Università degli Studi di Parma
Facoltà di Scienze
Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Progettazione e Realizzazione di un Sistema di Archiviazione di Documenti mediante Barcode

Candidato:
Filippo Dallavalle

Relatore:
Dott. Federico Bergenti

Anno Accademico 2008/2009

Thanks to ...

Prima di tutto voglio ringraziare la mia famiglia, in particolare i miei genitori: senza il vostro continuo "incitamento" e i vostri consigli (che non ho ancora imparato a seguire del tutto...) probabilmente avrei rinunciato all'università un paio di anni fa. Grazie di cuore.

Uno dei vantaggi di arrivare alla meta con qualche *annetto di ritardo è che hai l'occasione di conoscere meglio le persone che incontri lungo la via: in questi anni ho avuto l'opportunità di incontrare molte persone, che mi hanno aiutato a superare i momenti (e gli esami) difficili con un sorriso e con cui ho stretto una forte amicizia. Partendo dai "compagni" di vecchia data voglio ricordare Tino, Tsabo, Kusu, Fabio (Sozzi): grazie per l'aiuto che mi avete dato a partire dalle lunghe giornate del primo anno in via D'Azeglio, quando ogni scusa era buona per andare al Dulcamara a fare colazione e ancora si pranzava con Fantasia al parco ducale o in aula party. Un posto d'onore spetta ai mitici componenti del "Wolfe Team": Gianni, Berna, Daro, Boni, Teddy, Mino, Bisius e Amos (anche se per breve tempo): grazie per i lunghi pomeriggi e le serate passate insieme! Grazie a Laura, Stefania, Erika, Andrea e a tutti i matematici a cui ho rotto le scatole innumerevoli volte per avere spiegazioni che si sono sempre rivelate essere illuminanti. Grazie a tutti i "nerd" (e non) del condominio che sono qui oggi e che non ho ancora nominato: non ho scritto i vostri nomi perché se vi elencassi tutti potrei scrivere una nuova tesi solamente su di voi!*

Grazie a Anna, Laura, Chicca, Giulia, Cristina, Davide, Sergio, Antonio, Giulio per avermi dato la vostra forza, i vostri consigli e il vostro esempio per arrivare a questo importante traguardo.

Infine, voglio ringraziare il professor Bergenti, Mario, Luigi, Lorenzo e gli altri della SSDI, che sono stati sempre disponibili e pronti a chiarire i miei dubbi in questi ultimi e caotici mesi.

Indice

Capitolo 1

Introduzione

L'argomento affrontato in questa tesi è quello di implementare, attraverso l'uso di apposite *Java Libraries*, un lettore di codici a barre (più precisamente *barcode* appartenenti allo standard PDF417) al fine di riconoscere e, successivamente, automatizzare l'archiviazione di documenti PDF all'interno di un *database*.

Questo progetto nasce dalla necessità di ENEL di digitalizzare il suo "armadio di edificio", che contiene tutta la documentazione riguardante gli impianti, i complessi, i fabbricati e gli altri immobili di proprietà dell'azienda. Per questo motivo ENEL ha richiesto alla SSDI (Sistemi e Soluzioni Documentali Integrate) di realizzare una *web-application* che si occupasse della digitalizzazione e della successiva archiviazione dei suddetti documenti.

I documenti originali in formato cartaceo vengono scannerizzati e, successivamente, salvati in formato PDF; dopo questa fase sono pronti per essere catalogati all'interno del *database*. Per archiviare un documento, l'utente ha due opzioni:

- Caricamento in modalità *sincrona*

- Caricamento in modalità *asincrona*

La scelta dipende dal numero e dalla dimensione dei file da caricare.

Il caricamento in modalità *sincrona* consente l'upload del file tramite interfaccia web e garantisce l'immediato caricamento del file; per motivi di prestazioni del sistema, tale funzionalità è disponibile solo per file "leggeri" (file con dimensione fisica inferiore ad un parametro di sistema configurabile). Il sistema consente all'utente l'upload di un singolo file (contenente un solo documento) tramite la seguente sequenza operativa:

1. L'utente richiede l'archiviazione del file indicando il numero protocollo del documento (selezionato da elenco) ed il file da caricare (con il relativo percorso della cartella di origine).

2. Il sistema verifica se il file da caricare è di dimensione superiore ad una soglia configurabile da sistema (file "pesante"). Se il file è "pesante" la funzionalità non può essere utilizzata ed il sistema propone all'utente la modalità di caricamento singolo di file "pesanti".

3. Il sistema verifica che il file da caricare sia relativo al documento indicato utilizzando le informazioni acquisite tramite il codice a barre presente sulla prima pagina; in caso di errore (es.: il file non corrisponde al documento specificato) non si procede al caricamento.

4. Il sistema rinomina automaticamente il file associandogli un nome uguale al numero protocollo.

5. Il sistema archivia il file utilizzando gli attributi già definiti per il documento.

6. Il sistema restituisce all'utente l'esito del caricamento. In caso di esito negativo il sistema restituisce un apposito diagnostico.

Il caricamento in modalità *asincrona*, invece, permette il caricamento massivo di un insieme di file (leggeri o pesanti) associati ad un elenco di documenti, o di un singolo file "pesante". Con tale modalità il caricamento è fatto in una fase successiva alla richiesta, con tempistiche dipendenti dal carico del sistema. Il monitoraggio dell'esito del caricamento può essere fatto dall'utente tramite un'apposita funzionalità.
L'operatività è la seguente:

1. L'utente deposita i file da caricare in una cartella di rete condivisa.

2. Tramite apposita interfaccia utente si richiede al sistema il caricamento dei file, specificando quale dei file presenti nella cartella condivisa devono essere caricati.

3. Il sistema effettua il caricamento dei file richiesti in modalità *asincrona*; tramite il codice a barre presente sulla prima pagina del documento il sistema associa automaticamente il file al profilo documento e rinomina automaticamente il file associandogli il nome uguale al numero protocollo.

4. Se il file è "pesante" il sistema crea il file di anteprima.

5. Il sistema restituisce l'esito del caricamento su apposita interfaccia utente.

Durante la mia esperienza di tirocinio presso la SSDI ho avuto occasione di occuparmi della parte relativa al caricamento *asincrono* dei documenti.
Anzitutto è stato necessario individuare e testare degli strumenti che potessero essere utili a soddisfare le specifiche di ENEL. Tra le possibili soluzioni già in commercio sono state individuate le seguenti librerie:

- *Aspose.Pdf.Kit for Java™*

- *JPedal™* (Java Pdf Extraction Decoding Access Library)
- *iText*
- *jPDFImages™*
- *Barcode Xpress™*

Per quanto riguarda il riconoscimento dei codici a barre presenti all'interno dei PDF, *Barcode Xpress* si è subito dimostrata essere la soluzione ideale; ben più difficile è stato trovare una o più librerie in grado di dividere e esportare come immagini le pagine dei documenti. La conversione in formato JPEG, GIF o PNG è necessaria sia perché queste immagini devono essere visualizzate come anteprima del documento sia per il fatto che la library *Barcode Xpress* non ammette come *input* un file PDF, ma solo JPEG, GIF, TIFF o PNG.

Le prime due librerie ad essere testate sono state *JPedal* e *Aspose.Pdf.Kit* ma in entrambi i casi l'immagine risultante dalla conversione del PDF presentava una risoluzione troppo bassa, impedendo a *Barcode Xpress* di effettuare una lettura corretta del *barcode*. Aumentando — attraverso appositi metodi delle due suddette librerie — la risoluzione delle immagini in *output*, le dimensioni di queste aumentavano a tal punto da renderle inadatte al loro ruolo di anteprima del documento. Questi fattori hanno portato all'uso delle due librerie trattate nel prossimo capitolo: *iText* e *jPDFImages*.

1.1 La simbologia PDF417

Prima di iniziare l'analisi delle *Java library* utilizzate, è utile fare un breve accenno ai *barcode* ed in particolare alla simbologia scelta da ENEL per effettuare il riconoscimento dei suoi documenti.

Il codice a barre PDF417 è un *barcode* bidimensionale che permette di comprimere una grande quantità di informazioni in uno spazio limitato, immagazzinando più informazioni rispetto ad un *barcode* "tradizionale" di tipo lineare. Codificando l'intero set di caratteri ASCII, il codice a barre PDF417 può contenere fino a 2.700 caratteri. Il *barcode* PDF417 consiste di un numero di righe variabile da 3 a 90, ciascuna delle quali è simile ad un piccolo *barcode* lineare. Ogni riga ha:

1. Una "quiet zone". Questa consiste in un minimo spazio bianco prima dell'inizio del *barcode*.

2. Un *pattern* iniziale, che identifica l'appartenenza del *barcode* alla simbologia PDF417. Ogni simbologia di *barcode* possiede un *pattern* iniziale e un *pattern* finale univoci.

