser actualizado obligatoriamente.

- res_tingtun: no posee claves primarias, actúa como una relación entre las tablas “pdf” y “props_tingtun”, y su único atributo es el resultado del análisis de una propiedad en concreto. Este dato se formatea como una cadena de texto ya que, además de poder aprobar o suspender el análisis, también existe la posibilidad de que esa propiedad deba comprobarse manualmente.
- props_tingtun: su clave primaria es un identificador similar a la tabla “pdf”. Por lo demás, se mantiene igual que en el modelo lógico. En esta tabla se guardan las propiedades que analiza Tingtun, y estas propiedades están recopiladas, también, en un fichero en formato CSV para facilitar la inserción de los datos en la tabla.
- res_pave: en esta tabla hemos terminado añadiendo dos columnas más, a parte de la clave foránea de la tabla “pdf”. Estas columnas son “cantidad” y “pagina”, y la necesidad de incorporarlas surgió después de experimentar con los resultados de PAVE: esta herramienta detecta errores para cada página del documento PDF y, de vez en cuando, encuentra un mismo error repetido varias veces. Por ello, en la columna “cantidad” guardamos el número de repeticiones del error y en la columna “pagina” el número de la página a la que pertenece.

Como se puede comprobar, el modelo de la base de datos es sencillo. Está normalizado hasta la segunda forma normal, y podríamos normalizarlo hasta su tercera forma si renombráramos la tabla “pdf” como “análisis”, moviendo la columna “url” a otra nueva tabla llamada “pdf” y añadiendo, a esta, una columna de identificadores que actúe como clave primaria. Consideramos que, al menos de momento, normalizarlo a la tercera forma, o superior, no merecería la pena, pues complicaríamos el modelo en lugar de simplificarlo.

4.3.2 Primer enfoque

Comenzamos por diseñar un diagrama de clases que siga el modelo del esquema general que hemos presentado anteriormente (ver Figura 11):

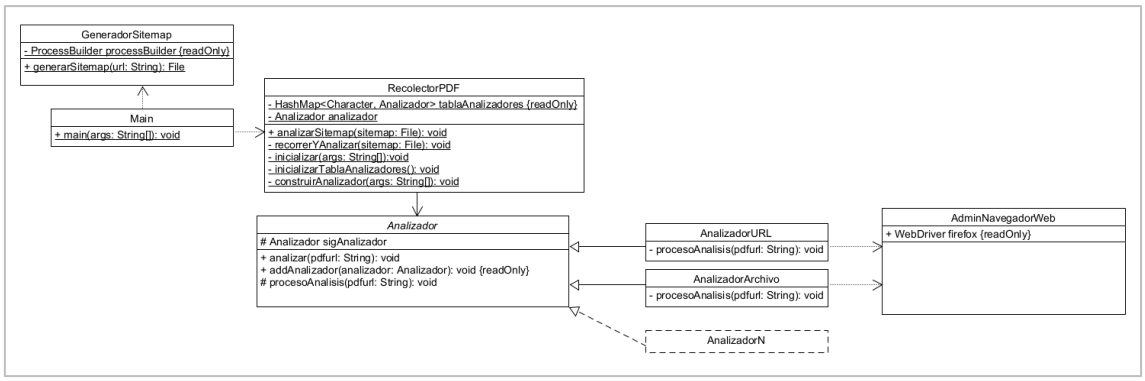

Figura 11 Diagrama

Procedemos a detallar la función de cada clase:

- Main: la primera clase del sistema que se ejecutará. Se encarga de, primero, utilizar la clase GeneradorSitemap para obtener el fichero *sitemap* y, después, pasarle el fichero a la clase RecolectorPDF.
- GeneradorSitemap: genera un nuevo *sitemap*, ejecutando el programa Sitemap Generator, o utiliza un fichero ya existente en nuestro disco duro, que corresponda al mismo sitio web que queremos analizar.
- RecolectorPDF: lee el fichero *sitemap* y extrae cada URL perteneciente a un documento en formato PDF, pasándola a los analizadores (clases que extienden a la clase Analizador) correspondientes. Es por esto último que esta clase posee un HashMap de caracteres y de la clase Analizador: cada caracter identifica un Analizador, y los caracteres que recibirá esta clase durante su ejecución serán aquellos introducidos como argumentos de entrada del programa. Además, esta clase se encarga también de construir el analizador que integra todos los seleccionados por el usuario.
- Analizador: clase abstracta que representa la estructura mínima que deberán poseer las clases encargadas de realizar el análisis de accesibilidad. Posee un atributo llamado sigAnalizador, que guarda el siguiente analizador que se utilizará durante la ejecución del programa. La intención de esto era crear una especie de lista enlazada de analizadores, de modo que, al terminar el primer analizador su ejecución, este arranque el análisis del segundo, y así sucesivamente hasta que todos los analizadores hayan terminado su proceso. La función “addAnalizador” sería la encargada de asignar un nuevo analizador a la cadena. También sería necesario una clase extra, la cual contuviera esta cadena de analizadores y ofreciera la función que ejecutara el análisis de todos ellos, pero este diseño, como veremos posteriormente, será finalmente desechado. Por otro lado, la función “procesoAnalisis” es una función privada, que contiene todos los pasos necesarios para interactuar con la herramienta web y obtener los resultados, mientras que la función “analizar” será la función pública, que ejecuta a “procesoAnalisis” y después llama a la función “analizar” del siguiente analizador.
- AnalizadorTingtun: utiliza Selenium para navegar en Tingtun y obtener los resultados de análisis.
- AnalizadorPAVE: igual que “AnalizadorTingtun”, pero para la herramienta PAVE.
- AdminNavegadorWeb: en un principio se creó esta clase para gestionar el uso del controlador web que utiliza Selenium para funcionar, ya que se ejecutarían varios análisis concurrentemente. Más adelante, una vez profundizado en el funcionamiento de Selenium, se desechó esta clase debido a su modo de funcionar, pues era necesario ejecutar tantos controladores web como análisis concurrentes quisiéramos realizar, uno por cada hilo. Por lo tanto, pensamos que cada clase “Analizador” tendría que ser la encargada de gestionar su propio controlador web. Pero, al final, vimos que la ejecución se ralentizaba considerablemente, ya que instanciar y apagar un nuevo controlador web es costoso, además de que no apagar el controlador, una vez utilizado, provoca que se queden procesos huérfanos en nuestro sistema operativo. Por estos motivos crearemos una clase llamada “DriverPool”, que se encargará de solucionar este problema, como veremos en la siguiente sección.

Como se puede comprobar, todavía no hemos mencionado nada al respecto de la comunicación con la base de datos. Durante el desarrollo de nuestro proyecto, esta tarea la dejamos para el final, una vez fuéramos capaces de extraer los resultados de análisis. Así el desarrollo era ordenado y, por lo tanto, más fácil de gestionar. Además, supusimos que implementar la

comunicación con la base de datos no requeriría de una refactorización de código sustancial. En la siguiente sección introduciremos la gestión de la comunicación con la base de datos.

4.3.3 Mejoras del diseño

4.3.3.1 La clase "RecolectorPDF"

En primer lugar, la clase "RecolectorPDF" realiza demasiadas funciones. Para que el código sea lo más limpio y modular posible, es recomendable que cada clase se dedique a una tarea concreta. De esta forma, cuando queramos realizar cambios al código, será sencillo identificar las clases involucradas en la funcionalidad que queremos modificar, además de que el acoplamiento del sistema se verá reducido. Esto implica que el número de clases que tendremos que modificar, como consecuencia de haber modificado "RecolectorPDF", también disminuirá.

Dicho esto, la clase "RecolectorPDF" debería dedicarse únicamente a la lectura del fichero *sitemap* generado, identificando las URL que referencien documentos PDF y pasando estas a otra clase que los analice. Por ello, crearemos la clase "EjecutorAnalisis", que se encargará de la construcción y ejecución de los analizadores.

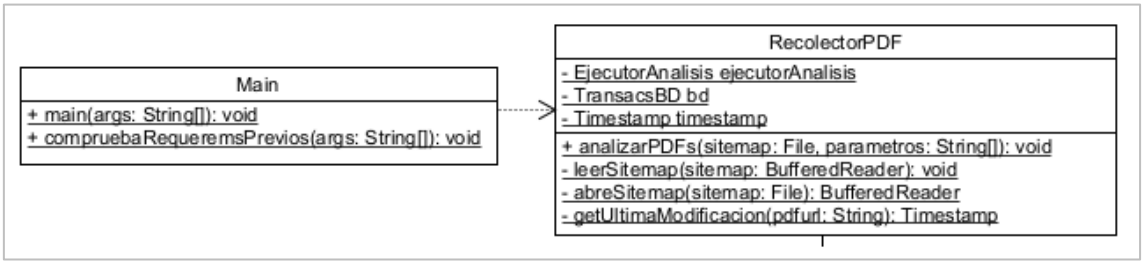

Figura 12. Las clases "Main" y "RecolectorPDF" en nuestro diagrama de clases

Atributos:

- "ejecutorAnalisis": instancia de "EjecutorAnalisis", al cual enviará las URL de los documentos PDF encontrados.
- "bd": instancia de la clase "TransacsBD", clase encargada de realizar las transacciones necesarias con la base de datos, la cual detallaremos en la sección 4.3.3.7.
- "timestamp": posee la fecha y hora del momento en que se insertó la URL del sitio web raíz en la base de datos, ya que es una clave foránea en la tabla "pdf".

En cuanto a sus funciones, apenas hay cambios:

- "analizarPDFs": misma función que "analizarSitemap", y esta vez recibe también los parámetros de entrada del programa, para acabar enviándoselos al "EjecutorAnalisis". También guarda la URL raíz del sitio web en la base de datos.
- "leerSitemap": misma función que "recorrerYAnalizar". Recorre el fichero *sitemap* y realiza las llamadas correspondientes a "EjecutorAnalisis". Cabe destacar que, si en los argumentos de entrada del programa se especifica la opción de ignorar aquellos documentos PDF más antiguos que una fecha indicada, es aquí donde se llama a la función "getUltimaModificacion" para comprobar la fecha.
- "abreSitemap": prepara el fichero *sitemap* para la función "leerSitemap".
- "getUltimaModificacion": consulta la fecha de la última modificación del documento, dato que también se almacena en la tabla "pdf" de la base de datos.

4.3.3.2 La clase "EjecutorAnalisis"

Esta clase gestionará la construcción de los analizadores indicados por el usuario, así como su ejecución. Las funciones dedicadas a esta tarea, que antes poseía "RecolectorPDF", pasan a esta clase.

Además, como adelantamos anteriormente, la estructura similar a una lista enlazada de analizadores será reemplazada por una lista de analizadores, usando la clase "List" que proporciona Java. En esta lista se guardarán los tipos de analizadores, indicados por el usuario, que analizarán los documentos PDF. Así, en la función "analizar", tan solo tendremos que recorrer la lista y ejecutar cada analizador que contenga.

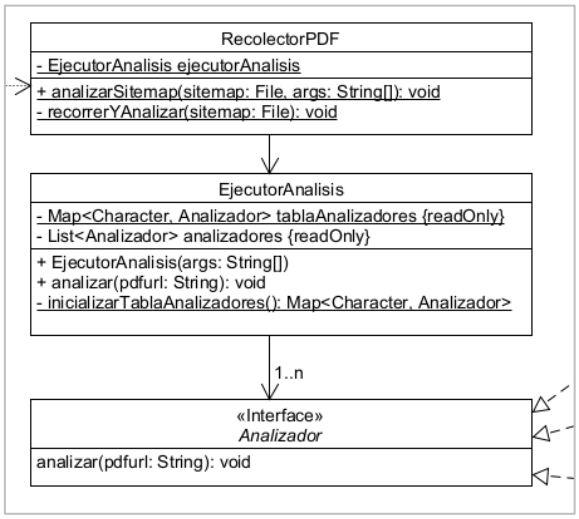

Figura 13. Estructura de las clases "RecolectorPDF", "EjecutorAnalisis" y "Analizador"

Pero al implementar los analizadores de este modo, nos cruzamos con un problema: dado que esta lista de analizadores tiene objetos previamente instanciados, se estarán utilizando siempre los mismos objetos para todos los análisis que se realicen. Si la ejecución de estos análisis se realizara secuencialmente, este hecho no resultaría ningún problema, pero nosotros pretendemos ejecutar los análisis concurrentemente (como veremos a continuación). Al usar los mismos objetos para analizar varios documentos PDF a la vez, tendríamos que lidiar con los siguientes hechos:

- Cada objeto utiliza un único controlador web para navegar por sus respectivas herramientas de análisis.
- Cada objeto se comunica con la base de datos para guardar los resultados a través de, también, un único objeto.
- Cada objeto tiene atributos que, si se usara el mismo durante la ejecución concurrente de varios hilos, estos se verían modificados de maneras impredecibles. A esto se le llama condición de carrera.

