



**UNIVERSITAT
JAUME·I**

Grado en Ingeniería Informática

Trabajo Final de Grado

**MEJORA Y REFACTORIZACIÓN DE UNA
BIBLIOTECA DE CÓDIGO ABIERTO PARA LA
EXTRACCIÓN DE METADATOS EN DOCUMENTOS
PDF**

Autor:

Juan Bautista García Traver

Supervisor:

Manuel Diago García

Tutor Académico:

Isabel Gracia Luengo

Fecha de lectura: 15 de julio de 2019

Curso académico 2018/2019

Resumen

En este documento se describe el análisis, desarrollo e implementación del proyecto de final de grado del grado de Ingeniería Informática, el cual ha tenido lugar durante la estancia en prácticas en Irene Solutions SL, localizada en Burriana.

El proyecto nace de la necesidad de mejorar PdfTagger, una de las librerías con las que trabajan en la empresa. Dicha librería permite a partir de un documento PDF y un conjunto de metadatos, aprender la estructura de un archivo PDF para posteriormente extraer metadatos de documentos PDF con la misma estructura.

Para poder llevar a cabo este trabajo, se ha realizado un estudio del entorno ya existente en la empresa, así como de las distintas herramientas empleadas.

Con la implementación de algunas de las mejoras detalladas en el documento se ha conseguido aumentar el porcentaje de metadatos extraídos correctamente con respecto al estado inicial.

Palabras clave

Documentos PDF, Extracción de metadatos, Librería PdfTagger, Librería iText, C#, Expresiones Regulares

Keywords

PDF Documents, Metadata extraction, PdfTagger Library, iText Library, C#, Regular Expressions

Índice general

Capítulo 1	7
Introducción	7
1.1 Contexto y motivación del proyecto	7
1.2. Objetivos del proyecto	8
1.3. Estructura de la memoria	8
Capítulo 2	11
Descripción del proyecto	11
2.1. Escenario inicial	11
2.2. Descripción de las tecnologías y herramientas	14
2.3. Control de versiones	16
Capítulo 3	17
Planificación del proyecto	17
3.1. Metodología	17
3.2. Planificación	17
3.3. Estimación de recursos	19
3.4. Estimación de costes	20
3.5. Seguimiento del proyecto	22
Capítulo 4	25
Análisis y diseño del sistema inicial	25
4.1. Conocimientos previos	25
4.1.1. Estructura de un archivo PDF	25

4.1.2. Expresiones regulares	26
4.2. Análisis del sistema	27
4.2.1. Flujo de aprendizaje	28
4.2.2. Flujo de extracción	30
4.3. Diseño de la arquitectura del sistema	32
4.4. Interfaz del sistema	32
4.5. Diagramas de clases de PdfTagger	34
Capítulo 5	41
Desarrollos e implementación de mejoras	41
5.1. Conocimiento de las tecnologías	41
5.2. Desarrollo de pequeño software para identificación de cadenas de texto mediante expresiones regulares	44
5.3. Desarrollo de pequeño software para identificación de cadenas de texto por posicionamiento de coordenadas	52
5.4. Estudio y desarrollo de mejora basada en extracción del tipo de fuente del texto	55
5.5. Estudio y desarrollo de mejora basada en extracción por tipo, tamaño y color de fuente en PdfTagger empleando PdfClown	59
5.6. Extracción por tipo, tamaño y color de fuente en PdfTagger con iText	64
5.7. Rutina de eliminación de resultados falsos positivos	65
5.8. Iteración de patrones sobre la página correspondiente	68
5.9. Identificación e implementación de nuevos tipos de patrones	70
Capítulo 6	71
Validación y pruebas	71
6.1. Conjunto de muestra	71
6.2. Procedimiento de las pruebas	71
6.3. Resultados	72

Capítulo 7	75
Problemas surgidos en los desarrollos	75
7.1. Extracción de XML de un Documento PDF	75
7.2. Extracciones con PdfClown	76
Capítulo 8	79
Conclusiones	79
Bibliografía	81

Capítulo 1

Introducción

En este capítulo se introducen las principales motivaciones para realizar el proyecto, los objetivos esperados una vez finalizado y la estructura de este documento.

1.1 Contexto y motivación del proyecto

El proyecto propuesto, presentado en este documento, se realiza en el marco de las prácticas enfocadas en el Proyecto Final de Grado de Ingeniería Informática. La motivación de estas prácticas se basa en la posibilidad de aplicar los conocimientos adquiridos durante los distintos cursos académicos del grado, así como poner en práctica los conceptos más específicos del itinerario de Tecnologías de la información.

Este proyecto tiene lugar en la empresa Irene Solutions SL bajo la supervisión de Manuel Diago García. Esta empresa presenta dos frentes de trabajo, dentro del sector de la gestión documental para la pequeña y la gran empresa. Por una parte, trata la gestión documental desde el punto de vista de la facturación y la contabilidad, teniendo como clientes tanto autónomos como empresas. Por otra parte, nos encontramos con el desarrollo e implantación de distintos programas enfocados a agilizar y, hasta cierto punto, automatizar este tipo de trabajo con enfoque hacia la gran empresa. Gran parte del abanico de estos programas está desarrollado bajo licencias públicas de código abierto GNU AFFERO v3 [2].

En concreto, el proyecto se centra en una de sus bibliotecas de código abierto, PdfTagger [11]. Dicho software se basa en la extracción de metadatos estructurados a partir de los objetos desestructurados de un documento PDF, de manera que pueda obtenerse cierta información relevante para los clientes. A partir del uso de esta librería, así como del análisis del entorno competitivo, se ha observado la necesidad de desarrollar distintas mejoras con tal de llegar a un producto que pueda, incluso, superar a la competencia.

El desarrollo del proyecto (consistente en el análisis del entorno, la propuesta de mejoras, su desarrollo, implementación y posterior evaluación para comprobar la mejora sobre los resultados) resulta de gran interés, dado que en esta estancia se da la oportunidad de poder realizar el estudio y la implantación de las distintas mejoras del software sobre un entorno real.

1.2. Objetivos del proyecto

El objetivo de este proyecto es proponer e implementar mejoras al software de código abierto actual denominado PdfTagger de la empresa Irene Solutions S.L.

El software deberá ser estudiado y analizado, con tal de entender su funcionamiento. Además, también se estudiarán las distintas herramientas y bibliotecas de software empleadas en su desarrollo y ejecución.

Una vez estudiado el software, se procederá a la identificación de mejoras con el objetivo de obtener una versión mejorada para aplicar en el producto final.

A continuación se deberá desarrollar, implementar y evaluar las distintas mejoras. La finalidad de la evaluación es obtener métricas que nos permitan observar si las mejoras desarrolladas suponen un incremento en el número de datos identificados correctamente sobre los objetos no estructurados de los documentos PDF.

El objetivo principal puede ser descompuesto en el siguiente subconjunto de objetivos:

- Identificar e implementar nuevas estrategias de localización de datos.
- Identificar e implementar nuevos tipos de patrones.
- Ampliar los catálogos de expresiones regulares usados en la biblioteca del software.
- Ampliar la documentación del software, de cara a permitir una mayor comprensión del código por terceros.
- Mejorar el control interno de traducción de expresiones regulares.

Con el transcurso de la estancia, además de los subobjetivos citados, también añadimos:

- Identificación y eliminación de resultados no deseados (falsos positivos).

1.3. Estructura de la memoria

La estructura de esta memoria consta de las siguientes partes.

En primer lugar, el primer capítulo, donde se expone la motivación y los objetivos que impulsaron la elección de las prácticas para la realización del proyecto.

Seguidamente, en el segundo capítulo, se realiza una descripción general del proyecto. Esto incluye la descripción de: la empresa donde se ha realizado la estancia, el alcance del proyecto, el software de la empresa Irene Solutions SL, así como las herramientas y metodologías empleadas para estudiar y mejorar el software actual.

A continuación, en el tercer capítulo, se expone la planificación inicial del proyecto, así como el seguimiento del proyecto efectuado.

Después, en el cuarto capítulo, se detalla el análisis realizado sobre el software de la empresa, además de los distintos desarrollos que se han ido realizando para mejorarlo.

En el quinto capítulo, se exponen distintos desarrollos e implementaciones de las nuevas mejoras.

En el sexto capítulo, se exponen las pruebas llevadas a cabo para poder comprobar la eficiencia de cada mejora.

En el séptimo capítulo, se exponen los problemas encontrados durante el desarrollo de las prácticas, así como mejoras no factibles.

Por último, en el octavo capítulo, se presentan las conclusiones. En este caso, se detalla la valoración del periodo de la estancia y se aporta una opinión personal.

Capítulo 2

Descripción del proyecto

En este capítulo se detalla cómo está estructurado el entorno actual de trabajo, las tecnologías empleadas y el control de versiones entre los distintos avances sobre el software.

2.1. Escenario inicial

El proyecto cuya propuesta se presenta en este trabajo se va a desarrollar en la empresa Irene Solutions SL. Esta empresa, creada en el año 2014, se dedica a la gestión documental empresarial. Por un lado trata la gestión documental desde el punto de vista de la facturación. Por otro, trata el desarrollo de aplicaciones que apoyen la gestión documental, proporcionando servicios de automatización contable y gestión a los clientes. La empresa cuenta con 2 empleados: Jesús Juárez, administrativo enfocado en la gestión documental y el trato directo con autónomos y empresas, y Manuel Diago, desarrollador de todo el software utilizado en la empresa y supervisor en esta estancia de prácticas.

Este proyecto consiste en mejorar el software, ya presente en la empresa. Dicho software se encuentra desarrollado en conjunción con la librería PdfTagger [11] (cuyo código se encuentra disponible en GitHub) sobre la cual se fundamenta el desarrollo del proyecto.

Esta biblioteca de código abierto utiliza como dependencia la biblioteca de iTextSharp 5 [12], está desarrollada en .NET por Irene Solutions SL y distribuida mediante la licencia Affero General Public License [2]. Su objetivo es servir de apoyo en la obtención de datos estructurados a partir de documentos PDF. Su funcionalidad radica en tomar como entradas metadatos obtenidos con anterioridad de archivos PDF y el archivo PDF del cual se han obtenido, identificar patrones con los que extraer los datos buscados y almacenarlos para utilizarlos en posteriores procesamientos de documentos PDF de la misma categoría y tipo que el documento de entrada. Para trabajar con la librería, se dispone también de una interfaz gráfica desarrollada en Windows Forms.

El cometido del alumno es conseguir que se incremente la tasa de extracción de datos válidos a partir de distintos documentos empresariales.

A continuación se detallará el funcionamiento del software que implementa la biblioteca PdfTagger.

El software parte de los siguientes valores de entrada:

- Un conjunto de metadatos estructurado. Véase un metadato como un trozo de texto (un número, una fecha, un grupo de palabras, etc.). Estos metadatos se pasan al programa mediante campos de texto sobre la interfaz del software en cuestión, donde el usuario puede introducir las salidas que espera obtener (un número de factura, la fecha de emisión, etc.).
- Un documento PDF estructurado. Donde *estructurado* se entiende como un archivo PDF el cual contiene una capa/lienzo sobre el que se encuentra el texto que contiene el documento PDF en cuestión. Esta “capa” es transparente para el ser humano pero visible por el software. Visto de otra manera, en un documento PDF escaneado, tendríamos por un lado la parte “imagen” del documento escaneado y, por otro, la parte “texto” correspondiente al texto representado en la imagen. Realmente todo esto es algo más complejo, cosa que explicaremos posteriormente, en la sección 4.1.1. *Estructura de un archivo PDF*.

Una vez obtenidos estos elementos de entrada, se realiza sobre la parte “texto” del PDF una búsqueda con la intención de identificar información de localización de los metadatos que deseamos obtener sobre el texto o el marco del PDF. Esto es, la posición de un conjunto de texto o grupo de palabras sobre el documento gráfico, así como el texto que precede/sucede a este conjunto de texto o grupo de palabras. Como ejemplo, para un número de factura, podría ser su posición en el documento (coordenadas X e Y) o el texto que le precede (por ejemplo “Número de factura: ”).

Llegados a este punto, debemos definir el concepto de patrón. Un patrón consiste en un objeto con una serie de propiedades que nos permite localizar un determinado metadato sobre un archivo PDF. Dependiendo del tipo de extracción, estos patrones contienen diferentes propiedades. Estas propiedades pueden ser: las coordenadas del texto sobre el documento PDF, una expresión regular que coincida con un trozo de texto que precede al metadato que se ha localizado, etc. Estos patrones se generan en el flujo de aprendizaje, consistente en identificar dónde se encuentran los metadatos sobre un documento PDF. Los patrones se almacenan en un fichero en formato XML con la finalidad de ser empleados en posteriores iteraciones de extracción de metadatos posteriores. Este fichero XML toma como

nombre un metadato que permita identificar el tipo de morfología del documento PDF (por ejemplo, el NIF del proveedor en los documentos de factura), de manera que la extracción de metadatos se generalice y automatice para un determinado grupo de documentos con el mismo formato.

A mayor aprendizaje sobre conjuntos de documentos del mismo tipo, mayor conocimiento por parte del sistema sobre cómo extraer dichos datos de entrada sobre el texto del documento PDF; con el consecuente incremento en el número de aciertos.

Finalmente, teniendo unos patrones efectivos, podemos permitir al sistema realizar la extracción de los datos de manera automatizada; asumiendo que existe un porcentaje de fallos, el cual se debe haber rebajado a un nivel asumible con la etapa de entrenamiento/aprendizaje anterior.

Un sistema como éste, en caso de ejecutarse en un tiempo adecuado, permitiría agilizar el proceso de clasificación e inserción de documentos empresariales, tales como facturas, Documentos Únicos Administrativos (DUAs) de importación, albaranes, etc., así como mejorar el tiempo de trabajo dedicado por el administrativo de esta tarea. Se debe entender por tiempo adecuado, un intervalo de tiempo menor al que emplearía una persona que ejerza un cargo de administrativo (con experiencia y fluidez) en introducir los datos de un conjunto de facturas en un sistema informatizado. De manera que el sistema logre realizar esta tarea en un intervalo de tiempo relativamente menor que el que debería dedicarle dicho administrativo.

El alcance de este sistema incluye la gestión de documentos en formato PDF desde que son recibidos como entrada al programa hasta que los datos necesarios son procesados y extraídos para su distribución hacia plataformas ERP (sistema de planificación de recursos empresariales). No incluye su integración con los ERP ni con sistemas de gestión de bases de datos.

Para la correcta ejecución de pruebas sobre el software, se dispone de una base de datos con distintos documentos empresariales, aproximadamente 10.000 documentos, los cuales se han proporcionado para el desarrollo de este proyecto con la condición de mantener la confidencialidad de los datos.

2.2. Descripción de las tecnologías y herramientas

En este apartado se describe el software empleado en el desarrollo del proyecto.

- **C# [5]**

Lenguaje de programación orientado a objetos, realizado por Microsoft para el desarrollo de aplicaciones sobre la plataforma .NET. Consta de una sintaxis similar al lenguaje JAVA. Este lenguaje es el empleado en el backend del software.

- **Microsoft Visual Studio Community 2017 [3]**

Entorno de desarrollo realizado por Microsoft. Se trata de una herramienta multiplataforma la cual soporta distintos lenguajes como C#, Visual Basic, etc. así como entornos de desarrollo web como ASP.NET.

La principal razón de uso de este entorno se debe a las directrices por parte del supervisor, además de ser el entorno empleado en la empresa. Esto es debido al soporte del lenguaje de programación C#, utilizado en el desarrollo del proyecto.

Por otra parte, según podemos observar en el TOP IDE de Popularity of Programming Language [16] (figura 2.1) se trata del entorno de desarrollo más usado en todo el mundo hoy en día [9]. Seguramente debido al gran soporte de actualizaciones de seguridad que recibe por parte de Microsoft.

Worldwide, Apr 2019 compared to a year ago:

Rank	Change	IDE	Share	Trend
1		Visual Studio	22.82 %	-3.2 %
2		Eclipse	20.83 %	-3.3 %
3		Android Studio	17.9 %	+6.9 %
4		NetBeans	6.74 %	-0.1 %
5	↑	IntelliJ	4.87 %	+0.6 %
6	↑↑↑↑	Visual Studio Code	4.64 %	+1.5 %
7	↑↑	pyCharm	4.28 %	+0.9 %
8	↓↓↓	Sublime Text	3.98 %	-0.4 %
9	↓↓	Atom	3.67 %	-0.6 %
10	↓↓	Xcode	3.31 %	-0.4 %

Figura 2.1. Top 10 IDEs en fecha abril de 2019.

En la siguiente gráfica (figura 2.2), proveniente del mismo sitio web que la figura 2.1 [16], podemos observar la tendencia a lo largo del tiempo, donde en el último periodo, Eclipse ha descendido frente a Visual Studio.

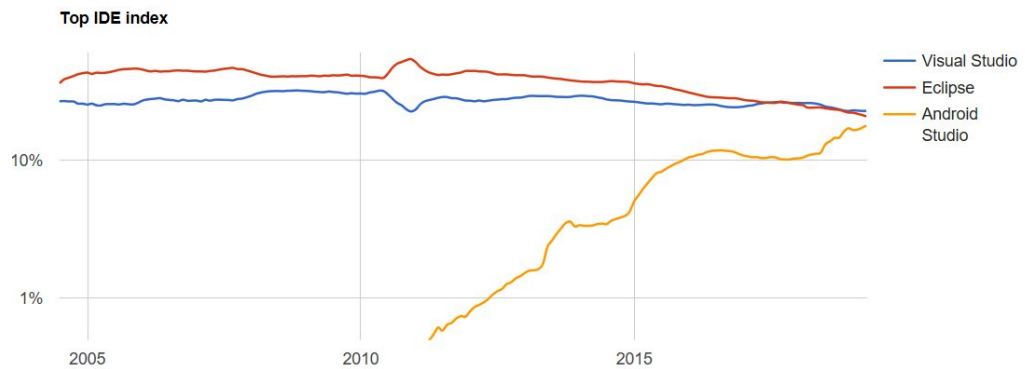


Figura 2.2. Gráfica comparativa de IDEs a lo largo del tiempo en fecha abril de 2019.

- **Windows Forms**

Interfaz de programación para aplicaciones gráficas incluida en el Microsoft .NET Framework. Permite el uso de elementos propios de la API de la interfaz gráfica de Windows. Usado en el frontend del software (la interfaz que acompaña a la biblioteca para facilitar su uso).

- **iTextSharp 5 [12]**

Biblioteca Open Source desarrollada en .NET por Bruno Lowagie, entre otros, y distribuida mediante la licencia Affero General Public License [2].

Facilita la creación y manipulación de archivos PDF. Se trata de la variante de iText 5 de JAVA.

- **PdfTagger [11]**

Biblioteca Open Source desarrollada en .NET por Irene Solutions SL y distribuida mediante la licencia Affero General Public License [2].

Su objetivo es servir de apoyo en la obtención de datos estructurados a partir de PDFs. Utiliza como dependencia la biblioteca de iTextSharp 5.

- **RegEx Storm [15]**

Plataforma web empleada en el uso, análisis y creación de las distintas expresiones regulares empleadas en el software.

- **Sandcastle**

Generador de documentación de software desarrollado por Microsoft. Produce documentación de tipo MSDN-Style a partir de la información del código ensamblado en .NET y la documentación XML encontrada en el código fuente del proyecto del ensamblado.

- **GitHub**

Plataforma Cloud destinada a almacenar proyectos de código fuente de aplicaciones, tanto libres como privativas, mediante el uso del control de versiones de Git. Desarrollado en el lenguaje de programación Ruby.

2.3. Control de versiones

En este proyecto se ha utilizado GitHub para llevar un control de las modificaciones que se han ido realizando sobre el proyecto. Todas las modificaciones se han hecho sobre un fork del repositorio original, con tal de no modificar el correcto funcionamiento del programa principal.

Por otra parte, para el proceso de aprendizaje, también se han creado otros repositorios a modo de entornos de prueba con los que poder implementar y ejecutar distintos desarrollos que vayan surgiendo sobre las bibliotecas a fin de mejorar su entendimiento y saber qué aplicar luego en las modificaciones y mejoras.

El uso de esta herramienta es debido a que, en cualquier proyecto de desarrollo de software, es necesario llevar un control sobre el trabajo realizado en cada versión del producto. De manera que, de ser necesario, se tenga la posibilidad de poder realizar el trabajo en cualquier dispositivo, ya sea el ordenador del trabajo o el de casa; así como poder volver a una versión anterior en caso de producirse algún error importante.

Capítulo 3

Planificación del proyecto

En este capítulo se introduce la metodología empleada en el desarrollo del proyecto, los recursos necesarios y los costes. Por otra parte también se describe el seguimiento realizado.

3.1. Metodología

En relación a la metodología de trabajo empleada, en este proyecto se aborda una metodología predictiva [7]. El uso de esta metodología se debe a la existencia de unos objetivos claros, previamente identificados, y desarrollados hasta su consecución. Por otra parte, el proyecto es llevado a cabo por una sola persona, por lo que no requiere de reuniones en equipo. Por último, la empresa sólo posee un trabajador enfocado en la programación, por lo que ésta ha sido la metodología llevada a cabo por la empresa hasta ahora.

3.2. Planificación

Inicialmente, la planificación de este proyecto se ha realizado mediante el siguiente desglose de tareas (tabla 3.1) que muestra las distintas fases en las que se basará mi desarrollo.

Para esta planificación, se han identificado una serie de tareas a realizar, como antesala al desarrollo de las mejoras, con tal de comprender el marco del proyecto de software ya establecido en la empresa y las distintas herramientas empleadas antes de realizar el desarrollo. Estas tareas, y su respectiva duración, se han establecido bajo las recomendaciones del supervisor, teniendo en cuenta su experiencia en este campo.

Las distintas etapas se encuentran establecidas por horas. En este caso, el horario de trabajo consiste en 24 horas semanales, repartiéndose en 4 horas por día laboral (lunes a viernes) excepto el miércoles en que se realizan 8 horas. La fecha de inicio de las prácticas data del 12 de febrero de 2019, por su parte, la fecha de finalización, del 24 de mayo de 2019. Como se puede observar, en la tabla 3.1 la suma total son las 300 horas del tiempo de prácticas.