3. Un *codeword* "riga sinistra" contenente informazioni sulla riga (quali il numero di riga e quale livello di correzione dell'errore è utilizzato dalla stessa).

4. Da uno a trenta *codeword* di dati: i *codeword* sono un gruppo di barre e spazi che rappresentano uno o più numeri, lettere, o altri simboli.

    - Tutte le righe hanno lo stesso numero di *codeword*.
    - Ogni *codeword* contiene quattro barre e quattro spazi (da questo fatto deriva il 4 nella sigla).
    - La larghezza totale di un *codeword* è 17 volte la larghezza della più stretta barra verticale consentita (da qui deriva il 17 nella sigla).

- Ogni *codeword* inizia con una barra e termina con uno spazio.
- Ci sono 929 *codeword*, di cui 900 per i dati e 29 per funzioni speciali.
- Ogni *codeword* è stampato utilizzando uno dei tre possibili *cluster*:
  - Un *cluster* è un *pattern* barra-spazio per ciascuna delle 929 *codeword*.
  - Il numero di riga determina il tipo di *cluster* da utilizzare.
  - Il *cluster* è identico per tutti i *codeword* di una riga.
  - Lo scopo dei *cluster* è stabilire in quale riga (modulo 3) si trova il *codeword*, permettendo una scansione non orizzontale. Per esempio, la scansione potrebbe cominciare alla riga 6 all'inizio della riga e terminare alla riga 10 alla fine di questa.

5. Un *codeword* "riga destra", con altre informazioni riguardanti la riga.
6. Un *pattern* finale.
7. Una "quiet zone".

Questa simbologia permette la correzione degli errori, permettendo di riparare e leggere anche *barcode* danneggiati o incompleti. Inoltre è un formato di pubblico dominio, ovvero può essere liberamente utilizzato senza alcuna licenza.

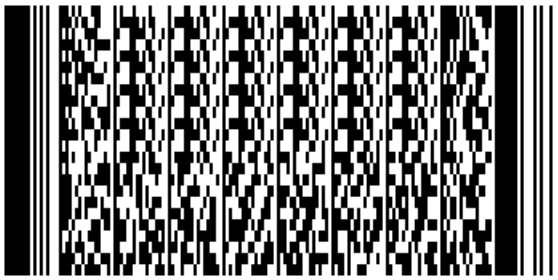

Figura 1.1: Un esempio di *barcode* PDF417

Capitolo 2

Gestione dei PDF in Java

Il PDF è il noto standard per i documenti introdotto da Adobe negli anni novanta, che consente di rendere le informazioni provenienti da un qualsiasi sistema, accessibili a chiunque e dovunque, persino su dispositivi mobili.
Le caratteristiche che ne hanno decretato il successo presso aziende e pubbliche istituzioni sono, principalmente:

- sicurezza, le informazioni inserite in un file PDF possono essere cifrate
- affidabilità, la resa è sempre fedele all'originale
- estensibilità, grazie ai *plug-in* che di continuo vengono sviluppati e che consentono di arricchire la già ampia gamma di informazioni stipabili in un file PDF
- indipendenza dalla piattaforma, un PDF è fruibile su moltissime piattaforme (Linux, Windows, MAC OS, etc.)
- standard aperto, che quindi può continuamente essere esteso ed arricchito

Vista la diffusione di questo formato, uno standard *de facto* per l'interscambio di documenti, è naturale cercare di automatizzare il processo di trasformazione dei documenti generati in altri formati, in documenti PDF da rendere disponibili via Web. Gli odierni *Application Server* forniscono nativamente strumenti di conversione dei documenti in RTF e HTML, ma non in PDF.

Nell'ambiente Java esistono svariate librerie (gratuite o a pagamento) per la gestione dei file PDF, ciascuna con i suoi pro e contro e la possibilità di eseguire numerose operazioni su questo tipo di file. Quelle utilizzate nella realizzazione di questo progetto — al fine di soddisfare le specifiche fornite da ENEL — sono due: la library *iText* e la library *jPDFImages*.

2.1 La Library iText

Creata da Bruno Lowagie e Paulo Soares, *iText* è una libreria Java *open-source* (pubblicata sotto licenza MPL e LGPL) che permette di automatizzare il processo di creazione e manipolazione dei file PDF, ad esempio in una delle seguenti situazioni:

- a causa del tempo o delle dimensioni, i documenti PDF non possono essere prodotti manualmente;
- il contenuto del documento deve essere calcolato in base ad un input dell'utente;
- il contenuto deve essere personalizzato;
- il contenuto del PDF deve essere utilizzato in ambiente web;
- i documenti devono essere creati in modalità *batch process*.

Sfruttando le numerose classi che fornisce, *iText* può essere utilizzata per:

- "passare" documenti PDF ad un browser;
- generare documenti dinamici a partire da un file XML o un database;
- utilizzare le numerose *features* interattive dei documenti PDF;
- aggiungere segnalibri, numeri di pagina, *digital watermark*, etc;
- dividere, concatenare e manipolare le pagine di un PDF;
- automatizare la compilazione di PDF *AcroForm*;
- gestire la cifratura dei documenti generati;
- aggiungere firme digitali ad un file PDF.
- ...

Le funzionalità di *iText* sono utili a coloro che necessitano di generare documenti *read-only* multipiattaforma contenenti testo, liste, tabelle e immagini; o a chi deve eseguire particolari modifiche a dei documenti già esistenti. Per poter essere utilizzata, *iText* richiede JDK 1.4 o superiore.

2.1.1 Perché iText?

Essendo *open-source*, *iText* è disponibile gratuitamente e questo fattore ha influito pesantemente sulla scelta di utilizzare questa libreria per la realizzazione del nostro progetto. Inoltre, il fatto che *iText* sia inclusa da molti software — tra cui Macromedia ColdFusion, Google Calendar, strumenti per la creazione di *report* come JasperReports e Eclipse/BIRT — nelle loro distribuzioni è stato per noi una garanzia delle sue funzionalità e della sua affidabilità.

2.1.2 Le classi

Analizziamo ora le principali classi generali (valide per un qualsiasi formato di output) che *iText* mette a disposizione. Grazie ad esse è possibile realizzare documenti ed esportarli in uno dei formati gestibili con *iText*, semplicemente modificando il *writer* del documento.

Classe	Descrizione
Document	rappresenta un generico documento di *iText* valido per qualsiasi formato di *output*
Chunk	è la più piccola parte di testo significativa che può essere aggiunta all'interno di un *Document*
Phrase	è semplicemente un insieme di *Chunk* che può essere inserito, in maniera immediata, all'interno di un *Document*
Paragraph	è un insieme di *Phrase* e/o *Chunk* opportunamente disposti e formattati
Section	è una parte del *Document* costituita da un insieme di *Paragraph* opportunamente disposti
Chapter	è per *iText* una speciale *Section* atta a contenere, appunto, un insieme di *Section* opportunamente sequenziate
PageSize	è un contenitore per un insieme predefinito di rettangoli che rappresentano altrettanti formati di pagine standard (es. A4, A3, etc.)
Font	è un contenitore per tutte le principali caratteristiche dei font digitali (es. font *family, size, color*, etc.)
Image	è rappresenta un elemento grafico (PNG, GIF o JPEG) da inserire nel documento
DocWriter	è una classe astratta che rappresenta un *writer* generalizzato a partire dal quale si effettuano le specifiche implementazioni per i *writer* di formati ben definiti (es. PDF o HTML)

Le classi appena illustrate in tabella sono la base di partenza teorica per la realizzazione di documenti *iText*, approfondiremo ora alcune particolarità riguardo la creazione di documenti PDF e la manipolazione di documenti PDF preesistenti e archiviati sul file system.

La creazione di PDF

La creazione di un documento PDF passa per la realizzazione di un *Document* di *iText*, il quale dovrà semplicemente essere scritto sull'*output stream* di un file con estensione `.pdf`. Per realizzare la scrittura del documento su un certo *stream* PDF, è necessario utilizzare l'apposito *writer*, ereditato dal *DocWriter*, che prende il nome di *PDFWriter*.

Il *PDFWriter* di *iText* è una specifica implementazione di *writer* che cura la scrittura di *Document* su di uno *stream* PDF. Sfrutta una *factory* che restituisce un'istanza statica del *writer*, a partire dalla quale, con chiamate successive sarà possibile redirigere l'*output* di uno stesso *Document iText* su file PDF diversi sul disco. L'utilità di questa modalità operativa può risiedere, ad esempio, nel voler aggiungere caratteristiche peculiari (quali *attachment*, immagini, commenti, etc.) ad un *Document* precedentemente creato e scritto in un file PDF, e di realizzare un nuovo PDF con le modifiche apportate; tutto ciò sfruttando un'istanza del *writer*.

