

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

TECNOLOGIE WEB T

**Progettazione e Realizzazione di Servizi Web per la
Gestione di Dati Personali in Documenti OOXML**

CANDIDATO

Lorenzo Mario Amorosa

RELATORE

Chiar.mo Prof. Paolo Bellavista

Anno Accademico 2018/2019

Sessione II

Indice

Introduzione.....	3
1 Scenario di lavoro.....	5
1.1 Minimizzazione dei dati personali.....	5
1.1.1 Anonimizzazione.....	5
1.1.2 Pseudonimizzazione.....	5
1.1.3 Altri trattamenti di “minimizzazione”.....	6
1.2 Punti di partenza per la progettazione del servizio.....	7
2 Definizione delle specifiche.....	9
2.1 Riconoscimento dei nominativi.....	9
2.1.1 Scelta della strategia di riconoscimento.....	9
2.1.2 Definizione dei pattern.....	12
2.1.3 Gestione delle omonimie.....	20
2.1.4 Formattazione dei documenti.....	23
2.2 Analisi dell’usabilità.....	25
2.2.1 Elenco dei nominativi da trattare esplicitamente espressi come dati di input.....	25
2.2.2 Elenco dei nominativi da trattare dedotti automaticamente da un dizionario.....	26
2.2.3 Soluzione ibrida adottata.....	31
2.3 Scelta dei formati da trattare.....	31
2.4 Privacy by Design.....	36
3 Architettura del servizio.....	38
3.1 I componenti software nell’architettura LAMP.....	38
3.2 Principi di progettazione per l’architettura.....	39
3.2.1 Single Responsibility Principle.....	39
3.2.2 Dependency Inversion Principle.....	40

3.3 Stile architetturale REST.....	40
3.4 Moduli software del servizio.....	42
3.4.1 Scelta dei componenti software.....	42
3.4.2 Logica di business e dinamiche di interazione.....	44
3.4.3 Le classi principali del progetto.....	48
4 Approfondimenti tecnologici.....	50
4.1 Analisi della struttura del formato OOXML.....	50
4.1.1 Linguaggi di markup.....	50
4.1.2 File packaging.....	50
4.1.3 Parti di un Documento OOXML.....	52
4.1.4 Analisi del Main Document.....	53
4.1.5 La libreria in ambiente java open source Docx4j.....	55
4.2 Ottimizzazioni del processo di minimizzazione.....	56
4.2.1 Riduzione del testo da trattare.....	56
4.2.2 Ordinamento del dizionario.....	59
4.3 Tecnologie complementari.....	65
4.4 Sviluppi futuri.....	66
4.4.1 Accesso al servizio web.....	67
4.4.2 Ampliamento dinamico del dizionario.....	67
4.4.3 Natura del documento e ricorrenza del nominativo....	68
4.4.4 Altri dati personali.....	68
4.4.5 Altri alfabeti.....	68
Conclusioni.....	69
Bibliografia.....	71
Indice delle figure.....	75
Appendice.....	76

Introduzione

Il recente Regolamento Generale europeo sulla Protezione dei Dati (UE) 2016/679 ([1], [2]) o *GDPR* (*General Data Protection Regulation*, come nel seguito sarà chiamato) ha modernizzato significativamente la normativa in materia di protezione dei dati personali, rendendola omogenea fra tutti gli stati membri. È bene notare che, fin dal titolo, il *GDPR* non riduce gli spazi per i trattamenti di dati personali, ma anzi ne protegge la "libera circolazione", dettando, però, regole definite e certe. Rinunciando ad una presentazione più completa del *GDPR*, saranno riportati nei successivi capitoli i concetti necessari alla discussione dell'argomento.

Enti ed organizzazioni, aventi a che fare con documenti contenenti dati sensibili, devono operare in maniera conforme al *GDPR*; potrebbero, di conseguenza, trarre beneficio da servizi specifici in grado di supportarli in queste esigenze.

L'oggetto principale della tesi sarà dunque, come riportato dal titolo, la progettazione e la realizzazione di un servizio per la gestione di dati personali.

La larga maggioranza dei documenti testuali viene generata con strumenti cosiddetti di "produttività individuale", cioè da word-processor. L'osservazione, ovvia nella sua evidenza, consente di introdurre una riflessione: quando un documento è generato da un'elaborazione automatica, magari per composizione di template con dati estratti da un database, l'individuazione dei "dati personali" nel documento prodotto può avvenire in modo rigoroso e senza incertezze, sulla base degli elementi di composizione del documento; ad es.: i campi del documento estratti da una anagrafica di soggetti sono certamente dati personali e come tali possono essere opportunamente trattati nel processo di elaborazione automatica (ad es. cancellati).

Ma, in quella larga maggioranza di documenti testuali generata con word-processor l'individuazione ed il trattamento dei dati personali vanno affrontati con altri approcci, discussi in questa tesi.

Si parte inizialmente con lo studio di una strategia di individuazione di pattern specifici all'interno del documento, in grado di riconoscere, ad esempio, nomi e cognomi come dati personali eminenti. Il problema si articola da un lato su algoritmi di ricerca supportati da

dizionari, dall'altro nell'analisi delle strutture XML originate da word-processor. Il problema trova la sua ricomposizione nell'implementazione del servizio web. Si applica in particolare un design architetturale REST nella progettazione di un'architettura LAMP. L'engine di processamento dei documenti viene sviluppato in Java, impiegando la libreria Docx4j.

1 Scenario di lavoro

1.1 Minimizzazione dei dati personali

Un concetto espresso in modo pervasivo dal *GDPR* è quello della "*minimizzazione*" dei dati personali trattati ed a maggior ragione dei dati personali pubblicati. In particolare l'Art. 5 ed il Considerando 39 prescrivono che i dati personali trattati siano "*adeguati, pertinenti e limitati a quanto necessario rispetto alle finalità per le quali sono trattati («minimizzazione dei dati»)*" [2].

Con questo fine, il *GDPR* delinea due possibilità organizzative e tecniche di "*minimizzazione*": l'anonimizzazione e la pseudonimizzazione.

1.1.1 Anonimizzazione

Sono anonimizzati i dati personali non (più) riferibili alle persone a cui sono appartenuti; si tratta di serie di dati, spesso ingenti, che sono stati definitivamente ed irreversibilmente separati da ogni riferimento alle persone che caratterizzavano. Sono le serie di dati utilizzate per fini statistici, scientifici, etc. E' importante notare che, come fissato dal Considerando 26 del *GDPR*, il Regolamento non si applica al "*trattamento di tali informazioni anonime*" [2]. Per questo, sottoporre database e documenti ad un procedimento, cioè un trattamento, di anonimizzazione consente di non doversi più preoccupare del *GDPR*.

1.1.2 Pseudonimizzazione

Fra le definizioni dell'Art. 4 del *GDPR*, la "pseudonimizzazione" viene indicata come "*il trattamento tale che i dati personali non possano più essere attribuiti a un interessato specifico senza l'utilizzo di informazioni aggiuntive, a condizione che tali informazioni aggiuntive siano conservate separatamente*" ([2]).

Traducendo in termini "operativi", si tratta del procedimento che, dal "record" dei dati di un interessato, separa i campi che caratterizzano l'interessato stesso da quelli che lo identificano, trasferendo questi ultimi in una nuova distinta tabella; nelle due tabelle vengono aggiunti i riferimenti che permettono la ricostruzione del record originario; il procedimento è completo

quando la nuova tabella con gli identificativi viene archiviata separatamente ed eventualmente cifrata. In margine si annota che in molti casi, come molti studi dimostrano, è possibile identificare, quindi "riconoscere", gli interessati utilizzando la sola tabella con i dati pseudonimizzati.

1.1.3 Altri trattamenti di "minimizzazione"

Un problema pratico che si incontra di frequente è quello di dover pubblicare documenti più o meno ampi che contengono dati personali. Spesso la pubblicazione è obbligatoria per legge: si pensi alle graduatorie relative a concorsi pubblici, alle sentenze dei tribunali, etc.

In questi frequenti casi, analizzando meglio le finalità per le quali sono pubblicati quei dati personali, si osserva che sono possibili e legittimi almeno due approcci per "minimizzare" il dato nel "trattamento di pubblicazione" e, dunque, per minimizzarne la propagazione, ad esempio attraverso motori di ricerca, ben oltre ogni ragionevole finalità. I contesti ed i corrispondenti approcci sono:

1. Situazioni in cui è necessario che l'interessato possa riconoscersi nel documento ed essere riconosciuto dagli altri soggetti menzionati nel contesto, ma non è plausibile la necessità di identificazione dell'interessato al di fuori di quel contesto specifico. A questo caso si riconducono le graduatorie di concorsi, gli elenchi per la formazione di classi, gli elenchi per convocazioni, etc. In tutti questi casi, nel processo di redazione del documento è più pratico trattare internamente i dati personali in forma completa; ma praticamente mai è necessario pubblicarli per intero, esponendo al pubblico accesso codici fiscali, indirizzi, recapiti telefonici etc. Per di più, nella maggior parte delle volte, è sufficiente pubblicare il solo cognome perché l'interessato conosca la sua posizione in graduatoria; ciò è spesso sufficiente anche a trasmettere l'informazione necessaria ai colleghi dell'interessato nella medesima graduatoria, o nella medesima classe, etc. Notare che, pubblicando il solo cognome, viene di fatto oscurato anche il sesso. Dunque, in questi casi è auspicabile cancellare una larga parte dei dati personali presenti nel documento.
2. Situazioni in cui il documento deve essere pubblicato affinché siano rese note le motivazioni che lo hanno originato, senza che sia necessaria la precisa identificazione dei soggetti che vi compaiono. Questo è il caso della pubblicazione di sentenze

giudiziarie di ogni grado. Le sentenze vengono notificate direttamente agli "aventi causa"; ma anche, come la legge prescrive, pubblicate al fine di costituire giurisprudenza. In questo secondo caso, risulta del tutto eccedente la pubblicazione dei reali nominativi degli interessati, che troverebbero verosimilmente la propria vicenda inutilmente indicizzata dai motori di ricerca negli anni a venire. Nella pubblicazione delle sentenze appare un approccio ottimale quello di sostituire con pseudonimi o con iniziali alterate i veri nominativi degli interessati presenti: il senso e le argomentazioni della sentenza restano pienamente comprensibili, come è necessario; il dato personale, del tutto superfluo ai fini della giurisprudenza, viene protetto.

In questi casi appare opportuno sostituire i dati identificativi dell'interessato, ed in particolare il nome e cognome, con pseudonimi o, ancora, con iniziali alterate, cioè non coincidenti con le iniziali reali.

Proprio questi tipi di trattamenti di "minimizzazione" sono l'oggetto di questa tesi.

1.2 Punti di partenza per la progettazione del servizio

La tesi è stata svolta in collaborazione con l'azienda AFA Systems (www.afasystems.it/gdpr) con la quale si sono discusse le problematiche di progettazione e realizzazione di un servizio generalizzato di minimizzazione dei dati personali presenti in un documento. Con l'intenzione di fornire il servizio ad un bacino d'utenza il più vasto e variegato possibile, si è pensato ad un'applicazione *web-based*, indipendente così dai singoli device sui quali poi sarà utilizzata. L'attenzione è stata concentrata sul trattamento dei nomi e dei cognomi (che da qui chiameremo nominativi), poiché sono quelli sempre presenti nei documenti (ad es. gli indirizzi sono presenti solo a volte), rappresentano gli elementi tramite i quali è immediato il riconoscimento della persona, non hanno formato predefinito (come ad es. i codici fiscali); altri dati personali (indirizzi, codici fiscali, etc.), potranno essere considerati in successive evoluzioni del progetto.

È utile notare che la parte iniziale della collaborazione con l'azienda è stata dedicata alla discussione delle specifiche, la cui più precisa definizione è parte rilevante di questa tesi.

Tra i requisiti che necessitano di un opportuno studio troviamo ad esempio:

- L'individuazione di metodi efficaci per il riconoscimento dei nominativi nei documenti

- L'identificazione delle migliori procedure di interazione con l'utente
- La scelta dei formati da trattare
- I vincoli non funzionali legati al rispetto del *GDPR*.

2 Definizione delle specifiche

2.1 Riconoscimento dei nominativi

2.1.1 Scelta della strategia di riconoscimento

Le difficoltà principali con cui ci si imbatte nel processo di riconoscimento dei nominativi riguardano la complessità strutturale dei documenti di testo, ma soprattutto l'intrinseca ambiguità del linguaggio naturale.

Le variabili ed imprevedibili strutture e formattazioni dei documenti introducono alcuni problemi significativi. Rendendo fortemente eterogeneo il contenuto dei documenti, infatti, esse non consentono di ricondurre il problema del riconoscimento dei nominativi ad uno o a pochi singoli casi, ma comportano lo studio di tutte le strutture e formattazioni possibili, rendendo quindi l'analisi molto generale. L'altra rilevante difficoltà presente sta nella complessità di effettuare il riconoscimento di una stringa testuale immersa in un insieme di elementi non tutti testuali. Ogni elemento di formattazione, che sia una tabella o una barra orizzontale, introduce infatti un proprio significato logico e semantico nel documento di cui bisogna tenere conto.

Il linguaggio naturale introduce anch'esso complessità: si pensi alle molteplici tipologie di proposizioni con cui possono essere articolati i periodi; ma i problemi principali derivano dalle sue ambiguità. Esse vengono classificate in diverse tipologie [3]; le principali sono le ambiguità sintattiche e lessicali.

Si ha ambiguità sintattica quando la sintassi di una frase può essere interpretata in diversi modi e, di conseguenza, la frase stessa assume significati diversi. Essa è presente, ad esempio, nelle seguenti frasi:

- *"Rapina in banca con rivoltella da centomila euro"*
- *"Luigi ha visto un uomo nel parco con il binocolo"*

L'esasperazione massima della problematica delle ambiguità sintattiche si presenta con l'*antinomia*, ossia un particolare tipo di paradosso che indica la compresenza di due affermazioni contraddittorie che possono essere entrambe dimostrate o giustificate.

L'antinomia di Epimenide o *Paradosso del mentitore*, nota fin dal VI secolo, è probabilmente uno dei più noti esempi: "*il cretese Epimenide afferma che tutti i cretesi mentono*". Se la proposizione è vera (i cretesi mentono) allora il suo significato implica che sia falsa (Epimenide mente e quindi i cretesi dicono la verità), ma se è falsa (i cretesi dicono la verità) ciò significa che è vera (Epimenide dice la verità e quindi i cretesi mentono). La proposizione appare contemporaneamente vera e falsa. A partire dagli anni venti del '900, sono state elaborate varie teorie per la risoluzione delle contraddizioni provocate dalle antinomie, soprattutto attraverso l'elaborazione di linguaggi multilivello o attraverso l'elaborazione di logiche polivalenti (quindi non-booleane).