N°	Tarea	Tiempo (h.)	Dependencias
1	Planificación	60 horas	
1.1	Análisis del entorno	20 horas	
1.1.1	Funcionamiento en el marco de la empresa	12 horas	
1.1.2	Vista general del software	8 horas	1.1.1
1.2	Estudio de las herramientas a usar	40 horas	1.1
1.2.1	IDE Visual Studio Community 2017	2 horas	
1.2.2	Windows Forms	1 hora	
1.2.3	GitHub	1 hora	
1.2.4	Librería PdfTagger	24 horas	
1.2.5	Librería iTextSharp	10 horas	
1.2.6	Expresiones regulares en C#	2 horas	
2	Preparación	88 horas	1
2.1	Creación de PDF mediante librería iText	24 horas	
2.2	Extracción de texto de PDF mediante iText	44 horas	2.1
2.2.1	Por límites contextuales	22 horas	
2.2.2	Por coordenadas	22 horas	
2.3	Automatización de extracciones	20 horas	2.2
3	Análisis	60 horas	2
3.1	Estudio del funcionamiento de PdfTagger	40 horas	
3.1.1	Búsqueda de posibles mejoras	36 horas	
3.1.2	Actualización de la documentación existente	4 horas	
3.2	Estudio de mejoras mediante iTextSharp	20 horas	3.1
4	Implementación	72 horas	3
4.1	Desarrollo de las mejoras	40 horas	
4.2	Adición al software de prueba	12 horas	4.1
4.3	Tratamiento de errores	20 horas	4.2

5	Prueba y puesta en marcha	20 horas	4
5.1	Prueba de eficiencia en el software final	12 horas	
5.2	Documentación de los nuevos procedimientos	8 horas	

Tabla 3.1. Desglose de tareas junto con su dedicación en horas y sus dependencias (aproximación inicial).

3.3. Estimación de recursos

La estimación de recursos se divide en 3 partes. Por un lado encontramos los recursos de software, por otro, los de hardware, los recursos humanos, y por último, los indirectos.

En cuanto a los recursos de software:

- **IDE Visual Studio Community 2017:** entorno de desarrollo del proyecto.
- **GitHub:** repositorio donde almacenar las distintas versiones de la evolución del proyecto.
- **iTextSharp 5:** biblioteca Open Source para la manipulación y creación de PDFs.
- **PdfTagger:** biblioteca Open Source para la extracción de metadatos de PDFs.
- **RegEx Storm:** plataforma web para el testeado de expresiones regulares sobre un texto.
- **Sandcastle:** generador de documentación de código fuente.

En cuanto a los recursos de hardware:

- **HP Pavilion 14-bf006ns:** ordenador portátil con Intel i5 7200U a 2'7GHz con sistema operativo Windows 10 Home.
- **Ratón inalámbrico VicTsing 2'4G.**

En cuanto a los recursos humanos:

- **Juan Bautista García Traver (yo):** analista y programador de software.

En cuanto a los costes indirectos:

- **Alquiler.**
- **Comunicaciones.**
- **Energía.**
- **Amortización mobiliario.**
- **Limpieza.**

3.4. Estimación de costes

En este apartado detallamos los costes monetarios para cada una de las partes.

Coste recursos humanos.

Basándome en el desglose de tareas y en los sueldos medios para un desarrollador junior de .NET según la plataforma web de indeed [4], los costes de recursos humanos para 1 desarrollador son 20.072 € al año. Desglosando este sueldo para contabilizar el precio por hora:

$$\text{Horas mes} = 40 \text{ horas semanales} * 4 \text{ semanas} = 160 \text{ horas / mes}$$

$$\text{Precio mes} = (20.072 \text{ € / año}) / 11 \text{ meses} = 1.824'73 \text{ € / mes}$$

$$\text{Precio hora} = (1.824'73 \text{ € / mes}) / (160 \text{ horas / mes}) = 11'40 \text{ € / hora}$$

Teniendo en cuenta que el proyecto conlleva 300 horas, por la duración de las prácticas:

$$\text{Coste analista y programador} = 300 \text{ horas} * 11'40 \text{ € / hora} = 3.420 \text{ €}$$

Coste recursos de software.

- **IDE Visual Studio Community 2017:** Según se detalla en la licencia del software [13], en el apartado de organizaciones, si el software se usa para contribuir en el desarrollo de un proyecto de código abierto, su coste es gratuito.
- **iTextSharp 5:** Según se detalla en la web del software [14] la licencia de uso de la biblioteca es AGPL. Esto supone que, en caso de utilizarla sobre tu software, debes publicar el código fuente de dicho software. En este caso el código fuente de PdfTagger está completamente publicado en GitHub [11], por lo que su uso no supone coste alguno.

Por lo demás, todos los recursos de software resultan gratuitos, por lo que su coste en este proyecto es nulo.

Coste recursos de hardware.

En cuanto al coste de los recursos de hardware:

HP Pavilion 14-bf006ns con sistema operativo preinstalado = 859 €

Ratón inalámbrico VicTsing = 8'79 €

Teniendo en cuenta que según la tasación de la empresa se amortizan en 4 años y que la estancia son casi 4 meses:

*4 años * 12 meses = 48 meses*

(859 € + 8'79 €) / 48 meses = 18'08 € /mes

*Total amortizado en estancia = 18'08 € * 4 meses = 72'32 €*

Costes indirectos.

Por último, en cuanto a los costes indirectos respecto de un mes según los datos proporcionados por el supervisor:

Alquiler: 318 € / mes

Comunicación: 218 € / mes

Energía eléctrica: 90 € / mes

Amortización mobiliario: 183 € / mes

Limpieza: 60 € / mes

Estos costes deben dividirse entre 3, ya que el proyecto es llevado a cabo solamente por el alumno en prácticas. Los costes totales indirectos entre 3 serían:

869 totales / 3 = 289'67 € / mes imputados al proyecto.

Teniendo en cuenta que la estancia empieza el 12 de febrero y termina el 24 de mayo, habría que considerar 4 meses, por lo que el precio total de los costes indirectos es:

*289'67 € / mes * 4 meses = 1.158'68 € / mes*

Sumario costes totales.

Una vez obtenidos los costes para cada sección, los costes totales del proyecto de 300 horas vienen detallados en la siguiente tabla (tabla 3.2):

Recurso	Coste
Analista y programador junior .NET	3.420'00 €
Amortizado dispositivos electrónicos	72'32 €
Costes indirectos	1.158'68 €
Total costes	4.651'00 €

Tabla 3.2. Costes totales.

3.5. Seguimiento del proyecto

Para el seguimiento del proyecto, se ha utilizado la herramienta de edición de documentos en la nube de Google Drive, Google Docs. La finalidad ha sido escribir un diario de trabajo a modo de tabla con 3 columnas:

- Día en el que se escribe la entrada del diario.
- Trabajo realizado durante la jornada de 4 horas (excepto los miércoles, los cuales han sido 8 horas).
- Posibles mejoras que hayan podido aparecer con el transcurso de las jornadas.

En el momento en que se ha empezado una mejora, esta celda se rellenaba de color gris, con tal de poder localizar posteriormente de manera fácil dónde empieza la redacción de cada mejora y poder estimar correctamente el tiempo que se le ha dedicado.

Gracias a este seguimiento personal diario, ha sido posible seguir el hilo del trabajo de cara a retomarlo en la jornada laboral siguiente; así como poder recuperar después los distintos detalles del trabajo realizado en el día a día para su redacción en este documento; a fin de no dejar pasar por alto algún componente importante.

Por otra parte, también se han realizado informes quincenales con:

- Pequeño sumario del trabajo realizado la quincena anterior.

- Resumen de la presente quincena.
- Aproximación del trabajo que se iba a realizar la quincena siguiente.

De esta manera, se ha intentado conseguir una cohesión con la planificación inicial e intentar no diferenciarse en exceso.

En referencia a este último aspecto, durante el desarrollo del proyecto, se han ido experimentando cambios sobre la planificación inicial. La planificación inicial se había realizado en base a la experiencia que tiene el supervisor en este campo. No obstante, no quita que pueda sufrir desviaciones debido a la dificultad de aproximar la curva de aprendizaje que iba a experimentar yo, como alumno en prácticas, en este proyecto. Es por ello que, las aproximaciones en coste de horas de los distintos periodos han variado en la práctica: algunos periodos han experimentando un aumento de horas dedicadas y otros han disminuido su duración.

Basándonos en el trabajo realizado y en el diario que se ha llevado a término en el día a día, se muestra una nueva tabla de desglose de tareas (tabla 3.3) con el coste real de horas de dedicación para cada tarea.

Como se puede observar respecto a la planificación inicial, el tiempo dedicado a la tarea de planificación se ha reducido considerablemente. Esto es debido a que la curva de adaptación al entorno fue más rápida de lo esperado. Por otra parte, la etapa de preparación también se ha visto reducida, ya que el aprendizaje de las distintas herramientas se realizó de manera satisfactoria.

No obstante, las etapas referentes al análisis y a la implementación han visto incrementada su dedicación. Esto ha sido debido a que el trabajo de búsqueda de posibles mejoras ha resultado ser más difícil conforme el software ha ido alcanzando cierto grado de madurez durante el transcurso del desarrollo. Además, el desarrollo de las mejoras y su tratamiento de errores también ha supuesto un hándicap donde muchas veces se ha debido deshacer el camino tomado y tomar otra vía de desarrollo con el fin de establecer mejoras factibles.

Nº	Tarea	Tiempo (h.)	Dependencias
1	Planificación	36 horas	
1.1	Análisis del entorno	8 horas	
1.1.1	Funcionamiento en el marco de la empresa	4 horas	
1.1.2	Vista general del software	4 horas	1.1.1
1.2	Estudio de las herramientas a usar	28 horas	1.1

1.2.1	IDE Visual Studio Community 2017	2 horas	
1.2.2	Windows Forms	1 hora	
1.2.3	GitHub	1 hora	
1.2.4	Librería PdfTagger	12 horas	
1.2.5	Librería iTextSharp	8 horas	
1.2.6	Expresiones regulares en C#	4 horas	
2	Preparación	52 horas	1
2.1	Creación de PDF mediante librería iText	8 horas	
2.2	Extracción de texto de PDF mediante iText	24 horas	2.1
2.2.1	Por límites contextuales	12 horas	
2.2.2	Por coordenadas	12 horas	
2.3	Automatización de extracciones	20 horas	2.2
3	Análisis	72 horas	2
3.1	Estudio del funcionamiento de PdfTagger	44 horas	
3.1.1	Búsqueda de posibles mejoras	40 horas	
3.1.2	Actualización de la documentación existente	4 horas	
3.2	Estudio de mejoras mediante iTextSharp	28 horas	3.1
4	Implementación	120 horas	3
4.1	Desarrollo de las mejoras	56 horas	
4.2	Adición al software de prueba	16 horas	4.1
4.3	Tratamiento de errores	48 horas	4.2
5	Prueba y puesta en marcha	20 horas	4
5.1	Prueba de eficiencia en el software final	12 horas	
5.2	Documentación de los nuevos procedimientos	8 horas	

Tabla 3.3. Desglose de tareas junto con su dedicación en horas y sus dependencias (coste real empleado).

Capítulo 4

Análisis y diseño del sistema inicial

En este capítulo presentamos un análisis del software inicial así como de sus flujos de ejecución.

4.1. Conocimientos previos

Con tal de comprender correctamente los puntos de este capítulo, primero se deben abordar los siguientes conocimientos fundamentales.

4.1.1. Estructura de un archivo PDF

Al contrario de lo que se puede pensar, un archivo PDF no cumple el formato WYSIWYG (What You See Is What You Get). Con esto se quiere decir que, lo que el usuario ve por la pantalla (por ejemplo un párrafo) no concuerda exactamente con cómo realmente está formado por dentro el PDF y lo que el programa visor está mostrando.

Un documento PDF es una especie de mezcla entre objetos que se referencian entre sí y un lenguaje de programación. Esto es, el PDF está formado por un tipo de estado gráfico. Es por eso que, cada vez que abrimos un PDF, lo que estamos viendo es el resultado de algún código que especifica información gráfica. Por ejemplo, para un archivo PDF en el que tuviéramos contenido solamente la frase “**Hola mundo**” en una parte concreta del documento, el programa interpretaría sobre el archivo información del tipo:

Ve a la posición 5, 300
Establece como fuente Helvetica - Bold
Establece el tamaño de fuente en 18
Establece el color en Blue
Dibuja los glyph “Hola”
Ve a la posición 22, 300
Dibuja el glyph “undo”
Ve a la posición 20, 300
Dibuja el glyph “m”

Como se puede observar, el orden en que se interpretan los gráficos no tiene por qué coincidir con el orden con el que el usuario lee normalmente el texto, como las líneas o los

párrafos. Además, los espacios no tienen por qué realmente existir para las instrucciones de dibujado de los gráficos; simplemente son posiciones sobre las que dibujar texto que pueden estar más o menos separadas. Incluso un trozo de texto que para el usuario es una palabra, para las instrucciones del programa pueden ser letras sueltas más o menos juntas. No hace falta ya especificar las formas de párrafos, las cuales coincidirían con texto puesto en una posición concreta u otra.

Todas estas instrucciones, con su correspondiente posición, fuente, texto, etc., se conforman en objetos, los cuales están numerados. Esta numeración sirve como referencia a una tabla al final del archivo PDF (denominada XREF) con la que poder interpretar posteriormente cada uno de los objetos.

Es por ello que, para poder realizar una extracción de texto de un archivo PDF en el orden de lectura normal del usuario, se debe:

- Leer la tabla XREF.
- Averiguar dónde se encuentra el objeto “inicio” de la página.
- Procesar dicho objeto y todos sus subobjetos haciendo uso de la tabla XREF.
- Procesar las instrucciones geométricas (recordando que los gráficos correspondientes al texto no tienen por qué seguir el flujo direccional de lectura del usuario).
- Ordenar todos los objetos visibles para el usuario, lo que vendrían siendo los objetos correspondientes al texto en el ejemplo anterior, en el orden que se supone que el texto debería ir escrito.
- Construir el texto a devolver.

4.1.2. Expresiones regulares

Las expresiones regulares son una notación para definir lenguajes regulares. Se suelen utilizar como una forma sencilla de describir patrones de texto [8].

Las expresiones regulares poseen un funcionamiento similar a los caracteres comodín del sistema operativo [6]. Por tanto, si queremos obtener de un directorio todos los ficheros de tipo C#, usaremos un identificador o patrón de búsqueda del tipo “*.cs”. De esta manera le estamos indicando al sistema que sólo queremos que nos muestre los ficheros cuya extensión sea “.cs”. Para el sistema operativo, el patrón de búsqueda “*.cs” sería una expresión regular.

Con tal de comprender mejor su aplicación, exponemos un ejemplo. Imaginemos que queremos validar un número de factura que sólo puede empezar por la letra F (de factura)

o la letra A (de abono) y siempre le siguen nueve dígitos. Una cadena de texto que cumpliera esta regla podría ser A018062019 ó F020022018; sin embargo, AB004 ó F0022300 no cumplirían esta regla.

Para poder escribir esta regla primero necesitamos conocer el significado de algunos símbolos de expresiones regulares:

- `[]`: Los corchetes, nos permiten dar una lista de caracteres, de los cuales esperamos que aparezca uno. Por ejemplo, si tenemos un número de factura que puede empezar por “X”, “Y” o “Z”, los corchetes lo indicarán de la manera `[XYZ]`.
- `\d`: Representa un dígito.
- `{n}`: N-veces que se va a repetir un elemento. En combinación con el símbolo del dígito, si queremos detectar un número de cinco cifras, expresaríamos `\d{5}`.

Con ello, para poder identificar el número de factura descrito, emplearíamos la expresión regular `[AF]\d{9}`. De manera que sobre una cadena de texto, seríamos capaces de detectar dicha subcadena tal y como se puede apreciar en la siguiente figura 4.1 empleando la web de RegEx Storm [15].

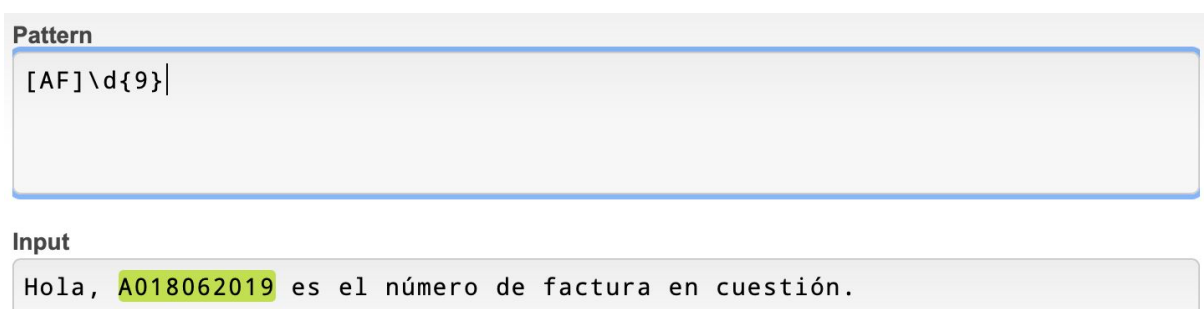


Figura 4.1. Ejemplo de uso de expresión regular sobre un número de factura.

4.2. Análisis del sistema

La biblioteca de PdfTagger tiene como finalidad servir de apoyo a los usuarios en la obtención de datos estructurados a partir de documentos PDF. Para ello, toma como entrada metadatos obtenidos con anterioridad de archivos PDF y el archivo PDF del cual se han obtenido, identifica patrones con los que extraer los datos objetivo y los almacena para utilizarlos en posteriores procesamientos de documentos PDF de la misma categoría y tipo que el PDF de entrada.

Para obtener todo el texto de un documento PDF, en PdfTagger se hace uso de la biblioteca iTextSharp 5, sobre la que se han sobrescrito algunas clases y métodos.

Con tal de comprender el funcionamiento del sistema, dividiremos el flujo de ejecución en dos subflujos: aprendizaje y extracción.

4.2.1. Flujo de aprendizaje

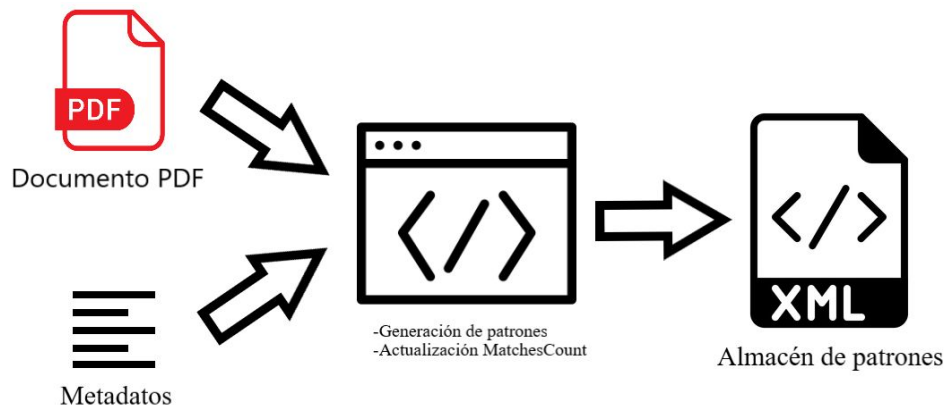


Figura 4.2. Esquema representativo del flujo de aprendizaje.

En primer lugar describiremos el flujo de aprendizaje (figura 4.2). En este flujo de ejecución el software aprende, mediante la generación de patrones de extracción, dónde localizar los distintos metadatos pasados por entrada sobre un archivo PDF concreto. De esta manera, será posible identificar en el flujo de extracción los metadatos de entrada en PDFs con la misma morfología.

La biblioteca contiene un extractor mediante el cual genera grupos de palabras (denominados “WordGroups”), líneas con estos grupos de palabras y un grupo que contiene todas las líneas de texto pertenecientes a una página del PDF. Con “genera” se quiere señalar que en el extractor se obtienen unas cadenas aleatorias; ya que esta extracción se produce a partir de los denominados “PdfTextChunks”, objeto que puede equivaler tanto a un solo carácter como a un párrafo del texto del PDF (este párrafo cobrará más sentido después de leer el *punto 4.5. Diagrama de clases de PdfTagger*).

Para entender el funcionamiento de la extracción sobre un documento PDF también es conveniente tener claro el *punto 4.1.1. Estructura de un archivo PDF*.

Teniendo en cuenta cómo está estructurado internamente un archivo PDF, tiene sentido que primeramente se obtengan los objetos “PdfTextChunks”, como también su procesado en el extractor de la biblioteca PdfTagger para generar los distintos tipos de grupos de cadenas de texto.

Después de generar estas agrupaciones de texto, empezamos a encontrar el valor añadido de la biblioteca sobre la de iText. En este punto la biblioteca realiza una búsqueda de

la posible existencia de los valores de entrada en el PDF, de modo que, para cada agrupación de texto, se realizan una serie de comparaciones. Para estas operaciones se compara:

- Cada uno de los metadatos de entrada.
- “WordGroups” (grupos de palabras a partir de los “PdfTextChunk”).
- “Lines” (líneas a partir de grupos de palabras).
- “PdfText” (todo el texto perteneciente a una página del PDF).

Con el fin de realizar de manera eficiente las comparaciones, se establecen una serie de jerarquías según el tipo de documento PDF sobre el que estamos extrayendo. Por ejemplo, en caso de pasar como parámetro de entrada un documento PDF de factura, se tendría cargada la jerarquía de conversores “InvoiceHierarchySet”, la cual contendría jerarquías de conversores a nivel de metadatos. Según si el metadato que pasamos por entrada es de tipo decimal, de tipo date... se cargarán los conversores para ese tipo, a fin de poder comprobar con éxito si el dato que pasamos por entrada se corresponde con el contenido en el PDF.

Para mejorar la comprensión de esta parte, exponemos un ejemplo concreto. Imaginemos que el texto contenido en el PDF contiene una fecha con formato “24 de abril de 2019”; sin embargo, el parámetro que nosotros hemos pasado por entrada es del tipo “24/04/2019”. Mediante la jerarquía correspondiente al tipo date, transformaríamos el texto al formato date, con el fin de que la comparación sea exitosa.

Llegados a este punto, en el momento en que se ha encontrado un acierto, la biblioteca crea un patrón. Este patrón contiene la información necesaria para permitir encontrar el metadato de entrada en sucesivos documentos PDF de estructura similar. Este patrón se genera a nivel de “WordGroups”, “Lines” y “PdfText”.