La creazione di RTF, HTML e XML

Il processo di generazione di documenti RTF, HTML e XML, a partire da un certo *Document*, è molto simile a quello descritto per il formato PDF. Il *writer RTFWriter* consente di scrivere su un *output stream* rediretto ad un file RTF. I file RTF, ovviamente, non supportano caratteristiche quali

watermark e cifratura del testo, caratteristiche specifiche del formato PDF (nel contesto di utilizzo *iText*). Con le nuove versioni della libreria, il *writer* RTF è stato perfezionato e rilasciato come *RTFWriter2* che è consigliabile utilizzare.

Quanto sopra vale, ovviamente, per i file XML e HTML, per i quali il processo di creazione è praticamente lo stesso.

La gestione di file PDF preesistenti

Come detto, *iText* è anche uno strumento molto potente per manipolare PDF preesistenti. Il *package* `com.lowagie.text.pdf` di *iText* è dedicato alla creazione e manipolazione dei file in formato PDF. Analizzando le classi all'interno del *package*, si può notare come siano ampie le possibilità di manipolazione di documenti PDF preesistenti. Ecco un breve elenco delle operazioni di modifica più utili e frequenti:

- *merge* (fusione) di due o più file PDF;
- *split* (suddivisione) di un file PDF in un certo numero di file;
- aggiunta di *digital watermark*;
- aggiunta di codici a barre;
- formattazione del testo in due o più colonne;
- inclusione di annotazioni;
- cifratura delle informazioni contenute in un PDF.

Nella tabella 2.1 a pagina seguente sono riportate le classi principali per la modifica di documenti in formato PDF.

Tabella 2.1: Le classi di iText per la modifica dei file PDF

CLASSE	DESCRIZIONE
PDFDocument	consente di trasformare un qualsiasi *Document* in un PDF di una o più pagine
PDFReader	consente all'applicazione di recuperare il contenuto di un file PDF stipato su disco
PDFContentByte	consente di prelevare (da un file) e posizionare (in un file) il contenuto in byte di testo ed immagini
PDFImportedPage	consente di importare un'intera pagina di un file PDF
Barcode	raggruppa un insieme di funzionalità atte a creare e gestire *barcode* di svariati tipi
MultiColumnText	consente di agire direttamente sulla formattazione del testo di un documento, e di renderne il testo su due o più colonne (identificate da appositi *Rectangle* di *iText*)
PDFAnnotation	consente di inserire delle note o annotazioni all'interno del documento, associandole ad una certa pagina
PDFEncryptor	consente di cifrare le informazioni presenti in un PDF preesistente: come *encryptor* (cifratore/codificatore), prende in input il PDF che si vuole cifrare e restituisce un PDF con lo stesso contenuto, ma cifrato
PDFStamper	consente di applicare nuovi contenuti alle pagine di un documento PDF. Questo contenuto può essere un qualsiasi oggetto che la classe *PdfContentByte* rappresenta, incluse pagine di altri PDF. Il PDF originale manterrà tutti i suoi elementi interattivi, come segnalibri e link

Concludiamo l'analisi della library *iText* con un esempio, che mostra come aggiungere un *watermark* e i numeri di pagina ad un documento già esistente:

```
IMPORT java.io.FileOutputStream;
IMPORT java.util.HashMap;

IMPORT com.lowagie.text.Element;
IMPORT com.lowagie.text.Image;
IMPORT com.lowagie.text.PageSize;
IMPORT com.lowagie.text.pdf.BaseFont;
IMPORT com.lowagie.text.pdf.PdfContentByte;
IMPORT com.lowagie.text.pdf.PdfReader;
IMPORT com.lowagie.text.pdf.PdfStamper;

PUBLIC CLASS AddWatermarkPageNumbers {
   PUBLIC STATIC VOID main(STRING[] args) {
      SYSTEM.OUT.println("Add watermarks and pagenumbers");
      TRY {
         // crea un reader per il documento ChapterSection.pdf
         PdfReader reader = NEW PdfReader("ChapterSection.pdf");
         INT n = reader.getNumberOfPages();
         // crea un PdfStamper che copierà il documento in un nuovo file
         PdfStamper stamp = NEW PdfStamper(reader, NEW FileOutputStream("↩
             watermark_pagenumbers.pdf"));
         // aggiunge il nuovo contenuto ad ogni pagina
         INT i = 0;
         PdfContentByte under;
         Image img = Image.getInstance("watermark.jpg");
         BaseFont bf = BaseFont.createFont(BaseFont.HELVETICA, BaseFont.↩
             WINANSI, BaseFont.EMBEDDED);
         img.setAbsolutePosition(200, 400);
         WHILE (i < n) {
            i++;
            // aggiunge il watermark al di sotto della pagina esistente
            under = stamp.getUnderContent(i);
            under.addImage(img);
            // aggiunge il numero di pagina
            over = stamp.getOverContent(i);
```

```
            over.beginText();
            over.setFontAndSize(bf, 18);
            over.setTextMatrix(30, 30);
            over.showText("page " + i);
            over.setFontAndSize(bf, 32);
            over.showTextAligned(Element.ALIGN_LEFT, "DUPLICATE", 230, ↩
                430, 45);
            over.endText();
         }
         // aggiunge una nuova pagina
         stamp.insertPage(1, PageSize.A4);
         over = stamp.getOverContent(1);
         over.beginText();
         over.setFontAndSize(bf, 18);
         over.showTextAligned(Element.ALIGN_LEFT, "DUPLICATE OF AN ↩
             EXISTING PDF DOCUMENT", 30, 600, 0);
         over.endText();
         // aggiunge una nuova pagina da un altro documento
         PdfReader reader2 = new PdfReader("SimpleAnnotations1.pdf");
         under = stamp.getUnderContent(1);
         under.addTemplate(stamp.getImportedPage(reader2, 3), 1, 0, 0, 1, ↩
             0, 0);
         // chiudendo il PdfStamper verrà generato il nuovo file PDF
         stamp.close();
      }
      catch (Exception de) {
         de.printStackTrace();
      }
   }
}
```

Listing 2.1: Aggiungere un *digital watermark* e i numeri di pagina

2.2 La Library jPDFImages

jPDFImages è una libreria Java che permette di convertire un documento PDF in un'immagine, ma anche di creare o modificare PDF contenenti immagini. *jPDFImages* svolge le seguenti funzioni:

- esportare singole pagine di un documento PDF come immagini JPEG, TIFF o PNG;
- importare immagini in documenti nuovi o già esistenti;
- esportare le pagine come oggetti *BufferedImage*, per processarle ulteriormente;
- salvare direttamente nel file system o come Java *output stream*.

2.2.1 Perché jPDFImages?

Come già accennato in conclusione del capitolo 1, la scelta di utilizzare la library *jPDFImages* è maturata in seguito agli scarsi risultati ottenuti nei test delle librerie che intendevamo inizialmente includere nel progetto. Le immagini generate da *JPedal* e *Aspose.Pdf.Kit* risultavano troppo pesanti per essere utilizzate come anteprima dei documenti ed in alcuni casi, pur avendo impostato la massima risoluzione possibile per l'immagine di output, il *barcode* contenuto nella pagina risultante non veniva letto correttamente.

2.2.2 Le classi

PDFDocument è la classe principale della library *jPDFImages*. Questa classe è utilizzata per caricare documenti PDF o per creare nuovi documenti e fornisce metodi per importare ed esportare immagini nel e dal documento.

PDFDocument possiede tre costruttori, per caricare un PDF dal file system, da un URL o da un *InputStream*. Tutti i costruttori accettano un parametro aggiuntivo, un oggetto che implementa *IPasswordHandler*, che sarà interrogato se il file PDF è protetto da password. Per i PDF che non sono criptati, questo secondo parametro può essere NULL.

```
PDFDocument pdfDoc = NEW PDFDocument(NEW URL("http://www.qoppa.com/content↩
    .pdf"), NULL);
```

Inoltre, la classe possiede un quarto costruttore che permette di creare un documento vuoto. Questo costruttore può essere utilizzato quando si deve creare un nuovo documento partendo da immagini preesistenti.