Si ha ambiguità lessicale, invece, quando una parola possiede più di un significato nella lingua a cui appartiene [4], in tal caso la parola è definita *polisemica*. In italiano, alcuni termini soggetti a questo tipo di ambiguità sono, ad esempio "*acuto*" o "*venti*". È questo genere di ambiguità che risulta critico per il riconoscimento dei nominativi. La difficoltà determinata dalle ambiguità sintattiche, infatti, riguarda l'individuazione corretta di soggetti, predicati e complementi di un periodo, mentre le ambiguità lessicali ostacolano la comprensione del significato del singolo lessema. Nel processo di riconoscimento è irrilevante determinare la funzione logica che il nominativo svolge nella frase, mentre è necessario essere certi che i termini che compongono il nominativo siano effettivamente dei nomi propri di persona o dei cognomi, non altri vocaboli del lessico comune.

In linguistica, l'intervento con cui si toglie ambiguità a una parola o a una frase prende il nome di "disambiguazione" [5]. Il problema della disambiguazione automatica (in inglese *Word Sense Disambiguation* o, abbreviato, *WSD*) riveste particolare importanza nelle ricerche sull'intelligenza artificiale e, in particolare, nell'elaborazione del linguaggio naturale. Si prevedono benefici della disambiguazione, ad esempio, in programmi di traduzione automatica, recupero dell'informazione o estrazione automatica di informazioni. Nell'analisi delle soluzioni esistenti in letteratura per la risoluzione delle ambiguità, ci si sofferma specialmente sulle ricerche incentrate sul trattamento dell'ambiguità lessicale, essendo essa la più rilevante per i nostri interessi.

La *WSD* richiede due input necessariamente: un dizionario per specificare i sensi che devono essere disambiguati e un corpus di dati linguistici da disambiguare. Nello scenario più realistico, si trattano testi le cui parole non sono note a priori e risulta molto onerosa la

produzione del corpus, essendo infatti necessaria la valutazione di un operatore umano per verificare la correttezza delle disambiguazioni effettuate dagli algoritmi (*supervised learning*).

Uno tra i principali dizionari semantici-lessicali utilizzati della lingua inglese è *WordNet* [6], mentre alcuni dei database equivalenti che trattano l'italiano sono *BabelNet* [7]), *ItalWordNet* [8] e *MultiWordNet* [9].

Un elemento importante da considerare è che le prestazioni di disambiguazione per la lingua inglese, impiegando *WordNet*, risultano corrette tra l'80% e il 90% delle volte [10], percentuali discrete ma non sufficienti per avere la totale garanzia.

I fattori che ostacolano la realizzazione di algoritmi di intelligenza artificiale per la disambiguazione sono molteplici:

1. Differenze tra dizionari impiegati: i database prima citati si basano su varie fonti che raggruppano semanticamente in maniera diversa i vocaboli, quindi programmi sviluppati con dizionari diversi generalmente hanno performance differenti.
2. Complessità della codifica di parte del discorso: per poter disambiguare correttamente un termine è importante riuscire a comprendere correttamente il contesto in cui è inserito, operazione non banale.
3. Varianza tra giudici: i supervisori dell'apprendimento degli algoritmi possono avere opinioni diverse, o semplicemente sbagliare, nella valutazione delle disambiguazioni, ciò porta ad algoritmi che hanno comportamenti diversi.
4. Impossibilità di applicare la disciplina della *pragmatica*, ossia la logica del *buon senso*: per identificare correttamente il senso di alcune parole, ad esempio nella comprensioni di anafore e catafore, è necessario applicare il buon senso.
5. Dipendenza del senso delle parole dai contesti: ogni scenario richiede la propria divisione del significato delle parole in sensi rilevanti. In un contesto informatico, ad esempio, il termine "*mouse*" deve essere ricondotto al dispositivo di puntamento, non al cognome del celebre personaggio Disney *Mickey Mouse*; viceversa dovrà invece avvenire in un contesto di letteratura a fumetti.

Laddove la *WSD* è una tecnica molto generale e che mira a risolvere un'ambiguità riguardante una qualunque parola, per le finalità del servizio oggetto di questa tesi è sufficiente risolvere le ambiguità dei nomi e dei cognomi.

Uno dei difetti che il servizio presenterebbe adottando un approccio WSD è legato alla imprevedibile formattazione dei documenti, che aggiunge informazione semantica al testo ma introduce complessità nella individuazione automatica delle parti che formano il contesto; un altro difetto dipende dalla vastità del lessico da elaborare, essendo i documenti da trattare forniti dai più disparati utenti, su qualunque genere di argomento e relativi ai più vari ambiti.

La strategia che sarà adottata per risolvere la problematica del riconoscimento dei nominativi sarà impostata attraverso un modello *pattern-based*, impiegando le *regular expression* (*regex*). Esse risultano comunemente usate per effettuare operazioni di ricerca o sostituzione in un testo, di conseguenza se si riesce ad individuare un pattern associato ad un nominativo dato, allora sarà possibile processare il documento ricercando le occorrenze del nominativo e "minimizzarlo" opportunamente.

Le *regular expression* risultano particolarmente efficaci poiché, se opportunamente progettate, possono identificare un nominativo indipendentemente dal significato linguistico del contesto in cui è calato, basandosi piuttosto sui singoli caratteri che compongono i lessemi analizzati; permettono, di conseguenza, di effettuare una valutazione estremamente minuziosa, riducendo al minimo le possibilità di errori.

Un algoritmo di risoluzione *pattern-based* risulta, inoltre, più efficiente nell'esecuzione, in generale, di un algoritmo di intelligenza artificiale; per fornire la risposta il più velocemente possibile ad un utente, fruitore del servizio via web, l'approccio di riconoscimento tramite pattern è il più indicato.

2.1.2 Definizione dei pattern

Nella progettazione del servizio si adotta, come detto, un approccio *pattern-based* per il riconoscimento dei nominativi. In linea di massima è opportuno che vengano riconosciuti più nominativi possibili e allo stesso tempo che la correttezza dell'identificazione di un nominativo sia garantita, quindi bisogna individuare dei pattern non troppo stringenti ma neppure troppo laschi. Per analizzare come i pattern devono essere strutturati si prende come caso di studio un generico nominativo, ad esempio *Lorenzo Mario Amorosa*. Nel documento esso può comparire esattamente come appena indicato, ma anche in altre plausibili varianti, in cui l'ordine dei termini viene alterato, si pensi ad esempio ad "Amorosa Lorenzo Mario" o "Mario Lorenzo Amorosa", o in altre varianti ancora in cui alcuni nomi non compaiono, come

in "Lorenzo Amorosa". Bisogna tuttavia supporre un limite alla variabilità: si osserva infatti che il cognome deve sempre comparire (il solo nome *Lorenzo* non è riconducibile a *Lorenzo Mario Amorosa*) e che esso inoltre deve essere necessariamente anteposto o posposto, ma non interposto, ai nomi (è poco ragionevole ricondurre "Lorenzo Amorosa Mario" a "Lorenzo Mario Amorosa").

Questa prima specifica permette di ricondursi a una soluzione generale, sufficiente nella maggior parte dei casi, ma che non risolve alcune criticità. Un nominativo può, in alcuni documenti, comparire manifestandosi unicamente attraverso il cognome (riferendoci all'esempio precedente, *Lorenzo Mario Amorosa* apparirebbe nella forma *Amorosa*). Sorge qui il problema che molti dei cognomi italiani hanno un significato proprio nel linguaggio comune; ad es., nella frase "*una relazione amorosa è bella*" è errato considerare la parola *amorosa* come cognome di un nominativo. Per risolvere questo genere di ambiguità si potrebbe pensare che per stabilire che il termine *Amorosa* sia un cognome sia sufficiente verificare che esso inizi con una lettera maiuscola, ma ciò può verificarsi anche perché la parola si trova ad inizio di frase. Inoltre, vari cognomi italiani possono iniziare con una lettera minuscola (*de Angelis*, *d'Onofrio*, etc.), quindi basare l'identificazione di un cognome sul fatto che la sua prima lettera sia in maiuscolo non è in generale un metodo valido. Volendo in maniera prioritaria garantire il corretto funzionamento del servizio, e quindi attuare le procedure di "minimizzazione" solo sui nominativi senza applicarle erroneamente ad altri termini, risulta necessario evitare il trattamento dei nominativi che si manifestano con i soli cognomi. Per via del contesto e dei significati che i cognomi possono assumere, infatti, risulta spesso impossibile distinguerli da parole del linguaggio comune.

Ci si concentra, quindi, nello studio di nominativi composti da un cognome seguito o preceduto da uno o più nomi.

Si rappresenta dunque in formato di *regular expression* il pattern attualmente ideato. Per semplicità espositiva, si definisce il tag <nomi> come l'insieme delle permutazioni di tutti i nomi del nominativo più l'insieme delle permutazioni di tutti i possibili sottoinsiemi di nomi del nominativo. Considerando il nominativo preso precedentemente come caso di studio, ad esempio, si avrebbe:

<nomi> = Lorenzo Mario|Mario Lorenzo|Lorenzo|Mario

Posto inoltre il tag <cognome> ad indicare il cognome contenuto nel nominativo e il tag <regex> ad indicare la *regular expression* associata al nominativo, si ottiene:

<regex> := <cognome> (<nomi>)|(<nomi>) <cognome>

Una osservazione sottile ma di fondamentale importanza per la corretta progettazione delle *regular expression* sta nella piena comprensione della semantica dell'operatore di scelta, espresso con il carattere *pipe* ("|"). Un qualunque *engine* di elaborazione delle *regex*, infatti, interrompe la valutazione di una stringa non appena può stabilire se tale stringa fa match o meno con il pattern dato, senza quindi necessariamente valutarlo nella sua interezza, come parimenti avviene nelle valutazioni *a corto circuito* delle espressioni logiche nei linguaggi di programmazione. Ogni nominativo avrà a sé associato un pattern che lo rappresenta in più possibili sequenze di caratteri; per effettuare una corretta "minimizzazione" dei dati è necessario che le sequenze contenenti tutti i nomi sia poste per prime, mentre quelle contenenti un singolo nome per ultime.

In questa prima formulazione della *regex*, inoltre, si è posto come separatore unicamente lo spazio bianco, ma alcuni documenti potrebbero contenere dei nominativi i cui termini sono separati da altri caratteri, come ad esempio una virgola nel caso di "Amorosa, Lorenzo Mario". Per poter quindi identificare un nominativo anche in questi casi, si potrebbe considerare un qualunque carattere di interpunzione come possibile separatore dei termini del nominativo.

Adottando questa soluzione si ha però come effetto collaterale che risultano critici i casi in cui nel testo sono presenti dei nominativi soggetti ad omonimia. Si consideri una generica frase contenente una sequenza di nominativi, ad esempio *Amorosa Lorenzo, Mario Giacomo e Fabio Rossi*, e si supponga che i nominativi da trattare siano *Amorosa Lorenzo Mario, Mario Giacomo* (in cui la parola *Mario* è il cognome) e *Fabio Rossi*. Gli ultimi due nominativi compaiono nella frase nella loro forma estesa, mentre il primo compare con il solo nome *Lorenzo* (eventualità possibile considerando la definizione del pattern precedentemente data). Considerando la virgola come carattere separatore dei termini del nominativo, la sequenza di parole *Amorosa Lorenzo, Mario* sarebbe ricondotta, venendo elaborata per prima, ad *Amorosa Lorenzo Mario*, mentre rimarrebbe non trattata la parola *Giacomo*, in quanto il cognome *Mario* che gli era associato è stato già identificato come un nome del nominativo *Amorosa Lorenzo Mario* ed il termine *Giacomo* preso singolarmente non rappresenta un nominativo.

Prima di valutare ulteriormente le problematiche relative alle omonimie, si possono scegliere due approcci per risolvere questo specifico caso:

1. Si riconduce una sequenza di parole ad un nominativo se e solo se tutti i suoi nomi sono contenuti nella sequenza
2. Si considerano come separatori dei termini contenuti nei nominativi solo spazi bianchi, tabulazioni e a capo, non gli altri segni di punteggiatura.

Entrambe le strategie sono valide, in quanto risolvono il problema garantendo la corretta identificazione dei nominativi, ma allo stesso tempo entrambe presentano lo svantaggio di ridurre le sequenze di parole riconducibili a dei nominativi, aumentando le possibilità che alcuni di essi non vengano trattati. Si consideri nuovamente il nominativo preso come caso di studio *Amorosa Lorenzo Mario*: applicando la prima strategia, non sarebbe possibile ricondurgli la sequenza *Amorosa Lorenzo*, mentre applicando il secondo metodo non sarebbe riconducibile la sequenza *Amorosa, Lorenzo Mario*.

Si decide di attuare la seconda strategia, in quanto è opportuno non imporre vincoli troppo stringenti sui nomi e poiché nella gran parte dei casi i termini dei nominativi nei documenti sono separati tra loro da caratteri quali spazi bianchi, tabulazioni ed a capo.

Un altro elemento su cui porre l'attenzione è la possibilità che i documenti da trattare contengano dei nominativi scritti interamente in maiuscolo o minuscolo, di conseguenza è conveniente che le *regular expression* siano progettate *case-insensitive*.

Un ulteriore punto su cui bisogna soffermarsi è la posizione all'interno di un periodo in cui un nominativo può comparire, in particolare si vogliono evitare quei casi critici in cui uno dei termini del nominativo è una sotto-stringa di un'altra parola del testo (si pensi ad *amorosa* in *clamorosa*). Come regola generale si può stabilire che è sempre necessario che un nominativo sia preceduto e seguito da un *carattere non alfabetico*. Un caso particolare si presenta quando il nominativo è posto ad inizio o a fine documento, situazione in cui quindi esso non è preceduto o non è seguito da alcun carattere: entrambe le posizioni sono da considerare corrette.

È opportuno ora definire il concetto di *carattere non alfabetico*, e ciò si può fare più facilmente ragionando sul problema in logica positiva; infatti risulta più semplice individuare

quei caratteri che rappresentano lettere piuttosto che quelli che rappresentano segni di interpunzione ed individuare ogni altro carattere che non può mai comporre una parola.

Inquadrando lo scenario di applicazione del servizio, molto probabilmente l'utente vorrà trattare un documento scritto nella lingua di uno dei paesi dell'Unione Europea, poiché il *GDPR* vige nei soli paesi membri dell'Unione. Si può, quindi, ipotizzare che i documenti trattati possono sì contenere parole e nominativi stranieri, ma che i caratteri contenuti siano appartenenti all'alfabeto latino (*Latin script*), usato in molti stati nel mondo e da tutti i principali stati europei (fatta eccezione per la Grecia ed alcuni stati che scrivono in caratteri cirillici). Inoltre, testi scritti in altri alfabeti, come esempio il cinese, l'arabo o il cirillico, vengono generalmente traslitterati. Considerare, quindi, *caratteri non alfabetici* tutti i caratteri diversi dalle lettere contenute nell'alfabeto latino sarebbe di conseguenza estremamente riduttivo ed inoltre in questo modo non si terrebbe conto delle lettere accentate, molto utilizzate anche nella lingua italiana.