- En los “WordGroups” y “Lines” almacenamos las coordenadas del rectángulo que contiene dicho conjunto de texto y la expresión regular generada en su identificación.
- En los “PdfText” almacenamos la expresión regular generada en su identificación, la expresión regular del conjunto de texto que precede y la del que sucede.

Por otra parte, independientemente del tipo de patrón con el que nos encontremos, se almacena también un contador denominado “MatchesCount”, el cual acumula el número de veces que un patrón obtiene el mismo texto que el metadato pasado por entrada (un acierto), así como el tipo de metadato al que corresponde ese patrón (“GrossAmount” para el total de factura). Esto es, en sucesivas iteraciones de aprendizaje, si el patrón generado ya estaba almacenado, simplemente actualiza el contador de aciertos.

Puntualizar que, para cada metadato pasado por entrada, dado que éste puede aparecer un distinto número de veces en el texto del PDF, pueden llegar a generarse bastantes patrones distintos. Esta casuística conlleva consecuencias que analizaremos posteriormente, en la sección 5.7. *Rutina de eliminación de resultados falsos positivos*.

Generados todos los patrones para cada metadato de entrada, se almacenan en un archivo XML. A este archivo XML se le denomina con un identificador en concreto, con la intención de poder localizar los patrones correspondientes a PDFs de su misma morfología en posteriores iteraciones, tanto de aprendizaje como de extracción. De esta manera, su lectura en el flujo de extracción se realiza de manera más intuitiva. Es en este momento donde finaliza el flujo de aprendizaje.

4.2.2. Flujo de extracción

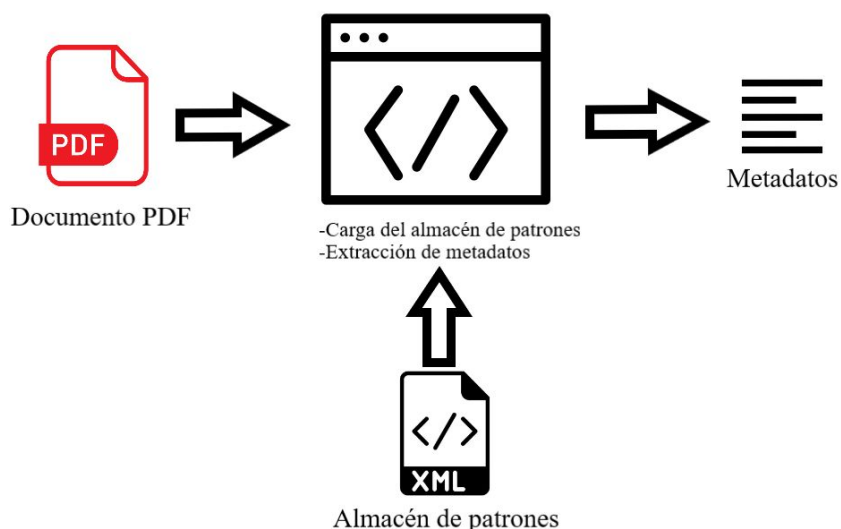


Figura 4.3. Esquema representativo del flujo de extracción.

Para la correcta ejecución de este flujo (figura 4.3), es necesario haber ejecutado primero el flujo de aprendizaje; de otra manera, no será posible extraer ningún tipo de metadato. Sin embargo, su ejecución no tiene por qué realizarse después de una iteración en el flujo de aprendizaje. Esto es, es frecuente el caso en que la morfología del PDF analizado esté bien estructurada, lo que comporta que con 3 iteraciones del flujo de aprendizaje sobre 3 documentos PDF distintos que compartan la misma morfología basten para poder realizar ilimitadas iteraciones del flujo de extracción sobre otros documentos de la misma morfología.

En el flujo de extracción también recibimos como parámetro de entrada un archivo PDF; no obstante, ahora los metadatos vamos a proporcionarlos como salida para el usuario.

La extracción del texto del PDF para poder obtener los metadatos se realiza de la misma manera que en el flujo de aprendizaje, haciendo uso de la sobrescritura realizada en la biblioteca de iText.

En primer lugar, carga el almacén de patrones que hemos generado en la iteración del flujo de aprendizaje. Como hemos comentado, los archivos XML se almacenan con un identificador de localización como nombre del archivo. Por ejemplo, para las facturas, cada XML se denomina con el NIF del proveedor, ya que normalmente una gran parte de los proveedores utilizan la misma morfología en los distintos PDFs de factura que generan. Con ello, se carga únicamente el almacén de patrones correspondiente a dicho formato de documento.

Una vez cargado el almacén de patrones, procedemos a la extracción de los metadatos resultado, correspondiente a la jerarquía del documento PDF pasado por entrada. En el caso de ejemplo hablaremos del correspondiente al “InvoiceMetadata”, el cual contiene los metadatos a extraer: “GrossAmount”, “IssueDate”, etc. Esta extracción se almacenará en un objeto de tipo “PdfTagExtractionResult”.

En este punto, realizamos la extracción sobre el texto a nivel de patrón y grupo de texto. Esto es, para cada grupo de texto:

- En el caso de los “WordGroups” y “Lines”, comprobamos que se aproximen:
 - El tipo de metadato proporcionado por el patrón sobre el que estamos iterando y el del conjunto de texto.
 - Las coordenadas de localización del patrón y el conjunto de texto sobre el PDF, de manera que, si el rectángulo que conforma uno de los textos coincide dentro de un ratio con el otro, se trataría de la misma localización.
- En el caso de los “PdfText”, comprobamos que el texto coincida con la expresión regular, que contiene la parte del texto que precede y la que sucede.

Si pasa con éxito estas comprobaciones, realiza la extracción mediante el patrón sobre el conjunto de texto. El texto resultante de la extracción se almacena en el objeto “result” junto con el patrón usado, en forma de par clave-valor.

Teniendo rellenado el objeto “result”, procedemos a extraer los metadatos. Para ello primero ordenamos el listado de par clave-valor de resultados. Esta ordenación se realiza teniendo en cuenta el “MatchesCount”, el cual almacena el número de aciertos consistente en las veces que un patrón ha extraído el mismo texto que el metadato pasado por entrada en el flujo de aprendizaje. De manera que, para 2 patrones, se compara su “MatchesCount”; y el que contenga el mayor número de aciertos, se sitúa el primero.

A partir del objeto result se obtendrán los metadatos de la salida del programa. Para seleccionar de la manera más correcta posible el valor de estos metadatos, cogeremos el primer valor de la lista de pares, resultante de la ordenación por “MatchesCount”. De manera que se pueda evitar al máximo los resultados falsos positivos.

4.3. Diseño de la arquitectura del sistema

La arquitectura del sistema posee una arquitectura con matices de Model-View-Presenter. Para tener un acceso rápido a las funciones proporcionadas por la biblioteca se emplea una interfaz realizada mediante Windows Forms. Ésta funciona de la siguiente manera [1] (figura 4.4):

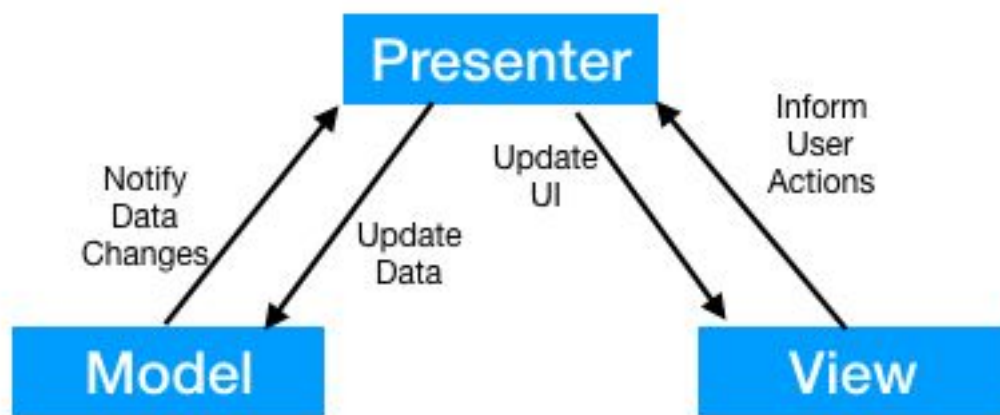


Figura 4.4. Diagrama MVP.

En este caso, el “Presenter” se corresponde con el archivo formInvoices.cs. Este fichero contiene escuchadores para los diferentes elementos presentes en la interfaz así como algunas comprobaciones referentes a los campos presentes en ésta.

Por otra parte, el “Modelo” se corresponde con el fichero formInvoicesModel.cs, el cual contiene llamadas para el relleno de los objetos que constituyen el documento PDF de entrada, los metadatos, y llamadas a distintos elementos de la biblioteca para poder realizar los flujos de aprendizaje y de extracción.

4.4. Interfaz del sistema

Para facilitar el uso de la biblioteca, Pdfagger dispone de una interfaz diseñada mediante las herramientas de Windows Forms. Dicha interfaz permite ejecutar los flujos de aprendizaje y de extracción, así como visualizar el documento sobre el que estamos trabajando por pantalla.

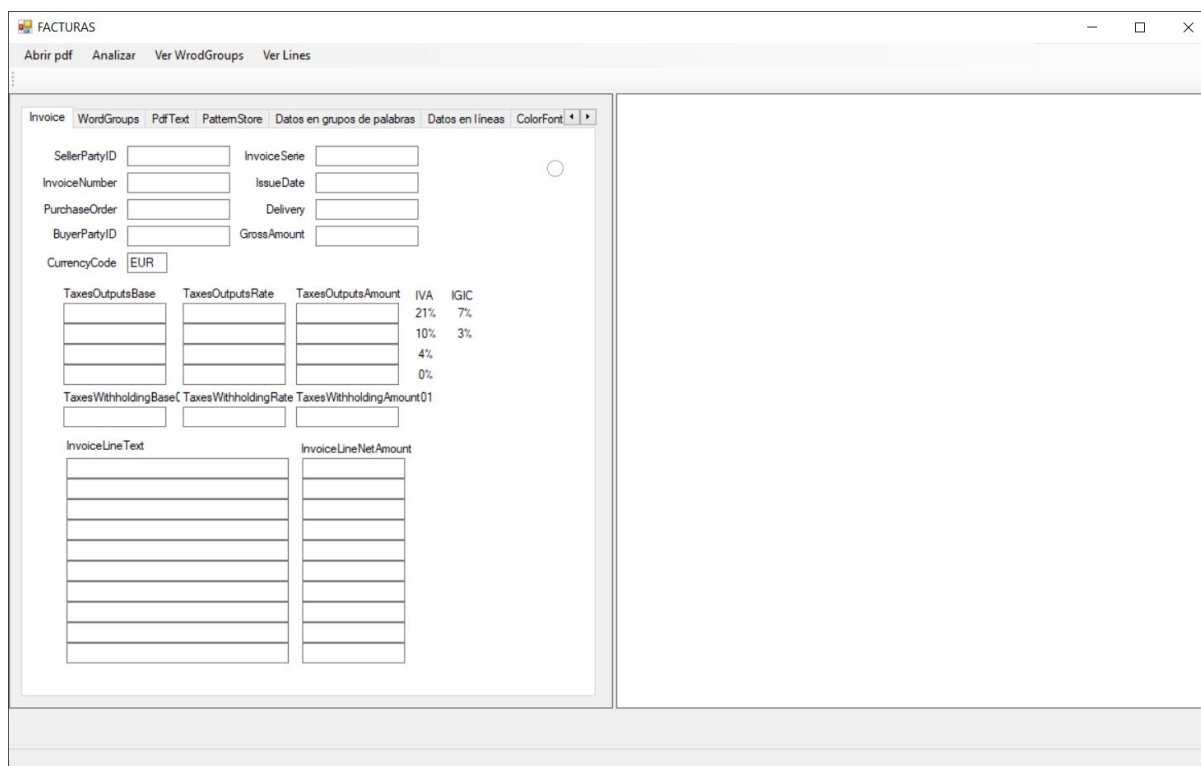


Figura 4.5. Interfaz de uso para la biblioteca PdfTagger.

Como se puede observar en la figura 4.5, en la parte izquierda de la interfaz se encuentran los campos mediante los que rellenar los metadatos que deseamos extraer, correspondientes a la clase “InvoiceMetadata”. Aunque la biblioteca está preparada para realizar extracciones sobre documentos PDF de cualquier ámbito, en este ejemplo, se ha enfocado en la extracción de metadatos de documentos de factura.

En la parte derecha encontramos un margen de color blanco, correspondiente al visor del documento PDF con el que estemos trabajando. En la parte superior encontramos el menú, sobre el que podemos abrir un PDF, así como realizar el flujo de aprendizaje mediante el botón analizar.

En caso de que la morfología del documento PDF de entrada estuviera ya analizada por el flujo de aprendizaje, al introducir el NIF del proveedor (primer campo correspondiente al SellerPartyID) se lanzaría automáticamente el flujo de extracción.

4.5. Diagramas de clases de PdfTagger

Los diagramas de clases presentes en este punto representan cómo está estructurada la biblioteca PdfTagger. En ellos podemos ver la representación de los distintos objetos de entrada y salida, así como las clases empleadas en los distintos flujos de ejecución.

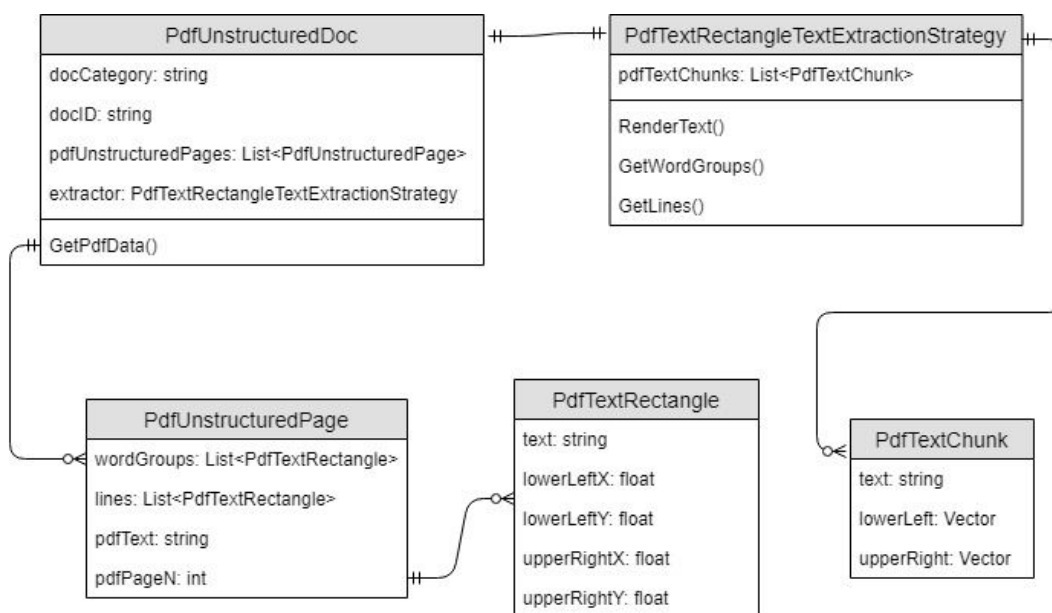


Figura 4.6. Diagrama de clases de la biblioteca PdfTagger correspondiente a un documento PDF.

En primer lugar en la figura 4.6 se detallan las clases que representan el archivo PDF de entrada al programa. Por un lado encontramos la clase “PdfUnstructuredDoc”, compuesta por distintas propiedades y métodos:

- **docCategory**: Representa la categoría del documento PDF. Por ejemplo, “Invoice” para las facturas.
- **docID**: Representa el identificador de la morfología del documento. Por ejemplo, este identificador se rellena con el NIF del proveedor en el caso de las facturas, con tal de encajar con el correspondiente fichero XML (el almacén de patrones).
- **pdfUnstructuredPages**: Representa el contenido de las páginas del documento PDF. Formado por una lista de objetos “PdfUnstructuredPage”.
- **extractor**: Motor del software mediante el que se extraen las cadenas de texto de los objetos del fichero PDF.

- **GetPdfData():** Extrae el texto del PDF, haciendo uso del objeto “extractor” el cual clasifica los datos para favorecer su posterior análisis, y rellena el objeto “pdfUnstructuredPages”.

Para poder extraer el texto, utilizamos la clase “PdfTextRectangleTextExtractionStrategy”. Posee una propiedad y varios métodos:

- **pdfTextChunks:** Listado compuesto de objetos “PdfTextChunk”, los cuales conforman el texto del documento PDF.
- **RenderText():** Procesa los objetos de un archivo PDF con tal de obtener distintas propiedades de él, como las cadenas del texto contenido o las coordenadas de estas cadenas sobre el documento PDF. Conforme lo procesa, rellena el listado de “pdfTextChunks”.
- **GetWordGroups():** A partir de los “PdfTextChunk” obtenidos del texto, intenta recrear el texto según lo interpretaría el usuario al leer por pantalla un documento PDF, en forma de palabras.
- **GetLines():** Parecido al método anterior, sin embargo, esta vez en forma de líneas.

El extractor hace uso de objetos denominados “PdfTextChunk”. Estos objetos conforman lo que, en una primera instancia, el extractor ha podido obtener a partir de los objetos del documento PDF. Se trataría, pues, de un primer procesado menos intrusivo de la información del PDF, por lo que estos objetos pueden representar desde un carácter, pasando por una palabra, hasta un párrafo entero.

Los “PdfTextChunk” están compuestos por:

- **text:** Contiene las cadenas de texto extraídas por el extractor.
- **lowerLeft:** Representa las coordenadas de la cadena de texto sobre el PDF, en concreto el punto de abajo a la izquierda.
- **upperRight:** Misma función que la propiedad anterior, esta vez el punto de arriba a la derecha de la cadena.

A partir de las coordenadas de los “PdfTextChunk”, el extractor calcula en los métodos “GetWordGroups()” y “GetLines()” si dicho PdfTextChunk se debe juntar con el anterior porque forman parte de una misma palabra/línea del documento o almacenarlo tal y como está extraído. Esta aglomeración se almacena en forma de “PdfTextRectangle”.

En cuanto a “PdfUnstructuredPage”, representa una página del documento PDF de entrada. Este objeto se rellena a partir de la información obtenida por el extractor en el método “GetPdfData()” de “PdfUnstructuredDoc”. En este objeto ya se puede apreciar cómo

se ha clasificado el texto en partes que el software cree que pueden ser palabras o líneas, con tal de conseguir una identificación de los datos más fiable. Es por ello que está compuesto de:

- **wordGroups:** Listado de “PdfTextRectangle” buscando formar palabras.
- **lines:** Listado de “PdfTextRectangle” buscando formar líneas.
- **pdfText:** Contiene todo el texto extraído de una página en un único string.
- **pdfPageN:** Número de página correspondiente al documento PDF.

Hay que especificar que esta clasificación sirve para poder establecer los rectángulos de coordenadas sobre el PDF, donde se localizan los grupos de palabras o líneas. En el caso del objeto que contiene texto al completo, esta información se pierde.

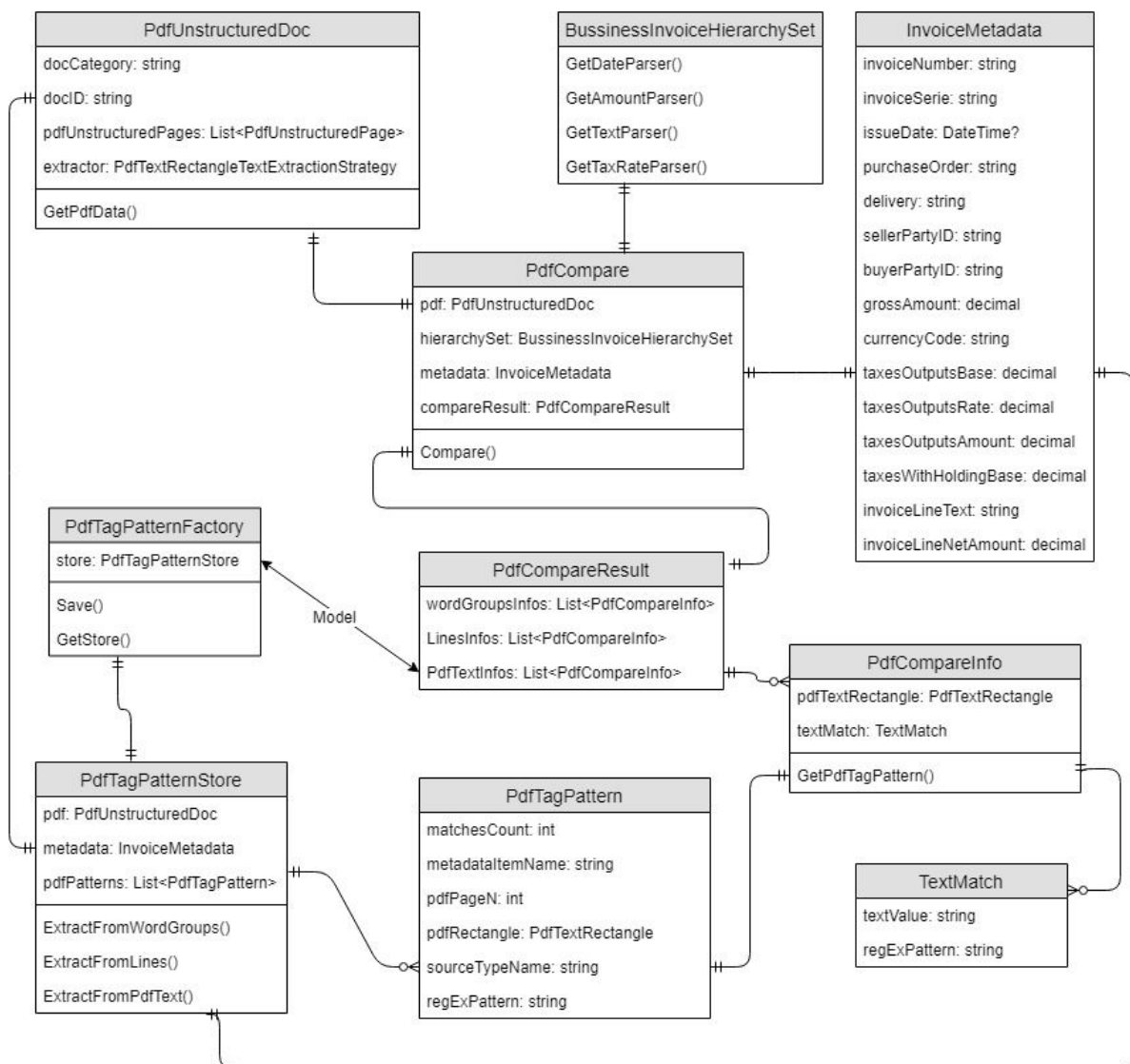


Figura 4.7. Diagrama de clases de la biblioteca PdfTagger para el flujo de aprendizaje.