Esportare le immagini

Dopo che un file PDF è stato caricato — creando un'istanza della classe *PDFDocument* — le pagine del documento possono essere esportate come immagini nel formato JPEG, TIFF o PNG.

```
//Esporta la prima pagina nei tre formati
pdfDoc.savePagesAsJPEG(0, "c:\\somefile.jpg",150,0.80f);
pdfDoc.savePagesAsTIFF(0, "c:\\somefile.tif",150,TIFFCompression.↩
    TIFF_FAX_GROUP4));
pdfDoc.savePagesAsPNG(0, "c:\\somefile.png",150);
```

Come si può osservare nell'esempio appena descritto, c'è un terzo parametro nell'invocazione di ogni metodo (fissato a 150 nell'esempio). Questo parametro rappresenta i DPI (*dot-per-inch*, punti per pollice) da utilizzare per il *rendering* dell'immagine. Il formato PDF non ha una risoluzione specifica, può contenere immagini di qualunque risoluzione, perciò questo valore deve essere passato alla libreria. Il valore dei DPI influisce direttamente sull'altezza e sulla larghezza dell'immagine.

Vi è inoltre un quarto parametro per i formati JPEG e TIFF; quando si desidera esportare il documento in formato JPEG, il quarto parametro rappresenta la qualità (ovvero la compressione) dell'immagine di output e può variare tra 0,10 e 1,00. Per il formato TIFF, il quarto parametro determina il tipo di compressione da utilizzare e deve essere uno dei valori predefiniti nella classe *TIFFCompression*.

Infine, *jPDFImages* consente di esportare una pagina come un oggetto della classe *BufferedImage*. Il metodo è il seguente:

```
BufferedImage pageImage = pdfDoc.getPageImage(0,150);
```

Come gli altri metodi precedentemente descritti, anche questo metodo ha tra i suoi parametri i DPI dell'immagine.

Importare le immagini

jPDFImages permette di importare immagini come nuove pagine di un documento; questo può essere un documento preesistente oppure un nuovo documento, a seconda del costruttore invocato nella creazione dell'oggetto *PDFDocument*. Una volta che l'oggetto *PDFDocument* è stato istanziato, il codice seguente può essere utilizzato per importare immagini dai formati JPEG, TIFF e PNG:

```
// Crea nuove pagine e vi importa le immagini
pdfDoc.appendJPEGAsPage("c:\\somefile.jpg");
pdfDoc.appendTIFFAsPages("c:\\somefile.tif");
pdfDoc.appendPNGAsPage("c:\\somefile.png");
```

Quando viene invocato uno di questi metodi, *jPDFImages* crea una pagina avente dimensione pari a quella dell'immagine scelta e quindi importa l'immagine in questa nuova pagina. La libreria controlla il valore dei DPI del file immagine per determinare la dimensione (in pollici) della pagina.

Nel caso in cui si scelga di importare un file TIFF contenente immagini multiple, *jPDFImages* creerà una nuova pagina per ogni immagine.

Ottenere le informazioni di base di un documento

Oltre a importare ed esportare immagini da un documento PDF, la library *jPDFImages* è in grado di ottenere le informazioni di base del documento. Per ricavare queste informazioni è necessario l'uso della classe *DocumentInfo*, accessibile come `PDFDocument.getDocumentInfo`. Attraverso questa classe è possibile ottenere informazioni riguardanti il documento, come il titolo, l'autore, il soggetto, le parole chiave, etc...

```
System.out.println(pdfDoc.getDocumentInfo().getTitle());
System.out.println(pdfDoc.getDocumentInfo().getAuthor());
System.out.println(pdfDoc.getDocumentInfo().getKeywords());
```

Per concludere, nell'esempio 2.2 a pagina seguente viene mostrato come utilizzare la library *jPDFImages* per esportare tutte le pagine di un documento PDF come immagini JPEG.

Listing 2.2: Conversione da PDF a JPEG

```
import com.qoppa.pdfImages.PDFImages;

public class PDFToJPEGs
{
    public static void main (String [] args)
    {
        try
        {
            PDFImages pdfDoc = new PDFImages("input.pdf", null);
            for (int count = 0; count < pdfDoc.getPageCount(); ++count)
            {
                pdfDoc.savePageAsJPEG(count, "output_" + count + ".jpg", ↩
                    150, 0.8f);
            }
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
    }
}
```

Capitolo 3

Barcode Xpress

Barcode Xpress è progettato per leggere, scrivere ed interpretare codici a barre mono e bidimensionali per lo sviluppo di applicazioni che prevedono l'archiviazione e l'elaborazione dei dati. *Barcode Xpress* è in grado di localizzare i codici a barre presenti nelle pagine di un documento, decodificare ed interpretare le informazioni che contengono e scrivere codici a barre.

3.1 Perché Barcode Xpress?

La possibilità offerta da *Barcode Xpress* di specificare la direzione di lettura dell'immagine in input consente di effettuare un controllo rapido della pagina e in un secondo momento, solo se necessario, di effettuare una ricerca approfondita (che ovviamente richiede un tempo di elaborazione maggiore). Inoltre è in grado di riconoscere svariati tipi di *barcode* semplicemente cambiando un campo della classe *ReadOptions* e per questo può essere ancora utilizzata anche nel caso in cui Enel scelga un nuovo tipo di *barcode* per codificare i suoi documenti.

3.2 Le classi

La library *Barcode Xpress* mette a disposizione dell'utente tre classi:

- *tasman.bars.BarCode*,
- *tasman.bars.BarReader*,
- *tasman.bars.ReadOptions*.

Le suddette classi sono tutte ugualmente necessarie nel processo di individuazione e decodifica dei *barcode*. Infatti, per leggere i codici a barre contenuti in un'immagine è necessario:

1. Creare un'istanza della classe *BarReader*.
2. Creare un'istanza della classe *ReadOptions*.
3. Impostare i campi dell'oggetto *ReadOptions* per specificare varie opzioni, quali:
   - la simbologia del *barcode* utilizzato
   - la direzione di lettura.
4. Invocare il metodo *BarReader.readBars(tasman.bars.ReadOptions)*. Questo restituisce in output un array di *BarCode*; ogni elemento dell'array corrisponde ad un *barcode* letto dall'immagine.

Ciascuno dei punti toccati nel precedente elenco è illustrato nel seguente esempio:

```
// 1: Crea un BarReader per l'immagine specificata.
BarReader br = NEW BarReader(myImage);
// 2: Crea un ReadOptions.
ReadOptions options = NEW ReadOptions();
options.readEast = TRUE;  // 3: Legge da sinistra a destra.
options.code128 = TRUE;   // 3: Specifica il tipo di barcode.
// 4: Legge i barcode contenuti nell'immagine.
BarCode[] bars = br.readBars(options);
FOR (INT i = 0; i < bars.length; i++)    // Stampa i risultati.
  System.out.println(bars[i].toString());
```

Una singola istanza dell'oggetto BarReader può essere utilizzata per leggere la stessa immagine più volte, con opzioni differenti. Ad esempio:

```
// 1: Crea un BarReader per l'immagine specificata.
BarReader br = NEW BarReader(myImage);
// 2: Crea un ReadOptions.
ReadOptions options = NEW ReadOptions();
options.readEast = TRUE;  // 3: Legge da sinistra a destra.
options.code128 = TRUE;   // 3: Specifica il tipo di barcode.
// 4: Legge i barcode contenuti nell'immagine.
BarCode[] bars = br.readBars(options);
IF (bars.length == 0) {
  options.readEast = FALSE;      // Nessun barcode trovato. Forse
  options.readWest = TRUE;       // l'immagine è stata scannerizzata
  bars = br.readBars(options);   // sotto-sopra. Controlliamo
}
```

3.2.1 La classe BarCode

Rappresenta un *barcode* contenuto in un'immagine.

Il metodo *BarReader.readBars()* restituisce un array di istanze di questa classe; questo array contiene un elemento per ogni *barcode* rilevato nell'immagine

analizzata. I metodi di questa classe forniscono dettagli sul *barcode*, tra cui la simbologia e la posizione all'interno dell'immagine.

Nella tabella seguente sono riportati alcuni metodi della classe *BarCode*:

METODO	DESCRIZIONE
getErrorCorrectionLevel()	restituisce un intero che indica il livello di correzione degli errori, o -1 se la simbologia non supporta livelli di correzione multipli
getReadDirection()	restituisce un intero che rappresenta la direzione di lettura del *barcode*
getString()	restituisce le informazioni contenute nel *barcode*
getSymbologyAsString()	restituisce la simbologia del *barcode*

3.2.2 La classe BarReader

Legge i *barcode* contenuti in un'immagine. Questa classe è dotata di:

- un costruttore, che accetta come parametro un oggetto della classe *java.awt.image.RenderedImage*
- il metodo *readBars*, che ha come parametro un oggetto della classe *ReadOptions* e restituisce un array di *BarCode*.