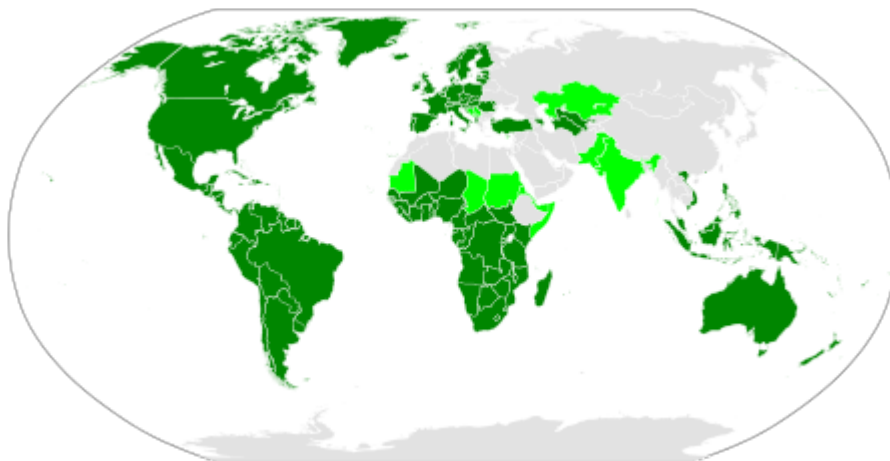


Figura 1. Diffusione mondiale dell'alfabeto latino

Per risolvere il problema facciamo riferimento alla codifica standard Unicode dell'alfabeto latino [11]; in essa è possibile individuare, oltre ai caratteri rappresentanti le lettere nella codifica *ASCII* classica, i caratteri rappresentanti le lettere nella codifica standard *ISO/IEC 8859-1* [12], encoding orientato principalmente alla rappresentazione delle lingue dell'Europa occidentale.

Caratteri latini nei primi due blocchi dello standard Unicode

U+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Block	#
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	C0 Controls and Basic Latin 0000–007F (identical to ASCII)	52
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_		
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o		
0070	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL		
00A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯	C1 Controls and Latin-1 Supplement 0080–00FF (identical to ISO/IEC 8859-1)	62
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿		
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï		
00D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß		
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï		
00F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ		

I caratteri indicati in rosso nella tabella sono tutti i possibili *caratteri alfabetici* che compaiono nei documenti scritti in 29 lingue diverse [12], tra le quali sono presenti l'italiano, l'inglese, lo spagnolo, il tedesco e il portoghese.

Aggiungendo pochi altri caratteri all'insieme dei *caratteri alfabetici* appena mostrati, si riesce a rappresentare ogni parola di altre 12 lingue [12], tra cui il francese, l'olandese, il ceco e il turco.

I caratteri mostrati in tabella corrispondono ad una parte dei primi due blocchi dello standard Unicode che codificano l'alfabeto latino e, come già accennato in precedenza, essi sono presenti negli standard *ASCII* e *ISO/IEC 8859-1*. I rimanenti *caratteri alfabetici*, invece, sono presenti in estensioni dello standard Unicode per il *Latin script*. Queste estensioni sono state realizzate per fornire il massimo supporto a tutte le lingue e contenenti molti simboli ad uso speciale, come ad esempio per la rappresentazione dei fonemi. Si osserva, inoltre, che i *caratteri alfabetici* definiti nelle estensioni dello standard Unicode sono presenti anche nella codifica *ISO/IEC 8859-2* o nelle versioni successive [12].

Definiti i “*caratteri non alfabetici*” come elementi di separazione tra un nominativo ed il testo in cui esso è inserito, occorre definire opportunamente la *regex* che al nominativo è associata.

Utili strumenti messi a disposizione dalle *regular expression* sono i gruppi speciali *lookahead* e *lookbehind*. Un pattern di un nominativo deve essere preceduto o seguito da una prestabilita

sequenza di caratteri, la quale però non è parte del nominativo. Utilizzando i due costrutti citati, è possibile far sì che nell'elaborazione di una stringa facciano match solamente le parole effettivamente appartenenti al nominativo, non i caratteri che lo delimitano dal resto del testo, e allo stesso tempo che un nominativo faccia match se e solo se preceduto o seguito da una certa sequenza di caratteri.

Per esprimere nella sintassi delle *regex* un carattere letterale in un qualunque alfabeto si può utilizzare il costrutto $\backslash p\{L\}$, che però risulta molto generale e troppo lasco per i requisiti considerati. Si può piuttosto valutare l'impiego del costrutto $\backslash p\{Latin\}$, il quale identifica un qualunque carattere alfabetico presente nell'alfabeto latino. Tra i caratteri corrispondenti al costrutto, però, ve ne sono alcuni che per le specifiche del servizio devono essere considerati *caratteri non alfabetici*, come ad esempio i caratteri dei fonemi, i segni diacritici e gli indicatori ordinali; di conseguenza è necessario individuare una strategia ad hoc per risolvere questa problematica.

Per chiarezza espositiva, si definisce il tag `<start>`, per indicare un qualunque carattere non alfabetico o l'inizio stringa, il tag `<end>`, per indicare un qualunque carattere non alfabetico o il fine stringa, ed il tag `<extra>`, per indicare i *caratteri alfabetici* non presenti negli standard *ASCII* o *ISO/IEC 8859-1*:

`<extra>:=`

`ŁłČčŘřŠšŽžİıĲĲÆæŸŸŌŌŪūBbĈĉĐđFfĜĝMmŠšŤťĀāĒēĪīŌōŪūİıĠġŠšŴŵŶŷŸŹŲų`

`<start>:=(?=[^A-Za-zÀ-ÖØ-öø-ÿ<extra>]|^)`

`<end>:= (?=[^A-Za-zÀ-ÖØ-öø-ÿ<extra>]|$)`

Si osserva che si sono utilizzati i gruppi speciali prima descritti e che si sono inseriti i caratteri dello standard Unicode presentati in precedenza. Si definisce nuovamente la *regular expression* associata ad un generico nominativo. Applicando i tag appena definiti, il costrutto `(?i)` che rende il pattern *case-insensitive* ed il costrutto `\s` che rappresenta un carattere qualunque tra i separatori non visibili, ossia `\r \n \t \f \v` e lo spazio bianco, si ottiene:

`<regex>:=(?i)(<start><cognome>\s+(<nomi>)|`

`(<nomi>)\s+<cognome><end>)`

Si osserva, inoltre, che in questa nuova formulazione della *regular expression* i nomi sono da intendersi separati tra loro da `\s+`.

A questo punto si è quasi giunti alla formulazione finale del pattern da associare ad un nominativo, rimangono solo da trattare alcuni casi critici non ancora risolti.

Si è già presentato in precedenza il problema legato al fatto che alcuni cognomi possono avere un significato proprio nel lessico comune e che ciò costringeva, quindi, ad abbandonare l'idea di trattare nominativi formati dal solo cognome. Questa problematica di ambiguità si presenta anche con alcuni nomi (si pensi, ad esempio, al nome *Gioia*). Ciò non rappresenta generalmente un problema, in quanto la coppia nome-cognome che forma il nominativo, presa complessivamente, non è soggetta ad ambiguità. Esistono, però, dei casi in cui questo non è vero. Si prenda in analisi il nominativo *Gioia Grande*: risulta evidentemente soggetto a rischio di ambiguità. Una soluzione che si può adottare, per risolvere questo caso critico, si basa sull'associazione di un pattern più stringente ai nominativi. In particolare, si osserva che i nomi propri di persona compaiono sempre ed obbligatoriamente, in un documento grammaticalmente corretto, con la prima lettera maiuscola [13]. I cognomi, invece, non sono soggetti ad una regola così stringente: un cognome iniziante con una lettera minuscola (come *de Rosa*) in alcuni casi, ad esempio se posto dopo un punto fermo, può comparire scritto con la prima lettera sia maiuscola che minuscola; naturalmente, in nessuna occorrenza un cognome che inizi con una lettera maiuscola potrà comparire con una minuscola.

Occorre quindi ridefinire, alla luce di queste osservazioni, la *regex* associata ad un nominativo, poiché precedentemente era stata posta interamente *case-insensitive*. Nel pattern, in particolare, i nomi dovranno sempre iniziare con una maiuscola, mentre i cognomi avranno questo vincolo solo se nel nominativo compaiono con la prima lettera maiuscola. Si mostra quindi quali sono i tag `<nomi>` e `<cognome>` associati a due nominativi, ad esempio *Gioia Grande* e *Antonio de Rosa*.

Per *Gioia Grande* si ha:

`<nomi> = G((?i)ioia)`

`<cognome> = G((?i)rande)`

Per *Antonio de Rosa* si ha:

`<nomi> = A((?i)ntonio)`

`<cognome> = (?i)de Rosa`

Si presenta dunque la definizione finale del tag <regex>. Nella definizione il tag <nomi> è definito in base al numero di nomi del nominativo ed il tag <cognome> è definito in base al carattere iniziale del cognome.

<regex>:=<start><cognome>\s+(<nomi>)|(<nomi>)\s+<cognome><end>

Vengono inoltre mostrati i valori dei due tag <nomi> e <cognome> per il nominativo preso come caso di studio in fase iniziale, ossia *Amorosa Lorenzo Mario*. Si ottiene:

<nomi> = L((?i)orenzo)\s+M((?i)ario)|

M((?i)ario)\s+L((?i)orenzo)|L((?i)orenzo)|M((?i)ario)

<cognome> = A((?i)morosa)

Infine, per completezza, viene mostrata la *regular expression*, associata a quest'ultimo nominativo, risolvendo tutti i tag che la compongono:

<regex>=(?<=[^A-Za-zÀ-ÖØ-öø-

ÿŁłČčŘřŠšŽžİİjĲÆŸŎŎŰŰBbĈĉDdFfĜĝMmŚśŤťĀāĒēĪīŌōŪūİıĠġŞşŴŵŶŵŸŹŲŲβ
]|^)(A((?i)morosa)\s+(L((?i)orenzo)\s+M((?i)ario)|M((?i)ario)\s+L((?i)orenzo)|L((?i)orenzo)|M((?i)ario))|(L((?i)orenzo)\s+M((?i)ario)|M((?i)ario)\s+L((?i)orenzo)|L((?i)orenzo)|M((?i)ario))\s+A((?i)morosa))(?<=[^A-Za-zÀ-ÖØ-öø-ÿŁłČčŘřŠšŽžİİjĲÆŸŎŎŰŰBbĈĉDdFfĜĝMmŚśŤťĀāĒēĪīŌōŪūİıĠġŞşŴŵŶŵŸŹŲŲβ]|\$)

2.1.3 Gestione delle omonimie

Nei ragionamenti che hanno portato alla formulazione della *regular expression* associata ai nominativi, si è tenuto conto di possibili ambiguità con termini appartenenti al linguaggio comune, risolte con l'introduzione nel pattern di stringhe *case-sensitive*, e di possibili nominativi posti in sequenza parzialmente omonimi (ossia aventi un nome o il cognome in comune tra loro), gestite con l'imposizione dei soli caratteri separatori non visibili come delimitatori delle parole componenti un nominativo. Per rendere l'analisi completa occorre valutare come ci si debba comportare in altri possibili casi di omonimia. Si osserva, per inciso, che nel pattern individuato nella sezione precedente il tag <nomi> è stato l'unico non

definito formalmente. La definizione di tale tag infatti dipende, oltre che dai nomi del nominativo, anche dall'insieme complessivo dei nominativi da trattare presenti nel documento.

Il caso più semplice di omonimia, che si verifica quando un solo nome o il solo cognome di un nominativo coincide con un nome o il cognome di un altro (come nel caso di *Lorenzo Mario Amorosa* e *Stefano Amorosa*), risulta già ben gestito dall'attuale formulazione del pattern. Infatti, un riconoscimento è determinato quando almeno un nome e il cognome del nominativo appaiono in sequenza.

Il problema si complica quando un nominativo ha due o più componenti in comune con un altro. Si osserva che se l'omonimia riguarda solo i nomi dei nominativi, ciò non risulta problematico, in quanto il cognome, supposto sempre presente nel pattern, funge da elemento di disambiguazione. Si considera da ora, quindi, che i nominativi abbiano il medesimo cognome e si analizzano i casi in cui anche uno o più nomi risultano in comune, attraverso degli esempi:

- $\text{Nomi_A} = \{\text{"Lorenzo"}, \text{"Mario"}, \text{"Luca"}\}; \text{Nomi_B} = \{\text{"Stefano"}, \text{"Mario"}\}$

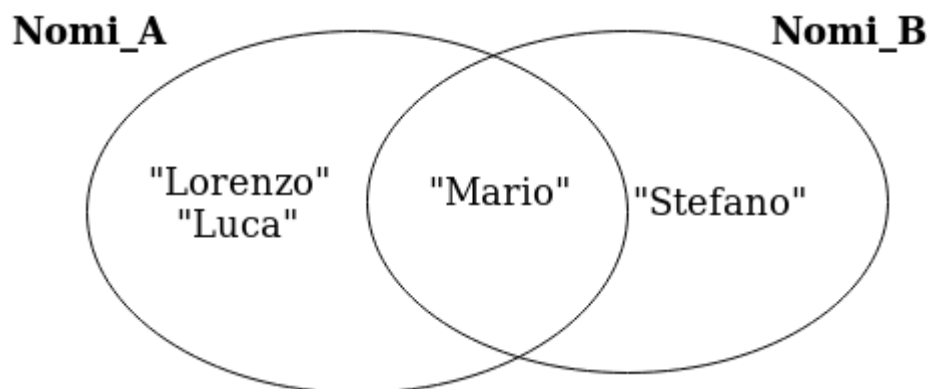


Figura 2. Diagramma di Venn (1)

In questo caso sono elementi di disambiguazione i nomi *Lorenzo* e *Luca* per il primo nominativo e *Stefano* per il secondo, bisognerà quindi far sì che almeno uno tra tali nomi compaia sempre nelle *regex* associate ai nominativi in questione. L'occorrenza *Mario* <cognome> rimane ambigua e non può essere trattata.

- $\text{Nomi_A} = \{\text{"Lorenzo"}, \text{"Mario"}, \text{"Luca"}\}; \text{Nomi_B} = \{\text{"Mario"}, \text{"Luca"}\}$

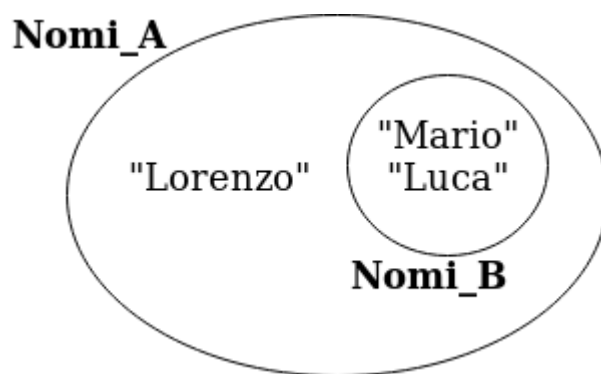


Figura 3. Diagramma di Venn (2)

L'insieme **Nomi_B** risulta un sottoinsieme di **Nomi_A**, di conseguenza solo il primo nominativo presenta degli elementi utili per risolvere l'ambiguità. Nel pattern relativo al primo nominativo dovrà essere presente almeno un tra i nomi non in comune, mentre il pattern del secondo nominativo rappresenterà inevitabilmente espressioni ambigue poiché riconducibili all'altro. Le soluzioni possibili sono due: rifiutarsi di effettuare il trattamento del secondo nominativo oppure decretare che esso è riconosciuto se e solo se appare nella sua forma estesa, presentando quindi tutti i nomi. Quest'ultima soluzione è ragionevole e la si sceglie, poiché così si aumentano, per quanto possibile, le sequenze "minimizzabili", e viene inoltre messo in conto di informare l'utente opportunamente sulla gestione di questo genere di omonimia per rendere le operazioni trasparenti.