Una vez se ha dado forma al documento PDF en el sistema es cuando empieza la ejecución de uno de los dos flujos de trabajo de la biblioteca.

En el caso del flujo de aprendizaje (figura 4.7) la información extraída del documento PDF se procesa en busca de coincidencias con los metadatos de entrada. En este caso, al estar enfocada la interfaz en los documentos de factura, los metadatos pasados representan información generalmente contenida en dichos documentos.

Es por ello que tenemos una clase “InvoiceMetadata”. Esta clase se corresponde con los metadatos de entrada con tal de facilitar el funcionamiento del programa. Las propiedades de esta clase están estrictamente relacionadas con los campos de introducción de texto de la interfaz.

Por otro lado, poseemos la clase “BussinessInvoiceHierarchySet”. Esta clase posee los conversores de texto correspondientes a cada metadato de entrada, a fin de realizar una comparación exitosa. Así, si tenemos un importe introducido por entrada de la manera “4.165’08” y en el texto del PDF figura como “4165.08”, la comparación no será exitosa. Es por ello que los convierte a un mismo tipo para que puedan cuadrar.

Procesados los datos de entrada (en un “PdfUnstructuredDoc” para el documento PDF y en un “InvoiceMetadata” para los metadatos a identificar), se ejecuta el análisis comparativo mediante la clase “PdfCompare”. Esta clase posee:

- **pdf:** Correspondiente al documento PDF pasado por entrada.
- **hierarchySet:** Contiene los conversores de datos, en este caso los correspondientes a los documentos de facturación.
- **metadata:** Permite tener acceso a los metadatos de la entrada, los cuales se deben comparar con el texto con tal de establecer coincidencias y poder posteriormente generar los patrones.
- **compareResult:** Contiene un catálogo con los resultados de las comparaciones. Este catálogo está clasificado según si las coincidencias se han identificado en los grupos de palabras (“wordGroups” en el sistema), en las líneas (“lines” en el sistema) o sobre todo el texto de una página (“pdfText”). Gracias a esta clasificación, es posible establecer si la extracción que realiza un patrón es del tipo *por coordenadas* o *por delimitación contextual*.
- **Compare():** Carga la jerarquía de conversores de los metadatos y compara cada uno de los metadatos pasados por entrada con todo el texto de todas las páginas del documento PDF, a fin de identificar estos metadatos y poder establecer posteriormente los patrones de identificación. A medida que encuentra coincidencias las añade al objeto “compareResult”.

Conforme se procesan los resultados, se rellena el objeto de la clase “PdfCompareResult”. Esta clase contiene los catálogos de los objetos identificados como “PdfCompareInfo”, todas las coincidencias de texto para cada metadato.

En “PdfCompareInfo” encontramos la información respecto a cada comparación exitosa obtenida; posee:

- **pdfTextRectangle**: Contiene las coordenadas del rectángulo que contiene la cadena de texto.
- **textMatch**: Contiene el texto sobre el que se ha identificado la coincidencia, así como la expresión regular que lo valida.
- **GetPdfTagPattern()**: Devuelve un objeto de la clase “PdfTagPattern” a partir de las propiedades (la información que ha obtenido sobre el string en referencia al documento PDF).

La clase “TextMatch” es la primitiva del objeto “PdfCompareInfo”. Contiene el texto referente a la coincidencia en un “textValue” y la expresión regular asociada “regExPattern”.

Cuando se ha procesado todo el documento PDF, se procede a generar los patrones y guardarlos en el almacén de patrones. Este flujo de ejecución se establece en la clase “PdfTagPatternFactory”. Esta clase posee:

- **store**: Objeto del tipo “PdfTagPatternStore”. Este objeto permite almacenar toda la información referente a la morfología de un documento PDF, con el fin de poder realizar extracciones de datos automatizadas.
- **Save()**: Permite guardar todas las coincidencias obtenidas sobre un documento PDF como patrones de identificación de metadatos, de manera que se establezcan modelos de documentos. Este guardado se realiza sobre un fichero XML, con tal de poder almacenar la información de manera estructurada y fácil de interpretar. Este método podría ser el denominado *generador de patrones*. Hace uso del método “GetPdfTagPattern()” presente en el objeto “PdfCompareInfo” para su generación.
- **GetStore()**: Carga un fichero XML específico. Este fichero se corresponde con el documento PDF cargado en el sistema, en el caso de las facturas, por el NIF del proveedor. De manera que pueda llegar a realizarse un flujo de extracción automatizado, sin la presencia de un usuario físico.

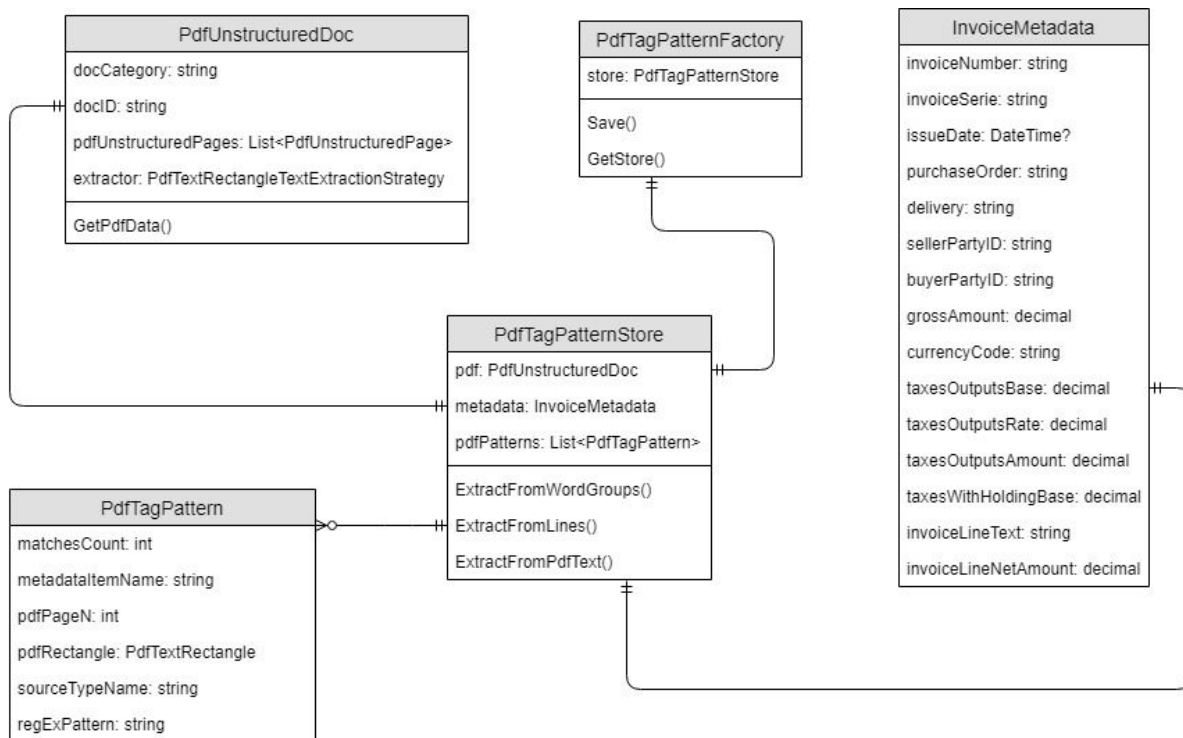


Figura 4.8. Diagrama de clases de la biblioteca PdfTagger para el flujo de extracción.

Como se puede observar, “PdfTagPatternFactory” también posee el método “GetStore()” que poco tiene que ver con el flujo de aprendizaje. Es por ello que esta clase tiene presencia también en el flujo de extracción (figura 4.8). Es el intermediario que permite almacenar en disco o cargar en el sistema un “PdfTagPatternStore”, un almacén de patrones.

La clase “PdfTagPatternStore” contiene:

- **pdf:** Correspondiente al pdf cargado en ese momento en el sistema y analizado por “PdfUnstructuredDoc”.
- **metadata:** Objeto de la clase “InvoiceMetadata”. Este objeto se encuentra en esta clase porque, al igual que en el flujo de aprendizaje el contenido de este objeto se proporciona como método de entrada, en el flujo de extracción es el objeto que se va a devolver relleno por lo que se haya podido extraer haciendo uso de los patrones de identificación aprendidos.
- **pdfTagPatterns:** Contiene un listado de objetos del tipo “PdfTagPattern”. Se trata de los patrones de identificación generados en la fase de aprendizaje.
- **ExtractFromWordGroups():** Haciendo uso del objeto “pdf” y de los “pdfTagPatterns”, los procesa en busca de coincidencias con los metadatos que se espera devolver. Para hallar estas coincidencias, se compara cada “WordGroup” con un “PdfTagPattern”, comprobando mediante el rectángulo

que se tiene guardado y la expresión regular si el texto contenido es un metadato a devolver o no.

- **ExtractFromLines()**: Similar al método anterior, pero sobre líneas.
- **ExtractFromPdfText()**: Similar a los anteriores pero sobre todo el texto obtenido sobre una página. En este caso no se tiene en cuenta las coordenadas del rectángulo, ya que sería el mismo tamaño que el documento PDF. Por otro lado, la expresión regular de este apartado está más elaborada, de manera que no sólo comprueba que el metadato coincida, sino también si el texto que le precede coincide con el que tiene almacenado como genérico de la morfología (delimitación contextual).

La clase “PdfTagPattern” representa un patrón. Es decir, una manera de identificar una cadena contenida dentro de un documento PDF, haciendo uso de las propiedades que se han podido obtener en el análisis del documento. Estas propiedades son:

- **matchesCount**: Contador. Representa el número de veces que el flujo de aprendizaje ha identificado el mismo patrón (dando indicios de que se trata de un patrón recurrente y susceptible de ser mejor que otros que se identifiquen un menor número de veces). Este contador se acumula una vez por cada ejecución del flujo de aprendizaje, no una vez por cada aparición del patrón en una misma ejecución del flujo.
- **metadataItemName**: Referencia a un metadato de la clase “InvoiceMetadata”. En caso de que el patrón permita identificar un importe total de factura, este campo llevará el valor de “GrossAmount”.
- **pdfPageN**: Relaciona con el número de página en el que se ha identificado el patrón.
- **pdfRectangle**: Representa el rectángulo continente con las coordenadas del texto identificado.
- **sourceTypeName**: Referencia al tipo de objeto sobre el que se ha identificado la coincidencia. Esta referencia puede ser “WordGroup”, “Lines” o “PdfText”.
- **regexPattern**: Contiene la expresión regular con la que verificar la validez del texto y averiguar si se trata de una coincidencia o no.

Capítulo 5

Desarrollos e implementación de mejoras

En este apartado se describen algunos desarrollos y cada una de las mejoras que se han identificado y desarrollado para su implementación con éxito.

Por motivos de privacidad y protección de datos, en las figuras presentes en este capítulo se han ocultado distintos datos personales.

5.1. Conocimiento de las tecnologías

Para este proyecto, el primer paso que se ha dado ha sido el aprendizaje y documentación acerca del software sobre el que se iba a trabajar el proceso de mejora, así como de las tecnologías empleadas.

Es por ello que, primeramente (después de las pertinentes instalaciones en el equipo de trabajo) se ha realizado un proceso de familiarización con el entorno de desarrollo del Visual Studio. En este proceso se han llevado a cabo distintas pruebas haciendo uso de Windows Forms como interfaz del software. Estos ejemplos han consistido en la realización de un formulario con distintos paneles y campos de entrada de datos, así como mostrar los datos de salida por otro panel. La finalidad de estas pruebas ha sido entender cómo funcionan los engranajes de esta herramienta.

Por otra parte, también se ha realizado un estudio del flujo de ejecución de la biblioteca PdfTagger. Esto es, comprender sus entresijos a fin de poder establecer qué mejoras son plausibles de realizar.

Además, se ha llevado a cabo la generación de un archivo PDF haciendo uso de las herramientas que proporciona la biblioteca de iText, con tal de entender también su funcionamiento y la integración de la biblioteca con la de PdfTagger. En el ejemplo que presentamos (figura 5.1) se muestra la creación de un PDF con un título con formato y dos párrafos.

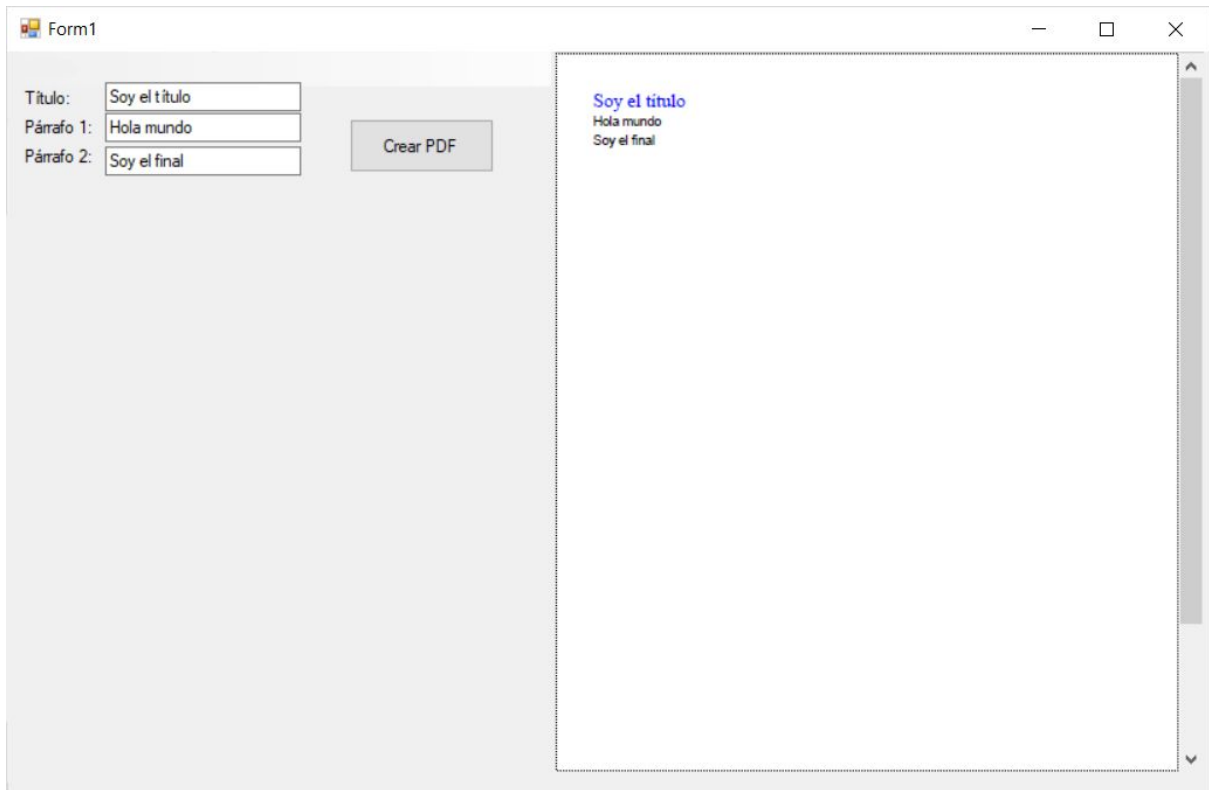


Figura 5.1. Ejemplo de creación de PDF.

El código que ha permitido realizar este PDF es el siguiente (figura 5.2).

```
Document doc = new Document();
PdfWriter.GetInstance(doc, new FileStream("ejemploCreación.pdf", FileMode.Create));
doc.Open();

Paragraph title = new Paragraph();
title.Font = FontFactory.GetFont(FontFactory.TIMES, 18f, BaseColor.BLUE);
title.Add(textBox1.Text);
doc.Add(title);

doc.Add(new Paragraph(textBox2.Text));
doc.Add(new Paragraph(textBox3.Text));
doc.Close();

string url = @"ejemplo.pdf";
webBrowser1.Navigate(url);
```

Figura 5.2. Código para la creación del PDF.

En el código es posible apreciar cómo se usan distintas clases de la biblioteca de iText para generar el PDF, tales como Document, PdfWriter... Para poder mostrar el PDF por pantalla una vez generado, se usa el componente nativo de Windows Forms de tipo

WebBrowser, el cual usa el navegador de Internet Explorer y su complemento de Adobe Reader para poder mostrar el PDF sin tener que integrar paquetes externos en el IDE.

En la figura 5.3 podemos observar parte del código interno del documento PDF generado por el anterior algoritmo. Es posible apreciar cómo en algunos objetos se encuentra:

- El texto que hemos introducido: líneas 68, 73 y 76.
- Las fuentes correspondientes a cada cadena: en las líneas 47 y 48, “F1” corresponde al objeto “7 0” (línea 88) y “F2” al objeto “8 0” (línea 98). En las líneas 66 y 72, “F1” y “F2” se usan para dar formato a las cadenas.
- La tabla XREF (línea 107) comentada en el punto 4.1.1. *Estructura de un archivo PDF*.

```

33 %% Page 1
34 %% Original object ID: 5 0
35 4 0 obj
36 <<
37   /Contents 5 0 R
38   /MediaBox [
39     0
40     0
41     595
42     842
43   ]
44   /Parent 3 0 R
45   /Resources <<
46     /Font <<
47       /F1 7 0 R
48       /F2 8 0 R
49     >>
50   >>
51   /Type /Page
52 >>
53 endobj
54
55 %% Contents for page 1
56 %% Original object ID: 3 0
57 5 0 obj
58 <<
59   /Length 6 0 R
60 >>
61 stream
62 q
63 BT
64 36 806 Td
65 0 -16 Td
66 /F1 18 Tf
67 0 0 1 rg
68 (Soy el título)Tj
69 0 g
70 0 0 Td
71 0 -18 Td
72 /F2 12 Tf
73 (Hola mundo)Tj
74 0 0 Td
75 0 -18 Td
76 (Soy el final)Tj
77 0 0 Td
78 ET
79 Q
80 endstream
81 endobj
82
83 6 0 obj
84 151
85 endobj
86
87 %% Original object ID: 1 0
88 7 0 obj
89 <<
90   /BaseFont /Times-Roman
91   /Encoding /WinAnsiEncoding
92   /Subtype /Type1
93   /Type /Font
94 >>
95 endobj
96
97 %% Original object ID: 2 0
98 8 0 obj
99 <<
100   /BaseFont /Helvetica
101   /Encoding /WinAnsiEncoding
102   /Subtype /Type1
103   /Type /Font
104 >>
105 endobj
106
107 xref
108 0 9
109 0000000000 65535 f
110 0000000052 00000 n
111 0000000133 00000 n
112 0000000349 00000 n
113 0000000458 00000 n
114 0000000697 00000 n
115 0000000903 00000 n
116 0000000950 00000 n
117 0000001085 00000 n
118 trailer <<
119   /Info 2 0 R
120   /Root 1 0 R
121   /Size 9
122   /ID [<9348942c05b262b262b262b262b262>]
123 >>
124 startxref

```

Figura 5.3. Código interno del documento PDF de ejemplo.

5.2. Desarrollo de pequeño software para identificación de cadenas de texto mediante expresiones regulares

Una vez entendido el proceso de generación de un PDF, se ha procedido a la extracción de texto de dicho PDF mediante la biblioteca de iText, así como a la identificación y extracción de una cadena de texto concreta haciendo uso de expresiones regulares [8]. Para poder obtener un número de factura mediante expresiones regulares, configuramos que el texto que preceda a este número (pueda ser la cadena “Nº Factura”) permita identificar qué número de factura corresponde a ese documento PDF.

Llegados a este punto, puede parecer que el uso de este patrón con expresiones regulares, permitiría detectar cualquier número de factura de cualquier documento. No obstante, nada más lejos de la realidad, los documentos PDF (e hilando más fino en este proyecto, los documentos de factura) son tan diversos y pueden llegar a estar tan mal estructurados que sea imposible identificar algo tan aparentemente simple como un número de factura.

Dicho esto, veamos un ejemplo de identificación de número de factura sobre una factura ejemplo mediante expresiones regulares. En este caso, la factura del ejemplo en cuestión (figura 5.4) es una factura de muestra de la empresa de prácticas: Irene Solutions S.L.



IRENE SOLUTIONS, S.L
 B12959755
 PLAZA ESTANY COLOMBRI 3 B
 12530 BURRIANA
 CASTELLON
 964 05 39 29



DOCUMENTO	NÚMERO	PÁGINA	FECHA
Factura	1 000300	1	15/12/2018

N.I.F.	AGENTE	FORMA DE PAGO
██████████		RECIBO DOMICILIADO

ARTÍCULO	DESCRIPCIÓN	CANTIDAD	PRECIO UNIDAD	SUBTOTAL	DTO.	TOTAL
006.001	Cuota mensual asesoria	1,00	39,00	39,00		39,00

TIPO	IMPORTE	DESCUENTO	PRONTO PAGO	PORTES	FINANCIACIÓN	BASE	I.V.A.	R.E.
21,00 10,00 4,00	39,00					39,00	8,19	

OBSERVACIONES:

TOTAL: 47,19

Vencimientos	Importe	Domiciliación	Oficina	Número de cuenta
15/12/2018	47,19	██████████	██████████	██████████

Figura 5.4. Factura ejemplo Irene Solutions S.L.