Per effettuare una lettura è necessario istanziare un oggetto *BarReader* e quindi invocare il metodo *readBars*, come mostrato in entrambi gli esempi precedenti.

3.2.3 La classe ReadOptions

Permette di specificare le opzioni di lettura. La classe *ReadOptions* possiede venticinque campi booleani, ciascuno dei quali associato ad una specifica simbologia di *barcode*. Le simbologie supportate sono:

Code 11	*Code 128*	*Code 32*
Code 39	*Code 93*	*Australia Post 4 State*
Codabar	*Data Matrix (ECC 200)*	*EAN13*
EAN8	*Interleaved 2 of 5*	*Intelligent Mail*
ITF-14	*Micro QR*	*Patch Codes*
PDF417	*Planet*	*Postnet*
QR	*RM4SCC*	*RSS-14*
RSS Limited	*Telepen*	*UPC A* e *UPC E*

Oltre a questi, *ReadOptions* possiede quattro campi per specificare la direzione di lettura dell'immagine:

CAMPO	DESCRIZIONE
readEast	legge l'immagine orizzontalmente, da sinistra a destra. Inizializzato a TRUE dal costruttore
readNorth	legge l'immagine verticalmente, dal basso all'alto. Inizializzato a FALSE dal costruttore
readSouth	legge l'immagine verticalmente, dall'alto al basso. Inizializzato a FALSE dal costruttore
readWest	legge l'immagine orizzontalmente, da destra a sinistra. Inizializzato a FALSE dal costruttore

Per concludere, *ReadOptions* possiede altri due campi che meritano particolare attenzione:

- *scanInterval*
- *scanBarsToRead*

scanInterval

L'intervallo di scansione. Impostarlo a uno significa effettuare la scansione di ogni riga o colonna di pixel dell'immagine; un valore di due comporta la scansione ogni due pixel, e così via.

Aumentare questo valore può portare ad una lettura più veloce dell'immagine, ma aumenta la probabilità di non riconoscere *barcode* "stretti" o malformati. Ridurre questo valore può produrre l'effetto opposto: una lettura più lenta dell'immagine ma un migliore riconoscimento dei *barcode* (*scanInterval* è inizializzato a cinque dal costruttore). Aumentare il valore di questo campo non comporta automaticamente una lettura più rapida dell'immagine: se il *BarReader* individua una struttura che potrebbe essere un *barcode*, allora l'intervallo di scansione viene temporaneamente ridotto per effettuare una migliore analisi.

Il valore di questo campo è irrilevante nella ricerca di *barcode* bidimensionali (i.e. *Data Matrix*, *QR Code*, *Micro QR*).

scanBarsToRead

Inizializzato a uno dal costruttore, *scanBarsToRead* rappresenta il numero di *barcode* che l'oggetto *BarReader* deve individuare all'interno di una immagine contenente tre o più colori. Il valore di questo campo è irrilevante quando si analizzano immagini con due soli colori (il bianco e il nero, ad esem-

pio). Se il numero di *barcode* riconosciuti è inferiore al numero specificato in questo campo, l'immagine viene analizzata per variazioni di colore, contrasto e illuminazione ed in seguito viene effettuata una nuova scansione.

Impostare questo valore al numero di *barcode* contenuti nell'immagine a colori che si vuole analizzare permette di migliorare sensibilmente le prestazioni.

Capitolo 4

Implementazione

In questo capitolo tratteremo di come le tre Java *Libraries* sinora descritte sono state integrate nel software realizzato da SSDI per ENEL. Come accennato a pagina 3, durante il mio periodo di tirocinio ho avuto occasione di occuparmi della parte del progetto relativa al caricamento *asincrono* dei documenti.

Utilizzando la modalità di caricamento *asincrono*, l'utente esegue l'upload dei suoi documenti in una cartella specifica, sulla quale il sistema effettua un controllo ciclico per verificare la presenza di nuovi file (*polling*). Se la cartella non è vuota, il sistema sceglie il primo file che non è ancora stato processato e ne esegue la lettura, ricercando il *barcode* contenuto nel documento e, in seguito al suo corretto riconoscimento, provvede all'archiviazione del file, spostandolo in una directory specifica e associandolo ad un nuovo *record* del database. Se il processo di riconoscimento ha esito negativo, il file viene spostato in un'altra directory, che contiene tutti i file rigettati dal sistema.

4.1 La classe BarcodeRdr

Per prima cosa è stata creata la classe *BarcodeRdr*, che sfrutta le classi fornite dalla library *Barcode Xpress* per eseguire la lettura e il riconoscimento del *barcode* presente sulla prima pagina del documento. Questa classe fornisce la *function getPDF417Barcode*, la quale ha come parametro di input il *path* completo — in formato STRING — dell'immagine di cui vogliamo effettuare la scansione e restituisce in output il valore contenuto nell'eventuale *barcode* rilevato, anch'esso in formato STRING. La *function getPDF417Barcode* svolge il suo compito attraverso i seguenti passaggi:

1. Viene creata una nuova istanza dell'oggetto *BarReader* a partire dal *path* che è stato passato come input alla funzione.

2. Viene creato un nuovo oggetto *ReadOptions*, con cui specifichiamo che:

   - effettueremo una scansione orizzontale, da sinistra verso destra,
   - utilizzeremo la simbologia *PDF417*.

3. Viene creato un vettore di oggetti *BarCode*, che avrà come elementi i *barcode* riconosciuti a seguito della scansione.

4. Viene effettuata la scansione dell'immagine in input, utilizzando il metodo *readBars* della classe *BarReader*.

A questo punto, effettuiamo un controllo sulla dimensione del vettore di *BarCode*; se questo contiene almeno un elemento, la funzione restituisce immediatamente in output il valore del *barcode* associato al primo elemento del vettore. Se invece il vettore risulta essere vuoto, vengono modificati i campi dell'oggetto *ReadOptions* in modo da effettuare una nuova scansione sia orizzontale che verticale, in entrambe le direzioni possibili (da destra

a sinistra, dall'alto al basso e vice versa). In seguito a questa seconda — lenta — scansione, se non è stato ancora riconosciuto alcun *barcode* all'interno dell'immagine, la funzione termina e restituisce NULL.

Il codice completo della classe *BarcodeRdr* è a pagina 31.

Listing 4.1: La classe BarcodeRdr

```
public class BarcodeRdr {
   public String getPDF417Barcode(String imagePath){
      String barcode = null;
      File imageFile;
      try {
         BufferedImage image = null ;
         imageFile = new File(imagePath);
         image = ImageIO.read(imageFile);
         BarReader br = new BarReader(image);
         ReadOptions options = new ReadOptions();
         // Di default, la scansione avviene da sinistra a destra
         options.readEast = true;
         // Il barcode utilizzato da Enel è del tipo PDF417,
         // di conseguenza impostiamo l'opzione corretta
         options.codePDF417 = true;
         BarCode[] bars = br.readBars(options);
         // Se il barcode non è stato individuato,
         // esegue una scansione approfondita
         if (bars.length == 0 &&  options.readNorth == false &&
            options.readSouth == false && options.readWest == false)
         {
            options.readNorth = true;
            options.readSouth = true;
            options.readWest = true;
            System.out.println("No barcode found. Advanced searching");
            bars = br.readBars(options);
            if (bars.length == 0 ){
               System.out.println( "No barcode found." );
               return null;
            }
            barcode = bars[0].getString();
         }
         else barcode = bars[0].getString();
      } catch (Exception ex) {
         ex.printStackTrace();
         return null;
      }
      if (imageFile != null) imageFile.delete();
      return barcode;
   }
}
```

4.2 La classe ImageExtractor

ImageExtractor utilizza i metodi della classe *PDFDocument* per convertire in formato PNG la prima pagina del file PDF il cui nome e *path* completo vengono richiesti in input dalla funzione *convert*. Effettuiamo la conversione solo della prima pagina perché il *barcode* che contiene le informazioni necessarie alla catalogazione del documento viene posto sempre ed esclusivamente su questa pagina. Una volta lette le suddette informazioni, il file può essere archiviato correttamente e l'immagine PNG che abbiamo creato viene cancellata. La risoluzione dell'immagine generata è di 350 DPI, in modo da facilitare il riconoscimento del *barcode*.

```
public class ImageExtractor {
   public String convert(String filesPath, String fileName) {
      /* Crea una stringa contenente il nome del nuovo file,
       * che ha lo stesso nome del file di input,
       * ma estensione png */
      String imageName = fileName.substring(0,fileName.length()-4);
      imageName = imageName + ".png";
      String imagePath = filesPath + "//" + imageName;
      try {
         /* Crea un'istanza PDFDocument a partire dal path completo
            del file di input */
         PDFDocument pdfDoc = new PDFDocument (filesPath + "//" + fileName
             , null);
         // Fissiamo i dpi a 350 e convertiamo solo la prima pagina
         pdfDoc.savePageAsPNG(0, imagePath, 350);
      } catch (Throwable t) {
        System.out.println("Exception "+ t.toString() +" in "+ imageName);
      }
      return imageName;
   }
}
```