- $\text{Nomi_A} = \text{Nomi_B} = \{ \text{"Lorenzo"}, \text{"Mario"}, \text{"Luca"} \}$

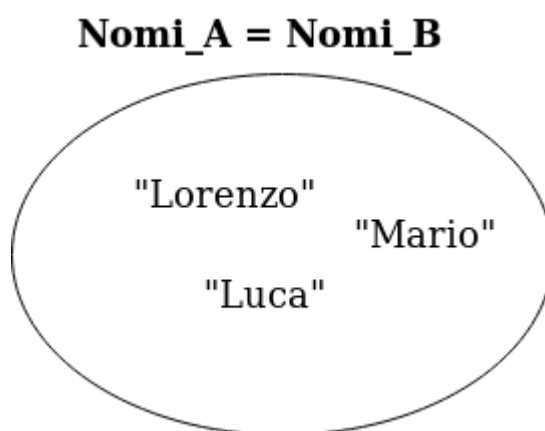


Figura 4. Diagramma di Venn (3)

Qualora l'insieme dei nomi dei due nominativi sia identico risulta impossibile distinguerli e non possono in alcun modo essere trattati, poiché l'ordine con cui compaiono i nomi di un nominativo non rappresenta un elemento di disambiguità. I dati appartenenti a persone completamente omonime contenuti in uno stesso documento, quindi, non sono "minimizzabili" distintamente.

Si osserva che è di fondamentale importanza che i documenti che gli utenti forniscono siano grammaticalmente corretti, in quanto un errato uso dei segni di interpunzione può rendere impossibile l'applicazione delle *regular expression*. Si prenda ad esempio una sequenza anomala di caratteri in cui non è corretto l'uso delle virgole, come "*Amorosa Mario Rossi Giacomo*": supponendo che nel documento si vogliano trattare i nominativi *Amorosa Mario*, *Rossi Mario* e *Rossi Giacomo*, risulta impossibile identificare quali tra questi siano rappresentati nella sequenza.

In conclusione, risultano quindi individuate le strategie che permettono di formulare *regular expression* anche per i nominativi soggetti ad omonimia.

2.1.4 Formattazione dei documenti

I documenti prodotti in enti ed organizzazioni presentano generalmente formattazioni e non sono puramente testuali. Si vuole qui considerare quale debbano essere le interpretazioni più adatte da dare agli elementi grafici per rendere sempre efficace il riconoscimento dei nominativi. In particolare, quindi, non si tratteranno gli elementi di punteggiatura, come parentesi o virgolette, poiché già compresi nelle *regex*, bensì tutti gli elementi che in un pattern composto da caratteri non sono espressi. Si inizia dunque la rassegna:

- Elementi di formattazione del testo

I font, la dimensione dei caratteri, il grassetto, il corsivo, l'evidenziazione e tutti i possibili elementi di modifica della apparizione grafica del testo non alterano il significato dei lessemi coinvolti. Se il cognome di un nominativo è posto in grassetto ed il nome no, ad esempio, la coppia nome-cognome rappresenta sempre il nominativo iniziale; lo stesso vale per gli altri elementi citati.

- Aree di comparizione del testo

In genere ogni documento è formato da più sezioni, ha un corpo principale ed eventuali titoli, note a piè pagina, intestazioni ed altre possibili aree. Il trattamento di "minimizzazione", secondo il *GDPR*, deve essere effettuato al documento in ogni sua parte, ma ogni sezione deve essere elaborata in maniera indipendente dalle altre sezioni in quanto rappresenta un blocco logico a sé. Ciò significa che, ad esempio, bisogna individuare eventuali nominativi presenti nel titolo, ma non bisogna considerare nominativo una sequenza di parole che è posta in parte nel titolo e in parte nel primo paragrafo del testo.

- Elementi blocco

Gli elementi blocco, come ad esempio le immagini, possono causare un'interruzione netta in un paragrafo, suddividendolo quindi in più blocchi logici, che devono essere analizzati separatamente; tuttavia nel caso in cui questi elementi siano posti *fluttuanti*, ossia ancorati ai bordi della pagina, il testo scorre in un unico flusso e forma quindi un unico blocco logico.

- Divisione in sillabe

Un problema rilevante nella formattazione del documento è che spesso, per esigenze estetiche, contiene delle parole divise in sillabe poste su righe diverse e separate da un trattino. Ovviamente se un nome va a capo a fine linea non perde il suo significato semantico, bisognerà quindi continuare a riconoscerlo.

- Tabelle

Molti documenti, specialmente quelli contenenti ingenti quantità di nominativi, possono essere strutturati in forma tabellare. Trascurando una trattazione per esteso delle modalità con cui i nominativi potrebbero comparire nelle tabelle, si considera esclusivamente il caso di gran lunga più ricorrente. In genere, infatti, in una tabella contenente dei nominativi, essi sono presenti nella stessa colonna o in colonne contigue: l'una contenente i nomi, l'altra il cognome, o viceversa. Senza basarsi sull'intestazione delle colonne, si può valutare se il contenuto di due celle adiacenti poste su di una stessa riga faccia match con il pattern di un nominativo, ed in caso ciò accada si può affermare la coppia di celle individuata rappresenta un nominativo.

Si osserva che si sono fornite solamente le interpretazioni più plausibili e comuni a questi elementi grafici e non si è ideato alcun particolare algoritmo per la loro elaborazione. L'espressione di tali strutture, infatti, è fortemente determinata dal formato in cui il documento è redatto. Si rimanda, quindi, ai capitoli successivi per specificazioni ulteriori della soluzione.

Occorre notare, infine, che gli elementi di formattazione non sono descritti da stringhe nel formato delle *regular expression*: bisognerà quindi integrare gli elementi presentati in questa sezione con l'approccio *pattern-based* adottato.

2.2 Analisi dell'usabilità

2.2.1 Elenco dei nominativi da trattare esplicitamente espressi come dati in input

In questo caso, per usufruire del servizio, l'utente deve fornire un documento contenente una serie di nominativi da trattare. Una garanzia della corretta elaborazione del documento si ha richiedendo all'utente stesso quali siano i nominativi da trattare: in questo modo potranno essere "minimizzati" i dati (cioè i nominativi) di tutte e sole le persone espressamente richieste. In molti plausibili scenari, infatti, è necessario anonimizzare solo alcuni dei nominativi presenti nel documento; ad esempio in un atto giudiziario serve anonimizzare le parti in causa ma non i magistrati.

Questa configurazione del servizio permette quindi all'utente di avere il massimo controllo possibile del risultato. Tuttavia questo approccio è poco pratico nel caso in cui i documenti da trattare contengano grandi quantità di nominativi diversi e, magari, l'utente che ne richiede il trattamento non li conosce; si pensi ad esempio a lunghe graduatorie di concorsi o altro. Si osserva inoltre che anche per pochi nominativi l'usabilità può degradare, se l'inserimento viene fatto con estemporanea digitazione, esposta anche al rischio di possibili errori di battitura, con conseguenti noiose ripetizioni delle operazioni.

Si osserva, infine, che il sistema progettato è sufficientemente robusto nel trattare i nominativi in input espressi da un utente che ha digitato caratteri maiuscoli o minuscoli violando le convenzioni grammaticali. Supposto che il documento trattato debba essere grammaticalmente corretto, infatti, si ha la garanzia che i nomi ivi presenti comincino con una maiuscola; è sufficiente, quindi, forzare una conversione *to-upper-case* dei nomi inseriti dall'utente e le *regex* progettate funzionano correttamente. Un discorso a parte va fatto per i cognomi inseriti dall'utente, in quanto essi possono comparire con la prima lettera sia maiuscola che minuscola. L'inserimento di un cognome iniziante con una minuscola non crea grossi problemi, in quanto in tal caso la *regex* risultante sarebbe totalmente *case-insensitive*

per il cognome, mentre l'aggiunta di un cognome cominciante con una maiuscola determina una *regex case-insensitive* per la prima lettera del cognome; qualora quindi un utente inserisse un nominativo tutto in maiuscolo, le occorrenze del cognome, presenti nel documento, inizianti con una lettera minuscola non verrebbero riconosciute.

Per le altre lettere dopo la prima, sia per i nomi che per i cognomi, il pattern è stato progettato *case-insensitive*, quindi non emergono problemi.

In sintesi, il rischio che vengano introdotte delle ambiguità lessicali, incaricando l'utente dell'inserimento dei nominativi, è piuttosto basso ed il controllo che si ha sui nominativi è il massimo desiderabile, mentre i requisiti di usabilità risultano penalizzati.

2.2.2 Elenco dei nominativi da trattare dedotti automaticamente da un dizionario

Se si desidera privilegiare la tematica dell'usabilità nella progettazione del servizio, risulta necessario individuare delle strategie che semplifichino il più possibile i compiti che devono essere svolti dall'utente. L'unica operazione che inevitabilmente resta a suo carico è l'upload del documento da trattare; non è infatti necessario richiedergli l'elencazione dei nominativi dei nominativi da trattare, in quanto questi possono essere dedotti automaticamente impiegando dei dizionari, dei quali va quindi valutato il contenuto e le modalità di utilizzo.

Si scarta subito l'ipotesi di dedurre automaticamente i nominativi basandosi unicamente sul fatto che le iniziali di nomi e cognomi siano maiuscole; come già argomentato infatti l'uso delle lettere maiuscole non è limitato solo a questi usi e, inoltre, non si può avere la certezza che un cognome inizi con una maiuscola.

Sono disponibili fortunatamente alcuni dizionari di nomi ed altri di cognomi, in diverse lingue; si fa qui riferimento a quelli di "Data World" (un'azienda focalizzata sulla raccolta, produzione e pubblicazione di dataset: <https://data.world>) [14].

Si inizia l'analisi inquadrando le dimensioni che un dizionario contenente nomi o cognomi avrebbe. Risalta subito all'occhio la differenza tra il numero di termini contenuti nei due casi: facendo riferimento ai soli nomi e cognomi italiani, risultano esistenti circa 350.000 cognomi [15] e circa 9.000 nomi, dei quale circa 5.000 maschili e circa 4.000 femminili [14]; il numero dei cognomi è quindi quasi 40 volte più grande del numero dei nomi.

Si ipotizza, per chiarire le idee, di applicare questi dizionari in uno scenario reale, in cui, ad esempio, si vuole trattare un documento di 10 pagine contenente 5.000 parole. Si trascurano inoltre i meccanismi che permettono di ricondurre un nome od un cognome ad un nominativo per concentrarsi unicamente sul numero di confronti necessari da effettuare nell'elaborazione. Nel peggiore dei casi possibili, ossia quando nessuna parola del documento compare tra i termini del dizionario, e dove occorre quindi confrontare ogni singola parola del documento con tutti i termini del dizionario, per semplice moltiplicazione si ottiene che l'impiego di un dizionario di nomi darebbe luogo a 45.000.000 confronti, mentre un dizionario di cognomi ben a 1.750.000.000 confronti. Se invece le parole nel documento fossero 50.000 si avrebbero 450.000.000 di confronti nel primo caso e 10.750.000.000 nel secondo. In sintesi, sebbene il rapporto tra il numero di confronti nei due casi rimanga sempre costante (circa 1:40) indipendentemente dalla lunghezza del documento, la differenza tra il numero di confronti cresce proporzionalmente al numero di parole che vengono sottoposte all'elaborazione. Per quantificare, infine, attraverso un unità di tempo, la differenza esistente tra l'impiego delle due diverse tipologie di dizionario, si realizza un semplice programma java, di cui si riporta in appendice il contenuto del metodo principale, che realizza i confronti necessari attraverso l'uso di *regular expression*.

Eseguendo il test più volte, inserendo fino a 10 occorrenze della parola "Lorenzo" nel testo, si ha un tempo di elaborazione medio di circa 3 ms, con prestazioni di picco di 2 ms, ottenibili con poche occorrenze del nome presenti, e tempo di attesa massimo di 6 ms, dovuto alla presenza invece a una presenza più frequente del nome nel testo. Questo fenomeno si comprende meglio ricordando che, come spiegato nel capitolo precedente, le *regular expression* ottimizzano la ricerca, considerano le stringhe del testo trattato solo finché necessario, ossia fino a quando non vi è certezza che esse corrispondano o non corrispondano ad un match. Ogni volta che nel testo si presenta una parola che inizia con una lettera diversa dalla 'L' di "Lorenzo", la ricerca procede direttamente con la valutazione della parola successiva, ignorando i rimanenti caratteri della parola.

Risulta, invece, più onerosa la verifica della corrispondenza tra una stringa ed il pattern desiderato, poiché in questo caso vanno elaborati tutti i caratteri della parola. Come caso limite, si è posta come stringa da elaborare la sequenza di caratteri "Lorenzo " ripetuta 500 volte: il tempo di esecuzione medio del programma risultante, a parità di piattaforma, è stato di circa 25 ms.

Un'altra diretta conseguenza di questo meccanismo di confronto è che all'aumentare del numero di parole nel documento non corrisponde un eccessivo aumento del tempo di esecuzione: considerando un documento di 5000 parole, con 0 occorrenze del nome "Lorenzo" il tempo di esecuzione medio risulta di 9 ms, con 10 occorrenze di 10 ms, con 100 occorrenze di 15 ms.

Occorre precisare, prima di formulare ulteriori ragionamenti, che in documento di 5.000 parole potranno essere presenti al più 2500 coppie nome-cognome; in un dizionario con più di 2.500 termini almeno uno di questi non contribuirà nell'individuazione di un nominativo. Se poi sono presenti altre parole oltre ai nominativi nel documento, la percentuale di nomi che presenterà 0 occorrenze crescerà di conseguenza. Ad ogni modo, si sceglie di sviluppare i ragionamenti considerando necessario il tempo medio di 10 ms per individuare le occorrenze di un termine di un dizionario in un documento di 5.000 parole; questo tempo è sicuramente maggiore rispetto al caso reale, ma valutando per eccesso questo valore è possibile trascurare il tempo impiegato nelle altre operazioni di routine a supporto dell'algoritmo di ricerca.

Ritornando all'esempio di partenza, considerando quindi necessari 10 ms per individuare le occorrenze di un termine di un dizionario in un documento di 5.000 parole, risulta che l'identificazione di tutti i cognomi contenuti nel testo richieda circa 3.500 secondi, (quasi un'ora!), mentre l'individuazione dei nomi "solo" 90 secondi. I tempi di attesa che l'utente dovrebbe aspettare sono estremamente elevati e irragionevoli, specialmente se calati nel contesto nelle *web application*. Come primo espediente per ovviare al problema si decide di abbandonare completamente l'idea di impiegare un dizionario di cognomi, in quanto, seppur si individuassero delle soluzioni in grado di effettuare il riconoscimento di tutte le occorrenze di un termine in un documento indipendentemente dalla lunghezza in solo 1 ms (irreale), sarebbero comunque necessari 3 minuti e 50 secondi per l'elaborazione. Non verranno quindi neppure trattate possibili soluzioni che prevedono l'applicazione di entrambi i dizionari.