Por todos estos motivos, la ejecución de los análisis produciría errores imprevistos. La solución a esto es instanciar un nuevo objeto de cada tipo de analizador, por cada documento PDF. Así no ocurrirán condiciones de carrera, ni problemas al navegar por las herramientas web, ni colisiones al realizar transacciones a la base de datos. Esto se puede lograr si el contenido de la lista, en lugar de ser de objetos "Analizador", fuera de objetos de clase "Supplier", cuyo contenido a su vez fuera de clase "Analizador". En Java, la clase "Supplier" sirve para obtener nuevas instancias de una clase determinada.

Continuando con más cambios, si queremos aprovechar los recursos del ordenador, lo más conveniente será utilizar hilos. En este escenario es fácil ver dónde podemos usarlos: ya que "EjecutorAnalisis" tendrá que ejecutar los análisis para cada documento PDF que reciba de la clase "RecolectorPDF", podemos perfectamente repartir los análisis en varios hilos. Para ello, bastará con usar el servicio de ejecución de hilos que nos ofrece Java. Le asignaremos a "EjecutorAnalisis" un objeto de la clase "ExecutorService", el cual será un "thread pool" (o banco de hilos) de tamaño fijo, de modo que, según se vaya obteniendo cada documento PDF, se creará un hilo para que lo analice, hasta un límite máximo de hilos. El límite de hilos se podrá especificar por los argumentos de entrada y, en caso de no especificar una cantidad, se utilizará el doble de hilos que núcleos lógicos tenga nuestra CPU. Se ha escogido esta cantidad de hilos por ser la que menos tiempo de ejecución producía en consecuencia, descubierto después de probar el análisis de un mismo *sitemap* con distintas cantidades de hilos. Muy probablemente la cantidad de hilos óptima varíe por cada máquina que ejecute el programa.

Finalmente, todavía podemos reducir todavía más las tareas que debe realizar esta clase. "EjecutorAnalisis" debería dedicarse solamente a eso, a ejecutar los análisis de accesibilidad para cada documento PDF que reciba. Pero actualmente también gestiona la construcción de del analizador, manteniendo el HashMap con los tipos de analizadores disponibles. Estas gestiones debería realizarlas una clase separada, a la cual denominaremos "ConstructorAnalizador". Aplicando los cambios necesarios, la estructura de "EjecutorAnalisis" se puede ver en la Figura 14.

Un aspecto importante a tener en cuenta es que "EjecutorAnalisis" también hace uso de una nueva clase, llamada "DriverPool". Esta clase gestiona los controladores web que facilita Selenium, como detallaremos en la sección 4.3.3.3.

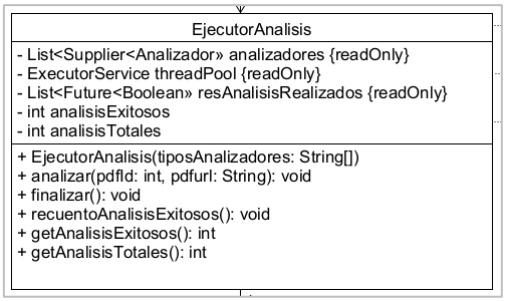

Figura 14. Estructura de la clase "EjecutorAnalisis"

Atributos:

- "analizadores": lista de *suppliers* de la clase "Analizador". Contiene los analizadores seleccionados por el usuario. Los análisis se ejecutarán creando instancias de la clase "Analizador" a partir de esta lista.
- "threadPool": banco de hilos. La clase "ExecutorService", ofrecida por Java, gestiona fácilmente el uso de una cantidad fija de hilos. Con esto ejecutaremos los análisis concurrentemente.
- "resAnalisisRealizados": lista con los objetos de clase "Future", los cuales poseerán el resultado del análisis realizado por cada hilo. Concretamente, si uno de los analizadores no termina su tarea debido a una excepción, su objeto "Future" correspondiente poseerá el valor booleano *False*. En caso contrario, el valor será *True*. Esta lista nos servirá para conocer la cantidad de documentos PDF que han sido completamente analizados (es decir, por todos los analizadores seleccionados).

Funciones:

- "EjecutorAnalisis": constructor de la clase. Aquí inicializamos la clase "DriverPool" para preparar los controladores web.
- "analizar": utiliza un hilo del "threadPool" para analizar un documento PDF y guardar el resultado en "resAnalisisRealizados".
- "finalizar": apaga el "threadPool" y los controladores web del "DriverPool".
- "recuentoAnalisisExitosos": consulta los objetos de la lista "resAnalisisRealizados" para contar la cantidad de análisis completados.

4.3.3.3 La clase "DriverPool"

Como ya adelantamos, esta clase nace por la necesidad de aumentar el rendimiento del programa. Instanciar y apagar un objeto de la clase "WebDriver", por cada analizador, es un gasto innecesario de recursos. Por ello, la clase "DriverPool" poseerá una cola de controladores web, y cada hilo irá recogiéndolos y depositándolos según corresponda. Esta cola contendrá tantos controladores como cantidad de hilos se utilicen para los análisis.

Cabe destacar que existe una forma más eficiente de gestionar el uso de los controladores por parte de los hilos, explicada en la sección 7.2.1.6. No la llegamos a implementar debido a la falta de tiempo.

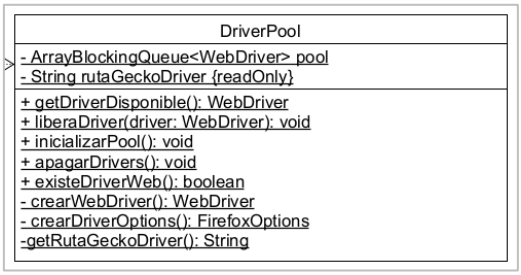

Figura 15. Estructura de la clase "DriverPool"

Atributos:

- "pool": cola de controladores. Su clase es "ArrayBlockingQueue", una clase facilitada por Java, que gestiona una estructura de tipo cola y puede ser utilizada de forma segura por varios hilos ejecutados concurrentemente.
- "rutaGeckoDriver": directorio donde se encuentra el controlador web facilitado por Selenium.

Funciones:

- "getDriverDisponible": obtiene un controlador disponible de la cola.
- "liberaDriver": deposita un controlador de vuelta a la cola.
- "inicializarPool": instancia el atributo "pool" y le inserta tantos controladores web como cantidad de hilos se hayan especificado en los argumentos de entrada del programa. Esta función es utilizada por la clase "EjecutorAnalisis" antes de recorrer el *sitemap*.
- "apagarDrivers": apaga todos los controladores de la cola. Esta función es utilizada por la clase "EjecutorAnalisis" al terminar de obtener los resultados de todos los análisis.
- "existeDriverWeb": comprueba que el ejecutable del controlador web se encuentre en el directorio indicado por "rutaGeckoDriver".

- "crearWebDriver": instancia un nuevo objeto de la clase "WebDriver". Es llamada por la función "inicializarPool".
- "crearDriverOptions": configura las opciones que queremos que posean los controladores web. Es llamada por la función "crearWebDriver".
- "getRutaGeckoDriver": determina el valor del atributo "rutaGeckoDriver". Dependiendo del sistema operativo desde donde se ejecute el programa (Windows o Linux), la ruta acabará en ".exe" o no.

4.3.3.4 La clase "ConstructorAnalizador"

Esta clase mantendrá el "HashMap" con los tipos de analizadores disponibles, además de encargarse de construir el analizador. Sencillamente traspasamos los atributos y funciones relacionados de la clase "EjecutorAnalisis" a esta clase. Cuando el "EjecutorAnalisis" necesite comenzar el análisis de un documento, simplemente tendrá que utilizar esta clase.

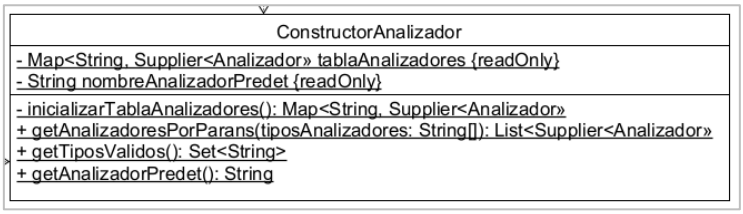

Figura 16. Estructura de la clase "ConstructorAnalizador"

Atributos:

- "tablaAnalizadores": objeto "HashMap" con los nombres de cada tipo de analizador disponible y su respectivo *supplier*.
- "nombreAnalizadorPredet": nombre del analizador predeterminado que se quiera utilizar, en caso de que no se especifique uno en los argumentos de entrada del programa.

Funciones:

- "inicializarTablaAnalizadores": inserta dentro de "tablaAnalizadores" los tipos de analizadores disponibles.
- "getAnalizadoresPorParams": a partir del texto de los argumentos de entrada del programa, crea una lista con los *suppliers* de los analizadores correspondientes.
- "getTiposValidos": devuelve un "Set", o conjunto, con los nombres de los analizadores disponibles. Utilizado para comprobar la validez de los argumentos de entrada del programa.
- "getAnalizadorPredet": devuelve el valor de "nombreAnalizadorPredet".

4.3.3.5 La clase "Analizador"

Como explicamos inicialmente, esta clase deja de actuar como si fuera un elemento perteneciente a una lista enlazada, por lo que su atributo "sigAnalizador" y su función "addAnalizador" desaparecen.

Añadimos una tercera función, llamada "crearBD", abstracta, que deberá devolver una instancia de la clase "TransacsBD". Con esta función, el analizador crea el objeto que utilizará para almacenar los resultados de análisis en la base de datos. Cada analizador puede, así, crear un objeto cuya clase herede a "TransacsBD", de modo que este posea funciones específicas para el

analizador que lo utiliza. Esta función será después utilizada por la función “analizar”, de modo que no será necesario que llamemos a la función “crearBD” en ninguna otra parte.

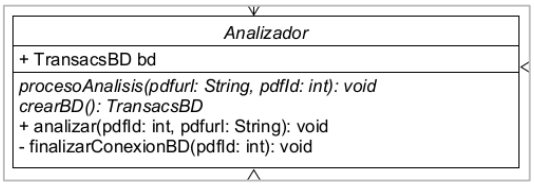

Figura 17. Estructura de la clase “Analizador”

¿Por qué llamamos a la función “crearBD” dentro de “analizar” en vez de hacerlo en la declaración de este atributo? Porque el constructor de “TransacsBD” comienza la conexión con la base de datos instantáneamente y, si instanciamos el objeto en el mismo momento en que lo declaramos, significaría que el propio constructor puede lanzar una excepción SQL, lo cual sería un problema para la clase “ConstructorAnalizador” que, recordemos, utiliza la clase “Supplier” para mantener su “HashMap” con todos los tipos de analizadores disponibles.

Atributos:

- “bd”: objeto que realizará las transacciones con la base de datos.

Funciones:

- “procesoAnalisis”: realiza los pasos necesarios para llevar a cabo el análisis de accesibilidad.
- “crearBD”: le da un valor al atributo “bd”.
- “analizar”: función pública, llamada por “EjecutorAnalisis”. Aquí se llama a las funciones “crearBD”, “procesoAnalisis” y “finalizarConexionBD”.
- “finalizarConexionBD”: aplica un *commit* a las transacciones realizadas. Irrelevante si dentro de “crearBD” especificamos que el objeto “bd” realice *commit* automáticamente.

4.3.3.6 La clase “AnalizadorWeb”

Recordemos que la clase “Analizador” es abstracta: cualquier nuevo analizador que se quiera implementar deberá heredarla. Dado que los dos analizadores que queremos añadir en este proyecto (Tingtun y PAVE) deben utilizar Selenium para navegar por los sitios web de sus respectivas herramientas, ambos necesitarán funciones y atributos en común. Además, un analizador no tiene por qué necesitar hacer uso de Selenium: perfectamente podríamos añadir, en un futuro, el código de un analizador que funcionara sin necesidad de herramientas web. Por estos motivos, creamos la clase “AnalizadorWeb”, que servirá como base para aquellos analizadores que deban acceder a sitios web o, dicho de otra forma, aquellos que deban hacer uso de Selenium.

Esta clase también es abstracta, y hereda a la clase “Analizador”. Proporcionará los objetos y funciones necesarios para usar la librería Selenium. Su atributo más importante es el controlador web, de clase “WebDriver”, proporcionado por Selenium. Es con este controlador con el que navegaremos por los sitios web y obtendremos los resultados que nos arrojen.

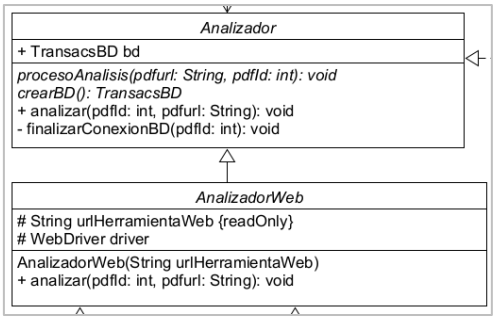

Figura 18. Estructura de la clase "AnalizadorWeb"

Atributos:

- "urlHerramientaWeb": URL del sitio web al que accederá para realizar los análisis. Deberá especificarse en el constructor de la clase.
- "driver": controlador web que utilizará para acceder a la herramienta.