La classe ImageExtractor

4.3 La classe PdfSplitter

Questa classe si occupa di generare le anteprime dei documenti archiviati. Le anteprime consistono non in file immagine ottenuti attraverso la conversione delle singole pagine del documento, ma in file PDF realizzati selezionando le prime tre pagine del documento originale e scartando le rimanenti. Ovviamente in questi file di anteprima non è presente la pagina che contiene il *barcode*, in quanto l'utente può visualizzare l'anteprima di un documento solo dopo che questo è stato riconosciuto ed archiviato all'interno del sistema. La classe *PdfSplitter* sfrutta numerose classi della library *iText*, tra cui:

- *PdfReader*, per leggere il file PDF originale e recuperare informazioni circa il numero di pagine totali e le dimensioni del foglio, utilizzando rispettivamente i metodi *getNumberOfPages* e *getPageSizeWithRotation*.
- *PdfCopy*, per creare il file di anteprima come copia delle prime quattro pagine del PDF originale, attraverso i metodi *getImportedPage* e *addPage*.
- *Image*, per inserire il marchio di anteprima nel PDF generato.
- *PdfContentByte*, per specificare la posizione del marchio di anteprima all'interno della pagina.

Come prima cosa viene invocata la *funtion generatePreview* sul file PDF originale, in modo da ottenerne una copia delle prime quattro pagine; questa copia viene creata all'interno della directory *preview*. La stringa *outPreviewFileName* — generata come output di questa prima funzione — sarà quindi passata in input alla *function dropFirstPage*, che provvederà ad eliminare la prima pagina dal file di anteprima, dato che questa contiene solamente il *barcode*.

Segue il codice completo della classe.

La classe PdfSplitter

```
import java.io.File;
import java.io.FileOutputStream;
import com.lowagie.text.Document;
import com.lowagie.text.Image;
import com.lowagie.text.pdf.PdfContentByte;
import com.lowagie.text.pdf.PdfCopy;
import com.lowagie.text.pdf.PdfImportedPage;
import com.lowagie.text.pdf.PdfReader;
import com.lowagie.text.pdf.PdfStamper;

public class PdfSplitter {
   public String generatePreview(String filename,String filenameOriginal) ↩
       {
      String outPreviewFileName = null;
      try {
         String filespath = System.getProperty("barcodereaderfilespath");
         System.out.println ("* barcode reader: Reading " + filespath + "↩
             //" + filename + " *");
         String filepath = filespath + "//" + filename;
         PdfReader reader = new PdfReader(filepath);
         int n = reader.getNumberOfPages();
         System.out.println ("* barcode reader: number of pages of the ↩
             file " + filename + " : " + n + " *");
         int i = 0;
         String outPreviewFile = filespath + "//preview" + ↩
             filenameOriginal;
         outPreviewFileName  = "preview" + filenameOriginal;
         System.out.println ("* barcode reader: Writing out preview : " + ↩
             outPreviewFile + " *");
         /* Crea un nuovo documento, che sarà poi il PDF di anteprima,
          * utilizzando le stesse dimensioni di pagina e
          * lo stesso orientamento del file originale */
         Document document = new Document(reader.getPageSizeWithRotation↩
             (1));
         FileOutputStream previewFileStream = new FileOutputStream(↩
             outPreviewFile);
         PdfCopy writer = new PdfCopy(document, previewFileStream);
```

```
        document.open();
        while ( i < 3 ) {
           PdfImportedPage page = writer.getImportedPage(reader, ++i);
           writer.addPage(page);
        }
        /* Una volta invocato il metodo close degli oggetti
         * document e writer, vengono applicate le modifiche
         * ai file cui questi oggetti si riferiscono */
        document.close();
        writer.close();

        // Istanzia l'oggetto img con l'immagine del marchio anteprima
        String watermarkpath = System.getProperty("watermarkpath");
        Image img = Image.getInstance(watermarkpath + "//watermark.jpg");
        if (img != null) {
           PdfReader readerP = new PdfReader(outPreviewFile);
           File watermarked = new File(filespath + "//temp" + ↩
               outPreviewFileName);
           PdfStamper stamp = new PdfStamper(readerP, new ↩
               FileOutputStream(watermarked));
           PdfContentByte under;
           img.setAbsolutePosition(200, 400);
           // Aggiunge il watermark di anteprima alla seconda pagina
           under = stamp.getUnderContent(1);
           under.addImage(img);
           stamp.close();
           // Cancella il file di anteprima creato dall'oggetto writer
           File preview = new File(outPreviewFile);
           preview.delete();
           /* Assegna al PDF appena creato il nome del file
            * che è stato appena cancellato */
           watermarked.renameTo(preview);
        }
     } catch (Exception e) {
        System.out.println("PDF Splitter generatePreview : " +
           e.toString());
     }
     return outPreviewFileName;
  }
```

```
    public String dropFirstPage(String filename) {
        String outFileName = null;
        try {
            String filespath = System.getProperty("barcodereaderfilespath");
            // Individua il path completo del file di input
            String filepath = filespath + "//" + filename;
            PdfReader reader = new PdfReader(filepath);
            int n = reader.getNumberOfPages();
            int i = 1;
            /* Assegna al nuovo file il prefisso new, in modo da
             * poterlo distinguere dall'originale */
            String outFile = filespath + "//new" + filename;
            outFileName = "new" + filename;
            Document document = new Document(reader.getPageSizeWithRotation↩
                (1));
            PdfCopy writer = new PdfCopy(document, new FileOutputStream(↩
                outFile));
            document.open();
            /* Il nuovo file conterrà tulle pagine dell'originale,
             * tranne la prima, che non viene copiata */
            while ( i < n ) {
                PdfImportedPage page = writer.getImportedPage(reader, ++i);
                writer.addPage(page);
            }
            /* Una volta invocato il metodo close degli oggetti
             * document e writer, vengono applicate le modifiche
             * ai file cui questi oggetti si riferiscono */
            document.close();
            writer.close();
        } catch (Exception e) {
            System.out.println("PDF Splitter dropFirstPage : " +
                e.toString());
        }
        return outFileName;
    }
}
```

4.4 La classe Runner

La classe *Runner* ha il compito di generare dei *thread* che eseguiranno le istruzioni contenute nella classe *PDFRunner*. Dato che la classe *Runner* viene eseguita all'avvio dell'*application server*, inizialmente mettiamo in stato di sleep il *thread* associato alla nostra classe, in modo da permettere all'*application server* di terminare correttamente la sua procedura di avviamento. Una volta terminata l'esecuzione del *thread* che ha generato, *Runner* si pone in stato di sleep per una quantità variabile di millisecondi (stabilita da una variabile di sistema), in modo da consentire l'esecuzione di altri *thread*. Una volta terminato questo tempo di attesa, la classe *Runner* riprende la sua esecuzione, generando un nuovo *thread* associato ad un nuovo oggetto *PDFRunner*.

La pagina 38 riporta il codice completo della classe.

Listing 4.2: La classe Runner

```
import java.util.Date;
public class Runner implements Runnable {
   public void run() {
     try {
        String brstartuptime = "40000";
        System.out.println("* barcode reader will start in : " + ↩
            brstartuptime + " ms *");
        /* Mette in sleep il nostro thread per permettere
         * all'application server di salire */
        Thread.sleep(new Long(brstartuptime));
     } catch (InterruptedException exc) {}
     // La classe Runner esegue il suo compito all'infinito
     while (true) {
         PDFRunner pdfrunner = new PDFRunner();
         Date day = new Date();
         Long id = day.getTime();
         // Il nome del thread è un ID random
         Thread threadpdf = new Thread(pdfrunner,new Long(id).toString());
         threadpdf.start();
         try {
            threadpdf.join();
         }
         catch (Exception e) {
            System.err.println("Eccezione in run : " + e.toString());
         }
         try
         {
         // Concede una opportunità di esecuzione agli altri thread
            String brpause = System.getProperty("barcodereaderpause");
            System.out.println("* barcode reader pause for : " + brpause +↩
                 " ms *");
            Thread.sleep(new Long(brpause));
         }
         catch (InterruptedException exc)
         {
          // Lo stato di sleeping è stato interrotto
          // da un altro thread
         }
      }
   }
}
```

4.5 La classe PDFRunner

La classe *PDFRunner* sfrutta tutte le classi di cui abbiamo parlato finora per:

- controllare la presenza di un nuovo file da processare,
- convertire la prima pagina del PDF in formato PNG,
- effettuare la scansione di questa immagine cercando l'eventuale *barcode* associato al documento.