I 90 secondi richiesti dal dizionario di nomi, invece, risultano anch'essi eccessivi, ma attraverso opportune valutazioni si possono individuare delle strategie che consentono la minimizzazione del tempo necessario all'elaborazione dei documenti. Tali metodi di ottimizzazione saranno trattati in un successivo capitolo, mentre ora ci si continua a concentrare sulle modalità di impiego del dizionario.

Per inciso, si osserva che i problemi di efficienza sono strettamente legati al riconoscimento automatico dei nominativi, in quanto in questo caso devono essere applicate quasi una decina di migliaia di *regex* diverse; nel caso in cui i nominativi da elaborare e le *regex* a loro associate siano noti, si ha a che fare con un numero esiguo di pattern da trattare e l'esecuzione del programma risulta infinitamente più rapida.

Per migliorare l'efficacia del servizio sarebbe opportuno introdurre nel dizionario anche nomi stranieri, sempre più diffusi in una società sempre più globalizzata. Il tempo di esecuzione medio del riconoscimento, già critico, crescerebbe ulteriormente: si decide allora di introdurre nel dizionario i soli nomi inglesi, in totale quasi 5500 [14]. Considerando sempre un tempo medio di esecuzione di 10 ms per nome, con un dizionario di circa 14.500 termini si avrebbe un tempo di esecuzione medio complessivo di 145 secondi, ossia pari a 2 minuti e 25 secondi, e risulta ancora possibile effettuare alcune ottimizzazioni che lo riducono ad un livello accettabile.

Una volta individuati i nomi contenuti nel documento occorre stabilire se essi fanno parte di un nominativo, progettando un'opportuna *regular expression*. È importante notare che per conseguire questo scopo bisogna individuare la soluzione più efficiente possibile.

Un nominativo, come già ripetuto più volte, è composto da uno o più nomi e da un cognome. Individuato un nome, la priorità che si ha è verificare se accanto ad esso sia presente un cognome. Finora i cognomi sono stati supposti non regolamentati da alcun pattern specifico, ma è necessario formularne almeno uno per consentirne il riconoscimento automatico. È ragionevole supporre che un cognome sia formato da massimo due parole, separate da spazio o apostrofo, sempre inizianti entrambe con una maiuscola, fatta eccezione per la prima parola laddove essa sia una preposizione semplice o articolata oppure un articolo, tenendo conto dei possibili troncamenti, come “d'” e “dell'”. Inoltre per verificare se un termine adiacente al nome trovato rappresenti un nome si evita di impiegare nuovamente il dizionario per non gravare ulteriormente sul tempo di esecuzione. Inoltre, si nota che avendo supposto un cognome composto da massimo due parole inizianti con una lettera maiuscola, un altro nome, oltre quello trovato dal dizionario, viene già automaticamente riconosciuto qualora il cognome sia composto da una sola parola. Rinunciando ad individuare nominativi composti da tre nomi o più, si formula una *regular expression* per il riconoscimento automatico in seguito al

identificazione di un nome, qui indicato con il tag <nome>, supponendo il cognome come precedentemente indicato e ipotizzando la presenza di un ulteriore nominativo.

```
<prep>:=d((a|e)(l(l[aeo]?)?|i|gli?)?|i)?|(ne|a|su)(l(l[aeo]?)?|i|gli?)?|l[aeo]?|co[i]n?|i[l]n?|gli|per
```

```
<cognome>:= (([A-Z][a-zA-ZÀ-ÖØ-öø-ÿ]*|<prep>)('|\\s))?[A-Z][a-zA-ZÀ-ÖØ-öø-ÿ]+
```

```
<second_name>:= [A-Z][a-zA-ZÀ-ÖØ-öø-ÿ]+
```

```
<regex>:=(<second_name>\\s)?(<nome>)\\s<cognome>|
```

```
<cognome>\\s(<nome>)(\\s<second_name>)?
```

Nella scrittura della *regex* si è prestata particolare attenzione all'utilizzo della simbologia per rendere l'espressione il più performante possibile, attraverso il massimo sfruttamento dei cortocircuiti logici e l'opportuno ordinamento dei caratteri (sono stati anteposti i caratteri comuni a più preposizioni od articoli). Inoltre, per alleggerire la *regex* si è rinunciato all'utilizzo dei caratteri dell'alfabeto latino non appartenenti ai primi due blocchi Unicode.

Si osserva tuttavia che in alcune circostanze, come nella frase "*di Amorosa Lorenzo non si hanno notizie*", accade che una parola ("*di*" in questo caso), non sia parte del cognome ma il programma non è in grado di riconoscerlo. Questa ambiguità è fortemente dipendente dal contesto e non è possibile trattarla senza significativi degrading delle performance; quindi è necessario rinunciare a trattarla, accettando riconoscimenti erronei. In altre situazioni, invece, dove magari si sta elaborando la stringa contenente il nominativo "*Mario Lorenzo*", risulta impossibile determinare quale tra i due termini sia il nome e quale il cognome; cioè significa che l'unico trattamento di "minimizzazione" automatico che risulta ragionevole applicare è la sostituzione del nominativo con uno pseudonimo, non il troncamento o l'alterazione dei nomi. In conclusione, con il riconoscimento automatico dei nominativi si migliora complessivamente l'usabilità del servizio, in quanto l'utente non deve digitare i nominativi, ma la latenza introdotta è notevole, si è soggetti a rischi di ambiguità e non è in alcun modo esprimibile la preferenza su quali nominativi si desidera "minimizzare" o meno. La soluzione così ideata non risulta quindi adeguata.

2.2.3 Soluzione ibrida adottata

Entrambe le tipologie di riconoscimento prima individuate hanno significativi problemi, si cerca quindi di sintetizzare una soluzione in grado di trarre i vantaggi dell'una e dell'altra. Risulta efficace, in particolare, che il documento venga inizialmente trattato tramite dizionari, evitando all'utente l'onere di specificare i nominativi, e che in seguito venga lasciata all'utente la possibilità di intervenire. Infatti, esso potrebbe voler esprimere delle preferenze su quali nominativi debbano essere trattati tra quelli individuati e gestire gli eventuali errori di riconoscimento dovuti a richieste di trattamento di nominativi aventi un nome non presente nel dizionario o anche dovuti a casi di ambiguità lessicale già presentati.

Una volta inserito il documento e terminata l'elaborazione tramite il dizionario, l'utente può sia richiedere l'immediato download del file ed effettuare le operazioni prima definite, attraverso un'opportuna interfaccia. Per fornire il supporto alle esigenze più disparate, si prevede la possibilità di:

1. eseguire download del file trattato unicamente con il dizionario
2. ripetere la "minimizzazione" del documento, specificando:
 1. nuovi nominativi
 2. quali nominativi tra quelli precedentemente individuati *non* si vogliono trattare
 3. quale parola/e dei nominativi individuati rappresenta il cognome (in tal caso, si possono utilizzare altri metodi, oltre alla sostituzione con pseudonimo, per il trattamento)
 4. quale parola/e dei nominativi individuati non compone il nominativo (situazione che si verifica quando ci si imbatte in una ambiguità).

Senza soffermarsi in questo momento sull'intera sequenza concreta di interazioni tra utente e servizio, si osserva che l'unico vincolo non funzionale da valutare resta il tempo medio di attesa dell'utente. L'usabilità complessiva è molto buona ed i vincoli funzionali sono soddisfatti.

2.3 Scelta dei formati da trattare

Una considerazione intuitiva, ed una buona prassi, è che un documento contenente dati personali sia pseudonimizzato o anonimizzato quanto prima possibile e cioè quando è ancora

solo nelle mani del suo autore: questo approccio scongiura che informazioni sensibili e dati personali in chiaro *sfuggano* in rete. Si può per questo ragionevolmente ipotizzare che i naturali destinatari del servizio siano gli stessi autori (creatori) che redigono il documento. Nello scenario tipico di utilizzo, infatti, il fruitore del servizio procederà al trattamento dei dati non appena avrà finito di scrivere il documento del quale, essendone autore, potrà scegliere il formato. Si osserva che potrebbe accadere che il documento sia redatto da terzi: in tal caso l'utente che richiede il trattamento può scegliere il formato in maniera indiretta concordandolo con il redattore. Avendo quindi l'utente la possibilità di stabilire il formato del proprio documento, risulta ragionevole progettare un servizio che lavori su un solo formato.

A questo punto occorre individuare quale sia il formato che maggiormente si presta alle finalità del servizio.

Una possibilità è quella di richiedere all'utente di inserire come documento da trattare un semplice file di testo in formato *txt*: in questo modo ci sarebbe il grande vantaggio di trattare file molto semplici, riducendo così al minimo la complessità realizzativa, e inoltre si avrebbe l'indipendenza dagli editor di testo utilizzati, in quanto tutti supportano i file in formato *txt*. Risulta tuttavia sconveniente utilizzare questo formato, poiché non ha alcuna capacità espressiva per gestire elementi complessi come tabelle, modifiche allo stile del testo e così via.

Bisogna quindi optare per un formato in grado di gestire questi elementi, al costo di aumentare la complessità della progettazione, ricordando sempre che è necessario allo stesso tempo che tale formato sia supportato da tutti i principali editor di testo.

Con facilità è possibile individuare quali sono i formati di testo più comuni oltre al *txt*: *doc*, *docx*, *odt*, *pdf*, *pages*, *rtf*, *tex*. Si passa ora dunque a valutare quale tra questi formati potrebbe meglio soddisfare i requisiti prima enunciati.

Facendo una cernita iniziale sulla base delle finalità per le quali un formato è utilizzato, si può immediatamente escludere *tex*, che trova impiego principalmente in ambito scientifico e matematico. In questo settore, infatti, non è richiesta generalmente l'applicazione delle procedura di trattamento dei dati.

Si può inoltre abbandonare l'idea di trattare documenti con estensione *pages*, formato proprietario di Apple, poiché utilizzati esclusivamente dall'omonimo editor di testo anch'esso

proprietario, e i documenti con estensione *rtf*, acronimo di *Rich Text Format*, formato proprietario di Microsoft che supporta una formattazione avanzata, poiché, pur gestito da vari editor, è un formato decisamente datato (ultima versione 1.9.1 risalente a marzo 2008 [16]).

Si esclude inoltre il formato *pdf* per motivi di usabilità: molti editor, infatti, consentono di esportare un documento in questo formato ma non permettono di importarlo. Un utente dopo aver effettuato l'anonimizzazione di un documento non può quindi proseguire modificandolo ulteriormente. Il *Portable Document Format*, infatti, è ideato per realizzare dei documenti destinati alla sola lettura. L'ultima esclusione, piuttosto immediata da effettuare, riguarda il formato *doc*, binario e proprietario di Microsoft. Esso infatti risulta, a partire dal 2006, soppiantato dal formato *docx*, sempre proprietario di Microsoft.

Si giunge infine a valutare quale tra gli ultimi due possibili formati rimanenti, ossia *docx* e *odt*, si presta meglio alle finalità del servizio. In via preliminare si osserva che entrambi i formati possiedono ottime capacità espressive, hanno una struttura interna di simile complessità e sono entrambi supportati dai principali editor di testo. È necessario quindi effettuare delle analisi più approfondite per poter sceglierne uno tra i due. La struttura del formato *docx*, sviluppato da Microsoft e formalmente denominato *Office Open XML Document (OOXML Document)*, è costituita da un file compresso *.zip* contenente un insieme di file *XML*. Il formato *OOXML* permette la rappresentazione, oltre che di documenti testuali, anche di fogli elettronici (formato *OOXML Workbook*, noto anche come *xlsx*) e di presentazioni (formato *OOXML Presentation*, noto anche come *pptx*) ([17], [18]). Il formato inoltre è stato inizialmente standardizzato nel 2006 dall'*ECMA* (come *ECMA-376*) e successivamente nel 2008 dall'*ISO* e dall'*IEC* (come *ISO/IEC 29500*) in una versione *transitional*, retrocompatibile con alcune versioni precedenti del formato contenenti elementi deprecati, e in una versione *strict*, dove tali elementi non sono ammessi. I due standard sono stati poi successivamente aggiornati e sono tutt'ora oggetto di revisioni [19].

Anche la struttura del formato open source *odt*, sviluppato dall'*OASIS* e formalmente denominato *OpenDocument Text*, è basata su uno *zip* contenente un insieme di file *XML*. Inoltre, il formato *OpenDocument Format (ODF)* permette di trattare fogli elettronici (formato *OpenDocument Spreadsheet*, noto anche come *ods*) e presentazioni (formato *OpenDocument Presentation*, noto anche come *odp*) [20]. Anche *OpenDocument Text* è stato

standardizzato, in particolare dall'OASIS stesso nel 2005 e dall'ISO/IEC nel 2006, ed è soggetto a revisioni e aggiornamenti [21].

In sintesi, basandosi sulla struttura dei documenti e sulla standardizzazione dei formati, non è ancora possibile individuare quale sia il formato migliore. L'unico elemento che potrebbe portare punti a favore del formato *odt* è che esso, a differenza del formato *docx*, è aperto, tuttavia, essendo le specifiche di entrambi i formati pubbliche, ciò non rappresenta un fattore determinante nell'elezione del formato. Entrambi i formati, inoltre, sono largamente supportati dai word-processor.

Le seguenti tabelle riepilogano il supporto all'uno ed all'altro formato dei *word-processor* più usati, ossia *Microsoft Office Word*, *LibreOffice Writer*, *OpenOffice Writer* e *Pages*.

Le tabelle sono tratte da wikipedia ([22], [23], [24]).