Funciones:

- "AnalizadorWeb": constructor. Llama a la función "getDriverDisponible" de la clase "DriverPool" para asignarse un controlador.
- "analizar": modifica la misma función de la clase "Analizador", con la etiqueta *override*. Lo único que hace es llamar a la misma función de la clase padre y, después, llamar a "liberaDriver", de la clase "DriverPool".

4.3.3.7 La clase "TransacsBD"

Pasemos a tratar la comunicación con nuestra base de datos. Como adelantamos anteriormente, cada analizador necesitará realizar transacciones separadas entre sí con la base de datos. Gracias a las capacidades que nos ofrece PostgreSQL y su librería en lenguaje Java, podemos gestionar las transacciones fácilmente. Cada nueva comunicación que se entabla con la base de datos está separada de las demás, y es el propio gestor de la base de datos quien se encarga de seguir el principio ACID: garantizar la atomicidad, consistencia, aislamiento y durabilidad de las transacciones. Esto significa que no tenemos que preocuparnos de escenarios como el de insertar, en una tabla, dos filas al mismo tiempo, resultando en que ambas sean asignadas el mismo identificador; el gestor evita estos errores. Y puesto que todas nuestras transacciones consisten en insertar nuevos datos a la base de datos (salvo la consulta de la tabla de propiedades de Tingtun, pero tampoco supone ninguna complicación), sin otro detalle de importancia más allá de realizar estas inserciones concurrentemente (cosa que, como hemos dicho, nos asegura el gestor de la base de datos), no es necesario que nos preocupemos de más detalles técnicos.

Por lo tanto, encomendaremos la comunicación con el gestor de la base de datos a esta clase, "TransacsBD". Como atributos, poseerá los valores necesarios para realizar la conexión, como el nombre de usuario, la IP, el puerto, etc. Estos datos los leerá de un fichero que deberá encontrarse en el mismo directorio que el programa, en formato PROPERTIES. Java nos proporciona una clase que nos permite leer fácilmente los ficheros en este formato (aunque tampoco tiene complejidad alguna: cada línea del fichero consiste en, primero, el nombre de una propiedad y, segundo, el valor que se le da a esa propiedad, ambos separados por el símbolo de igualdad). En cuanto a sus funciones, tendrá las necesarias para leer el fichero, iniciar la comunicación y finalizarla. Además de esas, destacamos dos funciones importantes: una para insertar, en la base de datos, el sitio web y otra para insertar el documento PDF analizado.

Por otro lado, existen dos detalles de diseño a destacar:

1. El constructor de esta clase inicia la conexión con la base de datos. Es decir, que al instanciar un nuevo objeto de "TransacsBD", se conecta directamente con la base de datos. Esto se ha realizado así con el único motivo de minimizar el código que debe escribirse dentro de los pasos que debe realizar un analizador, ya que, de todas formas, si se crea una instancia de "TransacsBD" no será por otro motivo más que para realizar transacciones con la base de datos.
2. El constructor recibe como parámetro de entrada un booleano para determinar si debe aplicarse el *commit* de las transacciones de forma automática. Esto es así para que, en el caso en que el análisis se vea interrumpido por cualquier motivo, se ofrezca la opción de guardar o no, en la base de datos, las transacciones que se hayan realizado previamente a la interrupción. Para entenderlo mejor, hay que tener en cuenta que los resultados del análisis de un solo documento PDF no se van recibiendo uno a uno, a lo largo del tiempo, sino que se reciben todos a la vez. Y que, además, los analizadores siempre deberán realizar las siguientes transacciones: primero, guardar el documento PDF (su URL) en la base de datos; después, guardar los resultados obtenidos de su análisis. Dicho esto, si ocurriera el caso en que el análisis de algún documento se viera interrumpido, muy probablemente el documento estaría guardado en la base de datos, pero no sus resultados, por lo que carecería de sentido mantener en la base de datos un documento sin un análisis de accesibilidad asociado. Por todo esto damos la opción de aplicar los *commit* automáticamente o manualmente.

"TransacsBD" será utilizada, además de por los analizadores, por la clase "RecolectorPDF", pues será esta la que inserte en la base de datos la URL del sitio web que esté siendo analizado. Tiene sentido que "RecolectorPDF" sea quien realice esta tarea, ya que es quien comienza los análisis de todos sus documentos PDF. En esta clase, a diferencia de los analizadores, el *commit* automático sí está activo, puesto que es la única transacción que realiza con la base de datos.

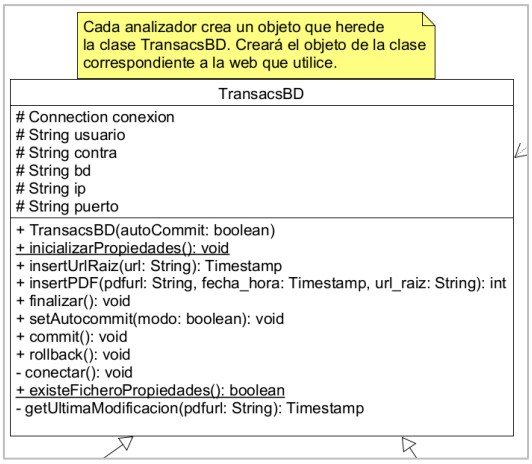

Figura 19. Diseño final de la clase "TransacsBD"

4.3.3.8 Las clases "TransacsBDTingtun" y "TransacsBDPAVE"

La clase "TransacsBD" no gestiona las transacciones de base de datos para el análisis mediante una herramienta web en concreto, sino que posee funciones generales que podrían necesitar cualquier tipo de analizador. Por ello, aquellos analizadores que necesiten transacciones concretas, deberán tener acceso a una clase que herede de "TransacsBD" y que posean las funciones necesarias.

Las clases “TransacsBDTingtun” y “TransacsBDPAVE” cumplen esta necesidad para los analizadores “AnalizadorTingtun” y “AnalizadorPAVE” respectivamente. Ambas clases poseen los nombres de las tablas de la base de datos a las que necesitan acceder, como atributos privados estáticos y constantes, para facilitar posibles modificaciones en el futuro. En cuanto a las funciones, ambos tienen en común una función destinada a almacenar los resultados de los análisis, pero cada una posee parámetros de entrada diferentes, al igual que las columnas de la tabla donde guardan los resultados. La única función que queda por mencionar es “getIdPropiedad”, de la clase “TransacsBDTingtun”, la cual sirve para obtener el identificador asociado al título de alguna de las propiedades que analiza la herramienta web.

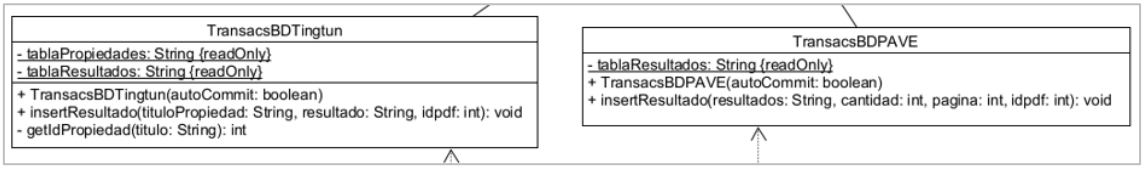

Figura 20. Diseño final de las clases "TransacsBDTingtun" y "TransacsBDPAVE"

4.3.3.9 La clase “ParamsEjecucion”

Por último, será necesario analizar los argumentos de entrada introducidos por el usuario. La sintaxis de estos argumentos debe ser la siguiente:

$$\text{url nombre } [-\text{x}_1\ \text{p}_1\ \text{p}_2\ \ldots\ \text{p}_n]\ [-\text{x}_n\ \text{p}_1\ \text{p}_2\ \ldots\ \text{p}_n]$$

Donde:
1. url es una cadena de texto con la URL de la web a analizar
2. $nombre$ es una cadena de texto que denominará al fichero $sitemap$ que se utilice
3. x es un caracter que representa una opción de ejecución
4. p es una cadena de texto que representa un valor

Ejemplo:
www.uah.es uah -a tingtun pave -n 3

Donde:
1. La opción a especifica los tipos de analizadores que se quieren utilizar
2. La opción n especifica el número de hilos

Ya que la sintaxis que pretendemos implementar para los argumentos de entrada es muy similar a la que usan la mayoría de programas, es muy probable que exista una librería que nos facilite este trabajo. En efecto, esta librería existe y se denomina Apache CLI[10]. Esta librería nos facilita el trabajo de análisis sintáctico de los argumentos de entrada, permitiéndonos implementar exactamente la misma sintaxis que hemos presentado. Lo único que nos queda por hacer es programar manualmente la comprobación de los valores que recibe cada opción, así como la asignación de valores predeterminados.

Todas estas tareas las realizamos dentro de una nueva clase, llamada “ParamsEjecucion”. Además, será esta clase la que guarde los valores de cada parámetro, de modo que estos valores estén accesibles por cualquier otra clase que los necesite. Por ejemplo, cuando la clase generadora del fichero *sitemap* necesite acceder al sitio web correspondiente, tan solo tendrá que consultar a la clase “ParamsEjecucion”. Por otra parte, como estos valores nunca se

[10] Sitio web oficial de la librería Apache CLI: https://commons.apache.org/proper/commons-cli/

modificarán durante la ejecución del programa, su visibilidad es privada para que solo puedan consultarse mediante funciones de tipo *get*. Así, nos aseguramos de que nunca los modificaremos por error.

4.3.3.10 Las clases "Log" y "CustomFormatter"

El objetivo de ambas clases es poder construir un fichero *log* durante la ejecución del programa, de modo que se reflejen los pasos realizados en cada momento y los errores que hayan podido surgir. La clase "Log" hace uso de la clase "Logger" facilitada por Java, se encarga de especificar la ruta donde se almacenará el fichero y de facilitar algunas funciones para registrar advertencias y errores en el *log*. La clase "CustomFormatter" se encarga del formato que tendrán los registros en el *log*.

4.3.3.11 Diagrama de clases final

En la figura Figura 21 podemos ver el diagrama de clases definitivo.

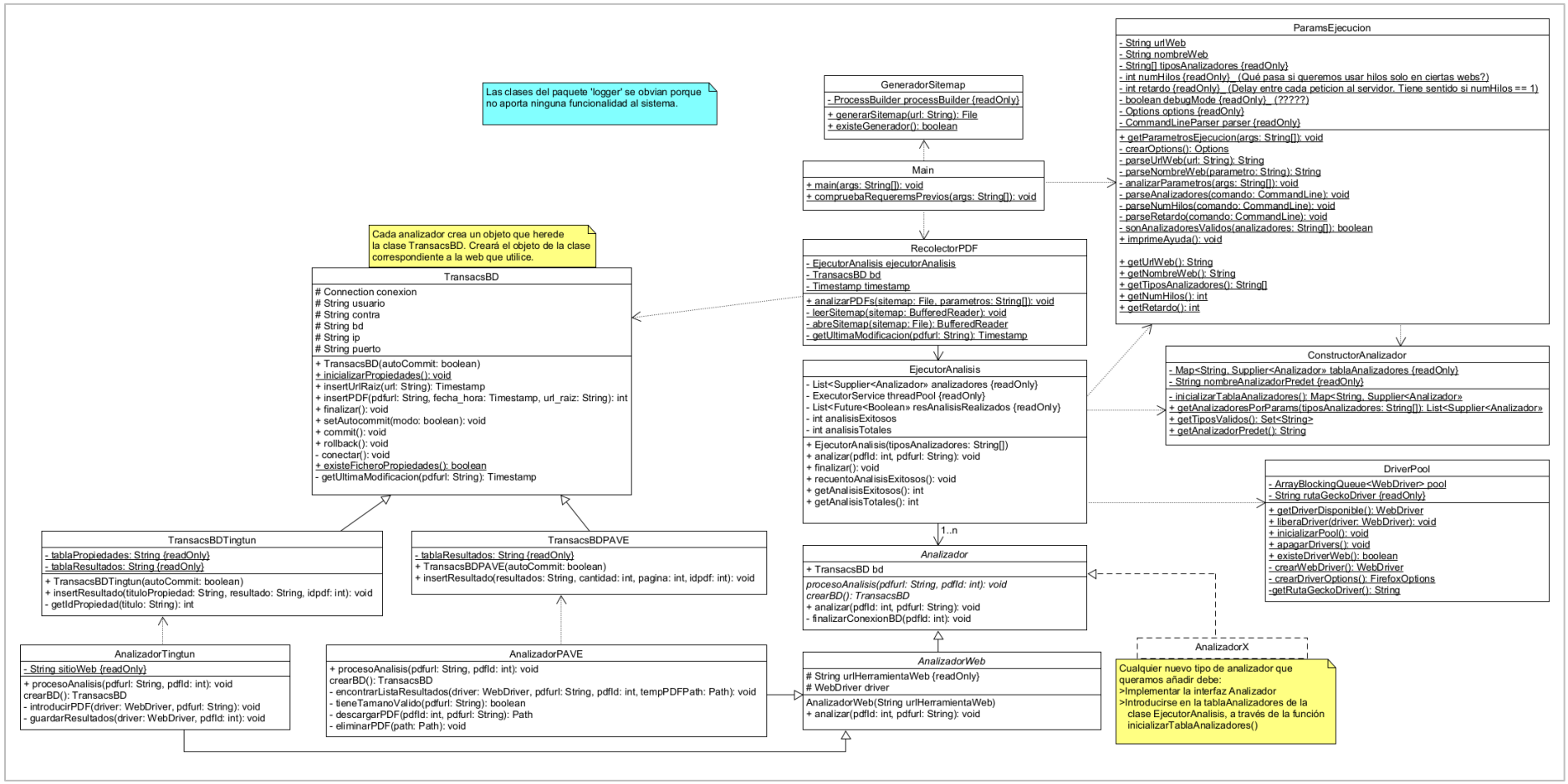

Figura 21. Diagrama de clases final

5 Detalles de desarrollo

En la sección 4 explicamos el papel de cada clase en nuestro programa, así como la estructura de nuestra base de datos. A continuación, detallaremos algunos aspectos del desarrollo, más concerniente al código del sistema.