Se il *barcode* non viene riconosciuto o se il numero di protocollo a cui è associato è già presente nel database, il file viene spostato nella directory riservata ai documenti rifiutati. Nel caso in cui da questa *query* risulti che il documento non è presente nel database, *PDFRunner* provvede — attraverso l'uso di *Java Beans* appositamente creati — ad inserire il documento, la sua anteprima e le informazioni che esso contiene nel sistema *FileNet*, che a sua volta si occuperà di scrivere queste modifiche nel database.

A pagina 40 e nelle pagine successive è riportato il codice della classe, mentre a pagina 46 vi è una breve introduzione ai *Java Beans* e a come sono stati utilizzati nella realizzazione di questo progetto.

Listing 4.3: La classe PDFRunner

```
import it.pr.ssdi.barcodereader.beans.DocumentoElettronico;
import it.pr.ssdi.barcodereader.beans.DocumentoElettronicoId;
import it.pr.ssdi.barcodereader.beans.LogDocumentiMassivo;
import it.pr.ssdi.barcodereader.beans.TipoImmagine;
import it.pr.ssdi.barcodereader.facade.DocumentoElettronicoManage;
import it.pr.ssdi.barcodereader.facade.LogDocumentiMassivoManage;
/* Tutte le classi finora importate sono dei Java Beans,
 * creati ed utilizzati per interfacciare il software da noi creato
 * con il database sottostante */
import it.enel.ae.barcodereader.isra.IDocument;
import it.enel.ae.barcodereader.isra.Document;
import it.enel.ae.barcodereader.isra.FilenetISRAConnectionLibrary;
import it.enel.ae.barcodereader.BarcodeRdr;
import it.enel.ae.barcodereader.pdfsplitter.PdfSplitter;
import it.enel.ae.barcodereader.imageExtractor.ImageExtractor;
import java.io.File;
import java.io.FileInputStream;
import java.util.Date;
import java.util.List;

public class PDFRunner implements Runnable {
   private LogDocumentiMassivoManage ldmm = new LogDocumentiMassivoManage↩
       ();
   private DocumentoElettronicoManage dem = new DocumentoElettronicoManage↩
       ();

   public void run()  {
      LogDocumentiMassivo item = null;
      File objFile = null;
      File objFileOriginal = null;
      File objFilePreview = null;
      String filename = null;
      String filenamepreview = null;
      String filenameOriginal = null;
      String filespath = System.getProperty("barcodereaderfilespath");
      String filespathrejected = System.getProperty("↩
          barcodereaderfilesrejpath");
      System.out.println("* Barcode reader filespath "+ filespath + " *");
      String imageName = null;

      try {
         List<LogDocumentiMassivo> ldmList =  ldmm.getListaLogToProcess();
```

```
        System.out.println("* Barcode Reader is alive *");
        if (ldmList.size() >0) {
           System.out.println("* Barcode Reader got a file to process*");
           item = (LogDocumentiMassivo)ldmList.get(0);
           filenameOriginal = item.getDocumento();
           objFileOriginal = new File(filespath + "//" + filenameOriginal↩
               );
           try {
              // Esegue la conversione del PDF in PNG ...
              filenameOriginal = item.getDocumento();
              ImageExtractor imageexct = new ImageExtractor();
              imageName = imageexct.convert(filespath, filenameOriginal);
           } catch (Exception e) {
              File imageFile = new File(filespath + "//" + imageName);
              imageFile.delete();
              throw new BarcodeReaderException(ExceptionCostanti.↩
                  BARCODE_IMAGE_ERROR ,"No barcode image extracted from ↩
                  document.");
              // Sposta il file tra i rigettati
           }
           // ... effettua la scansione per riconoscere il barcode e ...
           BarcodeRdr reader = new BarcodeRdr();
           String numeroprotocollo = reader.getPDF417Barcode(filespath + ↩
               "//" + imageName);
           DocumentoElettronico depresence = null;
           /* ... legge il numero di protocollo, controllando che
            * il documento non sia già presente nel database */
           if (numeroprotocollo != null) {
              depresence = dem.getDocumentoElettronicoByNumeroProtocollo(↩
                  new Integer(numeroprotocollo));
              if (depresence != null)
                 throw new BarcodeReaderException(ExceptionCostanti.↩
                     BARCODE_PRESENCE_ERROR ,"Document already present in↩
                      FilenetIS.");
           }
           if (numeroprotocollo != null && depresence == null) {
              try {
                 System.out.println("* Barcode reader: protocol number "+↩
                      numeroprotocollo +" *");
                 PdfSplitter pdfs = new PdfSplitter();
                 filename = pdfs.dropFirstPage(filenameOriginal);
                 filenamepreview = pdfs.generatePreview(filename,↩
                     filenameOriginal);
```

```
            System.out.println("* Barcode reader: filespath:   "+ ↩
                filespath +"//"+ filename +" *");
            objFile = new File(filespath + "//" + filename);
        } catch (Throwable t) {
            throw new BarcodeReaderException(ExceptionCostanti.↩
                BARCODE_PATH_ERROR ,"Files path error from system.↩
                properties.");
            // Sposta il file tra i rigettati
        }
        IDocument docreturned = null;
        IDocument docreturnedpreview = null;
        try {
            objFilePreview = new File(filespath + "//" + ↩
                filenamepreview);
            // Istanzia un oggetto FileInputStream
            FileInputStream objFileInputStreamPreview = new ↩
                FileInputStream(objFilePreview);
            // Recupero le dimensioni del file
            long lengthPreview = objFilePreview.length();
            /* Istanzia un'array di byte passando
             * la lunghezza del file (long) */
            byte[] objArrayBytePreview = new byte[(int) ↩
                lengthPreview];
            // Ciclo e inserisco nell'array di Byte
            objFileInputStreamPreview.read(objArrayBytePreview);
            // Chiude il file stream
            objFileInputStreamPreview.close();
            // Inserisce le anteprime in FileNet
            FilenetISRAConnectionLibrary isra = new ↩
                FilenetISRAConnectionLibrary();
            String user = System.getProperty("filenetisuser");
            String pwd = System.getProperty("filenetispwd");
            isra.setISRAUserName(user);
            isra.setISRAPassword(pwd);
            IDocument previewdoc = new Document();
            previewdoc.setClassName(System.getProperty("filenetisdoc↩
                "));
            previewdoc.setFileName("prev" + numeroprotocollo +  ".↩
                pdf");
            previewdoc.setContentType("application/pdf");
            previewdoc.setContent(objArrayBytePreview);
            docreturnedpreview = isra.addDocument(previewdoc);
            objFilePreview.delete();
```

```
            FileInputStream objFileInputStream = new FileInputStream↩
                (objFile);
            long length = objFile.length();
            byte[] objArrayByte = new byte[(int) length];
            objFileInputStream.read(objArrayByte);
            objFileInputStream.close();
            // Inserisce il file principale in Filenet
            IDocument doc = new Document();
            doc.setClassName(System.getProperty("filenetisdoc"));
            doc.setFileName(numeroprotocollo +  ".pdf");
            doc.setContentType("application/pdf");
            doc.setContent(objArrayByte);
            docreturned = isra.addDocument(doc);
            objFile.delete();
         } catch(ApplicationException e) {
            throw new BarcodeReaderException(ExceptionCostanti.↩
                ISRA_NOP,"Exception insert document in FilenetIS");
            // Sposta il file tra i rigettati
         } catch(Throwable t) {
            throw new BarcodeReaderException(ExceptionCostanti.↩
                ISRA_NOP,"Exception insert document in FilenetIS");
            // Sposta il file tra i rigettati
         }
         if  (docreturned != null && docreturnedpreview != null && ↩
             docreturned.getID() != null) {
            try {
               // Inserimento del nuovo documento nel db
               DocumentoElettronico de = new DocumentoElettronico();
               DocumentoElettronicoId dei = new ↩
                   DocumentoElettronicoId();
               Long docIdLong = docreturned.getID();
               dei.setIdContent(new Integer(docIdLong.intValue()));
               dei.setNumeroProtocolloDoc(new Integer(↩
                   numeroprotocollo).intValue());
               de.setId(dei);
               de.setAssociato(1);
               de.setAnteprima(0);
               de.setDescrizione(filenameOriginal);
               TipoImmagine ti = new TipoImmagine();
               ti.setIdTipoImmagine(1);
               ti.setNome("PDF");
               ti.setDescrizione("application/pdf");
               de.setTipoImmagine(ti);
               dem.setDocumentoElettronico(de);
```

```
                    // Inserimento dei dati relativi all'anteprima nel db
                    DocumentoElettronico deant = new DocumentoElettronico↩
                        ();
                    DocumentoElettronicoId deiant = new ↩
                        DocumentoElettronicoId();
                    Long docPreviewIdLong = docreturnedpreview.getID();
                    deiant.setIdContent(new Integer(docPreviewIdLong.↩
                        intValue()));
                    deiant.setNumeroProtocolloDoc(new Integer(↩
                        numeroprotocollo).intValue());
                    deant.setId(deiant);
                    deant.setAssociato(1);
                    deant.setAnteprima(1);
                    deant.setDescrizione(filenamepreview);
                    TipoImmagine tiant = new TipoImmagine();
                    tiant.setIdTipoImmagine(1);
                    tiant.setNome("PDF");
                    tiant.setDescrizione("application/pdf");
                    deant.setTipoImmagine(tiant);
                    dem.setDocumentoElettronico(deant);
                    System.out.println("* barcode reader : doc id ↩
                        returned from filenet =  " + docreturned.getID().↩
                        toString() + " *");
                    /* Tutto ok: imposta a true il campo processato sulla
                     * tabella log_documento_massivo e aggiorna lo stato
                     */
                    item.setProcessato(1);
                    item.setDataprocessato(new Date());
                    item.setStato(1);
                    item.setNote("Documento inserito con successo");
                    ldmm.updateLog(item);
                    System.out.println("* barcode reader : Document ↩
                        processed with success *");
                } catch (Exception e) {
                    throw new BarcodeReaderException(ExceptionCostanti.↩
                        BARCODE_DOC_ERROR ,"No Doc present in Documento ↩
                        Armadio.");
                }
            } else {
                throw new BarcodeReaderException(ExceptionCostanti.↩
                    BARCODE_DOC_ID_ERROR ,"No Doc Id returned from ↩
                    FilenetIS.");
                // Sposta il file tra i rigettati
            }
```

```
            } else {
                File imageFile = new File(filespath + "//" + imageName);
                imageFile.delete();
                throw new BarcodeReaderException(ExceptionCostanti.↩
                    BARCODE_READER_ERROR ,"No protocol number extracted ↩
                    from barcode.");
                // Sposta il file tra i rigettati
            }
        }
      } catch (BarcodeReaderException e) {
        /* NON ok: imposta a true il campo processato
         * sulla log_documento_massivo e aggiorna lo stato a 2 */
        if (item != null)  {
            item.setProcessato(1);
            item.setDataprocessato(new Date());
            item.setStato(2);
            item.setNote(e.getMessageWithoutCause());
            ldmm.updateLog(item);
            // Sposta il file tra i rigettati
            objFileOriginal.renameTo(new File(filespathrejected + "//" + ↩
                filenameOriginal));
            System.out.println("* barcode reader: Document process ko *");
        }
      } finally {
        if (objFile != null)  objFile.delete();
        if (objFileOriginal != null)  objFileOriginal.delete();
        if (objFilePreview != null)  objFilePreview.delete();
      }
    }
}
```