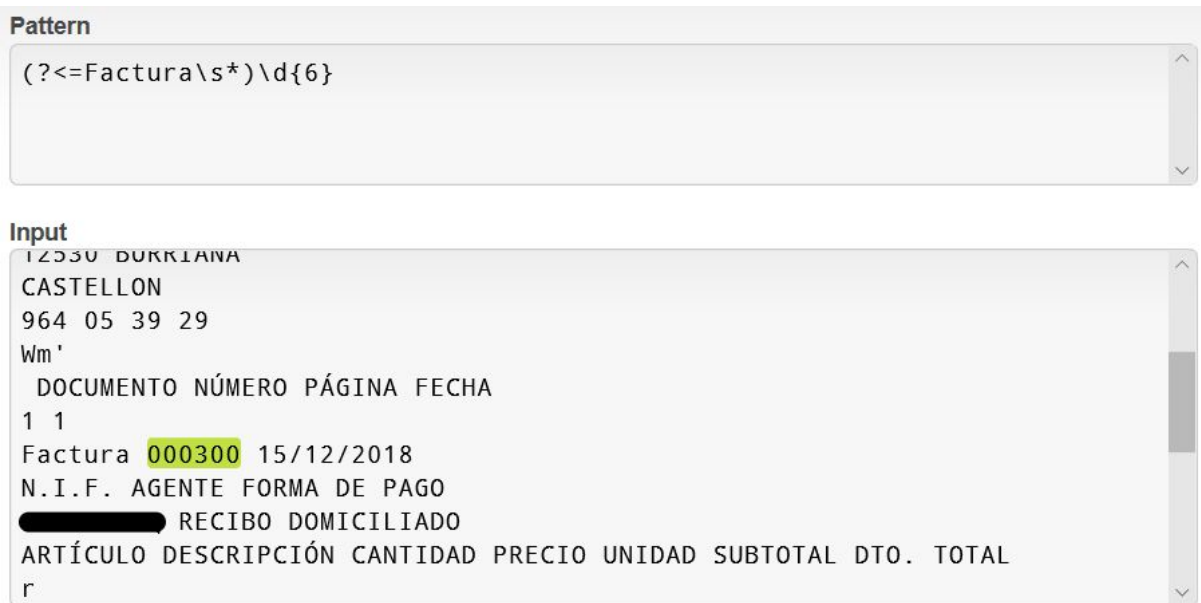


Figura 5.6. Evaluación RegEx del número de factura sobre la factura ejemplo Irene Solutions S.L.

En el anterior ejemplo observamos la expresión regular “`(?<=Factura\s*)\d{6}`”. Mediante esta expresión indicamos que si precede “Factura ” nos devuelva la cifra que le sigue. Esta expresión está compuesta por:

- `?<=` → *Positive look ahead*. Indica “en caso de que preceda” el texto que sigue a la expresión.
- `\s*` → donde el `\s` es el símbolo del espacio y el `*` que puede aparecer ninguna o varias veces.
- `\d{6}` → donde el `\d` simboliza un dígito y el `{6}` sirve para indicar el número de cifras que se va a contener (en este caso 6).

Una vez tenemos la expresión regular mediante la cual podemos detectar este número de factura, es posible detectar los números de factura de todas las facturas que posean la misma estructura (en este caso, las facturas emitidas por Irene Solutions SL). En el momento de cambiar el proveedor de emisión, la validez de la expresión regular sería nula.

En caso de querer averiguar también el total de la factura, emplearíamos la misma metodología de análisis, de manera que obtendríamos lo siguiente (figura 5.7).

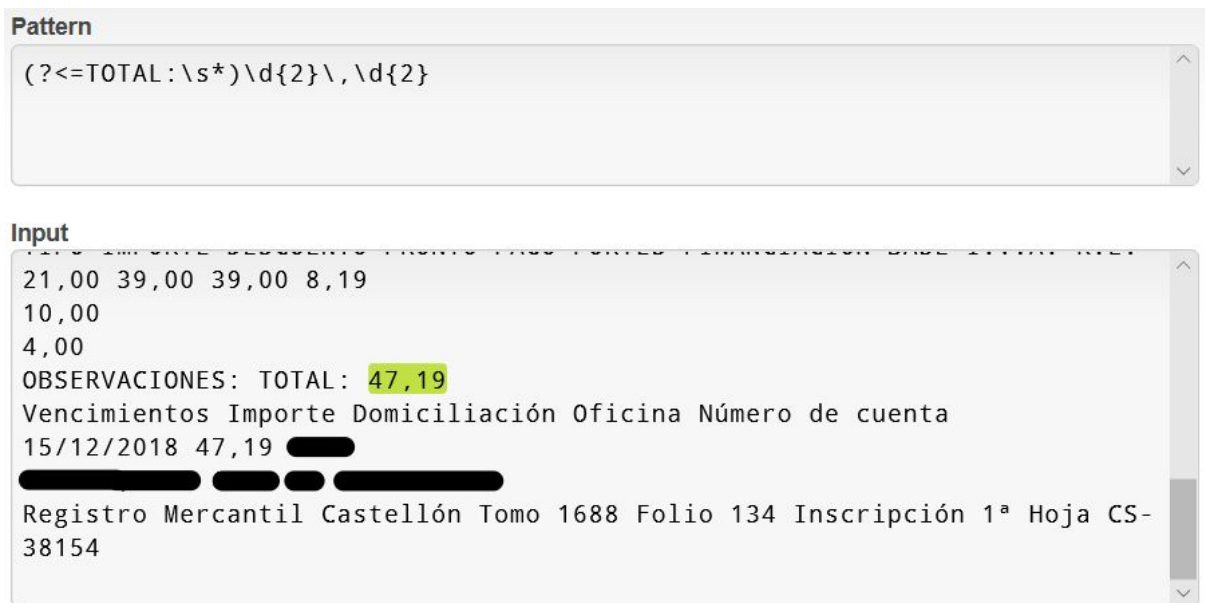


Figura 5.7. Evaluación RegEx del importe total sobre la factura ejemplo Irene Solutions S.L.

Hay que tener en cuenta que, en ambos casos, en el momento que nos pasen una factura donde el número contenga más cifras, las expresiones regulares fallarán y no encontrarán el número. No obstante, mediante una pequeña modificación en ambas expresiones (sustituir `\d{2}\,\d{2}` por `\d*\,\d{2}`) podemos seguir obteniendo el mismo resultado a la vez que nos abrimos a identificar un abanico más grande de resultados (hay que realizarlo con cautela, ya que es posible que en ciertos casos nos arroje un resultado “falso positivo”).

Cómo es posible observar en las siguientes imágenes (figuras 5.8 y 5.9) el cambio surtiría el mismo efecto que el original.

Pattern

```
(?<=TOTAL:\s*)\d*\,\d{2}
```

Input

```
TIPO IMPORTE DESCUENTO PRONTO PAGO PORTES FINANCIACIÓN BASE I.V.A. R.E.
21,00 39,00 39,00 8,19
10,00
4,00
OBSERVACIONES: TOTAL: 47,19
Vencimientos Importe Domiciliación Oficina Número de cuenta
15/12/2018 47,19 ██████████
██████████ ██████████ ██████████ ██████████
Registro Mercantil Castellón Tomo 1688 Folio 134 Inscripción 1ª Hoja CS-
38154
```

Figura 5.8. Evaluación RegEx del importe total sobre la factura ejemplo Irene Solutions S.L con modificación.

Pattern

```
(?<=TOTAL:\s*)\d*\,\d{2}
```

Input

```
Cuota mensual asesoria 1,00 239,00 239,00 239,00
TIPO IMPORTE DESCUENTO PRONTO PAGO PORTES FINANCIACIÓN BASE I.V.A. R.E.
21,00 239,00 239,00 50,19
10,00
4,00
OBSERVACIONES: TOTAL: 289,19
Vencimientos Importe Domiciliación Oficina Número de cuenta
23/12/2018 289,19 ██████████
██████████ ██████████ ██████████ ██████████
Registro Mercantil Castellón Tomo 1688 Folio 134 Inscripción 1ª Hoja CS-
38154
```

Figura 5.9. Evaluación RegEx del importe total sobre otra factura ejemplo Irene Solutions S.L con modificación.

Al igual que este cambio, para conseguir tener una expresión regular fuerte y consistente, habría que tener en cuenta también el separador de miles y el caso referente a que el separador decimal pueda convertirse en un punto en vez de una coma. En este caso, una expresión regular de este tipo sería de esta forma (figura 5.10).

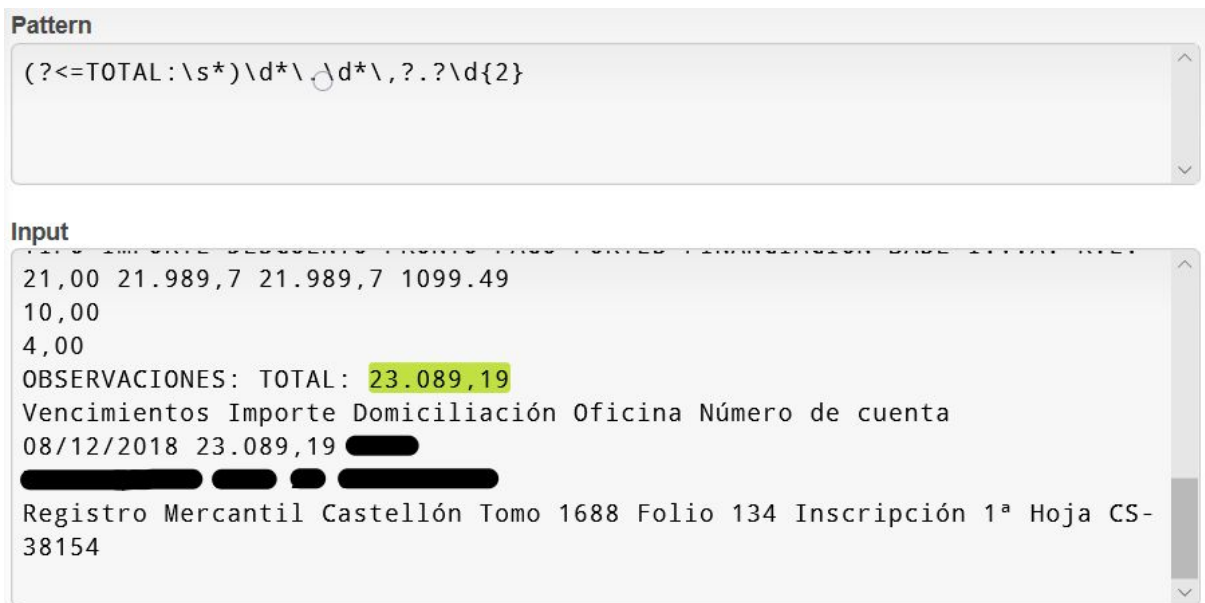


Figura 5.10. Evaluación RegEx del importe total sobre otra factura ejemplo Irene Solutions S.L con generalización.

Teniendo un poco más claro el funcionamiento de las expresiones regulares sobre un texto, podemos ver en funcionamiento un pequeño software desarrollado para su comprensión.

Para el anterior ejemplo de factura, en caso de que siempre tuviéramos que extraer información de facturas de la misma morfología (de Irene Solutions SL) podríamos desarrollar un algoritmo que, dado un documento PDF como parámetro de entrada, extrajera el número de factura, su fecha de emisión y el importe total (parámetros de salida bastante recurrentes en el mundo de gestión empresarial). Este algoritmo se vería de la siguiente manera (figura 5.11).

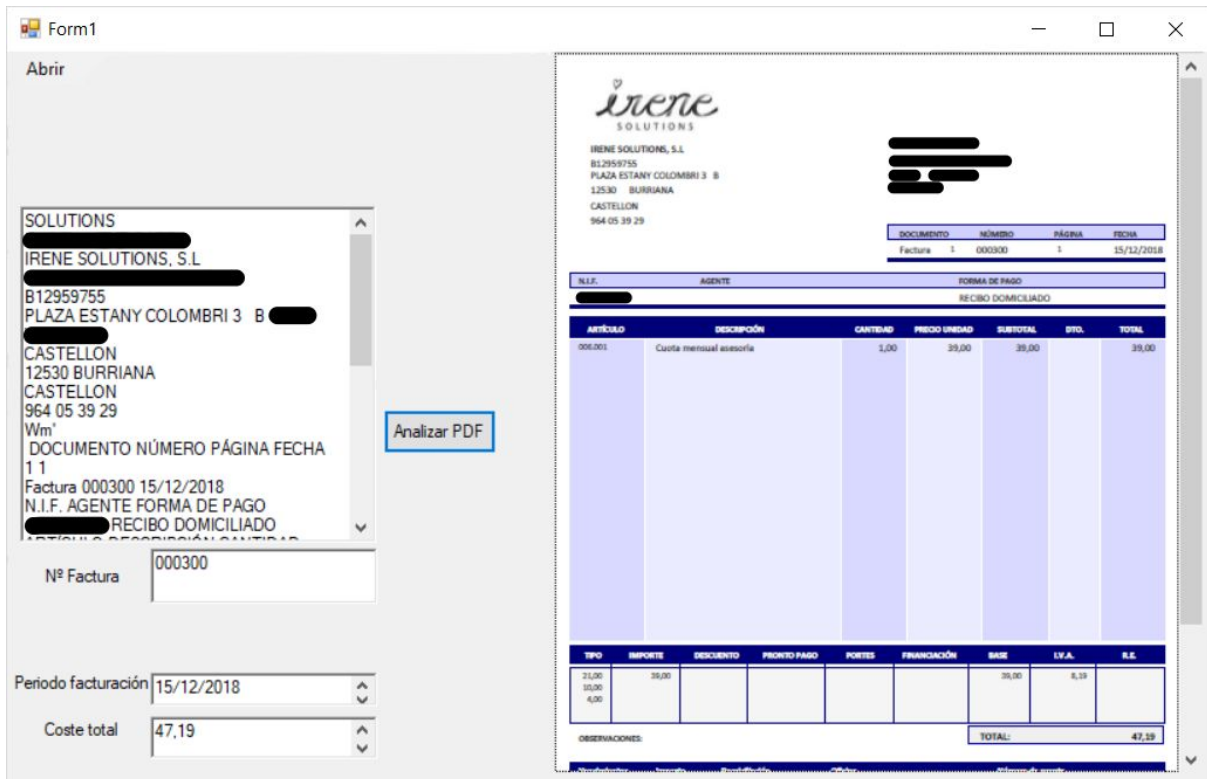


Figura 5.11. Extracción de texto con iText mediante expresiones regulares.

La figura 5.12 muestra parte del código que permite realizar dicha extracción, mediante expresiones regulares. Como se puede observar en el código, usamos algunas clases de iText:

- Por un lado, **PdfReader** nos permite realizar la lectura sobre un fichero PDF.
- Por otra, **LocationTextExtractionStrategy** nos devuelve el texto relativo al PDF, manteniendo la traza y posición que posee el texto sobre dicho fichero. También trata de devolver el texto de manera consistente al modo de lectura del usuario, es decir, ordena el texto: primero por la orientación, luego de manera perpendicular y después por la distancia paralela entre el resto de textos. Siendo más concretos, si un texto posee la misma distancia en perpendicular pero una distancia paralela diferente, se trataría de textos en la misma línea.

```

PdfReader reader = new PdfReader(file);
LocationTextExtractionStrategy strategy = new LocationTextExtractionStrategy();

for (int page = 1; page <= reader.NumberOfPages; page++)
{
    string texto = PdfTextExtractor.GetTextFromPage(reader, page, strategy) + "\n";
    richTextBox1.AppendText(texto);
    ExtraccionRegex(texto);
}
reader.Close();

ExtraccionRegex(string textoExtraido)
{
    string nFactura = @"(?<=Factura\s*)\d{6}";
    Regex exReg = new Regex(nFactura);

    Match coincidencia = exReg.Match(textoExtraido);
    if (coincidencia.Success)
    {
        string match = coincidencia.Groups[0].Value;
        richTextBox2.Text = match + "\n";
    }
}

```

Figura 5.12. Código de la extracción de texto con iText mediante expresiones regulares.

5.3. Desarrollo de pequeño software para identificación de cadenas de texto por posicionamiento de coordenadas

A priori podemos llegar a pensar que la extracción mediante expresiones regulares es bastante buena; sin embargo, como el texto no siempre coincide en las distintas extracciones de un PDF de una misma morfología, es posible que este tipo de patrones falle en su extracción. Es por ello que, una solución más a las problemáticas que se nos presentan a la hora de extraer un texto, podría hallarse en la posición de este texto.

¿Por qué extracción por posicionamiento? Después de observar un número reiterado de veces distintas facturas, se puede llegar a apreciar que en un gran número de este tipo de documentos, muchos de los metadatos que se quieren extraer (la posición del número de factura, la fecha de emisión de esta, su coste total...) suelen recaer en la misma posición del documento. Realizar una extracción por posicionamiento también puede reducir gran cantidad de resultados falsos positivos que la extracción mediante expresiones regulares no logra.

Teniendo esto en cuenta, sobre la factura ejemplo vista en el punto anterior, podemos realizar un desarrollo simple mediante el cual, pasando las coordenadas de lo que queremos extraer, obtengamos el texto contenido en el rectángulo formado por dichas coordenadas. En el software de prueba realizado, las coordenadas de cada rectángulo se introducen directamente en el código; sin embargo, dichas coordenadas podrían pasarse como

entrada de distintas maneras. Por ejemplo, a partir de un trazo rectangular que marcara el usuario con el puntero sobre la interfaz de gestión del PDF.

El resultado de este software se vería de la siguiente manera (figura 5.13).

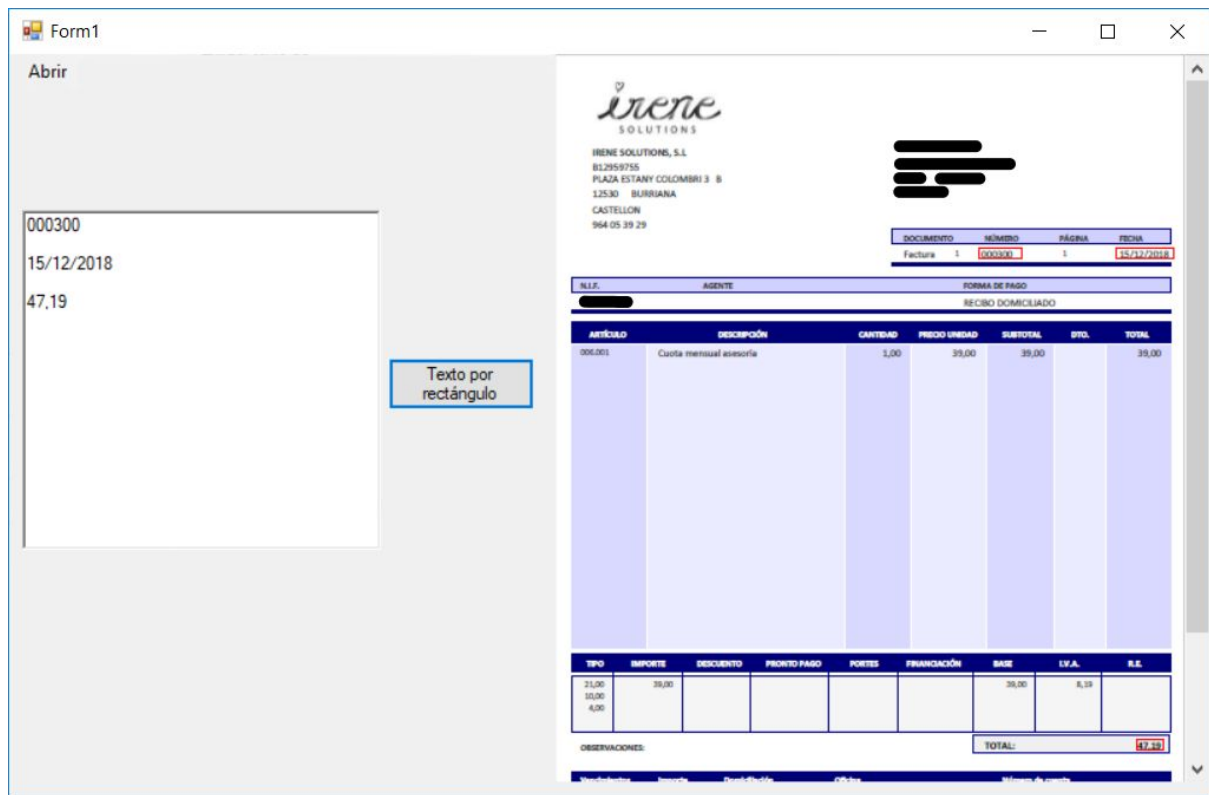


Figura 5.13. Extracción de texto con iText mediante posicionamiento por coordenadas.

La figura 5.14 muestra parte del código que permite realizar dicha extracción. Como podemos observar, en el código utilizamos distintas clases y métodos de la librería de iText. Encontramos:

- **PdfReader**, para poder abrir y leer el documento PDF.
- **PdfStamper**, para poder añadir nuevo contenido a los documentos PDF. En este caso, imprimir sobre un nuevo fichero los rectángulos, donde se localizan los datos que queremos extraer, fusionados con el documento PDF de entrada.
- **PdfContentByte**, lienzo gráfico sobre el que dibujamos los rectángulos.
- **RenderFilter**, definición de filtros sobre el texto.

```

Rectangle fecha = new Rectangle(530, 650, 585, 660);
PdfReader reader = new PdfReader(file);
PdfStamper stamper = new PdfStamper(reader, new FileStream("Rectangled.pdf",
    FileMode.Create));

PdfContentByte canvas = stamper.GetOverContent(1); // Lienzo sobre el que se imprimen los
rectángulos donde se localizan los metadatos a extraer

DrawRectangle(canvas, fecha); // Dibuja los rectángulos sobre el documento PDF

webBrowser1.Navigate("Rectangled.pdf");

RenderFilter filter = new RegionTextRenderFilter(fecha);
ITextExtractionStrategy strategy = new FilteredTextRenderListener(new
    LocationTextExtractionStrategy(), filter);

richTextBox1.AppendText(PdfTextExtractor.GetTextFromPage(reader, 1, strategy));

stamper.Close();
reader.Close();

```

Figura 5.14. Código de la extracción de texto con iText mediante posicionamiento por coordenadas.

Hay que tener en cuenta que, mediante este código, la única limitación a la hora de extraer el texto es que se encuentre en esa posición. Si por alguna razón, en sucesivas iteraciones de documentos de factura de la misma morfología, el texto contenido en dicha posición no coincide con el tipo de metadato esperado, éste devolverá falsos positivos. Una manera de evitar este tipo de falsos resultados sería la combinación de la extracción por posicionamiento con la extracción mediante expresiones regulares: se extraería el texto contenido en una región concreta y, previamente a devolverlo por salida, se realizaría una comprobación mediante expresiones regulares.