5.1 Sitemap Generator

El programa Sitemap Generator tuvo que modificarse ligeramente para poder introducir la URL del sitio web del cual queríamos obtener el *sitemap*.

```
const SitemapGenerator = require('sitemap-generator');

// create generator
const generator = SitemapGenerator('http://example.com', {
  stripQuerystring: false
});

// register event listeners
generator.on('done', () => {
  // sitemaps created
});

// start the crawler
generator.start();
```

Figura 22. Código en JavaScript del fichero principal de Sitemap Generator

Como podemos comprobar en la Figura 22, la URL está *hard-coded* en el script principal. Para poder especificar la URL, basta con reemplazar "`'http://example.com'`" por "`process.argv[2]`". Así, el primer argumento de entrada (después de la llamada al script en sí mismo) será la URL del sitio web del que queremos obtener el *sitemap*.

Por último, para poder especificar la ruta de la carpeta donde queremos guardar los *sitemap* generados, tenemos que insertar una nueva opción dentro del creador del generador, es decir, adyacente a la línea "`stripQuerystring: false`". Esta opción es "`filepath: process.argv[3]`". La lista de opciones que podemos especificar en este script se puede encontrar en el repositorio del código fuente[11].

5.2 Selenium

La librería de Selenium para Java ofrece funciones que nos permiten interactuar con los elementos de un sitio web cualquiera. Por ejemplo, podemos buscar un botón en concreto mediante diversos métodos: identificándolo por su etiqueta HTML, su ID, su nombre de clase, usando un Selector CSS, etc. Y, después, realizar acciones sobre el mismo, como por ejemplo hacer clic. Evidentemente, estas acciones deben especificarse en el código y, para poder determinar el modo de acceder a los elementos que nos interesen, nos tenemos que apoyar en una capacidad muy común en los navegadores web: consultar el código HTML y CSS del sitio web al que estamos accediendo. Por ejemplo, con el navegador Firefox bastaría con hacer clic derecho en cualquier espacio vacío del sitio web y seleccionar la opción "Inspeccionar". En ese momento, se nos abrirá un recuadro donde podemos consultar el código mencionado, así como modificarlo y ver el resultado de esa modificación.

Pero este proceso puede realizarse con mayor facilidad. Selenium ofrece una extensión para navegadores que permite registrar los elementos con los que interactúa el usuario. Esto facilita

[11] Enlace a la lista de opciones de Sitemap Generator: https://github.com/lgraubner/sitemap-generator#options

mucho codificar los pasos que debe realizar cada analizador de nuestro sistema. Por ejemplo, basta con activar esta extensión mientras interactuamos con Tingtun para que nos genere el código Java correspondiente. Esta extensión sirve para automatizar el acceso al sitio web, insertar el documento que queremos analizar y ejecutar su análisis, pero, para recoger los resultados, no queda otra opción que hacerlo manualmente.

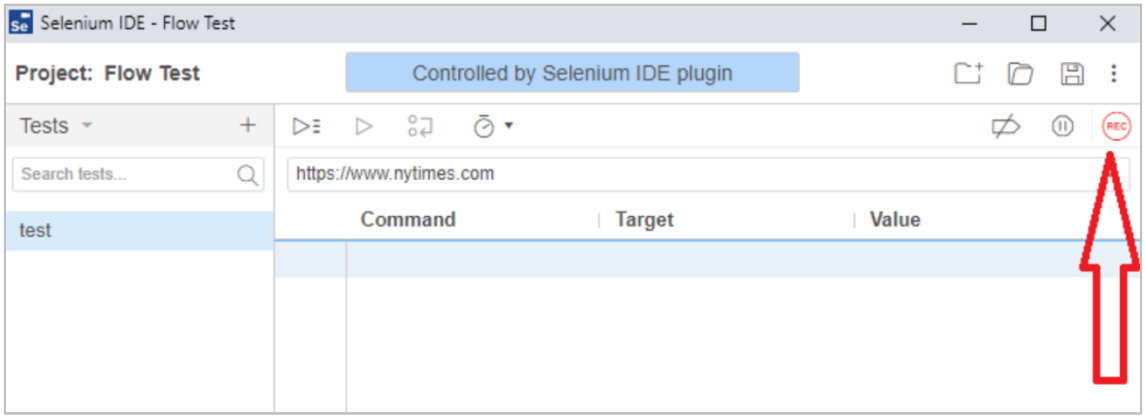

Figura 23. Interfaz gráfica de la extensión de Selenium para navegadores web

Como podemos ver en la Figura 23, basta con crear un nuevo test, especificar la URL del sitio web al que queremos acceder, y pulsar el icono rojo con el texto “REC”, el cual comienza la grabación de las interacciones que realicemos durante nuestra navegación. Al finalizar, nos permitirá obtener el código correspondiente en el lenguaje que elijamos, dentro de las opciones que ofrece.

5.2.1 Problemas de Selenium

Selenium es una herramienta muy potente, pero existen dos problemas a tener en cuenta al utilizarlo.

Primero, buscar los elementos de interés del sitio web al que estemos accediendo puede tomar mucho tiempo. Por ejemplo, el simple hecho de comprobar la existencia de un botón puede demorarse más tiempo del que se esperaría, a pesar de que el sitio web haya cargado rápidamente. Este problema se acentúa especialmente al realizar pruebas con *sitemaps* con gran cantidad de documentos. Investigando, descubrimos que las funciones básicas que se utilizan para acceder a determinados elementos del sitio web, es decir, “findElement” y “findElements”, realizan una espera implícita. ¿Qué quiere decir esto? La librería de Selenium te permite realizar esperas explícitas para que, al cumplirse cierta condición, se acceda al elemento web correspondiente. Si no se encuentra el elemento y un tiempo límite de espera, especificado por el programador, se agota, entonces se lanzará una excepción. Si en lugar de realizar estas esperas explícitas utilizamos, directamente, “findElement” o “findElements”, estas funciones esperarán implícitamente. Por lo visto, la espera implícita es notablemente más ineficiente que la espera explícita, por lo que tuvimos que refactorizar el código. Gracias a estos cambios, hemos podido experimentar un ahorro de tiempo de hasta 1 minuto, para documentos PDF individuales, aunque la diferencia iba variando en magnitud.

En segundo lugar, aunque esto no es un problema propio de Selenium, es que, al intentar recoger los resultados del sitio web de PAVE, a veces se lanza una excepción de clase “StaleElementReferenceException”. Esta excepción puede ser lanzada por dos posibles motivos:

1. El elemento web, al que estamos intentando acceder, deja de existir. Esto puede ocurrir por diversos motivos, ya sea porque la página web se actualizó, o porque se modificó el elemento al que queríamos acceder, porque cambiamos de página web, etc.
2. El elemento web ya no se encuentra dentro del DOM. El término DOM es un acrónimo de *Document Object Model*, un estándar de W3C que especifica el procedimiento para acceder a un documento. Existen tres especificaciones: Core DOM, para todo tipo de documentos; XML DOM, para ficheros en formato XML; y HTML DOM, para ficheros en formato HTML. Los sitios web usan el HTML DOM para especificar cómo se gestionan los elementos que contienen. A través de este estándar, las librerías adecuadas, como Selenium, son capaces de acceder a los elementos web de la página por la que naveguemos. Si un elemento web desaparece del DOM, no podremos acceder a su información.

Las soluciones que se proponen ante este problema son muy básicas: recargar la página; intentar acceder varias veces al elemento en cuestión, mediante el uso de bloques *try-catch*; o que la condición por la que esperemos acceder al elemento sea que haya sido recientemente actualizado dentro del DOM. Probamos las tres opciones, pero ninguna parece solucionar por completo este problema, ya que sigue ocurriendo de vez en cuando.

6 Pruebas

6.1 Ejecución del programa y problemas presentados

Antes de empezar con las pruebas sobre sitios web, vamos a mostrar brevemente qué ocurre cuando introducimos argumentos de entrada erróneos.

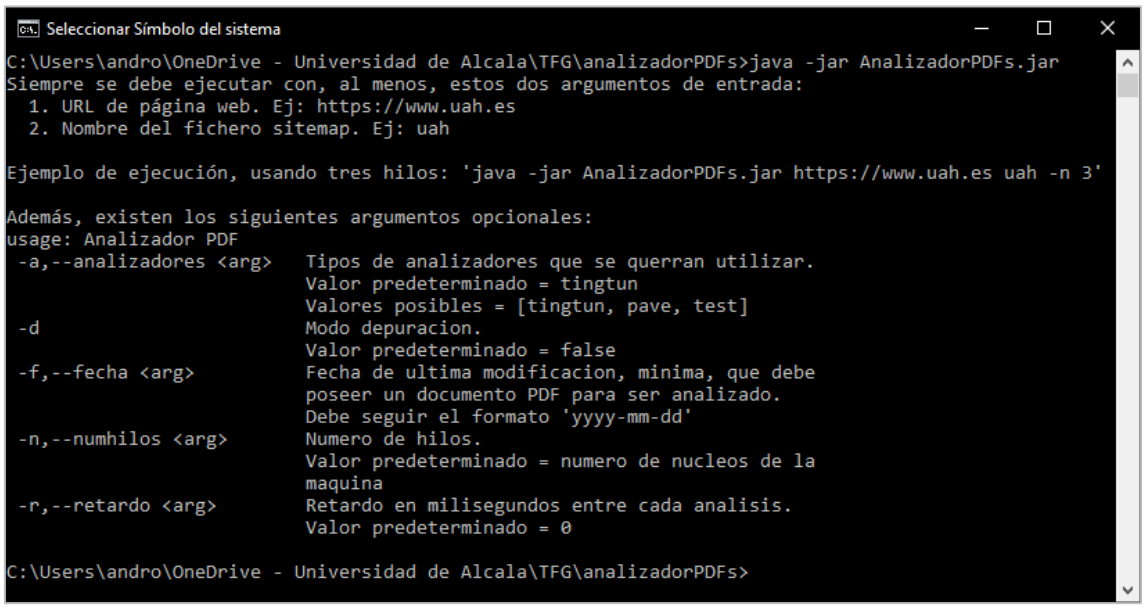

Figura 24. Respuesta del programa al ejecutarlo sin introducir un solo argumento de entrada

Como vemos en la Figura 24, al ejecutar el programa sin especificar ni un solo argumento de entrada, este nos muestra una explicación de la sintaxis que deben seguir los argumentos para que el programa pueda funcionar. También nos indica las opciones que tenemos disponibles para la ejecución. Cabe destacar que las opciones "-d" y "-r" fueron planteadas en un principio, pero no se llegaron a implementar por no considerarse, al menos de momento, necesarias. Se pretendía hacer una distinción entre un modo de depuración y un modo de ejecución normal, donde en uno se muestran detalles de los errores, mientras que en el otro solo la información mínima de su ejecución. El retardo fue pensado por la posibilidad de que ciertas herramientas web pudieran limitar su uso si detectaban un gran número de solicitudes, en un pequeño margen de tiempo, provenientes de una misma dirección IP, pero este no fue el caso con Tingtun y PAVE.