4.6 Un accenno ai Java Beans

Le *Java Beans* (letteralmente, chicchi di Java) sono classi scritte in linguaggio di programmazione Java secondo una particolare convenzione. Sono usate per incapsulare molti oggetti in un singolo oggetto (il *bean*), così da poter passare il *bean* invece degli oggetti individuali. Al fine di funzionare come una classe *Java Bean*, una classe di un oggetto deve obbedire a certe convenzioni in merito ai nomi, alla costruzione e al comportamento dei metodi. Queste convenzioni rendono possibile avere dei *tool* che possono usare, riusare, sostituire e connettere *Java Bean*. Le convenzioni richieste sono:

1. La classe deve avere un costruttore senza argomenti.

2. Le sue proprietà devono essere accessibili usando *get*, *set* e altri metodi, seguendo una convenzione standard per i nomi.

3. La classe dovrebbe essere serializzabile (capace di salvare e ripristinare il suo stato in modo persistente).

4. Non dovrebbe contenere alcun metodo richiesto per la gestione degli eventi.

Una proprietà è un singolo attributo pubblico. Le proprietà possono essere in lettura/scrittura, sola lettura o sola scrittura, e possono essere di vari tipi: semplici, indicizzate, *bound* e *constrained*. Una proprietà semplice rappresenta un singolo valore e può essere definita da una coppia di metodi *set*/*get*. Il nome della proprietà deriva dal nome di tali metodi. La sola presenza del metodo *get* indica che la proprietà è in sola lettura, la sola presenza del metodo *set* indica che la proprietà è in sola scrittura, mente la presenza di entrami i metodi indica una proprietà in lettura e scrittura.

Nella realizzazione di questo progetto sono state utilizzate numerose classi *Java Bean*, ciascuna associata ad una tabella del database, in modo da rendere più efficiente tutto il software. Se, per un qualsiasi motivo, si deve modificare una tabella del database (per esempio aggiungendo o rimuovendo un campo) è sufficiente modificare la sola classe *Bean* associata a questa tabella, aggiungendovi — o cancellandone — le proprietà relative al campo che è stato modificato.

In conclusione di questa breve introduzione ai *Java Beans*, portiamo ad esempio la classe *TipoImmagine*, utilizzata all'interno della classe *PDFRunner*.

```
public class TipoImmagine implements java.io.Serializable {
   private Integer idTipoImmagine;
   private String nome;
   private String descrizione;
   public TipoImmagine() {}
   public TipoImmagine(Integer idTipoImmagine) {
      this.idTipoImmagine = idTipoImmagine;
   }
   public TipoImmagine(Integer idTipoImmagine, String nome, String ↩
       descrizione, Set documentoElettronicos) {
      this.idTipoImmagine = idTipoImmagine;
      this.nome = nome;
      this.descrizione = descrizione;
   }
   public Integer getIdTipoImmagine() {
      return this.idTipoImmagine;
   }
   public void setIdTipoImmagine(Integer idTipoImmagine) {
      this.idTipoImmagine = idTipoImmagine;
   }
   public String getNome() {
      return this.nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
   public String getDescrizione() {
      return this.descrizione;
   }
   public void setDescrizione(String descrizione) {
      this.descrizione = descrizione;
   }
}
```

Capitolo 5

Conclusioni e sviluppi futuri

In questo lavoro di tesi abbiamo illustrato una parte del sistema di archiviazione documenti realizzato da SSDI per ENEL. Tali documenti cartacei vengono inizialmente scannerizzati — e quindi digitalizzati — per poi essere inviati al nostro sistema, che provvederà a riconoscerne la tipologia e ad effettuarne la corretta archiviazione all'interno del database. Il riconoscimento della tipologia dei documenti avviene attraverso la lettura e la decodifica di un codice a barre (*barcode*) stampato sulla prima pagina di ciascun documento. La simbologia adottata da ENEL per i suoi *barcode* è lo standard PDF417.

In particolare abbiamo descritto le librerie e le classi utilizzate per la realizzazione del caricamento *asincrono* dei documenti.

È doveroso chiarire che l'intero sistema non è ancora completo, e quindi alcune classi potranno essere ulteriormente modificate e migliorate. Tra le possibili migliorie, una sicuramente potrebbe essere l'implementare via codice un metodo per capire come è stato scannerizzato il foglio in cui è presente il *barcode* e impostare di conseguenza la direzione di lettura nella *library Barcode Xpress* (mentre al momento la lettura avviene sempre — in un primo momento — orizzontalmente, da sinistra a destra e in caso di mancato

riconoscimento viene effettuata una seconda scansione del foglio in tutte le altre direzioni). Inoltre la fase di testing del nostro *barcode reader* non è stata esaustiva, in quanto ENEL ha fornito soltanto una decina di documenti di prova.

Bibliografia

[1] iText Homepage.
http://www.lowagie.com/iText/.

[2] JavaBean - Wikipedia - L'enciclopedia libera.
http://it.wikipedia.org/wiki/JavaBean.

[3] jPDFImages Developer Guide.
http://www.qoppa.com/pdfimages/guide/guide.html.

[4] PDF417 - Wikipedia - L'enciclopedia libera.
http://en.wikipedia.org/wiki/PDF417.

[5] Smartscan Xpress Barcode.
http://www.microway.com.au/catalog/pegasus/
smartscanbarcode.stm.