Una posible implementación de esta comprobación sería modificar el código antes de añadir el contenido a “richTextBox” con el siguiente fragmento (figura 5.15). El resultado es la combinación de la parte de extracción por coordenadas con la verificación por expresión regular.

```

Regex fechaFac = new Regex(@"\d{2}/\d{2}/\d{4}");
Match comprobación = fechaFac.Match(PdfTextExtractor.GetTextFromPage(reader, 1, strategy));
richTextBox1.AppendText(comprobación.Groups[0].Value);

```

Figura 5.15. Código de la extracción de texto con iText mediante posicionamiento por coordenadas y validación por RegEx.

En la figura 5.16 podemos ver cómo, a pesar de estar detectando los 3 campos de texto, al restringir con la validación de la expresión regular, sólo se obtiene la fecha; con las demás regiones, conforme estaba previsto, no se encuentra ninguna coincidencia.

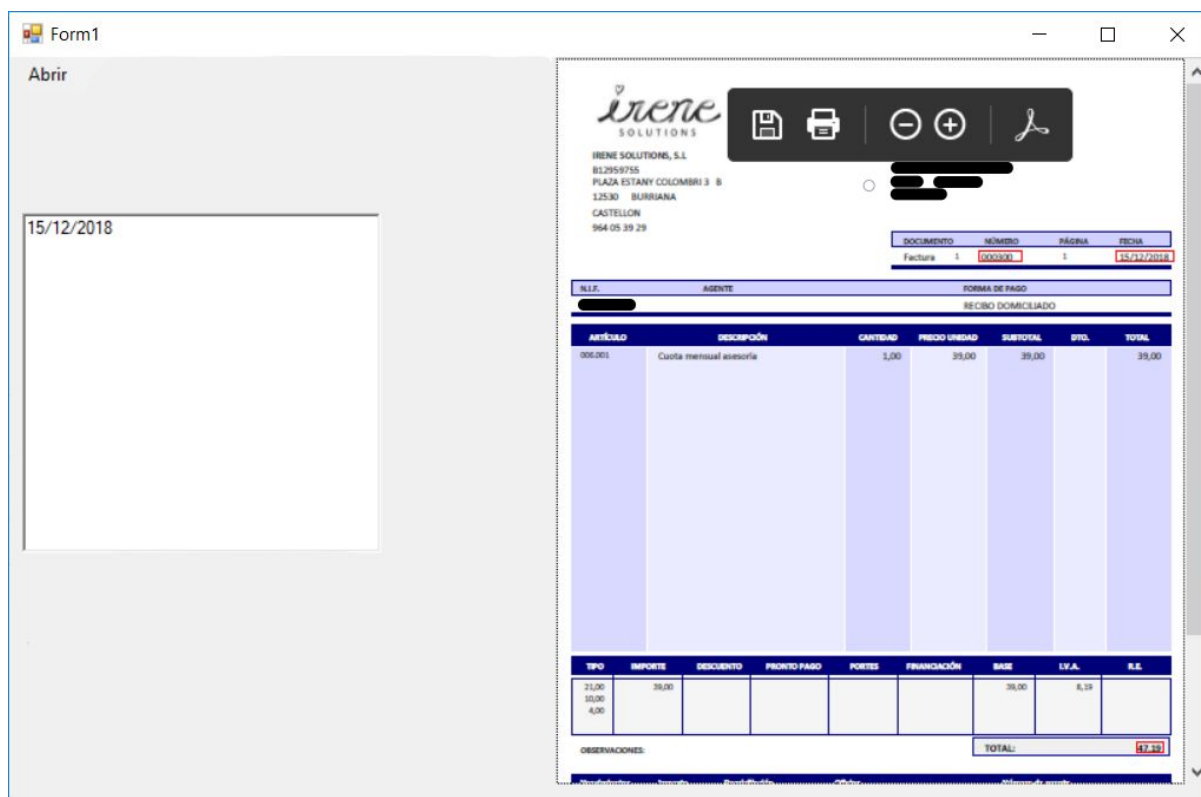


Figura 5.16. Extracción de texto con iText mediante posicionamiento por coordenadas y validación por RegEx (en este caso la fecha).

Para esta restricción por validación, en caso de que el formato de fecha sobre el PDF cambiara (por ejemplo, 15 de diciembre de 2018), habría que utilizar otro tipo de expresión regular genérica que pudiera detectar dicho formato de fecha; de otra forma, en la extracción no sería posible devolver un resultado.

Este tipo de comprobaciones con vistas al cambio de formato de la fecha (y de otros metadatos requeridos en el proceso de extracción) están ya controladas por la biblioteca de PdfTagger, mediante el uso de distintas jerarquías para cada tipo de metadato.

5.4. Estudio y desarrollo de mejora basada en extracción del tipo de fuente del texto

Tras el análisis de la morfología de los distintos documentos de factura que recibe cada día la empresa, en vistas a desarrollar posibles mejoras, se ha podido observar en un gran número de facturas el empleo de distintas tipografías en su contenido. Este hecho ha llevado a la investigación de nuevos métodos de extracción basados en el tipo de fuente del texto.

Esta investigación se ha realizado sobre el código fuente ya presente en las bibliotecas de iText y PdfTagger, con la intención de comprobar hasta qué punto era posible obtener la información de la fuente mediante iText y sobrescribir PdfTagger para hacer uso de esta información en el flujo de extracción.

La biblioteca de iText es capaz de extraer la fuente a partir de los objetos gráficos contenidos en un archivo PDF.

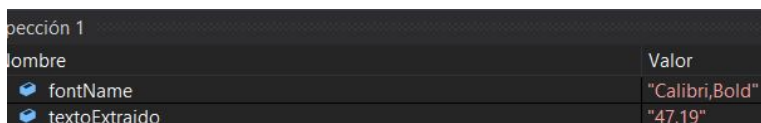
Haciendo uso de sus herramientas, por tanto, comprobamos mediante la factura de ejemplo usada en los puntos anteriores que, efectivamente, se extrae el tipo de fuente que conforma el texto. Para realizar esta comprobación, se realiza una modificación en la clase extractora (PdfTextRectangleTextExtractionStrategy vista en el apartado 4.5. Diagrama de clases de PdfTagger) para poder ver el tipo de fuente de cualquier grupo de texto extraído.

Estas modificaciones consisten en importar en el método “RenderText()” de nuestro extractor no sólo caracteres y sus coordenadas en el documento PDF, sino también el nombre de la fuente (figura 5.17).

```
string fontName = renderInfo.GetFont().PostscriptFontName;  
string textoExtraido = renderInfo.GetText();
```

Figura 5.17. Extracción de fuente del texto de un PDF.

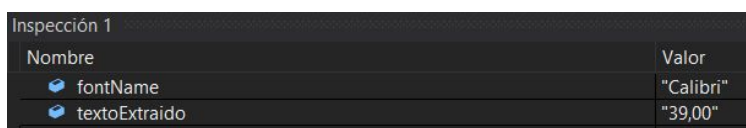
Una vez llevadas a cabo las modificaciones, realizamos una inspección sobre el documento de ejemplo para estudiar los resultados.



Nombre	Valor
fontName	"Calibri,Bold"
textoExtraido	"47,19"

Figura 5.18. Extracción de fuente del texto de un PDF.

En la figura 5.18 podemos observar cómo para el texto “47,19”, correspondiente al coste total de la factura de Irene Solutions S.L., la fuente es Calibri en Negrita. En cambio, en la figura 5.19 podemos observar cómo, si nos fijamos en otro coste que aparezca en la factura, la fuente es sólo Calibri.



Nombre	Valor
fontName	"Calibri"
textoExtraido	"39,00"

Figura 5.19. Extracción de fuente del texto de un PDF.



IRENE SOLUTIONS, S.L
 B12959755
 PLAZA ESTANY COLOMBRI 3 B
 12530 BURRIANA
 CASTELLON
 964 05 39 29



DOCUMENTO	NÚMERO	PÁGINA	FECHA
Factura	1 000300	1	15/12/2018

N.I.F.	AGENTE	FORMA DE PAGO
		RECIBO DOMICILIADO

ARTÍCULO	DESCRIPCIÓN	CANTIDAD	PRECIO UNIDAD	SUBTOTAL	DTO.	TOTAL
006.001	Cuota mensual asesoria	1,00	39,00	39,00		39,00

TIPO	IMPORTE	DESCUENTO	PRONTO PAGO	PORTES	FINANCIACIÓN	BASE	I.V.A.	R.E.
21,00 10,00 4,00	39,00					39,00	8,19	

OBSERVACIONES:

TOTAL: 47,19

Vencimientos	Importe	Domiciliación	Oficina	Número de cuenta
15/12/2018	47,19		8 7 24	



Figura 5.20. Documento de factura usado como ejemplo.

Si analizamos la factura al completo (figura 5.20), se obtiene que ningún coste o número (que no sea el importe total) posee negrita.

De esto se deduce que, para todos los documentos de factura que en su morfología contengan un tipo de fuente en concreto para los metadatos que queramos obtener (y no se repita en el resto del texto), habremos obtenido un patrón a priori bastante robusto a la hora de realizar futuras extracciones. Bastante robusto porque, al realizar la extracción, si incluimos esta información en las propiedades del patrón, las extracciones apuntarán directamente a este texto, independientemente de la posición del dato o el texto que le preceda, quitando de esta manera una gran cantidad de ruido de las extracciones.

En el caso de que hubiera más de un dato con el mismo tipo de fuente, con la comprobación por expresión regular para un metadato concreto, estaríamos también filtrando posibles falsos positivos, por lo que estaríamos obteniendo igualmente un patrón robusto.

Sin embargo, una vez desarrollado el código que ha permitido ejecutar este tipo de extracción, en el momento de iterar sobre un gran conjunto de facturas (donde encontramos distintos tipos de morfologías, mejor o peor realizadas) se observa cómo esto no es tan sencillo. En el escenario actual, los distintos documentos de factura no presentan una gran elaboración a nivel de diseño. Esto supone que, en algunos casos, se va a encontrar:

- Ambigüedad con el tipo de fuente. La misma fuente presente a lo largo de toda la factura; por lo que el principal factor de robustez del patrón se pierde.
- Superposición del mismo texto. Con tal de realizar un trozo de texto en negrita, muchos diseños emplean la superposición del mismo texto, consiguiendo así que la letra aparezca más gruesa. Esto supone una problemática de cara a la extracción, ya que aparece el mismo dato distintas veces sin realmente aportar ningún tipo de información relevante para su extracción.
- Tipo negrita circunscrito a valores gráficos. La propiedad de negrita no estaría presente en la definición de la fuente, se trataría de dibujo gráfico, por lo que no sería detectable por esta vía de extracción.

Llegados a este punto, se podría pensar que la solución a la problemática presente pasa por combinar este tipo de patrón con las coordenadas del texto sobre el PDF. Sin embargo, una de las razones que ha llevado a desarrollar este patrón tiene que ver con la razón de ser de documentos de factura con líneas dinámicas. La naturaleza de esta morfología hace imposible que se pueda llegar a detectar con éxito la posición del coste total. Esto es debido a que, dependiendo del número de filas de productos que posee la factura, el coste total se situará en un distinto nivel de la coordenada de las Y (la coordenada de las X permanecería inmutable en muchos casos). En caso de combinar la posición con el tipo de

fuelle, si la fuente es la misma en todo el documento, el valor de esta propiedad nos sería irrelevante.

Por otro lado, combinar esta propiedad con la extracción por expresiones regulares no aportaría ningún tipo de información adicional. Esto es debido a que, en el caso de la extracción por patrones de expresiones regulares, esta extracción se realiza sobre todo el texto de una página del PDF (podría verse como tratado como un único grupo de texto), no en distintos grupos de texto/palabras como la extracción por posición; lo que conlleva a la pérdida de la información del tipo de fuente, al combinarse todo el texto en un mismo grupo de texto.

En síntesis, este método de extracción nos ha permitido analizar conjuntos de facturas con un mayor grado de detalle en muchos casos donde las extracciones anteriores no han llegado, apuntando directamente al dato precisado de extracción y eliminando una gran parte del ruido que pueda encontrarse. Sin embargo, al igual que en muchos casos se ha conseguido un patrón más robusto que en otros tipos de extracciones, se ha podido observar documentos de facturas donde el patrón presentaba casuísticas muy genéricas y no ha aportado ningún tipo de información. Este hecho nos lleva a la búsqueda de nuevos métodos de extracción, donde podamos utilizar de un mayor número de propiedades presentes en los objetos de la estructura del PDF.

5.5. Estudio y desarrollo de mejora basada en extracción por tipo, tamaño y color de fuente en PdfTagger empleando PdfClown

Basándonos en el análisis de distintos documentos PDF, podemos observar cómo muchos de ellos poseen en su morfología varias estructuras de texto con diferentes tamaños de fuente o colores.

Un claro ejemplo de este tipo sería el siguiente documento de factura (figura 5.21).

Documento emitido por IBERDROLA CLIENTES, S.A.U. - domicilio fiscal: C/ Tomate Redondos 1, 28033 Madrid domicilio social: Plaza Euzkadi 5, 48009 Bilbao. inscrita en el Registro Mercantil de Bilbao, tomo 5448, folio 19, hoja B-43981, inscripción 1ª - CIF A-95758389

IBERDROLA CLIENTES, S.A.U.
CIF A-95758389



IBERDROLA

FACTURA DE ELECTRICIDAD

Página 1 / 3

DATOS DE FACTURA

Periodo de facturación 03/12/2018 – 01/01/2019
Número de factura 21190102010320280
Fecha de emisión de factura 2 de enero de 2019
Fecha prevista de cargo 02/01/2019
Factura con lectura real
Titular [REDACTED]
CIF titular [REDACTED]
Referencia contrato suministro [REDACTED]

Remite: IBERDROLA CLIENTES, S.A.U. Apartado de Correos 61175 28080 Madrid

IN 999 M 5 0639148451 0 1 08 S110 003605 007589 20190102



06391484510023999010130105900630102019

TOTAL IMPORTE FACTURA: 435,83 €

Dirección de suministro: [REDACTED]

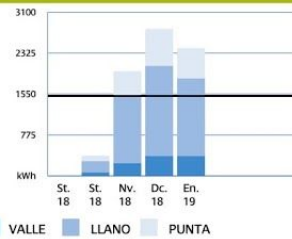
RESUMEN DE FACTURACIÓN

ENERGÍA	354,45 €
SERVICIOS Y OTROS CONCEPTOS	5,74 €
IVA 21% s/360,19 €	75,64 €

TOTAL A PAGAR 435,83 €

> ver detalle de facturación y consumo en el reverso

EVOLUCIÓN DE CONSUMO



Este gráfico muestra la evolución de su consumo.
Su consumo medio diario en este último periodo facturado ha sido: 15,02 €
Su consumo medio diario en los últimos 4 meses ha sido: 12,86 €



El apartado DATOS RELACIONADOS CON SU SUMINISTRO recoge toda la información necesaria para conocer las características y datos de su contrato.



Atención al Cliente: Consultas, gestiones y reclamaciones



Su gestor personal [REDACTED]



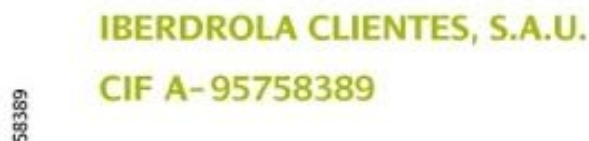
Atención Averías de Red: [REDACTED]



www.iberdrola.es

Figura 5.21. Documento de factura de Iberdrola.

En este documento de factura podemos encontrar distintos datos susceptibles de extracción, con diferentes tonalidades y tamaños respecto al resto del documento (figuras 5.22, 5.23 y 5.24).



50389
IBERDROLA CLIENTES, S.A.U.
CIF A-95758389

Figura 5.22. CIF del documento de factura de Iberdrola.



Referencia contrato suministro [REDACTED]
TOTAL IMPORTE FACTURA: 435,83 €

Figura 5.23. Coste total 1 del documento de factura de Iberdrola.



IVA 21% s/360,19 € 75,64 €
TOTAL A PAGAR 435,83 €

Figura 5.24. Coste total 2 del documento de factura de Iberdrola.

En las imágenes anteriores se observa que, por ejemplo, el CIF del proveedor posee una tonalidad verde. Por otro lado, en el caso de los totales de la factura, podemos encontrar uno con tonalidad blanca y otro verde. Además, ambos tienen un tamaño mayor que el resto del texto.

Según estas evidencias, en caso de conseguir extraer las propiedades referentes al color y al tamaño de la fuente, podríamos estar delante de una gran herramienta a la hora de ejecutar la extracción de metadatos.

En una primera instancia, según los conocimientos adquiridos del estudio de las herramientas hasta este punto, se han encontrado limitaciones en la biblioteca de iText. Después de analizar el código fuente dedicado a la extracción de texto, no se han hallado referencias a la obtención del color y tamaño de la fuente. Al parecer, la máxima información que la biblioteca permite extraer de los objetos de un archivo PDF (que nos pueda resultar interesante en nuestro trabajo) reside en la posición y la fuente como elementos característicos. Probablemente se deba a un mayor enfoque, por parte de los desarrolladores de la biblioteca, en el proceso de creación del PDF que en su posterior extracción (según se

ha podido analizar, el funcionamiento de estos dos flujos por parte de la biblioteca es completamente distinto).

Debido a este factor, se ha procedido a buscar otra biblioteca Open Source que permita obtener este tipo de información en las extracciones. Esta búsqueda ha sido complicada, debido a la poca dedicación que se le da al código abierto en este ámbito de la gestión empresarial.

Una de las bibliotecas que más se ha acercado al referente impuesto ha sido la de PdfClown [10]. La razón de decantarnos por esta biblioteca reside en uno de los ejemplos de ejecución que proporcionan en su repositorio (figura 5.25).

```
12 namespace org.pdfclown.samples.cli
13 {
14     /**
15      <summary>This sample demonstrates how to retrieve text content along with its graphic attributes
16      (font, font size, text color, text rendering mode, text bounding box...) from a PDF document;
17      it also generates a document version decorated by text bounding boxes.</summary>
18     */
19     public class TextInfoExtractionSample
20         : Sample
21     {
```

Figura 5.25. Extracto del Sample de TextInfoExtraction de PdfClown.

Dada esta descripción (aunque en el código de ejemplo no hay rastro de la extracción del color) parece interesante analizar el código fuente.

Si en PdfTagger denominábamos “WordGroups” a las agrupaciones de texto y “PdfTextChunks” a su primitiva (visto en el *apartado 4.5. Diagrama de clases de PdfTagger*), en PdfClown pasan a llamarse “TextStrings” y “TextChars” respectivamente. Estos “TextStrings” se encuentran dentro de la clase “TextStringWrapper”, donde encontramos las propiedades que nos interesan. Algunas de ellas son:

- *font*, de tipo Font (clase que posee el nombre de la fuente).
- *fontSize*, para obtener su tamaño en double.
- *fillColor* y *strokeColor*, de tipo Color (dependiendo de cómo se haya creado el documento PDF se rellena uno u otro).

Con ello, se muestra el resultado de una pequeña implementación donde, efectivamente, mediante esta biblioteca se puede extraer el tipo, color y tamaño de la fuente (figura 5.26).

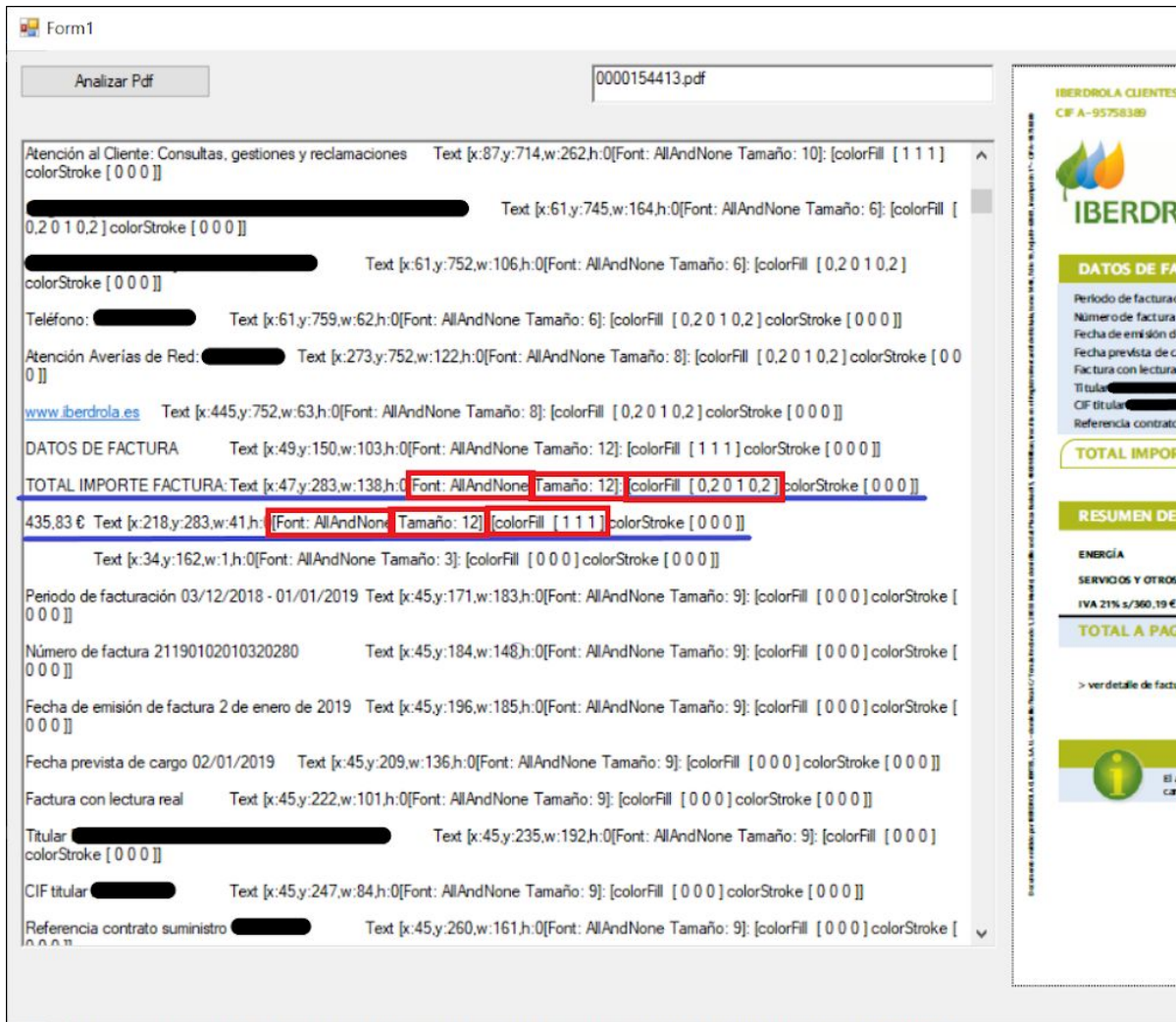


Figura 5.26. Ejemplo de implementación de la biblioteca de PdfClown.