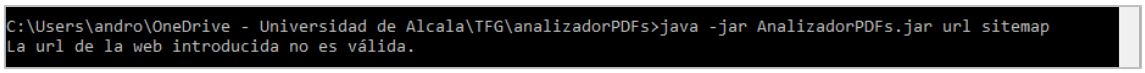

Figura 25. Error del programa al introducir una URL inválida

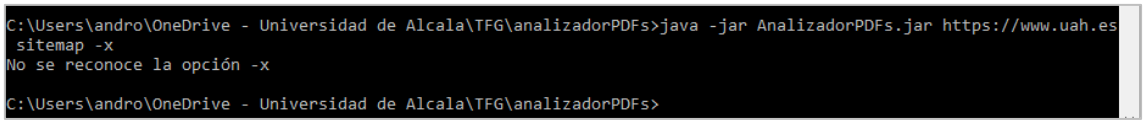

Figura 26. Error del programa al introducir una opción inexistente

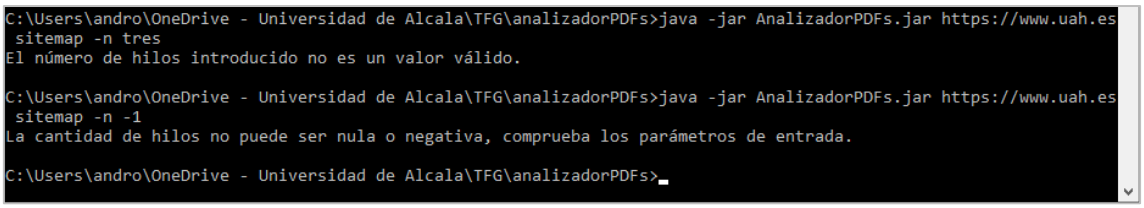

Figura 27. Errores del programa al introducir valores inválidos en la cantidad de hilos

En las figuras Figura 25, Figura 26 y Figura 27 podemos ver algunos ejemplos más de la respuesta del programa al introducir argumentos erróneos. En definitiva, el programa no comenzará si existe algún parámetro que debe ser corregido.

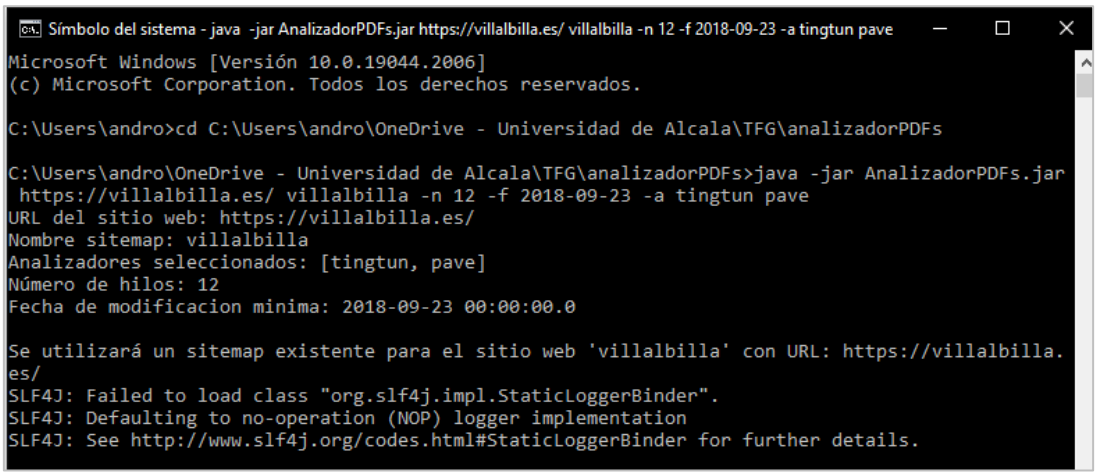

Figura 28. Captura de una de las ejecuciones realizadas

Como se puede ver en la Figura 28, el programa nos indica la configuración seleccionada. Además, vemos que, como ya se generó previamente un *sitemap* para el sitio web oficial del ayuntamiento de Villalbilla, y como el nombre que le hemos querido dar al *sitemap* coincide con el ya generado, nuestro programa se ha saltado la generación del fichero para pasar, directamente, a analizar los documentos PDF que contiene.

Procedamos ahora a explicar los mensajes que podemos recibir, tanto en la consola como en los *logs*, al ejecutar nuestro programa.

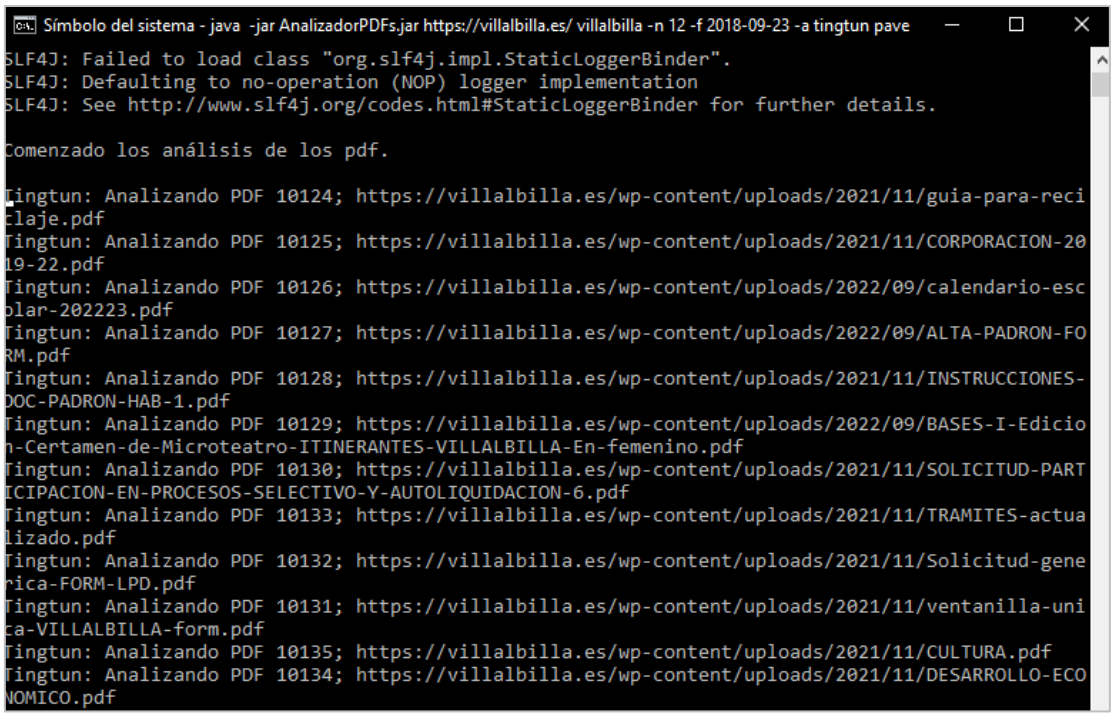

Figura 29. Comienzo de los análisis de los primeros 12 documentos PDF

En la Figura 29 vemos cómo el programa nos va a visando de los análisis que se están realizando en cada momento, indicándonos el analizador, el ID del documento PDF en la base de datos, y la URL de este. Los errores que vayan surgiendo también son mostrados en la consola. Cuando un documento PDF no consigue ser analizado, por el motivo que sea, el programa seguirá funcionando siempre que pueda. Es decir, que un documento sin analizar no impide el análisis de los demás.

Según se va ejecutando el programa, también se va generando un fichero *log* con las acciones que va realizando en cada momento. Así, podemos ver la hora exacta en la que comienza la ejecución, en la que termina, en la que ocurren los posibles errores, etc.

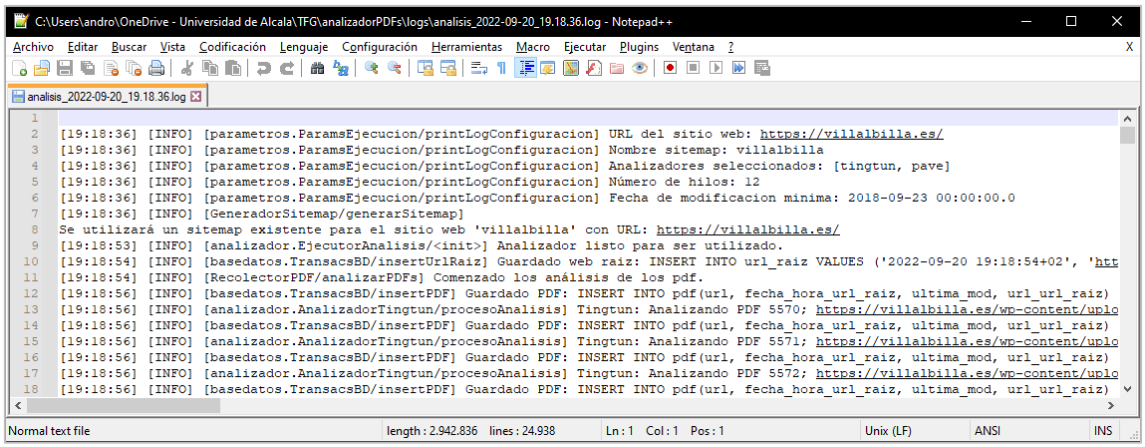

Figura 30. Ejemplo de fichero log

Al comenzar a ejecutar estas pruebas, descubrimos que existían ciertos escenarios donde nuestro programa no era capaz de analizar muchos documentos PDF. El problema principal fue no configurar correctamente los pasos que debía realizar, con ayuda de Selenium, al interactuar con los analizadores web. El sitio web que más problemas causó en este aspecto fue PAVE, ya que su interfaz gráfica de usuario consta de una única página web, donde dentro se organizan

los distintos menús, accesibles a través del uso del puntero, requiriendo abrir muchos desplegables, especialmente a la hora de recoger los resultados de los análisis. Esto provocaba que, si no se medían bien los tiempos de espera por las transiciones de cada acción, el programa se quedara atascado hasta un límite de tiempo máximo, antes de desistir en el intento de recoger los resultados. Sí, Selenium permite que el controlador web espere hasta que cierto elemento web sea visible pero, teniendo en cuenta que el sitio web de PAVE suele dejar de funcionar con frecuencia (descubierto, también, a base de probar con muchas ejecuciones del programa), dejar un margen de tiempo demasiado grande provoca una ralentización innecesaria del programa, y más teniendo en cuenta que, si el servidor web está caído, ninguno de los documentos PDF restantes podrán ser analizados, implicando una pérdida de tiempo enorme.

Otro problema, inherente a la utilización de herramientas web que no son propias, es que a veces estas dejan de funcionar por motivos que no están a nuestro alcance. A esto le tenemos que sumar que Tingtun limita la cantidad de documentos analizables por día a unos pocos cientos documentos, no sabemos exactamente cuántos.

```
11384 [15:18:24] [INFO] [analizador.AnalizadorTingtun/procesoAnalisis] Tingtun: Analizando PDF 6723; https://www.inclusion.gob.es/oberaxe/fich
11385 [15:18:25] [INFO] [basedatos.TransacsBDTingtun/insertResultado] PDF 6714 APROBADO - Propiedad 8
11386 [15:18:27] [ERROR]
11387 org.openqa.selenium.TimeoutException: Expected condition failed: waiting for visibility of element located by By.cssSelector: ul[id='rst
11388 Build info: version: '4.2.1', revision: 'ac4d0fdd4a'
11389 System info: host: 'ADLEYMD', ip: '192.168.56.1', os.name: 'Windows 10', os.arch: 'amd64', os.version: '10.0', java.version: '17.0.3.1'
11390 Driver info: org.openqa.selenium.firefox.FirefoxDriver
11391 Capabilities {acceptInsecureCerts: true, browserName: firefox, browserVersion: 104.0.2, moz:accessibilityChecks: false, moz:buildID: 202
11392 Session ID: dcefec3f-ff89-47c2-9afb-e6cb563a7c71
11393     at org.openqa.selenium.support.ui.WebDriverWait.timeoutException(WebDriverWait.java:87)
11394     at org.openqa.selenium.support.ui.FluentWait.until(FluentWait.java:231)
11395     at analizador.AnalizadorTingtun.guardarResultados(AnalizadorTingtun.java:79)
11396     at analizador.AnalizadorTingtun.procesoAnalisis(AnalizadorTingtun.java:41)
11397     at analizador.Analizador.analizar(Analizador.java:16)
11398     at analizador.AnalizadorWeb.analizar(AnalizadorWeb.java:26)
11399     at analizador.EjecutorAnalisis.lambda$analizar$0(EjecutorAnalisis.java:48)
11400     at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
11401     at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)
11402     at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
11403     at java.base/java.lang.Thread.run(Thread.java:833)
```

Figura 31. Ejemplo de error recogido en un fichero log

Cuando la ejecución finaliza, vemos un mensaje final como el mostrado en la Figura 23. Ahí nos indica el número de documentos PDF que se han intentado analizar (sin contar aquellos cuya fecha de última modificación no se encuentra en el margen seleccionado), y el número de aquellos que no pudieron ser completamente analizados (es decir, aquellos que no pudieron ser analizados por, al menos, uno de los analizadores).

```
9392
9393 [15:00:25] [INFO] [analizador.AnalizadorPAVE/procesoAnalisis] ÉXITO PAVE: PDF: 312;
9394 [15:11:33] [INFO] [analizador.AnalizadorPAVE/procesoAnalisis] ÉXITO PAVE: PDF: 345;
9395 [15:11:34] [INFO] [RecolectorPDF/leerSitemap]
9396 >>> Fin de los análisis. PDF totales: 379. PDF sin analizar por completo: 133 <<<
```

Figura 32. Mensaje final después de terminar todos los análisis. El último mensaje se muestra tanto en los logs como en la consola

6.2 Plan de pruebas del sistema

Hablemos ahora de las pruebas realizadas. Hemos ejecutado el programa con distintos sitios web, los cuales alojan, a su vez, distintas cantidades de documentos PDF. Esta es la lista de los sitios web:

- Ayuntamiento pequeño: https://villalbilla.es/
- Ayuntamiento medio: https://www.ayto-alcaladehenares.es/
- Comunidad Autónoma: https://www.madrid.es/
- Ministerio: https://www.inclusion.gob.es/
- JRC: https://joint-research-centre.ec.europa.eu/

El sitio web de la Comunidad de Madrid no pudo analizarse, debido a que el script que estamos usando, Sitemap Generator, no fue capaz de generar el *sitemap*. Por lo visto, Sitemap Generator no es capaz de generar *sitemaps* para algunos sitios web. Este problema lleva reportado por diferentes usuarios, en el repositorio oficial del programa, desde el año 2020, pero es evidente que no ha sido solucionado. Se desconoce el motivo del error.