Supporto fornito dagli editor di testo al formato docx

	Microsoft Office Word	LibreOffice Writer	OpenOffice Writer	Pages
Version	2013, 2011 for Mac	All versions	3.0	'08
Operating systems	Windows, Mac OS X	Windows, OS X, Linux, Unix, Android	Windows, Linux, Unix, Mac OS X	Mac OS X
Office suite	Microsoft Office		OpenOffice.org	iWork
Developer	Microsoft	The Document Foundation	Apache OpenOffice	Apple Inc.
License	proprietary	MPL	LGPL	proprietary
ECMA-376	yes	yes	yes	yes
ISO/IEC 29500:2008	yes	yes		
Notes			Import only	

Supporto fornito dagli editor di testo al formato odt

	Microsoft Office Word	LibreOffice Writer	OpenOffice Writer
Version	2007 SP2	4.0.3	3.0.0
Operating systems	Windows	Unix-based systems, Mac OS X, Solaris	Windows, Linux, Unix-based systems, Mac OS X, Solaris
Office suite	Microsoft Office		OpenOffice.org
Developer	Microsoft	The Document Foundation	Apache OpenOffice
License	Proprietary	MPL	LGPL
ISO/IEC 26300:2006	yes	yes	yes
Notes	some limitations	Multiple ODF versions supported (ISO/ODF 1.0/1.1/1.2/1.2 Extended)	adjustable ODF version (ISO/ODF 1.2)

Si possono effettuare le seguenti osservazioni:

- *Microsoft Office Word* è, in ambiente Microsoft, il word processor maggiormente usato; è molto usato anche in ambiente Apple Mac. Esso offre il pieno supporto al formato *OOXML Document*; solo parzialmente invece gestisce il formato *OpenDocument Text*: il processamento di file con estensione *odt* con questo editor comporta la perdita di alcune informazioni secondarie.
- *LibreOffice Writer*, l'editor open source maggiormente utilizzato, supporta pienamente entrambi formati. Nato con lo scopo di gestire i file con formato *OpenDocument Text*, attualmente supporta completamente anche il formato *OOXML Document*.
- *OpenOffice Writer*, un altro degli editor di testo tra i più utilizzati, supporta pienamente *odt*, mentre è solo in grado di importare i file con estensione *docx*.
- *Pages*, il word process installato a default sui Mac della Apple e, di conseguenza anche maggiormente usato su questi dispositivi, non offre supporto al formato *odt*, ma elabora correttamente i documenti con estensione *docx* ([25],[26]).

Queste osservazioni evidenziano che quindi è impossibile adottare un formato supportato da tutti gli editor; la scelta va allora indirizzata verso quello maggiormente supportato dai word processor più popolari. Per avere un'indicazione di *popolarità*, e quindi per poter comprendere

meglio quali tra gli editor prima citati siano i più usati, si può fare un'analisi, tramite motori di ricerca, di quanti file con una certa estensione siano presenti in rete. È possibile, ad esempio, ricercare su Google i file che contengono nel proprio nome una determinata sequenza di caratteri o aventi una certa estensione (*filetype*). Sono qui presentati i risultati delle interrogazioni riguardo ai file aventi formato *docx*, *doc* e *odt* ed il cui nome contiene uno dei caratteri, fra i più frequenti, "1" o "a" o "e". L'investigazione è stata effettuata inserendo nella barra di ricerca di Google le stringhe *1 filetype:docx*, *a filetype:docx* e così via.

Formato cercato	Documenti con "1" nel nome	Documenti con "a" nel nome	Documenti con "e" nel nome
docx	16.900.000	12.800.000	8.730.000
odt	736.000	667.000	507.000
doc	32.300.000	26.300.000	21.500.000

Effettivamente il formato più diffuso è il *doc*; tuttavia esso è supportato per retro-compatibilità anche dagli editor che supportano formati *OOXML*, con i quali non si ha difficoltà nel convertire i documenti dal formato *doc* al *docx*. In conclusione, quale che sia l'editor più diffuso, è ragionevole adottare il formato *OOXML Document* per le finalità del servizio, in quanto molto diffuso, ben supportato dagli editor e avente ottime capacità espressive. Di conseguenza i documenti che saranno elaborati dovranno essere forniti dall'utente in tale formato. In successive sezioni verrà approfondita la struttura dei documenti *OOXML*.

2.4 Privacy by design

L'Art. 25 del *GDPR*, con i Considerando 75 e 78, ha per titolo e per oggetto la "protezione dei dati fin dalla progettazione e protezione per impostazione predefinita" [2], perfettamente in linea con i concetti di "minimizzazione", già richiamati.

Il senso dell'Art. 25 viene spesso sintetizzato con l'espressione "*PbD - privacy by design, privacy by default*" [2]. Il principio fissato nell'Art. 25 dovrà sempre più essere tenuto ben presente nell'ambito dell'ingegneria del software, in quanto impone di adottare nuove cautele nella realizzazione delle applicazioni software.

Del principio *PbD* si è ben tenuto conto nella progettazione descritta in questa tesi. In particolare, relativamente alla *privacy by design*, si è fatto in modo che nessun dato personale permanga registrato sul sistema di esecuzione, o altrove, al termine dell'applicazione. Anche laddove l'applicazione termini in modo anomale non vi sono strutture dati superstiti, che restano memorizzate sul sistema.

Marginale in questa tesi è la nozione di *privacy by default*, giacché l'applicazione non offre all'utente alcuna opzione che possa indurlo in errore ed acconsentire a trattamenti dei dati personali, oltre quello realizzato dall'applicazione.

È utile osservare quanto sia importante esprimere queste impostazioni di progettazione fra le condizioni d'uso presentate all'utente: esse costituiscono uno dei presupposti affinché l'utente abbia piena fiducia (*trust*) nel servizio, lo utilizzi, lo divulghi, lo renda un servizio di successo.

3 Architettura del servizio

3.1 I componenti software nell'architettura LAMP

Il servizio è realizzato in forma di *web-based application* poiché, come già illustrato nell'introduzione, si intende renderlo disponibile per il massimo numero di utenti possibili. Nella progettazione e realizzazione dell'architettura web si adotta la piattaforma *LAMP*, il cui nome deriva dall'acronimo dei componenti open source che la realizzano (il sistema operativo Linux, il server HTTP Apache, il sistema per la gestione di database relazionali MySQL ed il linguaggio di programmazione PHP). Poiché il modello è composto da quattro livelli, spesso si parla anche di *stack LAMP*. Uno degli elementi di forza del modello *LAMP* è che i componenti dei vari *layer* sono sostituibili, a seconda delle esigenze, con dei componenti più adatti, tipicamente open source ([27], [28], [29]). Basando la piattaforma su un sistema operativo della famiglia di Microsoft Windows, ad esempio, si ottiene uno *stack WAMP*, mentre se si usa un sistema operativo Mac OS si realizza uno *stack MAMP*. Un'architettura web basata sul modello *LAMP* può, inoltre, essere integrata con software open source che offre utili funzionalità, come, ad esempio, il monitoraggio (*Nagios*), il load balancing (*Linux Virtual Server*), la rilevazione di accessi illeciti (*Snort*) e il security testing (*netsniff-ng*).

Si valutano ora brevemente la diffusione dei componenti tradizionali dello *stack LAMP* e le alternative maggiormente usate per i vari *layer*.

Le distribuzioni Linux risultano essere la scelta più comune per l'ultimo *layer* dello *stack*. Secondo W3Techs, infatti, nell'ottobre del 2013, il 58.5% dei web server a livello globale avevano come sistema operativo la distribuzione Debian o Ubuntu, mentre il 37.3% una tra RHEL, Fedora e CentOS [30].

Tra i web server, Apache risulta essere maggiormente usato. Secondo le stime fatte da Netcraft, a giugno del 2014 circa il 51.9% del milione di siti web più trafficati al mondo usavano un web server Apache, il 19.2% nginx ed il 12.4% Microsoft [31].

I principali *DBMS* relazionali, oltre a MySQL, che possono essere presenti nello *stack LAMP* sono MariaDB e PostgreSQL, mentre tra i *DBMS NoSQL* MongoDB risulta essere il più comune.

Nello stack, infine, il ruolo di linguaggio di programmazione, solitamente svolto dal linguaggio PHP, spesso è ricoperto dai linguaggi Perl o Python.

Si osserva, ad ogni modo, che le informazioni sulla diffusione delle tecnologie non rappresentano un fattore vincolante nella scelta delle stesse, in quanto la piattaforma *LAMP* è anche aperta verso molti altri componenti oltre quelli citati; di conseguenza la scelta delle tecnologie da impiegare può essere effettuata basandosi principalmente sui requisiti e sulle specifiche del servizio.

3.2 *Principi di progettazione per l'architettura*

3.2.1 *Single Responsibility Principle*

Per rendere il servizio maggiormente manutenibile ed estensibile, si presta particolare attenzione al rispetto del *Single Responsibility Principle (SRP)* [32]. Si cerca, quindi, di fattorizzare il software in più moduli, suddividendo tra questi le elaborazioni da svolgere. Un altro importante beneficio che si ottiene dalla fattorizzazione del codice è la riusabilità dei componenti. Ogni singolo modulo, infatti, svolge il proprio compito in maniera indipendente dal resto della applicazione ed è, quindi, riutilizzabile in altri scenari simili senza dover essere re-implementato. Inquadrando meglio questa metodologia di progettazione con i requisiti del servizio e la piattaforma web *LAMP*, si individuano i due principali compiti a carico dell'applicazione:

- La gestione dell'interazione web con l'utente per la trasmissione del documento e dei nominativi da trattare
- Il processamento del documento per effettuare la "minimizzazione" dei dati.

Questi due differenti *task* saranno, quindi, portate a termine da componenti diversi.

Progettando la struttura dell'applicazione in questo modo, si disaccoppia completamente la logica di business del servizio dalle interfacce web di comunicazione con l'utente. In un futuro sviluppo sarà possibile, quindi, realizzare nuove interfacce utilizzando altre tecnologie senza dover modificare la logica applicativa.

3.2.2 *Dependency Inversion Principle*

Un altro principio di design architetturale che risulta importante nella progettazione è il *Dependency Inversion Principle (DIP)* [33]. Esso si traduce nella realizzazione di componenti che non dipendono dalle specifiche implementazioni degli altri moduli del sistema, bensì dalle loro *astrazioni*. In relazione all'estensibilità e manutenibilità del prodotto software, infatti, l'impiego di moduli dipendenti direttamente dalla definizione dei metodi presenti in altri componenti risulta problematica: il modulo dipendente deve essere re-implementato ad ogni variazione del modulo da cui dipende. Nella programmazione orientata agli oggetti, per rispettare questo principio, si su può fare uso di classi astratte o interfacce, costrutti che definiscono i moduli senza implementarli completamente o senza implementarli affatto.

3.3 *Stile architetturale REST*

Il modello architetturale del servizio che si sta definendo presenta una serie di caratteristiche, come ad esempio l'indipendenza dei moduli, che permettono di realizzarlo secondo il paradigma delle *RESTful API*. L'espressione "Representational State Transfer" e il suo acronimo "REST" furono introdotti nel 2000 nella tesi di dottorato di Roy Fielding [34], uno dei principali autori delle specifiche dell'Hypertext Transfer Protocol (HTTP). Roy Fielding descrive il *Representational State Transfer* come uno stile architetturale ("*architectural style*"), ovvero un'astrazione degli elementi di un'architettura all'interno di un sistema hypermedia distribuito. *REST* non specifica i dettagli dell'implementazione dei componenti e della sintassi del protocollo, ma definisce i ruoli dei componenti, i vincoli sulla loro modalità di interazione e la loro interpretazione. Riassumiamo quindi i principi che deve rispettare una architettura *REST*:

1. Architettura Client-Server

In una architettura *REST* viene data particolare importanza ai principi *SRP* e *DIP* prima citati, più generale si può affermare che l'architettura sposi il paradigma noto con il termine di "*Separation of Concerns*". Una *RESTful API* applica questi principi in un'architettura Client-Server, capace di supportare l'evoluzione indipendente della logica lato client e della logica lato server.

2. Stateless

La comunicazione tra utente e fornitore del servizio deve essere senza stato, quindi

ogni richiesta del client deve contenere tutte le informazioni di cui il fornitore necessita per l'erogazione del servizio offerto. Questa ipotesi viene fatta per rendere una *RESTful API* facilmente scalabile orizzontalmente.

3. Uso della cache

In un Architettura *REST* i messaggi di risposta inviati dal server devono esplicitamente indicare se possono essere memorizzati nella cache del client o di componenti middleware per il riutilizzo nelle richieste successive.

4. Interfaccia uniforme

I componenti devono essere in grado di comunicare attraverso un'interfaccia uniforme e le risorse devono essere trasferite in un formato standard. Inoltre l'utente deve poter effettuare la navigazione tra le risorse di suo interesse tramite collegamenti ipertestuali. Per inciso, si osserva che il protocollo HTTP usato correttamente rispetta i requisiti di un architettura *REST* e che nelle implementazioni delle *RESTful API* il formato più usato per la modellazione dei dati è JSON.

5. Layered System

In un sistema a livelli, componenti intermedi (come i proxy) possono essere collocati tra client e server per intercettare il traffico per scopi specifici; ad esempio il caching o la sicurezza. Una soluzione basata su *REST* può essere composta da più livelli architettonici, indipendenti l'uno dall'altro.

6. Code on Demand

Questo vincolo facoltativo è inteso principalmente a consentire aggiornamenti alla logica all'interno dei client (come i browser Web) indipendentemente dalla logica lato server. Una *single page application*, ad esempio, rispetta in pieno questo punto.

Un'applicazione progettata sul modello dell'architettura *REST* ha quindi gli indiscutibili vantaggi di:

- consentire l'evoluzione indipendente delle diverse componenti
- avere un'interfaccia utente portabile con altri tipi di piattaforme
- permettere agevolmente la replicazione delle macchine server
- non vincolare moduli server e client a linguaggi e tecnologie specifiche.

L'architettura del servizio oggetto di questa tesi si trova già perfettamente in linea con alcuni dei vincoli discussi, ma sono necessarie alcune considerazioni. Il punto 1 ("architettura Client-

Server") è chiaramente soddisfatto. Il punto 2 ("Stateless"), direttamente collegato con i punti 3 e 5 ("Uso della cache", "Layered System"), può essere rispettato con facilità; a ogni richiesta di "minimizzazione" del client corrisponde la restituzione del documento trattato dal server, non c'è quindi necessità di avere uno stato nell'interazione.

Analizzando le caratteristiche di una applicazione stateless in relazione al principio *privacy by design* illustrato in precedenza, si osserva che l'assenza di stato determina la semplificazione delle problematiche critiche relative al principio. Non vengono conservate, infatti, informazioni contenenti dati sensibili, poiché dopo aver effettuato l'invio della risposta, sarà subito eliminato sul server il file elaborato.

Nei capitoli precedenti tuttavia si è detto che l'utente può ripetere più volte il trattamento di uno stesso documento nella stessa sessione di interazione; essendo il vincolo dell'efficienza già difficile da rispettare per via dell'elaborazione onerosa del documento, si può progettare una modalità alternativa del funzionamento con stato del servizio, applicabile in assenza di replicazione delle macchine server e sfruttando le ripetute richieste di trattamento per uno stesso documento. Si può infatti memorizzare lato server il documento inviato dal client inizialmente e, alle successive richieste di trattamento del medesimo documento, evitarne la ritrasmissione da client a server. Per rispettare il *PbD*, al termine della sessione di interazione il documento verrà rimosso dal server. Si nota, inoltre, che per predisporre il servizio alla gestione delle due diverse modalità di funzionamento si può utilizzare un parametro di configurazione a livello applicativo. I punti 4 e 6 infine ("Interfaccia uniforme", "Code on Demand") si possono rispettare utilizzando il formato JSON per la trasmissione dei documenti e dei nominativi, e usando localmente sul client Javascript per offrire tutte le funzionalità del servizio in un'unica pagina html.