En este documento en concreto, vemos como la fuente utilizada (AllAndNone) no varía a lo largo de todo el texto; lo cual, en este caso, no aporta mucha información.

Sin embargo, encontramos característico el hecho de que la línea del importe de factura posea un tamaño mayor al del resto del texto; permitiendo poder apuntar directamente a dicho importe en la extracción.

Por otra parte, también encontramos como punto diferencial que los TextStrings correspondientes a “TOTAL IMPORTE FACTURA: ” y “435,83€” tienen un color distinto al del resto del documento. En el caso de la figura 5.26, se ha señalado la salida para el TextString de la cifra del coste total de factura de color blanco; no obstante, también encontramos unas líneas más abajo en la salida del programa (apreciable en la captura de la factura a la derecha) el TextString correspondiente al coste total de color verde.

Llegados a este punto, es posible que nos encontremos delante de una herramienta de extracción potencialmente más rica en información referente al texto que la extracción por

posición o por delimitaciones contextuales, vistas en anteriores puntos de este documento. Teniendo en cuenta, en el momento de realizar una extracción, propiedades como el color, el tamaño y el tipo de fuente, es posible apuntar directamente a un dato sin necesidad de requerir posicionamiento frente al documento (ya sean coordenadas o cadenas que preceden al dato), y evitar de esta manera devolver resultados falsos positivos.

Por ejemplo, gracias a la inclusión de esta función, en un documento PDF sobre el que el número de filas sea determinante en la posición correspondiente al coste total, será posible extraer este coste total en todas las facturas en las que una de las propiedades comentadas sea característica frente al resto del documento.

Con todo esto, es posible solucionar muchas situaciones complejas, como puede ser revisar individualmente cada factura en busca de dicho dato al no poder extraerse por anteriores métodos.

5.6. Extracción por tipo, tamaño y color de fuente en PdfTagger con iText

Con el desarrollo del punto anterior, estamos añadiendo una nueva dependencia a la biblioteca de PdfTagger, con todo lo que ello conlleva. Poder aplicar dicha funcionalidad a la biblioteca de iText podría ahorrarnos costes a nivel de computación, líneas de código, etc.

Estudiando el código de iText más a fondo, se puede hallar algunas propiedades que permiten realizar un funcionamiento similar.

En concreto, añadiendo unas líneas de código en el método “RenderText()” de nuestro extractor podemos extraer el color de la fuente. Por otra parte, para obtener el tamaño de ésta podemos emularlo cogiendo la altura del rectángulo que conforma un “PdfTextChunk” (véase el *apartado 4.5. Diagrama de clases de PdfTagger*). Algunas de las modificaciones a realizar para obtener estas propiedades son las siguientes (figura 5.27).

```
string fillColor = renderInfo.GetFillColor()?.ToString(); // Color del texto.
string strokeColor = renderInfo.GetStrokeColor()?.ToString(); // Color del texto.

Vector curBaseline = renderInfo.GetBaseline().GetStartPoint();
Vector topRight = renderInfo.GetAscentLine().GetEndPoint();
iTextSharp.text.Rectangle rect = new iTextSharp.text.Rectangle(curBaseline[Vector.I1],
curBaseline[Vector.I2], topRight[Vector.I1], topRight[Vector.I2]);

double fontSize = Math.Round(rect.Height); // Tamaño de la fuente a partir del rectángulo.
```

Figura 5.27. Extracto de la clase TextRenderInfo de iText.

Sintetizando estas últimas líneas, ahora hemos podido emular la funcionalidad de la biblioteca de PdfClown, de manera que con iText estaríamos obteniendo también el color, tamaño y tipo de fuente empleada en cada uno de los “WordGroups” de un documento PDF. De esta manera, podemos dejar el desarrollo mediante PdfClown y seguir utilizando iText como dependencia única.

5.7. Rutina de eliminación de resultados falsos positivos

Llegados a este punto del desarrollo de las mejoras, se ha podido estudiar un gran conjunto de distintos documentos de factura. El resultado de este estudio implica haber revisado las distintas salidas que ha devuelto nuestra aplicación para cada uno de los documentos PDF.

En el transcurso de este estudio se ha observado que, por más que hayamos afinado las metodologías empleadas en la rutina de extracción, nos encontramos todavía con una gran cantidad de resultados incorrectos. Estos resultados incorrectos son los denominados falsos positivos. Un falso positivo implica un resultado potencialmente correcto el cual, al ser revisado, se comprueba que no lo es. Fruto de realizar distintas iteraciones en el flujo de aprendizaje del software, la biblioteca aprende patrones que permiten posteriormente extraer el texto mediante el flujo de extracción. En el transcurso de las iteraciones del flujo de extracción, la gran variedad de facturas que pueden pertenecer a lo que entendemos por una misma morfología produce que, en ocasiones, se extraigan resultados incorrectos a pesar de todas las validaciones desarrolladas; ya sea porque en un caso concreto el patrón se vuelve excesivamente genérico o porque un dato (un importe total) posee la misma estructura que otro (un importe de un solo producto de la factura). Hasta el momento, la biblioteca ha realizado el flujo de aprendizaje de manera que, cuando se ha encontrado un valor que coincide con el que el patrón tiene almacenado en ese momento, se acumula una unidad al contador MatchesCount; dándole a entender a la biblioteca que el patrón está aumentando su efectividad para sucesivas extracciones. Este funcionamiento implica que, en el caso de que la factura posea un valor factible y otro no factible pero de misma estructura, cuando la biblioteca ejecute el flujo de extracción, nos devuelva el valor no factible porque para el patrón aparece antes dicho valor. Dado que este problema es bastante frecuente, solventarlo puede agilizar bastante el trabajo con la biblioteca.

En base a lo descrito anteriormente, se ha llegado a la conclusión de que lo más interesante es incorporar a los patrones un contador de errores, denominado “ErrorsCount”. Para poner este contador en práctica, no obstante, hay que incluir un nuevo flujo de ejecución a la biblioteca. La necesidad de desarrollar un nuevo flujo de ejecución radica en la imposibilidad de incluir la funcionalidad en los ya presentes. Para poder acumular una unidad a este ErrorsCount, necesitamos tener una entrada de metadatos correctos con la que comparar si un patrón está extrayendo un resultado falso positivo. Para saber si un resultado es correcto, éste debe haber sido revisado por un usuario físico, por lo que (actualmente) no

es posible llevarlo a cabo de manera automatizada, como sí ocurre en el flujo de extracción. Además el flujo de extracción no recibe metadatos por entrada, sólo recibe el documento PDF. Por otra parte, tampoco es posible llevarlo a cabo mediante el flujo de aprendizaje, ya que en este flujo (a pesar de que sí que se reciben metadatos como entrada) se crean los patrones para su almacenamiento y no se utilizan para la extracción de texto, por lo que no tenemos aún la constancia de qué resultado van a devolver.

Dicho esto, se describe el funcionamiento del flujo desarrollado, al que se denomina como flujo de comprobación (figura 5.28). Este flujo debe actuar como un flujo de extracción, el cual se ejecute entre la etapa de aprendizaje y la de extracción. En este flujo se recibe como entrada metadatos que introduce el usuario (por lo que están validados, al igual que en el flujo de aprendizaje) y el documento PDF. Seguidamente se realiza una extracción de los metadatos con los patrones ya aprendidos en las iteraciones del flujo de aprendizaje. En este punto, comprobamos si el texto extraído por el patrón se corresponde con el que hemos pasado por entrada. En caso de que estos dos textos no coincidan, se actualiza el contador ErrorsCount de los patrones acumulando una unidad. Así, cuando se vaya a realizar un flujo de extracción automatizado, se realizan las pertinentes comprobaciones no solo al MatchesCount, sino también al ErrorsCount. Estas comprobaciones se realizan en la llamada a la ordenación de la lista de patrones de cara a posicionarlos por una mayor tasa de aciertos.

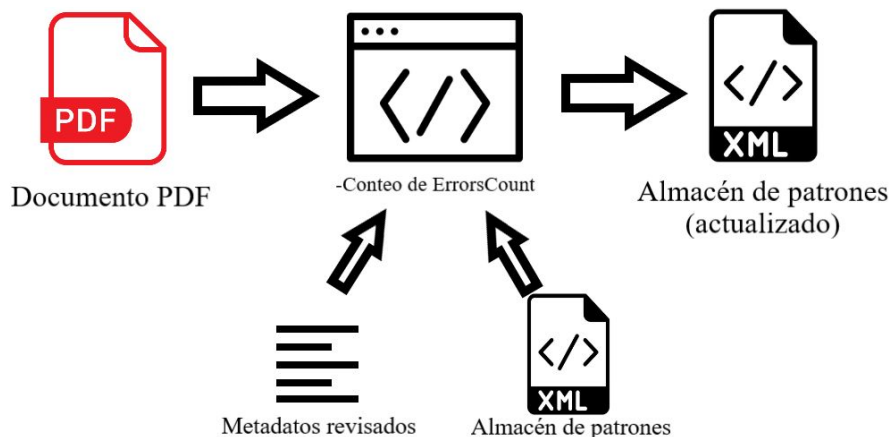


Figura 5.28. Flujo de comprobación.

Hasta ahora la biblioteca tenía implementada la ordenación de patrones de la siguiente manera (figura 5.29). En esta ordenación, si el MatchesCount de un patrón es mayor que otro, se posiciona primero; en caso contrario, se posiciona detrás.

```

if (MatchesCount > input.MatchesCount)
    return -1;
else
    return 1;

```

Figura 5.29. Método original usado en la ordenación de patrones.

Sin embargo, al haber añadido el contador de errores en cada patrón, esta ordenación carece de sentido. En el anterior código se estaría perdiendo el valor añadido de este contador. Es por ello que este método se ha modificado quedando de la siguiente manera (figura 5.30).

```
if ((MatchesCount - ErrorsCount) > (input.MatchesCount - input.ErrorsCount))
{
    return -1;
}
else if ((MatchesCount - ErrorsCount) == (input.MatchesCount - input.ErrorsCount))
{
    if (MatchesCount > input.MatchesCount)
        return -1;
    else
        return 1;
}
else
    return 1;
```

Figura 5.30. Método modificado, usado en la ordenación de patrones.

En el anterior código se observa como, ahora, se realiza una batería de comprobaciones más estricta.

- En primer lugar, partiendo de la diferencia de ambos contadores, si el valor de de la primera de las diferencias es mayor que el de la otra, significa que el primer patrón falla un menor número de veces (o acierta un mayor número de veces) que el segundo, por lo que tiene una mayor probabilidad de devolver un dato correcto.
- En caso de que la diferencia entre contadores de ambos patrones sea la misma; en segundo lugar, si el primer patrón posee una mayor tasa de aciertos que el segundo, también debería posicionarse el primero.
- Si las 2 anteriores condiciones fallan, puede significar que o poseen el mismo valor para los 2 contadores, o el segundo patrón es mejor que el primero, por lo que el segundo patrón debe posicionarse antes.

Mediante este nuevo desarrollo, se permite hilar más fino de cara a ordenar los patrones con un mayor coeficiente de resultados positivos. Dado que, aunque un patrón posea un alto valor de MatchesCount, si éste también comporta un alto valor de errores, no debe tomarse como un patrón a tener en cuenta en las extracciones; en tal caso se estaría ejecutando en una morfología demasiado genérica para las propiedades que posee.

5.8. Iteración de patrones sobre la página correspondiente

Con el transcurso de la estancia, se han detectado algunas partes de código susceptibles de refactorización. Una de ellas hace referencia a los patrones.

Hasta ahora, cuando se realiza el flujo de aprendizaje los patrones se aprenden de todas las páginas del documento PDF. No obstante, cada patrón es individual. En caso de que un mismo texto susceptible de extracción se encuentre en dos páginas distintas, se generan dos patrones (lo más probable es que el contenido de las propiedades de estos dos patrones sea distinto a la hora extraer la información del texto).

Por otra parte, en el flujo de extracción, a pesar de que cada patrón es individual y pertenece a un lugar concreto, se cargan todos los patrones en todas las páginas. Esto provoca que, en caso de tener almacenados una gran variedad de patrones para un tipo de morfología de documento PDF, cada patrón de este almacén se cargue para cada página y, por tanto, muchos patrones fallen en su extracción o, incluso, devuelvan falsos positivos. Además, sobrecarga los bucles de procesamiento innecesario donde, por ejemplo, en un almacén de 1300 patrones puede que sólo hiciera falta ejecutar 80 en una página en concreto.

En caso de desarrollar algún proceso que permitiese cargar los patrones sobre la página correspondiente, estaríamos aliviando en gran medida el procesamiento del bucle de extracción.

Llegados a este punto, procedemos a detallar el desarrollo realizado.

En la ejecución del software, se obtiene como entrada un documento PDF. Sobre este documento se realiza una extracción del texto que contiene. Seguidamente, sobre esta extracción, se realiza una serie de instrucciones que permiten generar los patrones correspondientes a los metadatos que se precisen extraer.

Como hemos comentado en el *apartado 4.5. Diagrama de clases de PdfTagger*, al guardar un patrón, guardamos el número de página que le corresponde en la variable “PdfPageN”. El valor de esta variable se obtiene del número de página perteneciente a su posición en el listado del objeto “PdfUnstructuredPages”, el cual contiene todas las páginas extraídas de un documento PDF.

Cuando un patrón se ha generado en el flujo de extracción haciendo uso de varios métodos, entre ellos los anteriores, se almacena en un documento XML, clasificado por tipo de morfología de documento PDF. En este XML, se reflejan todas las propiedades pertenecientes a cada patrón según el tipo de extractor usado. En el extracto de la figura 5.31 podemos observar un patrón donde se refleja cómo se guarda la página.

```

<PdfTagPattern>
  <MetadataItemName>InvoiceNumber</MetadataItemName>
  <PdfPageN>1</PdfPageN>
  <IsLastPage>>false</IsLastPage>
  <PdfRectangle>
    <Llx>468.114</Llx>
    <Lly>676.923035</Lly>
    <Urx>535.944031</Urx>
    <Ury>687.098</Ury>
  </PdfRectangle>
  <RegexPattern>[A-Z]{2} \d{2}/\d{4}</RegexPattern>
  <Position>0</Position>
  <SourceTypeName>ColorFontWordGroupsInfos</SourceTypeName>
  <MatchesCount>5</MatchesCount>
  <ErrorsCount>0</ErrorsCount>
  <FillColor>Color value[FF000000]</FillColor>
  <StrokeColor>Color value[FF000000]</StrokeColor>
  <FontName>Helvetica</FontName>
  <FontSize>8</FontSize>
  <CFTYPE>NA</CFTYPE>
</PdfTagPattern>

```

Figura 5.31. Extracto de un XML perteneciente a un tipo de morfología de documento PDF.

Teniendo todo esto, ahora en el flujo de extracción, una primera solución planteada ha sido comprobar para cada patrón iterado, si el “PdfPageN” coincide con el de la página que se está iterando en ese momento.

Sin embargo, después de analizar la mejora sobre varios documentos PDF, se detectó una manera de mejorar esta implementación. Esta mejora consiste en generar un diccionario a partir del listado de patrones de “PdfTagPatternStore”. En dicho diccionario, cada clave se correspondería con un número de página y su valor, una lista de patrones que pertenezcan a dicha página. La implementación se ha desarrollado de esta manera (figura 5.32).

```

PdfPatternsPage = new Dictionary<int, List<PdfTagPattern>>();

foreach (PdfTagPattern pattern in PdfPatterns) // Evitar que los bucles de extracción
recorran siempre todos los patrones independientemente del número de página.
{
    if (PdfPatternsPage.ContainsKey(pattern.PdfPageN))
        PdfPatternsPage[pattern.PdfPageN].Add(pattern);
    else
        PdfPatternsPage[pattern.PdfPageN] = new List<PdfTagPattern>() {pattern};
}

```

Figura 5.32. Extracto de la clase PdfTagPatternStore.

Con el primer desarrollo de la mejora, el bucle recorría todos los patrones y, en caso de no encontrarse en la página, no se realizaban las comprobaciones. De esta manera, al usar

un diccionario previa extracción, los bucles solamente recorren los patrones de la página correspondiente, por lo que se trata de un desarrollo más eficiente.

5.9. Identificación e implementación de nuevos tipos de patrones

Como comentamos en el *capítulo 4*, para realizar las comparaciones entre el texto extraído por el patrón y el metadato pasado por entrada, se establecen unas jerarquías de comparación. En estas jerarquías, dependiendo del documento PDF que se pase por entrada, se cargará una jerarquía en concreto, de manera que se carguen los conversores correspondientes a los metadatos que se quieren extraer. Para los casos de ejemplo que exponemos a lo largo de este documento, se cargan las jerarquías correspondientes al “BusinessHierarchySet”, al tratarse de documentos de gestión documental de empresas. Por otra parte, al tratarse en concreto de un documento de factura, se carga la “InvoiceHierarchySet”. Esta clase hereda de la mencionada anteriormente, ampliando los conversores necesarios para metadatos que sólo pertenezcan a las facturas.

El contenido de estas jerarquías son conversores anidados a un tipo de expresión regular. La finalidad de estos conversores es permitir una comparación exitosa entre datos que tengan el mismo significado. Por ejemplo, en el caso de que pasemos un documento PDF por entrada y queramos que la biblioteca aprenda una fecha, nosotros le indicaremos como metadato de entrada “29/03/2019”. Sin embargo, en el momento de extraer el dato, es posible que en el documento PDF se encuentre escrito de la manera “29 de marzo de 2019”. Mediante la aplicación de estas jerarquías de conversión, la fecha con separadores en formato “de” se convierten a “/” y los meses se convierten a número, permitiendo que el sistema luego pueda realizar la comparación en su formato primitivo (en este caso en el formato DateTime).

En el transcurso de la estancia se han ido detectando componentes del texto que aún no estaban parametrizados por la biblioteca para poder ser convertidos y comparados correctamente. En concreto, tomando como ejemplo las fechas del párrafo anterior, se pueden detectar formatos de fecha para los que la librería no estaba preparada. En caso de aparecer una fecha con el formato “2019/03/29”, el sistema no la detectaba. Indicando a la librería la expresión regular correspondiente a este formato “`\d{4}\d{2}\d{2}`” y que se ha de convertir a DateTime, en el momento en que el sistema se encuentre con este formato de fecha sobre el texto, será capaz de convertirlo a dicho formato y compararlo con el metadato pasado por entrada de manera exitosa.

Capítulo 6

Validación y pruebas

En este apartado se detallan el conjunto de pruebas realizado a cada desarrollo con tal de verificar si se trataba de una mejora factible o no; así como la mejora real sobre el funcionamiento de la biblioteca inicial.

6.1. Conjunto de muestra

Para poder evaluar los desarrollos realizados en un entorno lo más realista posible, se ha establecido un conjunto de muestra de 10.000 documentos de factura distintos. Sobre estos 10.000 documentos de factura, se han obtenido 2.599 morfologías de factura distintas, por lo que se trata de un conjunto de muestras bastante diverso.

Por otra parte, se ha obtenido un fichero en formato “.csv”, con estas columnas:

- **PdfID**: El nombre del fichero PDF.
- **BuyerPartyID**: El NIF del proveedor.
- **InvoiceNumber**: El número de factura del documento.
- **IssueDate**: La fecha de facturación del documento.
- **GrossAmount**: El importe total.

Nos hemos focalizado en estos datos porque, entre tan gran número de documentos, encontraríamos muchas carencias en caso de querer añadir más propiedades. Esto es debido a que cada proveedor crea el documento de manera distinta, no se ajustan a ningún estándar.

La finalidad de este documento es servir como metadatos de entrada en el flujo de aprendizaje. De modo que, con este archivo y con el conjunto de facturas anterior, tendríamos los dos objetos de entrada para realizar el flujo de aprendizaje.

6.2. Procedimiento de las pruebas

Para proseguir con las pruebas, se ha modificado el fichero Modelo “formInvoicesModel.cs” con tal de conseguir que el flujo de aprendizaje y el de extracción se realicen en automático sobre el directorio del conjunto de pruebas.

El procedimiento de actuación del flujo de aprendizaje es el siguiente. Para el directorio de documentos PDF, se recorre abriendo uno a uno los documentos PDF. Para cada PDF, se realiza una búsqueda sobre el documento CSV por el nombre del fichero. Una vez

obtenidos los metadatos a identificar correspondientes a ese fichero, se pasan como entrada al flujo de aprendizaje. Así reiteradamente para todos los documentos hasta completar el directorio y haber obtenido todos los ficheros XML correspondientes.

Por otra parte, el procedimiento de actuación para el flujo de extracción consiste en volver a recorrer el directorio de documentos PDF. Pero esta vez, se va a utilizar el conjunto de ficheros XML generados en la etapa de aprendizaje para realizar las extracciones. Por cada extracción en cada documento, se rellena un fichero CSV con las mismas columnas que el CSV pasado por entrada en la primera etapa.

El resultado es un fichero CSV que va a presentar huecos vacíos, debido a que las extracciones no son 100% perfectas y hay muchos documentos con carencias en información o mal estructurados, produciendo que nuestro software no encuentre los metadatos deseados.

6.3. Resultados

Para poder cuantificar los resultados, así como poder realizar una comparación entre el software original y el software mejorado, recorreremos el fichero CSV con las salidas obtenidas, y, para cada columna, realizamos un conteo de cuántos datos coinciden con los del fichero CSV original (el procedente de la base de datos, donde todos los datos son correctos).