Para la ejecución de las pruebas, hemos utilizado la misma configuración con cada sitio web:

- Número de hilos: 12 (el doble de unidades lógicas de procesamiento de la máquina)
- Fecha de modificación mínima: 23 de septiembre de 2018
- Analizadores seleccionados: Tingtun y PAVE

La fecha de modificación mínima seleccionada, el 23 de septiembre de 2018, corresponde a la fecha mínima a partir de la cual todos los documentos PDF, creados o modificados con posterioridad, deben cumplir con los criterios de accesibilidad, según el Real Decreto [1].

A continuación, mostramos la tabla con el tiempo que se ha necesitado para generar los ficheros *sitemap* de cada sitio web probado, así como el número de documentos PDF obtenidos:

Sitio web	**Cantidad de documentos**	**Tiempo en generar *sitemap* (minutos)**
Ayuntamiento Villalbilla	379	13
Ayuntamiento Alcalá de Henares	566	27
Ministerio de Inclusión	1398	24
Joint Research Centre	1123	138

Debido al límite de documentos que nos impone Tingtun, y a la frecuencia con la que PAVE deja temporalmente de funcionar, nos ha resultado imposible conseguir análisis completos de la mayoría de los sitios web con los que hemos probado. Solamente el sitio web del ayuntamiento de Villalbilla pudo ser analizado por completo, aunque cerca de un 35% de los documentos no pudieron ser totalmente analizados. La gran mayoría de los errores ocurrían por esperar demasiado tiempo a que cargara un elemento web de la herramienta, de modo que se pudiera acceder a los resultados. Pero el motivo concreto de llegar a esa situación es todavía desconocido, porque muchos documentos, que no eran analizados debido a eso, eran de poco tamaño, tanto en número de páginas (menos de 5), como en espacio que ocupaban (no llegaban ni a 1 MB).

Dicho esto, vamos a indicar algunos datos que hemos podido recopilar de las ejecuciones que hemos realizado:

- Web del ayuntamiento de Villalbilla: en una ejecución, tardó 2 horas y 15 minutos en analizar 379 documentos, de los cuales 133 no fueron completamente analizados, 75 no lo fueron por PAVE y 83 por Tingtun. En otra, 1 hora y 20 minutos, con 289 sin analizar completamente, 292 no lo fueron por PAVE y 154 por Tingtun. Si solo tomamos en

cuenta la primera ejecución, ya que tiene mejores resultados, el promedio de la velocidad de análisis es de 2,8 documentos analizados por minuto.

- Web del ayuntamiento de Alcalá de Henares: en una ejecución, tardó 1 hora y 50 minutos en analizar 566 documentos, de los cuales 545 no fueron analizados por PAVE y 301 por Tingtun. La web de PAVE dejó de funcionar poco después de comenzar, y Tingtun nos limitó su uso después de unos 260 documentos analizados. Si contamos el tiempo que dedicó a analizar los primeros 50 documentos, antes de que los errores fueran frecuentes, el programa analizó unos 4,16 documentos por minuto.
- Web del Ministerio de Inclusión: en una ejecución que no llegó a terminar por completo, analizó, con Tingtun, 563 documentos en 2 horas y 8 minutos. PAVE no funcionaba en ese momento, por lo que no analizó ni un solo documentos. Resulta en un promedio de unos 4,4 documentos por minuto.
- Web de Joint Research Centre: en una ejecución que no llegó a terminar por completo, por motivos similares a los ocurridos con la web del ministerio, analizó, tomando en cuenta solo los primeros 50 documentos, 24 documentos con Tingtun y 45 con PAVE. Tardó 10 minutos en hacerlo, por lo que analizó un promedio de 5 documentos por minuto.

Los resultados van variando por cada página y en cada momento, debido a las disponibilidades de las herramientas de análisis y a la variedad en los tamaños de cada documento.

También realizamos una prueba para comparar la velocidad de ejecución del programa según la cantidad de hilos que creáramos para los análisis. Hemos analizado 55 documentos PDF del sitio web del ayuntamiento de Villalbilla, con tres cantidades de hilos diferentes. Recordemos que la máquina desde la que hemos arrancado estas pruebas posee 6 unidades lógicas de procesamiento. Estos fueron los resultados:

Nº de hilos	Tiempo de ejecución (minutos)	Documentos analizados por Tingtun	Documentos analizados por PAVE	Velocidad promedio (documentos/minuto)
6	48	51	52	1,14
12	23	55	45	2,39
18	26	55	49	2,2

6.3 Resultados obtenidos del análisis

Mostraremos algunos ejemplos de los resultados que podemos encontrar en la base de datos, después de finalizar los análisis. Si queremos conocer los resultados obtenidos durante uno de los análisis realizados a la web del ayuntamiento de Villalbilla, basta con consultar la base de datos a través de la URL de la web y la fecha del análisis, datos que se pueden encontrar al comienzo de los ficheros *log*. En la Figura 33 podemos ver algunos resultados del análisis, concretamente aquellos obtenidos por Tingtun. En la Figura 34 encontramos los resultados de los análisis realizados por PAVE, los cuales, a diferencia de Tingtun, te indican también en qué página se encuentra cada error, así como la cantidad de veces que aparecen esos errores.

En el fichero “ScriptsBD.sql”, ubicado en la carpeta “Scripts”, hay algunas propuestas de consultas que se podrían realizar a la base de datos. Un ejemplo de estos scripts lo podemos ver en la Figura 35, donde figuran todos los aprobados y suspensos, determinados por Tingtun, de las propiedades analizadas que están almacenadas en nuestra base de datos.

accesibilidad_pdf/postgres@PostgreSQL 14

Query Editor | Query History

```
SELECT resultado, id_pdf, titulo
FROM (res_tingtun
    INNER JOIN (url_raiz INNER JOIN pdf ON url_raiz.url = pdf.url_url_raiz AND url_raiz.fecha_hora = pdf.fecha_hora_url_raiz)
        ON res_tingtun.id_pdf = pdf.id)
        INNER JOIN props_tingtun ON id_propiedad_props_tingtun = id_propiedad
WHERE url_raiz.fecha_hora = '2022-09-20 19:18:54+02' AND url_raiz.url = 'https://villalbilla.es/'
GROUP BY resultado, id_pdf, titulo
ORDER BY id_pdf
```

Data Output | Explain | Messages | Notifications

	resultado character varying (30)	id_pdf integer	titulo character varying (35)
1	APROBADO	5570	Correct Tab and Reading Order
2	APROBADO	5570	Document Permissions
3	APROBADO	5570	Document Title
4	APROBADO	5570	Form Fields
5	APROBADO	5570	Scanned Document
6	APROBADO	5570	Structure Elements (tags)
7	SUSPENSO	5570	Bookmarks
8	SUSPENSO	5570	Natural Language
9	APROBADO	5571	Correct Tab and Reading Order
10	APROBADO	5571	Document Permissions
11	APROBADO	5571	Document Title
12	APROBADO	5571	Scanned Document
13	APROBADO	5571	Structure Elements (tags)
14	SUSPENSO	5571	Bookmarks
15	SUSPENSO	5571	Natural Language
16	APROBADO	5572	Alternative Text for Images
17	APROBADO	5572	Correct Tab and Reading Order
18	APROBADO	5572	Document Permissions
19	APROBADO	5572	Document Title
20	APROBADO	5572	Heading Levels
21	APROBADO	5572	Scanned Document
22	APROBADO	5572	Structure Elements (tags)
23	APROBADO	5572	Table Elements
24	SUSPENSO	5572	Bookmarks

Figura 33. Ejemplo de resultados de Tingtun obtenidos al analizar el sitio web del ayuntamiento de Villalbilla

accesibilidad_pdf/postgres@PostgreSQL 14

Query Editor | Query History

```
SELECT resultado, cantidad, pagina, id_pdf FROM res_pave INNER JOIN
    (url_raiz INNER JOIN pdf ON url_raiz.url = pdf.url_url_raiz AND url_raiz.fecha_hora = pdf.fecha_hora_url_raiz)
        ON res_pave.id_pdf = pdf.id
WHERE url_raiz.fecha_hora = '2022-09-20 19:18:54+02' AND url_raiz.url = 'https://villalbilla.es/'
ORDER BY id_pdf
```

Data Output | Explain | Messages | Notifications

	resultado character varying (500)	cantidad smallint	pagina smallint	id_pdf integer
1	Illegal tab order	4	4	5574
2	Illegal tab order	3	3	5574
3	Some content is not marked.	73	3	5574
4	Illegal tab order	2	2	5574
5	Some content is not marked.	50	2	5574
6	Illegal tab order	1	1	5574
7	Some content is not marked.	24	1	5574
8	Illegal tab order	4	5	5574
9	Some content is not marked.	108	5	5574
10	Some content is not marked.	96	4	5574
11	Missing link alternate description	3	6	5575
12	Illegal tab order	1	2	5577
13	Some content is not marked.	104	2	5577
14	Missing link alternate description	3	2	5579
15	Structure element is incorrectly nested: Sect in Sect	7	2	5579
16	Missing link alternate description	2	1	5579
17	Figure has no alternative text	3	1	5579
18	Structure element is incorrectly nested: Sect in Sect	3	1	5579
19	Figure has no alternative text	7	2	5579
20	Missing link alternate description	5	1	5580
21	Structure element is incorrectly nested: Sect in Sect	3	1	5580
22	Figure has no alternative text	4	1	5580
23	Structure element is incorrectly nested: Sect in Sect	7	2	5581
24	Figure has no alternative text	24	2	5581
25	Structure element is incorrectly nested: Sect in Sect	4	1	5581
26	Figure has no alternative text	13	1	5581

Figura 34. Ejemplo de resultados de PAVE obtenidos al analizar el sitio web del ayuntamiento de Villalbilla

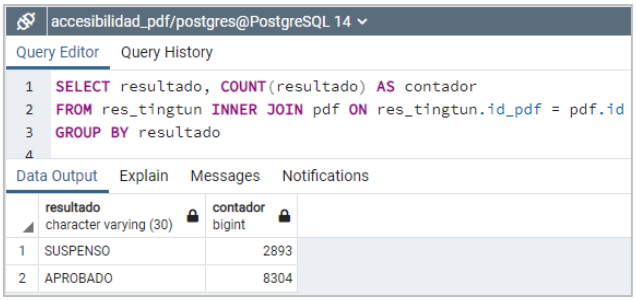

Figura 35. Ejemplo de consulta a la base de datos donde vemos todos los aprobados y suspensos de Tingtun

Vamos a comprobar ahora cuántos documentos carecen de algunas propiedades de accesibilidad, según los análisis que hemos podido realizar. Para los análisis realizados con Tingtun, con la consulta de la Figura 36 podemos obtener la cantidad exacta de esos documentos, y con la consulta de la Figura 37 podemos conocer, por cada propiedad de accesibilidad, la cantidad de documentos que las incumplen. Para los análisis de PAVE, en cambio, usaremos la consulta de la Figura 38 para conocer el número de documentos con problemas, y la consulta de la Figura 39 para ver qué propiedades se incumplen. Cabe destacar que, en el caso de PAVE, no podemos conocer exactamente la cantidad de documentos que considera completamente accesibles, ya que solamente se guardan los resultados de aquellos que sí tengan carencias.