3.4 Moduli software del servizio

3.4.1 Scelta dei componenti software

Si opera ora la scelta delle tecnologie da utilizzare per i componenti dei 4 layer dello *stack LAMP*. Per i due layer inferiori risulta piuttosto immediato decidere di usare le tecnologie presenti per default. Un sistema operativo Linux ed un web server Apache sono adatti all'architettura del servizio. Inoltre anche il linguaggio di programmazione PHP può essere

usato senza significativi problemi, anche se verranno fatte in seguito, in una sezione di questo capitolo, alcune considerazioni sulle problematiche di esecuzioni concorrenti di script PHP. Il linguaggio sarà impiegato in particolare per la realizzazione della logica necessaria a gestire l'interazione web con il cliente, mentre l'elaborazione del documento OOXML lato server sarà effettuata da un programma Java.

I due *task* principali del servizio sono delegati, quindi, a due programmi distinti realizzati con tecnologie diverse. Si concretizza in questo modo il principio *SRP* e, sfruttando l'*openness* della piattaforma LAMP, le funzionalità necessarie vengono realizzate attraverso i linguaggi più adatti. La scelta del linguaggio Java per l'implementazione del *main core* della logica di business è dovuta alla libreria open source in ambiente java Docx4j ([35],[36]). Questa libreria è realizzata per il processamento delle tre tipologie di formati OOXML (*docx*, *xslx*, *pptx*) e permette tutte le possibili operazioni di creazione, lettura e modifica su questo tipo di documenti. Anche questa libreria sarà approfondita in sezioni successive. Si osserva, per inciso, che esistono altre librerie equivalenti in ambiente .NET.

Nella scelta del linguaggio di programmazione usato per elaborare i documenti ed effettuare i confronti tramite *regular expression*, un dato non di poco conto da considerare è l'encoding delle stringhe. In Java, in particolare, una stringa è rappresentata internamente come un array di char, ognuno codificato da 2 byte in formato UTF-16 [37]; la codifica esprime in 16 bit tutti i caratteri definiti dallo standard Unicode, quindi è possibile scegliere Java.

Per il *layer* della persistenza, infine, l'impiego di un database MySql risulta una buona scelta. Si osserva che questo livello non verrà utilizzato nella tradizionale maniera prevista dallo *stack LAMP*. Infatti, generalmente, la base di dati viene interrogata direttamente dal componente che si occupa dell'interazione con l'utente (PHP) per il reperimento delle informazioni utili da restituire al client; nel servizio, invece, la base di dati sarà a supporto unicamente del programma Java, poiché gli unici dati persistenti da gestire sono i nomi del dizionario (si ricordi sempre il *PbD*) e solo i moduli dell'eseguibile Java devono utilizzarli. Se il dizionario contenessero unicamente i nomi (senza informazioni accessorie correlate) e fossero usati in sola lettura, un semplice file di testo poteva essere sufficiente per la rappresentazione. Tuttavia, poiché saranno individuate tecniche di ottimizzazione nell'impiego dei dizionari che richiedono operazioni di scrittura e ordinamento dei dati, risulta opportuno usare un database relazionale.

3.4.2 Logica di business e dinamiche di interazione

Vengono presentati in questa sezione gli aspetti principali dei funzionamenti dei moduli del servizio. Per mettere bene a fuoco quali siano le meccaniche di interazione e i flussi di esecuzione dei programmi, si propone il diagramma di sequenza del tipico scenario di utilizzo. In questo schema UML si descrive il caso d'uso dove un utente, dopo aver inserito un documento ed averlo ricevuto "minimizzato", esprime delle preferenze e richiede poi un secondo trattamento.

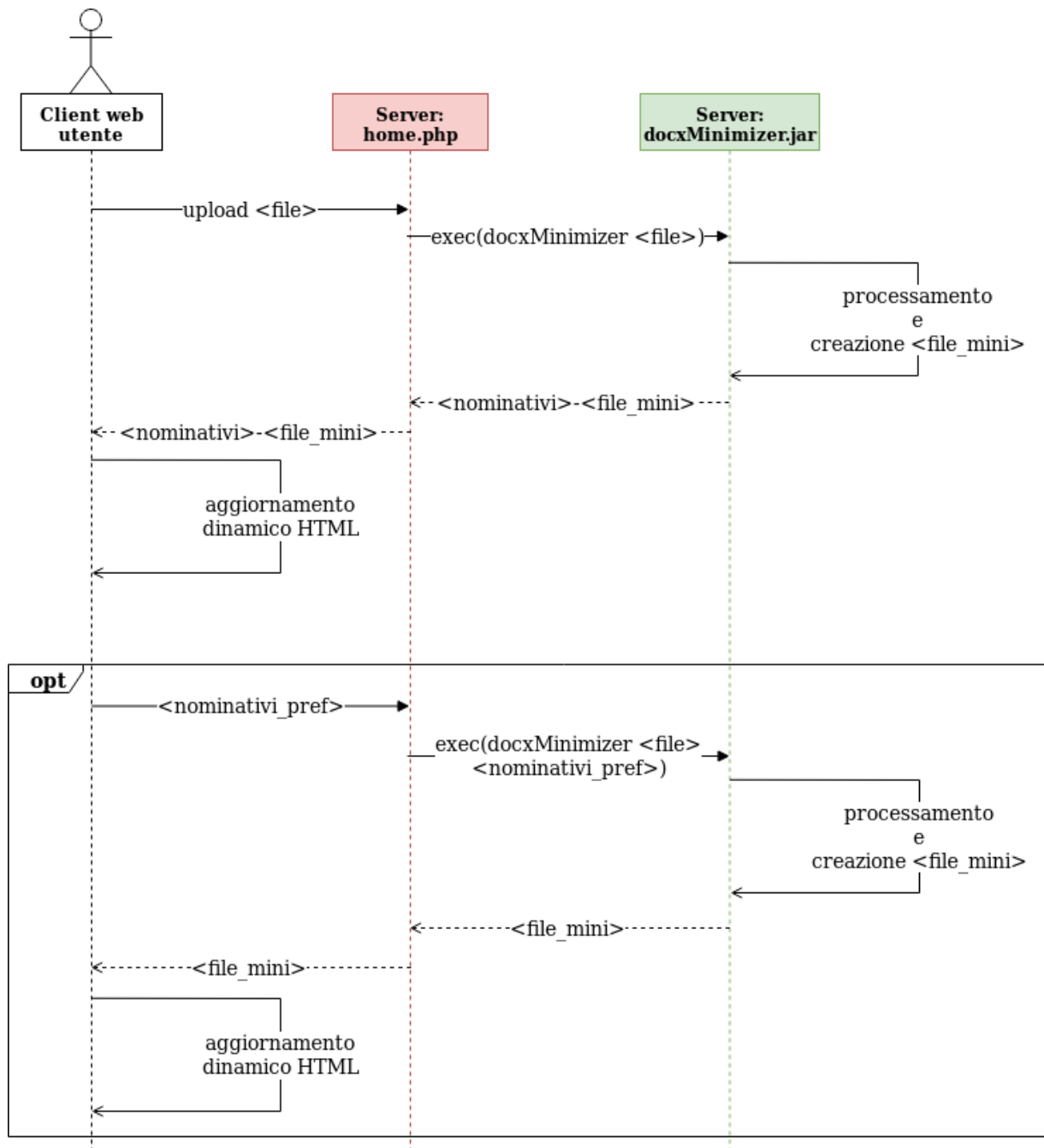


Figura 5. UML - Diagramma di sequenza

Si descrivono quindi ora le principali operazioni che vengono eseguite. All'avvio dell'interazione viene presentata una pagina PHP di benvenuto contenente le informazioni su come i dati personali vengano elaborati ed un *file-picker* per consentire l'upload di un documento *OOXML*; nel momento in cui l'utente confermerà il caricamento del documento, esso verrà trasferito sul server. Bisogna prestare particolare attenzione al comportamento del

sistema nel caso in cui più utenti richiedano contemporaneamente l'esecuzione del servizio, e cioè alle problematiche di esecuzioni concorrenti di script PHP; è necessario, in particolare, evitare situazioni in cui i nomi dei file inviati dagli utenti siano in conflitto. Analizzando più in dettaglio il problema, ogniqualvolta un web server Apache riceve una richiesta HTTP per una pagina PHP, si ha la generazione di un nuovo processo che esegue il codice presentato nella pagina PHP richiesta. Supponendo quindi che N utenti eseguano l'invio del file contemporaneamente, si avrebbe che sul server N processi diversi dovrebbero procedere con la scrittura di un file. Ovviamente se due o più file condividono il nome almeno uno di essi sarà sovrascritto. Per risolvere questo problema, supponendo di voler salvare tutti i documenti in una cartella temporanea, occorre modificare il filename dei documenti inviati dagli utenti per esser certi di non incorrere in sovrascritture o altri tipi di errori correlati. Una valida possibilità è l'inserimento di un *prefisso* univoco nel filename. Per la generazione di una stringa univoca da anteporre al filename, che permetta di distinguere file caricati da utenti diversi, si può direttamente usare il *session id* dell'utente. In PHP esso è determinato casualmente combinando [38]: l'IP del cliente, orario di attribuzione dell'id, un numero pseudo-casuale (con il *PHP Linear Congruence Generator*) e, se presente un "*random source*" sul sistema operativo del Client (spesso chiamato impropriamente "*seme*"), un ulteriore numero pseudo-casuale. Per introdurre ulteriore alea, necessaria se, ad esempio, un utente tenta l'upload di file omonimi (con contenuto interno diverso) e mantenga il medesimo *session id*, si utilizza un ulteriore generatore pseudo-casuale per produrre un altro numero con cui comporre il prefisso. Concatenando i due numeri generati si ha praticamente certezza di aver individuato un numero univoco nel sistema per il tempo in cui il servizio sarà erogato al cliente.

Una volta memorizzato il file, lo script PHP dovrà ricorrere all'invocazione del modulo Java, delegato al vero e proprio processamento del documento. Occorre quindi individuare un set di opzioni che consenta al modulo che gestisce l'interazione con il cliente di sfruttare al meglio il componente delegato all'elaborazione del documento, in qualunque circostanza; la totale indipendenza dei moduli, la loro sostituibilità e la replicazione possibile di macchine server sono solo alcuni dei fattori che rendono mutevole la configurazione del sistema. Va definito un protocollo di comunicazione, quindi, che astragga completamente dall'implementazione dei moduli o dalla locazione dei file.

Valutando i possibili flussi logici, anche con l'ausilio del diagramma di sequenza, si ha che i tipi di esecuzione sono due:

- "minimizzazione" con dizionario
- "minimizzazione" di specifici nominativi

Il protocollo di comunicazione tra i due moduli è costituito in entrambi i casi di una singola richiesta a cui segue una singola risposta. Più in particolare, lo script PHP invoca il comando Java esprimendo obbligatoriamente il path del file da trattare e opzionalmente l'elenco dei nominativi. A suo volta, il modulo Java restituisce i nominativi trovati, qualora non espressamente richiesti dallo script, e segnala la corretta terminazione dell'elaborazione. Si definiscono come canali di comunicazione standard del protocollo gli argomenti presi in input dal modulo di processamento del documento e lo standard output del processo che esegue tale programma. Inoltre è importante definire un formato standard per la comunicazione dei nominativi tra moduli; è necessario quindi scegliere un pattern che permetta di determinare univocamente la composizione dei nominativi. Un possibile formato, espresso tramite *regex*, è il seguente:

```
<nominativo> = ^(<nome>:)*<nome>;<cognome>$
```

```
<nominativi> = ^(<nominativo>\s+)+$
```

Dove nome e cognome sono una sequenza di caratteri Unicode come già discusso. Ulteriori opzioni importanti da definire nel protocollo per l'invocazione del programma che si occupa delle funzionalità di elaborazione sono le seguenti:

- path file da elaborare (opzione "-i <file_path>", obbligatoria)
- path con cui salvare il file elaborato (opzione "-o <file_path>", opzionale: per default viene creato un nuovo file automaticamente)
- abilitazione stampe verbose, da usare solo in fase di debug (opzione "-d", opzionale)

Una volta terminato il processamento, il modulo di gestione dell'interazione con l'utente dovrà accertarsi che l'*exit status* sia 0 e leggere dallo standard output del processo appena terminato, qualora si sia stata eseguita la "minimizzazione" con dizionario, l'elenco dei nominativi trovati. Se invece la "minimizzazione" ha avuto luogo con i nominativi già specificati non sarà scritto nulla sullo standard output. A margine si osserva che l'impiego dell'opzione "-d" deve sempre consentire una semplice individuazione dei nominativi trovati; in particolare, se

tale opzione è specificata, nello standard output i nominativi compariranno con un *header* riconoscibile ("*NOMINATIVO*="). Ritornando alla descrizione del flusso di esecuzione tipo, una volta che il modulo Java ha completato il trattamento del documento, lo script PHP ne effettuerà l'upload sul client e presenterà all'utente le opzioni già descritte nel capitolo sull'usabilità.

A questo punto l'interazione con client potrebbe concludersi, qualora si eseguisse il download e si chiudesse il browser, o continuare elaborando i *suggerimenti* dell'utente espressi dopo il primo trattamento. Le modalità di comunicazione dei moduli varierebbero leggermente come appena illustrato, mentre, a seconda della configurazione stateful o stateless del servizio, verrebbe eseguito o meno un nuovo upload del file dal client al server.

Si osserva infine che, per predisporre a successivi studi di ottimizzazione il servizio e per agevolare tutti i possibili sviluppi futuri, si applica largamente nella progettazione del modulo di elaborazione Java il *Dependency Inversion Principle*, in quanto si desidera realizzare classi estendibili ed intercambiabili. In particolare, si introducono delle classi astratte per rappresentare in maniera generica il concetto di "*persona*" e "*minimizzatore*". È possibile infatti realizzare successivi studi per trattare ulteriori dati personali oltre che ai nomi e cognomi; inoltre, in base ai tipi di documenti trattati o eventuali altri fattori utili, è possibile studiare differenti tecniche di "*minimizzazione*" realizzate da differenti algoritmi "*minimizzatori*". In particolare una "*persona*" dovrà sempre esporre un metodo "*minimizza*" che prende in input una stringa, la "*minimizza*" e la restituisce; il pattern con il quale si effettua la "*minimizzazione*" sarà fornito in implementazione a seconda dei dati sensibili di interesse. Un "*minimizzatore*" esporrà sempre un metodo "*work*" che, presa in input una lista di persone ed un documento *OOXML*, con l'ausilio della libreria Docx4j, fornirà in output un documento del tutto "*minimizzato*"; la definizione delle modalità con cui si estrapolano le informazioni dal documento e con cui si invoca il metodo "*minimizza*" della classe *persona* sono confinate nell'implementazione. Definendo delle classi astratte risulta, quindi, più veloce la realizzazione dei futuri componenti e si svincola il processo di "*minimizzazione*" dal tipo di dati che si "*minimizzano*" e viceversa.

3.4.3 *Le classi principali del progetto*

Le classi principali del progetto vengono quindi presentate in appendice. Esse sono:

- App.java, contenente il main
- Elaborator.java, che presenta il metodo "work"
- Persona.java, che presenta il metodo "minimizza"
- Testing.java, contenente alcune funzioni usate in fase di debug.