En la siguiente tabla (tabla 6.1) se muestran los resultados para cada una de las mejoras desarrolladas a lo largo de esta estancia. Estos resultados muestran el número de aciertos sobre 9.379 (número total de documentos analizados) así como el porcentaje de aciertos. Puntualizar que, no se muestra sobre 10.000 documentos porque, al tratarse de un entorno realista, hay documentos que están tan mal formados internamente que a la hora de analizarlos el programa detenía su ejecución por lanzar una excepción.

Tipo de mejora	Número de factura	Fecha de facturación	Importe total
Estado inicial del software	7327 (78'12 %)	8266 (88'13 %)	8583 (91'5 %)
Extracción por nombre de la fuente	7378 (78'67 %)	8280 (88'28 %)	8571 (91'39 %)
Extracción por color, tamaño y nombre PdfClown	7157 (76'30%)	8148 (86'87 %)	8140 (86'8 %)
Extracción por color, tamaño y nombre PdfClown (añadiendo a huecos vacíos)	7391 (78'8 %)	8287 (88'4 %)	8602 (91'7 %)
Extracción por color, tamaño y nombre con iText	7479 (79'74 %)	8379 (89'34 %)	8720 (92'97 %)
Extracción anterior añadiendo rutina de eliminación de falsos positivos	7531 (80'30 %)	8426 (89'84 %)	8882 (94'70 %)

Tabla 6.1. Resultados de las pruebas aplicadas sobre los distintos desarrollos.

- **Estado inicial del software**

Como se puede observar, partimos de un software el cual ya está lo suficientemente maduro como para estar corriendo en un entorno real y poder ofrecer un servicio de extracción de metadatos solvente. No obstante, nuestro cometido en este proyecto ha sido mejorar el software. Al situarse en un alto número de aciertos, la cantidad de porcentaje a mejorar es más pequeña. Eso no quita que, cualquier mejora que consigamos, refuerce la eficacia del servicio prestado al cliente final.

- **Extracción por el nombre de la fuente**

Con la primera mejora desarrollada, la extracción de la propiedad “nombre de la fuente”, conseguimos subir sutilmente el porcentaje de aciertos para el número de factura y la fecha. Sin embargo, en el caso del importe total de la factura, muchas de estas facturas poseen un tipo de fuente genérico a lo largo de toda la factura, produciendo que dicho patrón sea ineficaz.

- **Extracción por color, tamaño y nombre de la fuente con PdfClown**

Debido a esto, se buscaron otras vías de mejora (almacenar más información por cada metadato identificado, por ejemplo el color). En este punto desarrollamos y evaluamos la mejora con PdfClown. Y, al contrario de lo que se puede llegar a pensar, el hecho de conseguir extraer una mayor cantidad de información nos produjo una disminución en la cantidad de aciertos. Esto se debe a que, aunque obtengamos más información, si ésta no es de calidad (mismos colores a lo largo de toda la factura, misma tipografía, mismo tamaño de fuente) produce que los patrones obtenidos sean demasiado genéricos, extrayendo finalmente metadatos incorrectos.

- **Extracción por color, tamaño y nombre de la fuente con PdfClown (añadiendo a los huecos vacíos)**

Una primera solución evaluada, vistos los resultados anteriores, consistía en añadir mediante estos patrones sólo en huecos que la biblioteca original no pudiera haber rellenado, consiguiendo de esta manera aumentar los aciertos por descarte (rellenar con un metadato tiene una mayor probabilidad de acertar que dejar un campo vacío).

- **Extracción por color, tamaño y nombre de la fuente con iText**

Al aplicar esta misma solución mediante la biblioteca de iText (la extracción por color, tamaño y tipo de fuente comentada en PdfClown), conseguimos subir unos peldaños más en el número de aciertos. Esto se debe a que, en algunas facturas, PdfClown no era lo suficientemente efectivo en sus extracciones, dejándose objetos del contenido del documento PDF por analizar.

- **Extracción por color, tamaño y nombre de la fuente, así como rutina de eliminación de falsos positivos**

En última instancia, desarrollamos cómo poder eliminar los resultados falsos positivos, aplicando la rutina de eliminación de falsos positivos vista en el *apartado 5.7. Rutina de eliminación de resultados falsos positivos*. Gracias a este sistema, conseguimos eliminar todo patrón genérico que sólo añada al ruido al software y no permita extraer resultados correctos. Con ello, se consigue hacer más fiables no sólo los nuevos patrones desarrollados en el proyecto, sino también los ya presentes en la biblioteca. Esto se traduce en subidas de hasta un 3% de aciertos en relación a los resultados sobre la biblioteca original.

Capítulo 7

Problemas surgidos en los desarrollos

En este apartado se detallan algunos de los problemas que han surgido en el desarrollo e implementación de las distintas mejoras, así como su solución en caso de haber llegado a una.

7.1. Extracción de XML de un Documento PDF

Con el transcurso de la estancia, se observó que en la creación de los archivos PDF se puede hacer uso de diferentes herramientas, como por ejemplo Acroforms. Los Acroforms, sin entrar en muchos detalles, consisten en campos de formulario mediante los cuales se puede rellenar un documento PDF.

Por otra parte, al comienzo de esta estancia, se quería averiguar si en un PDF era posible comprobar si había objetos *paragraphs*, *titles*, *tables*, etc.

Es por ello que se intentó realizar una extracción XML de la estructura de un PDF, en busca de encontrar indicios que nos permitieran agilizar la extracción de datos concretos. Al tener como dependencia iText, en este proceso se han usado sus métodos.

En este caso nos basamos en el uso de una clase que proporciona iText para extraer un documento XML a partir de un documento PDF dado. Es posible que este documento PDF no tenga las etiquetas formadas, pudiendo no extraer nada.

Una vez probado el desarrollo sobre un conjunto de 10.000 facturas, sólo 1.100 de éstas devolvieron un documento XML (para el resto, se mostraba una excepción indicando que no contenía elementos estructurales). De entre estos 1.100 documentos XML, solamente 43 contenían información útil, el resto eran archivos menores de 1 KB (la mayoría mostraban 0 KB) que no proporcionaban información o estaban vacíos. Los documentos que dejaban entrever información se examinaron en busca de indicios que permitieran extraer metadatos por esta vía. En su estructura se observó cómo algunos de ellos contenían campos CDATA (figura 7.1).

```
<Sect><Sect><Sect><Sect><Sect><Div><P><alt><![CDATA[WARENWERT]]></alt></P>
<P>Netto merce</P>
</Div>
</Sect>
<Div><P><alt><![CDATA[kon_KWERT]]></alt></P>
<P>45,15</P>
</Div>
</Sect>
</Sect>
<Sect><Sect><Sect><Div><P><alt><![CDATA[kon_vorzeichen]]></alt></P>
<P>+</P>
</Div>
<Div><P><alt><![CDATA[kon_VTEXT]]></alt></P>
<P>Trasporto</P>
</Div>
</Sect>
<Div><P><alt><![CDATA[kon_KWERT]]></alt></P>
<P>11,00</P>
</Div>
</Sect>
</Sect>
```

Figura 7.1. Documento XML extraído de un documento PDF.

No obstante, la finalidad de estos campos consiste en que el texto contenido entre sus corchetes sea ignorados por el intérprete. De esta manera resultaba imposible apuntar al dato deseado con la iteración sobre los objetos hijo del XML; dado que en cada archivo, el orden era distinto.

Teniendo en cuenta el bajo ratio de documentos XML extraídos sobre un gran conjunto de facturas, así como los resultados insatisfactorios para los XML extraídos, esta vía de mejora quedó descartada.

7.2. Extracciones con PdfClown

Como se ha comentado en el *apartado 5.5*, se realizó una implementación en base a la biblioteca de PdfClown. Dicha implementación conllevó agregar una dependencia más a la biblioteca de PdfTagger, razón por la cual se analizó de manera exhaustiva su funcionamiento con tal de no entorpecer nuestra aplicación.

Con el transcurso de los documentos analizados, se observó que, para un conjunto de 10.000 facturas, encontrábamos 200 facturas sobre las cuales la librería no era capaz de realizar una extracción. A la hora de realizar las llamadas a los objetos de la estructura del PDF, se producían excepciones que impedían seguir con el correcto funcionamiento de la extracción. Un ejemplo de esta excepción lo encontramos en la siguiente imagen (figura 7.2); donde, para esta factura, resulta imposible extraer metadato alguno. Además, este documento

de factura es susceptible de extracción por tamaño, tipo y color de fuente, dado que, en el caso del importe total, es el único con ese tamaño en toda la factura.

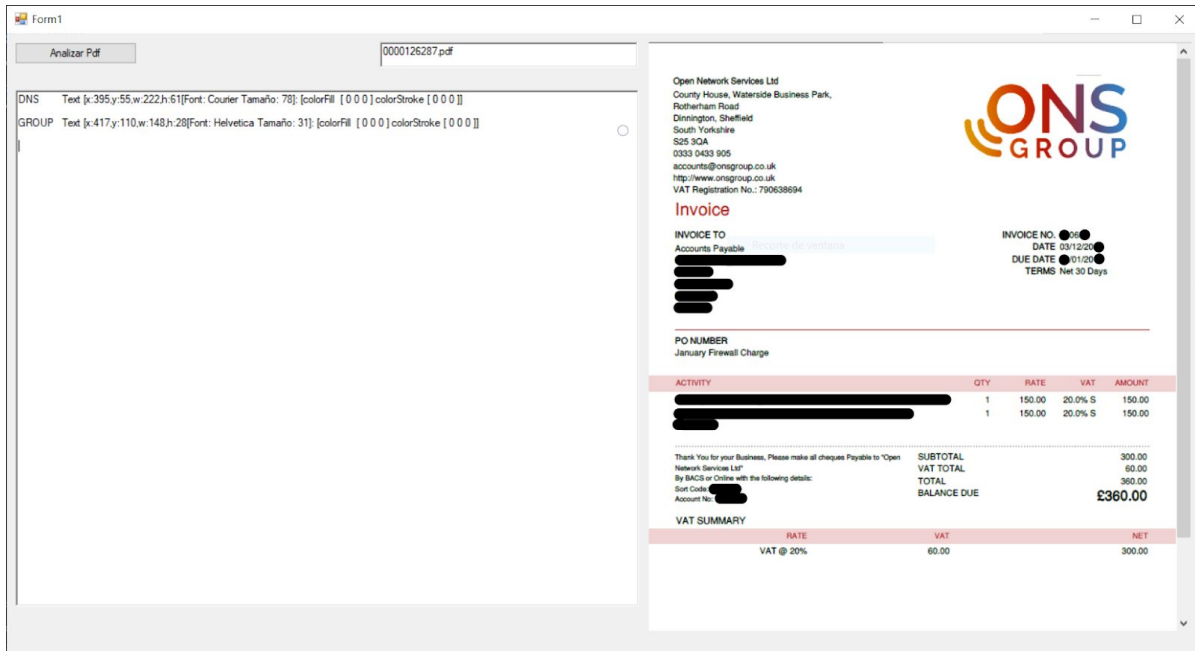


Figura 7.2. Ejemplo de extracción mediante PdfClown.

No obstante, estas 200 facturas, a la hora de realizar la extracción con iText no lanzan excepción. Un ejemplo de ello es el mismo documento de factura extraído con iText (figura 7.3). Por ello, una vez terminado el desarrollo con PdfClown, se comprobó la posibilidad de realizar la extracción por tipo, tamaño y color de fuente mediante la biblioteca de iText.

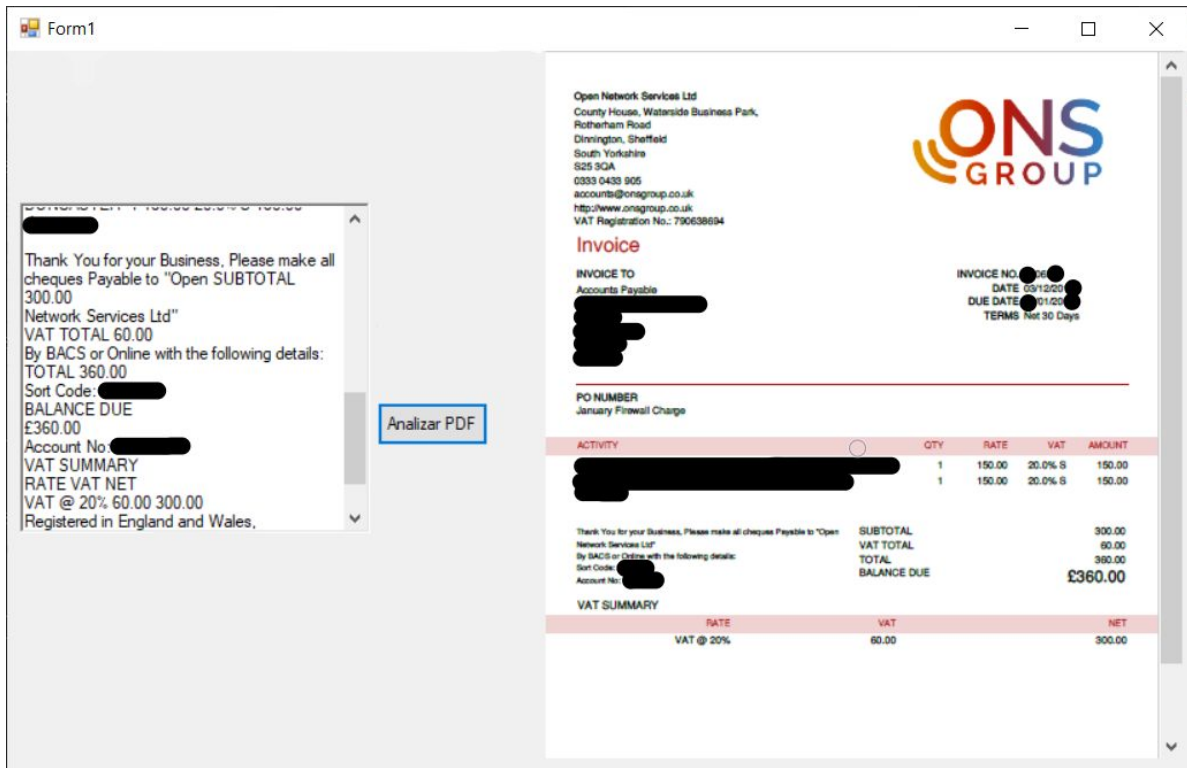


Figura 7.3. Ejemplo de extracción mediante iText.

Sin embargo, éste no ha sido el único de los fallos que nos ha dado la biblioteca. Habiendo desarrollado procedimientos para iterar un conjunto de facturas con tal de ejecutar el flujo de aprendizaje y extracción de manera automatizada (usando los datos de la base de datos como metadato de entrada para el aprendizaje), se ha obtenido un alto consumo de memoria RAM.

Después de depurar el código no se encontró la manera de solventarlo, por lo que, realizando una búsqueda exhaustiva, se dio con que se podía forzar el *Garbage Collector*. De esta manera, pasamos de tener un consumo, pasadas unas dos horas, de 1 GB de RAM a permanecer estable en unos 100 MB durante todo el transcurso de la ejecución, liberando al sistema de una gran carga considerable.

Hasta que se dio con dicha solución, con tal de iterar un conjunto de 10.000 documentos de factura (pudiendo así obtener resultados sobre un entorno realista para la funcionalidad desarrollada) podíamos emplear 3 días, debido a que el ordenador llegaba incluso a apagarse por quedarse sin memoria. En el momento de forzar al *Garbage Collector* y esperar a que éste hubiera limpiado las variables cada vez que se realizaba la extracción sobre un documento PDF, pasamos a emplear sólo algo menos de 3 horas. (Solamente con lanzar el *Garbage Collector* no bastaba, se seguía acumulando información en la memoria; había que decirle al programa que parara su ejecución hasta que se liberara esta memoria, lo cual supone milésimas de segundo de ejecución).

Capítulo 8

Conclusiones

En este proyecto se ha llevado a cabo un análisis de la biblioteca PdfTagger, perteneciente a la empresa Irene Solutions S.L. La funcionalidad de esta biblioteca consiste en la extracción de unos datos concretos a partir de documentos empresariales en formato PDF. En base a este análisis se han realizado distintas mejoras que han permitido evolucionar el sistema y ofrecer un servicio más consistente mediante el que proporcionar datos correctos al cliente final.

Hasta ahora, en el desarrollo del proceso de extracción de metadatos, la mayor parte de la industria enfocada en este sector se ha basado en posicionar la información a extraer dentro del marco que supone un documento PDF (tamaño A4 en vertical u horizontal). La gran mayoría de alternativas emplean, en su caso, una estrategia posicional (por coordenadas) para poder extraer un determinado dato; por tanto, cuando un dato no se encuentra en esa posición, no es posible realizar una extracción satisfactoriamente. La biblioteca PdfTagger ha aportado a este sector la extracción por delimitaciones contextuales, basada en identificar mediante expresiones regulares dónde se posiciona una determinada cadena entre todo el texto del PDF. Con la nueva mejora en el extractor, descrita en este documento (*capítulo 5*), la biblioteca se posiciona frente a la industria a un nuevo nivel, donde, teniendo en cuenta las características del texto contenido en un PDF (como pueden ser de qué tamaño, color o tipo de fuente se trata) es posible realizar una extracción de metadatos más eficaz y realista. Entendiendo por realista, proporcionar datos válidos y correctos (y evitar, por tanto, los resultados falsos positivos que puedan derivarse como salida del software).

Por otra parte, con el desarrollo del nuevo flujo de comprobación basado en el conteo de resultados no deseados (los denominados falsos positivos) se dota de un mayor grado de fiabilidad a la biblioteca. Mediante el conteo de errores, ahora es capaz de detectar no sólo cuándo un patrón devuelve un resultado correcto, sino también cuando un patrón devuelve un resultado erróneo. La combinación de estos dos conteos implica darle a la biblioteca un mayor grado de información respecto a la fiabilidad que puede ofrecer cada patrón, proporcionando de esta manera unas extracciones de mayor calidad.

En cuanto a la parte personal, la realización de esta estancia me ha permitido desarrollarme y mejorar tanto a nivel profesional como personal, además de darme la oportunidad de conocer de primera mano cómo es el funcionamiento de un software en un entorno real. También me ha permitido mejorar mis competencias y mi conocimiento acerca del uso y análisis de bibliotecas. He adquirido experiencia también en realizar trabajos de investigación acerca de cómo mejorar un software que ya se encuentre diseñado y corriendo en un entorno funcional. Este punto es algo que nunca se había dado en la carrera, donde

diseñas todos los sistemas desde cero, por lo que ha sido una manera de salir de lo ya puesto en práctica y realizar nuevos desarrollos con pocas directrices de por medio; ya que, en referente a realizar mejoras no existe nada escrito, el camino sobre el que va a evolucionar el software lo escribes tú. En este sentido, se ha tratado de una experiencia enriquecedora, donde la responsabilidad de llevar el proyecto hacia delante recae sobre uno mismo; con sus momentos de frustración cuando se ha sido incapaz de seguir evolucionando por una vía, y sus momentos de satisfacción cuando te encontrabas que la mejora que estabas llevando a cabo había dado resultados y habías conseguido aportar tu granito de arena al proyecto. Por otra parte también he podido conocer de primera mano cómo es el día a día de una empresa de software en pleno crecimiento, algo que no puedes llegar a apreciar en los distintos años dedicados al estudio.

Bibliografía

- [1] Arquitectura Model - View - Presenter. <https://www.journaldev.com/14886/android-mvp>
[fecha de consulta: 24 de mayo, 2019]
- [2] GNU Affero General Public License 3.0. <https://www.gnu.org/licenses/agpl-3.0.en.html>
[fecha de consulta: 18 de junio, 2019]
- [3] IDE Visual Studio Community 2017.
<https://visualstudio.microsoft.com/es/vs/community/> [fecha de consulta: 24 de mayo, 2019]
- [4] Indeed. Salarios para empleos de Programador/a junior .NET en España.
<https://www.indeed.es/salaries/Programador/a-junior-.NET-Salaries>
[fecha de consulta: 29 de mayo, 2019]
- [5] Libro manual. C# 5.0 in a Nutshell: The Definitive Reference. Joseph Albahari, Ben Albahari. 26 de junio de 2012 por O'Reilly Media.
- [6] Manual de expresiones regulares en C#. http://joaquin.medina.name/web2008/documentos/informatica/lenguajes/puntoNET/System/Text/Regex/C_Sharp_ExpresionesRegulares.pdf
[fecha de consulta: 19 de junio, 2019]
- [7] Metodología predictiva.
<https://blog.workep.com/es/metodologias-de-gestion-de-proyectos-tradicional-vs-agil>
[fecha de consulta: 24 de mayo, 2019]
- [8] Microsoft Docs. Documentación acerca de las expresiones regulares en C#. <https://docs.microsoft.com/es-es/dotnet/standard/base-types/regular-expression-language-quick-reference> [fecha de consulta: 19 de junio, 2019]
- [9] Noticia “Visual Studio Leads IDE Popularity List, IntelliJ Climbing Fast”.
<https://visualstudiomagazine.com/articles/2017/08/15/ide-ranking.aspx>
[fecha de consulta: 28 de mayo, 2019]
- [10] PdfClown. Librería Open Source para PDF. <https://pdfclown.org/>
[fecha de consulta: 24 de mayo, 2019]
- [11] Repositorio de la biblioteca PdfTagger. <https://github.com/mdiago/PdfTagger>
[fecha de consulta: 24 de mayo, 2019]
- [12] Software iText PDF. <https://itextpdf.com/es> [fecha de consulta: 24 de mayo, 2019]
- [13] Términos de la licencia de uso de Visual Studio Community.
<https://visualstudio.microsoft.com/es/license-terms/mlt553321/>
[fecha de consulta: 17 de junio, 2019]

[14] Términos de la licencia de uso de iText 5. <https://itextpdf.com/en/resources/faq/legal/itext-5-legacy/itext-java-library-free-charge-or-are-there-any-fees-be-paid> [fecha de consulta: 17 de junio, 2019]

[15] Testeador de expresiones regulares para .NET. <http://regexstorm.net/tester>
[fecha de consulta: 20 de junio, 2019]

[16] Top índice de entornos de desarrollo de software. <https://pypl.github.io/IDE.html>
[fecha de consulta: 24 de mayo, 2019]