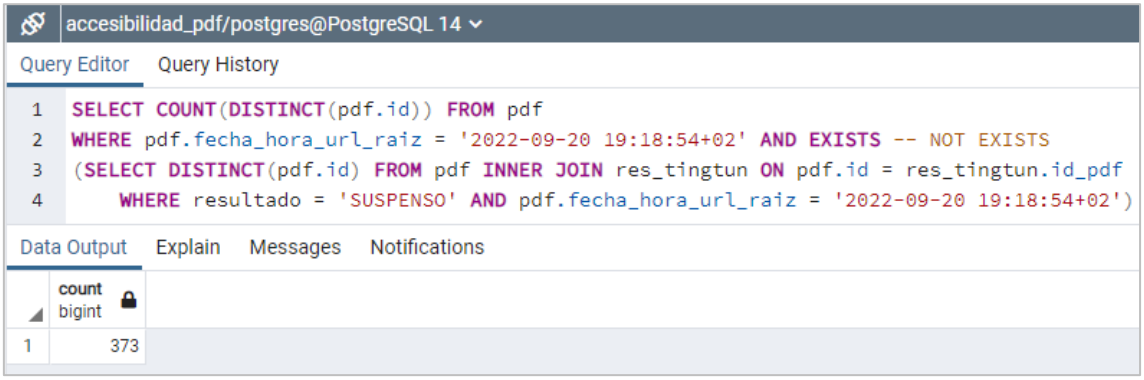

Figura 36. Consulta de los resultados de Tingtun que cuenta la cantidad de documentos que tienen algún error

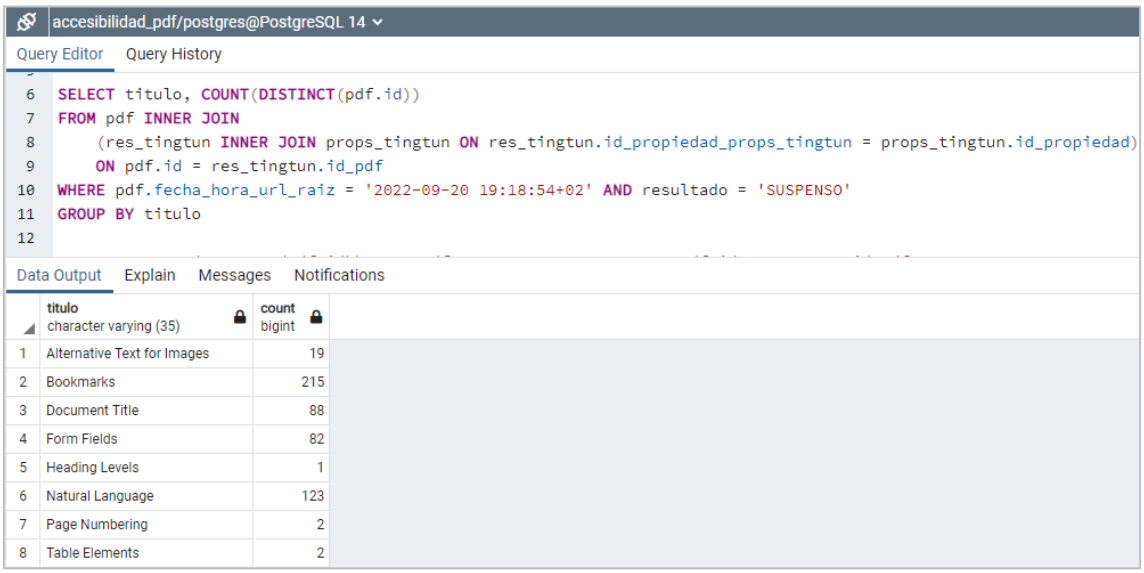

Figura 37. Consulta de los resultados de Tingtun que cuenta la cantidad de documentos que carecen de cada propiedad de accesibilidad

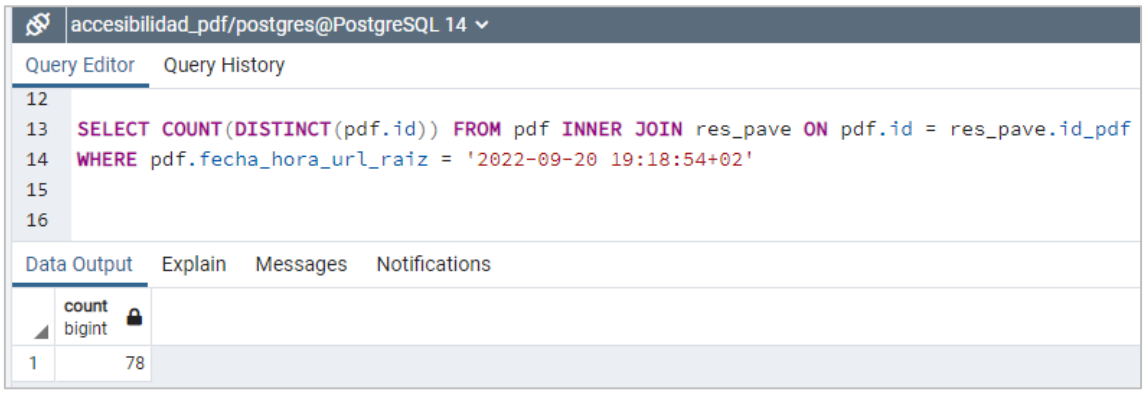

Figura 38. Consulta de los resultados de PAVE que cuenta la cantidad de documentos que tienen algún error

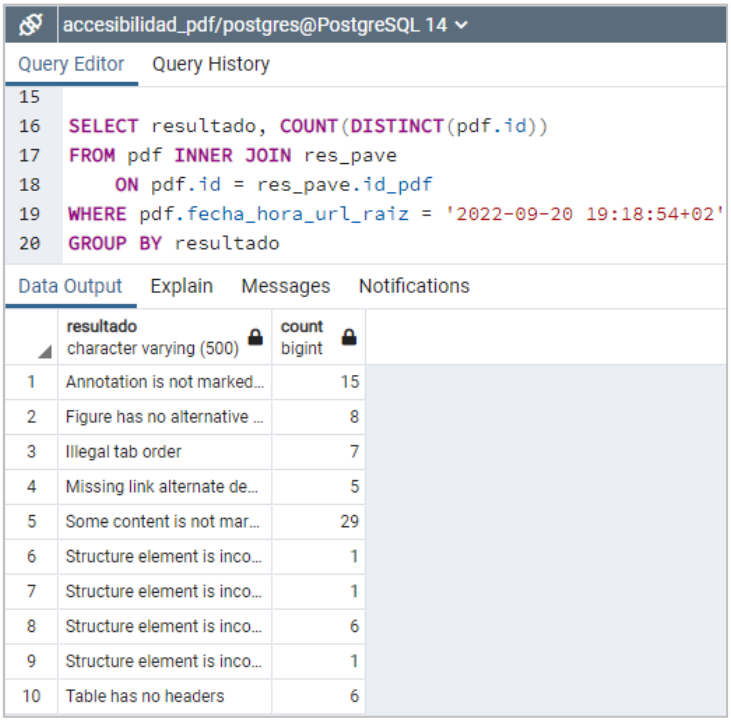

Figura 39. Consulta de los resultados de PAVE que cuenta la cantidad de documentos que carecen de cada propiedad de accesibilidad

Tabla con los resultados de Tingtun (los porcentajes de la columna más a la derecha se calculan con respecto al número de documentos a los que se les han detectado problemas):

Sitio web	**Nº documentos**	**Documentos CON problemas**	**Propiedad de accesibilidad**	**Documentos que incumplen la propiedad**
Ayuntamiento de Villalbilla	373	100 %	Alternative Text for Images	19 (5%)
			Bookmarks	215 (58%)
			Correct Tab and Reading Order	0
			Decorative Images	0
			Document Permissions	0
			Document Title	88 (24%)
			Form Fields	82 (22%)
			Heading Levels	1 (0,2%)
			Natural Language	123 (33%)
			Page Numbering	2 (0,5%)
			Scanned Document	0
			Structure Elements (tags)	0
			Submit Buttons	0
			Table Elements	2 (0,5%)

Tabla con los resultados de PAVE (los porcentajes de la columna más a la derecha se calculan con respecto al número de documentos a los que se les han detectado problemas):

Sitio web	**Nº documentos**	**Documentos CON problemas**	**Propiedad de accesibilidad**	**Documentos que incumplen la propiedad**
Ayuntamiento de Villalbilla	373	78 (21 %)	Annotation is not marked. Type: Widget	15 (19%)
			Figure has no alternative text	8 (10%)
			Illegal tab order	7 (9%)
			Missing link alternate description	5 (6%)
			Some content is not marked.	29 (37%)
			Structure element is incorrectly nested: Div in Span	1 (1%)
			Structure element is incorrectly nested: L in LI	1 (1%)
			Structure element is incorrectly nested: Sect in Sect	6 (8%)
			Structure element is incorrectly nested: Span in Lbl	1 (1%)
			Table has no headers	6 (8%)

En el caso de Tingtun, las propiedades que menos se cumplen son las siguientes:

1. *Bookmarks*, 58%: un PDF debe poseer marcadores para facilitar la navegación por el documento.
2. *Natural Language*, 33%: debe especificarse el lenguaje del documento, de modo que los lectores de pantalla puedan utilizar las voces del idioma correspondiente.

3. *Document Title*, 24%: el documento debe tener un título que refleje la temática de este; nos referimos al título que tiene como propiedad todo documento PDF, no a un título escrito dentro de sus páginas.
4. *Form Fields*, 22%: los campos de formularios, donde el usuario puede introducir datos, deben estar correctamente etiquetados.

La propiedad que menos se cumple, con diferencia, es la adición de marcapáginas al documento, aunque cabe destacar que Tingtun determina esta propiedad de un modo diferente a como lo hace Adobe Acrobat Reader. Para Tingtun, todo documento debe tener marcapáginas, sin excepción; mientras que, para Adobe, solamente deben poseerlos aquellos con una extensión de 21 o más páginas. Quizá eso explique, en cierta medida, que tantos documentos incumplan esta propiedad. Las siguientes dos propiedades más incumplidas están relacionadas con la falta de metadatos, y la tercera con la falta de etiquetación de los elementos, concretamente de los formularios. Las demás propiedades se incumplen con un 5% o menos de frecuencia. Aquellas con 0% de incumplimiento pueden

En el caso de PAVE, estas son las que menos se cumplen:

1. *Some content is not marked*, 37% (los elementos del documento deben estar todos debidamente etiquetados, para que el lector de pantalla pueda leerlos correctamente)
2. *Annotation is not marked*. *Type: Widget*, 19% (hay anotaciones de tipo widget en el documento que no están etiquetadas)

Las dos propiedades que menos se cumplen están relacionadas con la falta de etiquetación de los elementos del documento, coincidiendo con la cuarta propiedad que menos se cumple según Tingtun. Las demás propiedades se incumplen con un 10% o menos de frecuencia, relacionadas con la falta de texto alternativo, mala organización de los elementos del documento o falta de cabecera en las tablas.

7 Conclusiones y trabajos futuros

7.1 Conclusiones

Para este proyecto ha sido necesario conocer el estándar que especifica el formato PDF, los estándares que buscan garantizar su accesibilidad, la existencia de los ficheros *sitemap* y las nociones básicas del *scraping*. También ha requerido poner a prueba las habilidades de diseño de software, así como la gestión de tareas concurrentes y el tratamiento de los casos excepcionales. Por otro lado, ha sido una sorpresa descubrir que no existen alternativas, gratuitas y de calidad, al analizador de accesibilidad proporcionado por Adobe, teniendo en cuenta que la accesibilidad está cobrando tanta importancia en los últimos años.

Inicialmente, la expectativa era que este proyecto no sería demasiado complicado. Al fin y al cabo, la obtención de los documentos PDF lo hacemos a través de un script desarrollado por un tercero, y los análisis de accesibilidad los realizan, también, otras herramientas web. Nuestra tarea principal consistía en conectar estas herramientas, en leer el fichero *sitemap*, analizar los documentos PDF que contuviera y guardar los resultados en una base de datos. A pesar de todo, diseñar la arquitectura del sistema ha requerido de más trabajo del esperado, especialmente por intentar mantener cierta calidad. De hecho, todavía existen aspectos mejorables del diseño.

En definitiva, este proyecto ha servido como demostración de que el diseño y desarrollo del software es una tarea compleja, incluso para proyectos pequeños como este. Merece la pena mencionar, también, lo útil que resulta generar un *log* con las acciones que va realizando el programa, pues facilita identificar el foco de los errores que puedan suceder, tanto para el desarrollador que quiera solucionarlos como para el usuario que necesite saber hasta dónde ha llegado la ejecución del programa.

7.1.1 Problemas inesperados

Durante el desarrollo del proyecto y su puesta a prueba, nos hemos ido enfrentando a diversos problemas. Pudimos resolver la mayoría de ellos, aunque algunos todavía deben ser revisados.

7.1.1.1 Sitemap Generator no funciona siempre

Cuando ejecutamos el script Sitemap Generator con algunos sitios web, no es capaz de construirnos el *sitemap*. Un ejemplo es la web www.madrid.es/portal/site/munimadrid, de la Comunidad de Madrid, del cual nos produce un *sitemap* cuyo único contenido es esa misma URL. Si utilizamos la web www.xml-sitemaps.com, como pretendíamos en nuestro anteproyecto, podemos obtener un *sitemap* de esa web, pero con un límite de 500 URL. Es posible generar el *sitemap* a través de esa herramienta web para, después, usarlo en nuestro programa, pero no es una solución óptima. Por desgracia, no hemos encontrado otra alternativa. Este error, propio de Sitemap Generator, lleva reportado en su repositorio de GitHub desde el 7 de marzo de 2020, y no se ven indicios de que vaya a ser solucionado próximamente.

7.1.1.2 Driver de Selenium

Uno de los primeros problemas con los que tuvimos que lidiar fue con la gestión del *driver* de Selenium. Recordemos que, durante la ejecución del programa, cada analizador crea una instancia del *driver* o controlador web, con el que navegamos por Tingtun y PAVE. Estos *drivers* deben ser explícitamente apagados desde el código porque, si no, continuarán funcionando, aunque el programa finalice. La pequeña dificultad que esto provocó es que, cuando el programa fallaba por el motivo que fuera, se iban amontonando los *drivers* hasta que podía darse el caso de que la máquina se quedara sin memoria. La solución obvia es procurar apagar los *drivers* cada

vez que ocurra alguna excepción. Otra solución que hemos adoptado, por asegurarnos doblemente, es ejecutar un comando que finalice los procesos correspondientes a los *drivers*, al final del programa.

7.1.1.3 Obtención de los resultados de PAVE

Cuando PAVE comienza a analizar los documentos PDF, te redirige directamente a la página donde te mostrará los resultados, aunque todavía no haya finalizado su análisis. Además, durante el análisis, la página va reescribiendo los resultados, hasta que termina. Al acabar, no hemos encontrado que el sitio web muestre ningún tipo de notificación que lo indique, es decir, que no podemos saber cuándo finaliza el análisis exactamente. La solución que hemos aplicado es, simplemente, esperar durante un tiempo fijo antes de consultar los resultados. Pero esto implica que, si la duración del análisis es menor al tiempo que esperamos, estaremos perdiendo tiempo; y si la duración es mayor, entonces consultaremos unos resultados incompletos. La variación del tiempo necesario para realizar los análisis está influida tanto por la velocidad en la conexión de la red, como por el tamaño (en bytes) de los documentos que analicemos, por lo que podríamos intentar variar el tiempo de espera según el tamaño. Pero incluso esta solución no es adecuada para buena parte de los documentos ya que, a veces, PAVE simplemente no obtiene resultados, por lo que desperdiciaríamos tiempo esperando por unos resultados que nunca llegarán.

7.1.1.4 Límite de uso de Tingtun y PAVE

Al poner a prueba el sistema con distintos sitios web, descubrimos que la herramienta Tingtun limita el número de documentos que se pueden analizar en un solo día, aunque no especifican la cantidad de exacta.

Por otro lado, a veces ambas herramientas dejan de funcionar por motivos ajenos a nosotros. Cuando esto ocurre, no podemos hacer nada al respecto más que esperar a que vuelvan a estar operativas. Una solución clara a este problema es usar un analizador de accesibilidad que se ejecute localmente, pero desconocemos de una herramienta así, que esté abierta al público y sea gratuita.

7.1.1.5 Problema con los certificados SSL

Un certificado SSL es una clave que permite autenticar la identidad de un sitio web. Se utiliza para que los datos que se transmiten entre el cliente y el servidor se mantengan confidenciales. En nuestro caso, surge un problema al automatizar la conexión con el sitio web del que queremos obtener los documentos PDF, puesto que Java debe tener acceso a los certificados SSL para comunicarse de forma segura y, si no los tiene, no se establecerá la conexión. Una solución sería descargar los certificados para cada sitio web que lo requiera, algo que desconoceremos hasta que no probemos nuestro programa con el sitio web respectivo. Otra solución es desactivar este requerimiento. Puesto que tan solo necesitamos acceder a los documentos PDF, ignorar los certificados SSL no supondría ningún riesgo, pues no estamos enviando o recibiendo datos sensibles. (A no ser que le de a alguien por suplantar la identidad del sitio web y nos envíe documentos PDF con contenido ilegal, aunque la probabilidad es baja…)

7.2 Trabajos futuros

Probablemente el trabajo que más aportaría a nuestro proyecto es desarrollar un analizador de accesibilidad de los documentos PDF propio. Es decir, un analizador que se pueda ejecutar de forma local, sin tener que depender de otras herramientas web. No solamente permitiría obtener resultados de análisis completos, sino que la ejecución del programa también sería

mucho más rápida. Además, no nos tendríamos que preocupar de su disponibilidad, como ocurre con Tingtun y PAVE.

Como mencionamos con anterioridad en esta memoria, hicimos un pequeño intento de programa, paralelamente al desarrollo de este proyecto, que cumpliera este objetivo. Sería necesario utilizar una librería que nos permitiera acceder al contenido de los documentos PDF, en nuestro caso escogimos PDFBox. Esta librería nos permitió verificar la existencia de algunos aspectos necesarios, como el título del documento o el lenguaje de su texto, de forma muy sencilla. Puede que una de las tareas más complejas sea revisar que todos los elementos del documento estén correctamente etiquetados, ya que requiere leer el cuerpo del documento, cuyo contenido está representado en un lenguaje poco amigable basado en etiquetas, y sería necesario consultar el estándar del formato PDF para conocer bien cómo se estructura y qué se debe analizar exactamente.

Por otra parte, también podríamos automatizar el análisis de accesibilidad de las páginas web. Es lo que planteamos realizar en nuestro anteproyecto pero, como dijimos, no pudimos llevar a cabo. El estándar WCAG 2.0 establece las pautas que debe seguir un sitio web para considerarse accesible, por lo que bastaría encontrar alguna herramienta que comprobara esas pautas. Si no existiera tal herramienta, o fuera incompleta, podríamos desarrollarlo nosotros. En caso de que sí existiera, faltaría automatizar el análisis de todas las subpáginas del dominio, similarmente a lo que hemos realizado en este proyecto. El sitio web oficial de la organización que desarrolló el estándar WCAG, World Wide Web Consortium (W3C), nos ofrece una lista de herramientas relacionadas con la accesibilidad web[12]. Entre esas herramientas, podemos encontrar una[13] que se encarga de analizar la accesibilidad de cualquier sitio web que le insertes, a través de su URL. Los resultados te indican qué elementos no cumplen los requisitos necesarios y por qué. Quizá sea una herramienta adecuada, pero recordemos que está alojada en la web y que puede dejar de funcionar en cualquier momento. Desarrollar un analizador propio siempre será la opción más completa y robusta.

También podríamos ampliar, todavía más, nuestra área de acción, y echar un vistazo al estado de la accesibilidad respecto a otros formatos de información, como por ejemplo vídeos o audios. De nuevo, W3C nos ofrece algunos recursos para garantizar la accesibilidad de otros medios de información[14]. Podemos encontrar ayudas como los subtítulos, lenguajes de signo, etc. Sabemos que la generación automática de subtítulos es un área sobre la que se está trabajando desde hace ya un tiempo, teniendo ejemplos de ello servicios web como YouTube. También podemos encontrar trabajos relacionados con el lenguaje de signos, como la aplicación Hand Talk[15], para Android, que permite traducir palabras transmitidas por voz a lenguaje de signos, utilizando técnicas de machine learning y un muñeco 3D para ello. En definitiva, puede ser un campo interesante de explorar y ver qué aportaciones se pueden realizar.

7.2.1 Mejoras posibles

Existen aspectos que todavía se pueden mejorar de nuestro programa, explicaremos aquellas que hemos identificado:

[12] Lista de herramientas de accesibilidad ofrecida por W3C: https://www.w3.org/WAI/ER/tools/
[13] Herramienta de análisis de accesibilidad de sitios web: https://www.accessibilitychecker.org/
[14] Lista de herramientas de accesibilidad en formatos multimedia: https://www.w3.org/WAI/media/av/
[15] Sitio web oficial de la aplicación Hand Talk: https://www.handtalk.me/en/

7.2.1.1 *Mejor* presentación de los análisis incompletos

Durante la ejecución del programa, es común que algunos documentos PDF no hayan sido completamente analizados, es decir, que algunos analizadores no hayan podido terminar su trabajo. Por ejemplo, a veces PAVE no es capaz de analizar ciertos documentos debido a que superan el tamaño máximo permitido (actualmente 5MB). A pesar de que se deje constancia de esta clase de errores, tanto en los *logs* como en la propia terminal desde donde se ejecute, cuando la cantidad de documentos que posee el *sitemap* es muy grande, se vuelve muy tedioso leer la nube de texto y discernir la información que es de interés. Además, actualmente los únicos datos de interés que aporta el programa con respecto a esto, al final de su ejecución, es la cantidad total de documentos que ha intentado analizar, y cuántos de ellos se quedaron a medias o, directamente, sin poder analizar.

7.2.1.2 *Seguimiento del estado de ejecución*

Cuando estamos analizando los documentos de un *sitemap* considerablemente amplio, el análisis de todos ellos puede llegar a tomar varias horas hasta que finalice. Vendría bien poder mostrar algún tipo de seguimiento, es decir, los documentos que lleva analizados, cuántos le falta por analizar, quizá una estimación temporal, etc.

7.2.1.3 *Poder cancelar el uso de algunos analizadores*

De nuevo en el escenario en que estemos analizando muchos documentos PDF, a veces puede ocurrir que uno de los analizadores web que estemos utilizando deje de funcionar, ya sea porque se haya alcanzado el límite de documentos que nos permite analizar o por algún error proveniente de sus servidores. En estos casos, el programa intenta constantemente utilizar el servicio web a pesar de no estar disponible, esperando una respuesta durante un tiempo límite, lo que provoca una pérdida de tiempo considerable. Por ello, podría ser útil permitir al usuario cancelar el uso de ese analizador para el resto de la ejecución.

7.2.1.4 *Detectar automáticamente los documentos analizados recientemente*

Imaginemos que ejecutamos el programa para un *sitemap* grande y que, al finalizar, un porcentaje cualquiera de documentos no han sido analizados, por el motivo que sea. Sería muy conveniente poder ejecutar de nuevo el programa y que éste, automáticamente, se salte el análisis de aquellos documentos que ya hayan sido analizados previamente. Se podría agregar una nueva opción a los argumentos de entrada, que reciba una fecha a partir de la cual deban haber sido analizados los documentos por última vez, para que sean ignorados solo aquellos que hayan sido analizados por los analizadores seleccionados en el momento de ejecución. En lugar de una fecha, se podría contemplar un margen tiempo en segundos, minutos, etc., desde la ejecución del programa.

7.2.1.5 *Buscar otra alternativa para generar sitemaps*

Como ya habíamos mencionado anteriormente, el Sitemap Generator que estamos utilizando no es completamente robusto, puesto que existen algunos sitios web para los que no es capaz de generar un *sitemap*. Tenemos cuatro opciones: esperar a que su desarrollador solucione el problema, cosa que no se espera que ocurra pronto debido a su inactividad; arreglar nosotros mismos el programa, puesto que el código está en JavaScript y está abierto a modificaciones, pero llevaría mucho tiempo aprender su funcionamiento y encontrar la solución; desarrollar uno desde cero, lo cual es incluso más costoso que la opción anterior; y, por último, buscar otra alternativa ya desarrollada. Podríamos volver a utilizar www.xml-sitemaps.com, pero recordemos que limita el número de documentos a 500.

7.2.1.6 Mejor gestión de los controladores web

Anteriormente propusimos gestionar los controladores web mediante una clase llamada "DriverPool", que facilitara una cola con la misma cantidad de controladores que hilos analizadores. Además, sería esta clase la que se encargara de construir la cola e insertar los controladores, con las opciones correspondientes. Pues bien, existe la posibilidad de eliminar por completo el uso de una cola, aumentando el rendimiento del programa a base de evitar la extracción e inserción de los controladores. Es necesario crear una clase, llamémosla "AnalizadorCompleto", que contenga la lista de *suppliers* que tiene actualmente la clase "EjecutorAnalisis". También debe poseer, como atributo, un controlador web. Además, debe implementar a la clase "Runnable", de Java, que permite ejecutarla como un hilo. Dentro de la clase "EjecutorAnalisis", en la función que inicie los análisis de accesibilidad de un documento, crearemos instancias de la clase "AnalizadorCompleto" hasta llenar su *thread pool*. De esta forma, cada hilo tendrá su propio controlador web y no tendrá que ir extrayendo e insertando diferentes controladores por cada tipo de analizador que utilice.

8 Referencias

1. Boe.es. 2018. *BOE.es - BOE-A-2018-12699 Real Decreto 1112/2018, de 7 de septiembre, sobre accesibilidad de los sitios web y aplicaciones para dispositivos móviles del sector público.*. [online] Available at: <https://www.boe.es/diario_boe/txt.php?id=BOE-A-2018-12699> [Accessed 23 September 2022].
2. eur-lex.europa.eu. 2019. *Directiva (UE) 2019/882 del Parlamento Europeo y del Consejo, de 17 de abril de 2019, sobre los requisitos de accesibilidad de los productos y servicios (Texto pertinente a efectos del EEE)*. [online] Available at: <https://eur-lex.europa.eu/eli/dir/2019/882/oj> [Accessed 23 September 2022].
3. 14289-1:2014, I., 2022. *ISO 14289-1:2014*. [online] ISO. Available at: <https://www.iso.org/standard/64599.html> [Accessed 23 September 2022].
4. W3.org. 2022. *Web Content Accessibility Guidelines (WCAG) 2.0*. [online] Available at: <https://www.w3.org/TR/WCAG20/> [Accessed 23 September 2022].
5. 32000-1:2008, I., 2022. *ISO 32000-1:2008*. [online] ISO. Available at: <https://www.iso.org/standard/51502.html> [Accessed 23 September 2022].