Gli script in PHP sono stati forniti dall'azienda come implementazione minimale e provvisoria, da perfezionare a seguito della eventuale definizione sulle modalità di presentazione del servizio su Internet.

Una considerazione di implementazione comune a tutte le classi deriva del principio *Privacy by Design*, indifferentemente dalla modalità di configurazione stateful o stateless del servizio: è di fondamentale importanza che nel sistema non siano memorizzati permanentemente in alcun caso i documenti inviati dagli utenti del servizio. Per realizzare un sistema robusto, per fronteggiare crash improvvisi del client o congestioni della rete, è opportuno impostare dei timeout lato server che procedano automaticamente alla eliminazione dei documenti degli utenti dopo un periodo di inattività troppo prolungato. Qualora invece si dovesse verificare un interruzione critica del servizio a causa di un errore logico del server o semplicemente per via di un'interruzione dell'alimentazione della macchina server, bisogna considerare altri metodi; essendo il servizio realizzato con un sistema operativo Linux, si può configurare il demone "crond" per eseguire a ogni reboot e, per sicurezza, una volta al giorno l'eliminazione dei file usati dall'applicazione tutti contenuti all'interno della cartella temporanea.

4 Approfondimenti tecnologici

4.1 Analisi della struttura del formato OOXML

Come argomentato, il formato dei documenti elaborati dal servizio progettato dovrà essere *Office Open XML*, o *OOXML*; si tratta di un formato *XML-based* per documenti di testo, fogli elettronici e presentazioni, in grado di rappresentare grafici, diagrammi, immagini e altro materiale grafico. La specifica fu sviluppata da Microsoft e adottata dall'*ECMA International* come standard *ECMA-376* nel 2006. Una seconda versione fu rilasciata nel 2008, una terza nel 2011, una quarta nel 2012 ed una quinta tra il 2015 ed il 2016 ([39]). La specifica è stata adottata, inoltre dall'*ISO* e dall'*IEC* come standard *ISO/IEC 29500* a partire dal 2008 ([40], [41]).

4.1.1 Linguaggi di markup

Lo standard *ECMA-376* [39] include tre differenti specifiche di linguaggio per ognuna delle tre principali categorie di documenti:

- *WordprocessingML* per i documenti testuali
- *SpreadsheetML* per i fogli elettronici
- *PresentationML* per le presentazioni elettroniche
- *DrawingML* ed altri linguaggi di markup di supporto per la rappresentazione di disegni, forme e diagrammi.

La specifica include sia gli schemi *XML* sia i vincoli espressi per iscritto. Ogni documento conforme al formato dovrà, quindi, rispettare gli schemi *XML* e dovrà essere codificato in *UTF-8* o *UTF-16*. La specifica, inoltre, permette l'aggiunta di *custom tag XML* a supporto dei linguaggi di markup dati, per default nel formato *OOXML*, consentendo quindi la libera personalizzazione dei documenti.

4.1.2 File packaging

Oltre alla specifiche relative ai linguaggi di markup, la seconda parte dell'*ECMA-376* [39] definisce la struttura gerarchica dei file del formato adottando le *Open Packaging*

Conventions (OPC). *OPC*, una tecnologia *file-container* basata sul comune formato compresso *zip*, che stabilisce che tutti i file che riguardano una stessa entità devono essere raggruppati in un unico package [42]. Un documento *OOXML* è, infatti, un archivio *zip* contenente alcuni files *XML* (detti anche *parti*) organizzati in un singolo package. Questa frammentazione dei dati in più files rende più semplice e veloce l'accesso ai dati stessi e riduce le possibilità di una loro corruzione. Le *parti* possono contenere qualsiasi tipo di dato; per tenere traccia della relazione tra una *parte* e la sua tipologia di contenuto, senza basarsi sull'estensione del file, è presente nel package, nella cartella radice della gerarchia, il file denominato *[Content_Types].xml*, il quale mappa appunto queste associazioni. È mostrata qui ad esempio l'associazione tra la *parte* rappresentante il documento principale e il suo *ContentType*.

```
<Override
```

```
PartName="/word/document.xml"
```

```
ContentType="application/vnd.openxmlformats-officedocument.wordprocessingml.document.main+xml"/>
```

Le informazioni relative alle relazioni che ogni *parte* ha con le altre *parti*, con risorse esterne e con il package in sé sono mantenute separatamente dal contenuto della *parte* stessa. Tali informazioni sono presenti, infatti, all'interno delle cartelle denominate *_rels*: ne esiste una per il package nel suo complesso e una per ogni sotto-cartella del package contenente delle *parti* coinvolte in delle relazioni. In questo modo i riferimenti che una *parte* ha verso altre *parti* o risorse sono salvati una sola volta e possono essere aggiornati con semplicità se necessario, senza dover modificare il contenuto stesso delle *parti* coinvolte. È mostrato qui un esempio di relazione contenuta in *_rels/.rels* relativa al documento principale e al suo schema.

```
<Relationship
```

```
Id="rId1"
```

```
Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument" Target="word/document.xml"/>
```

Il documento principale, ossia il file *word/document.xml*, ha inoltre numerose relazioni con altre *parti*, come ad esempio *word/styles.xml* e *word/footer.xml*, e risorse esterne collegate con degli URI. Per descrivere queste relazioni si usa la cartella *word/_rels*.

4.1.3 Parti di un Documento OOXML

Vengono qui presentate le *parti* che sono caratteristiche esclusive di un *WordprocessingML package*, non presenti quindi negli altri due formati *OOXML*. È importante osservare che alcune di esse sono facoltative e sono presenti nel package solo se necessario.

Le *parti* relative al vero e proprio contenuto testuale sono:

- *Main Document*, contiene le informazioni principali ed è salvata come *word/document.xml*
- *Header*, contiene i dati relativi alle intestazioni ed è salvata in *word/header.xml*. Ogni sezione del documento può avere intestazioni diverse per la prima pagina, per le pagine pari e per le dispari, quindi possono esserci molteplici *parti header*
- *Footer*, contiene le informazioni relative al piè pagina ed è salvata in *word/footer.xml*. Può essere presente più volte, per motivi analoghi alla *parte header*
- *Footnotes*, contiene le eventuali note a piè pagina del documento ed è salvata come *word/footnotes.xml*
- *Endnotes*, contiene le note di chiusura del documento ed è salvata in *word/endnotes.xml*.

Sono presenti poi delle *parti* relative allo stile e alle impostazioni del documento:

- *Style Definitions*, contiene la definizione di un insieme di stili usati dal documento, salvata in *word/styles.xml*
- *Font Table*, contiene informazioni riguardo i font usati, salvata in *word/fontTable.xml*
- *Numbering Definitions*, contiene le definizioni sulla struttura delle liste contenute nel documento, salvata in *word/numbering.xml*
- *Document Settings*, specifica come il word processor debba trattare il documento (spell checking, gestione delle revisioni, etc.), contenuta in *word/settings.xml*
- *Web Settings*, contiene informazioni su come il documento debba essere convertito in *HTML* se richiesto, contenuta in *word/webSettings.xml*.

Sono presenti anche delle *parti* che rappresentano testo secondario:

- *Comments*, contiene i commenti sul documento inseriti attraverso un word processor

- *Glossary*, contiene dati testuali aggiuntivi secondari, ne è permessa una sola. Tutte le parti prima elencate, tranne il *Main Document*, possono comparire una seconda volta in relazione alla parte *glossary*.

4.1.4 Analisi del Main Document

Il file *document.xml*, elemento principale di un documento *WordprocessingML*, è costituito da due tipi di informazioni:

- *properties*, che definiscono lo stile e la formattazione del testo
- *stories*, che rappresentano il contenuto testuale delle varie parti del documento.

Le *properties* verranno trattate in misura minore. Queste informazioni sono espresse attraverso una struttura XML gerarchica che ha come elemento radice il nodo *document*. Esso ha due nodi figli:

- *background*, che contiene alcune *properties* che descrivono lo sfondo del documento
- *body*, che contiene tutti i rimanenti contenuti ed è potenzialmente molto complesso.

Si osserva che per le finalità del trattamento dei dati risultano di primario interesse le *stories* e che è opportuno effettuare un'analisi *bottom-up* del problema: anziché considerare la gerarchia a partire dal *body* (nodo padre) partiremo dalla rappresentazione più semplice del testo contenuta nel documento (nodi figli).

Facendo direttamente riferimento allo standard *ECMA-376* [39], si evince che l'unico nodo che contiene puro testo ha il nome *t* (*Text*). Questo elemento contiene una stringa rappresentante gli esatti caratteri che vengono poi mostrati negli editor a video. Il nodo *Text* non presenta figli e l'unico nodo padre che può avere è *r* (*Run*). Quest'altro nodo definisce una regione di testo con comuni proprietà (ad esempio l'uso del grassetto). Attraverso l'uso di tag *t* ed *r* quindi si rappresenta una regione di testo formattata con differenti elementi stilistici. Occorre però evidenziare che un nodo *r* può presentare molti altri figli oltre a *t*, come ad esempio immagini, in quanto rappresenta un'area generica del documento. Si supponga quindi di avere due nodi *Text* fratelli, ossia figli dello stesso nodo *r*: essi rappresenteranno semanticamente un unico blocco logico se e solo se compariranno a video come una sequenza

di parole continua. Questo fenomeno è facilmente identificabile anche nel file *XML* descrittivo: due sequenze di parole mostrate a video adiacenti compaiono infatti nel file *XML* in due righe consecutive; se invece due parti di testo dovessero essere intervallate ad es. da un'immagine nel documento *XML* ci sarebbe un tag *drawing* a dividere i due tag *t*. Dalla consecutività dei nodi *Text* nel file *document.xml* si può determinare quali siano i blocchi logici da trattare. Un insieme di *Run* infine può essere figlio di diversi nodi, ma ciò non è di rilevante importanza, in quanto ogni area di testo individuata dal nodo *Run* rappresenta già un blocco logico a sé e, di conseguenza, contiene sufficienti informazioni per la "minimizzazione". L'unico caso di rilievo nel quale bisogna valutare quale sia la gerarchia a monte di un nodo *Run* è nell'elaborazione di una tabella. In tal caso, in realtà, occorre verificare opportunamente la strutturazione della gerarchia processata. Per chiarezza espositiva viene mostrata la rappresentazione in formato *OOXML* di una tabella con una riga e due colonne (senza l'uso di *properties* per la formattazione stilistica).

```
<w:tbl>
  <w:tr>
    <w:tc>
      <w:p>
        <w:r>
          <w:t>Cella A</w:t>
        </w:r>
      </w:p>
    </w:tc>
    <w:tc>
      <w:p>
        <w:r>
          <w:t>Cella B</w:t>
        </w:r>
      </w:p>
    </w:tc>
  </w:tr>
</w:tbl>
```

Come si osserva in figura, in una tabella il nodo *r* sarà figlio del nodo *p* (*Paragraph*), il quale a sua volta sarà contenuto in un nodo *tc* (*Table Cell*). Celle di una tabella appartenenti ad una

stessa riga saranno presenti all'interno dello stesso *tr* (*Table Row*) ed infine tutte le righe della tabella saranno inserite del tag *tbl* (*Table*).

Per determinare quindi se due nodi *Text* sono scritti in due colonne adiacenti di una stessa riga (e quindi costituiscono un unico blocco logico) è necessario verificare che per i due nodi *Text*:

1. il nodo "padre" **non sia** in comune
2. il nodo "padre di 2° livello" sia *Paragraph* e **non sia** in comune
3. il nodo "padre di 3° livello" sia *Table Cell* e **non sia** in comune
4. il nodo "padre di 4° livello" sia *Table Row* e **sia** in comune
5. i nodi "padre di 3° livello" **siano** consecutivi.

Se tutte queste ipotesi sono soddisfatte, le due celle vanno "minimizzate" come unico blocco logico.

Si valutano infine gli ultimi vincoli emersi nell'analisi sull'interpretazione della formattazione di un documento. In particolare, descrivendo le parti che compongono un documento *OOXML*, era già emerso che alcune sezioni di testo, ossia note a piè pagina ed intestazioni, compaiono in altri file *XML* e non in *document.xml*. Questi file (*header.xml*, *footer.xml*, *endnotes.xml*, etc.) sono tutti raccolti in un package ("word") e la grammatica *XML* è la stessa usata per il Main Document. Si conclude osservando che il problema della divisione in sillabe delle parole non si pone, in quanto la divisione sillabica mostrata a video dai word-processor è calcolata dinamicamente; in particolare, su una porzione di testo viene applicata la sillabazione (ove necessario) se è presente tra le *properties* di quella regione di documento il tag *hyphenationZone*.

4.1.5 La libreria in ambiente java open source Docx4j

La libreria Docx4j offre completo supporto alla manipolazione di documenti *docx* (compressione e decompressione archivio zip, generazione file *XML* necessari, etc.). Offre inoltre un'utile mappatura in specifici oggetti Java della gerarchia *XML* presente nei vari file del formato. Uno tra gli elementi più interessanti è la tecnica con cui vengono recuperati i nodi dai file *XML*, in quanto si fa impiego del linguaggio standard W3C *XPath* ([44], [45]). Con *XPath* è possibile esprimere con una stringa un sottoinsieme di nodi presenti in una gerarchia *XML* non solo in base al loro nome o valore, ma anche in relazione alla posizione reciproca rispetto agli altri nodi. Riferendoci alle problematiche del servizio risulta, ad

esempio, significativamente agevolato il trattamento di dati personali in forma tabellare. Si osserva infine che la libreria attribuisce un id univoco ad ogni nodo del documento, di conseguenza se ne può trarre vantaggio nei confronti tra i nodi.

4.2 Ottimizzazioni del processo di minimizzazione

Durante l'analisi dei requisiti è emerso che l'efficienza costituisce un aspetto problematico del servizio. Si ricorda di aver stimato la durata di una prestazione media di circa 145 secondi, considerando dati:

- un documento di medie dimensioni (10 pagine ~ 5000 parole)
- tempo medio di 10 ms per individuare le occorrenze di un nome
- un dizionario di circa 14.500 parole

In linea di massima si individuano due generi di strategie che consentirebbero la riduzione del tempo medio di esecuzione: uno mirato a ridurre il più possibile il testo da sottoporre a "minimizzazione" automatica, l'altro a ottimizzare l'impiego del dizionario e dell'algoritmo di ricerca.

4.2.1 Riduzione del testo da trattare

Come già spiegato, l'approccio di ricerca *pattern-based* è poco sensibile all'aumentare della lunghezza del documento, ma da questo valore ha una dipendenza non trascurabile.

Nel documento è altamente probabile che ci siano interi periodi, se non pagine, dove non sia presente alcun nominativo. Anche all'interno di una frase che necessita di *minimizzazione* è plausibile che siano poche le coppie (al più le terne) di parole riconducibili ad una sequenza nome-cognome. È inessenziale che sezioni di testo che non necessitano di "minimizzazione" vengano processate. Si osserva per inciso che, ovviamente, questo ragionamento è corretto per via dell'indipendenza dal contesto di un approccio *pattern-based*.

Il pattern definito per la ricerca di un nominativo è molto restrittivo, in quanto accetta solo sequenze di parole costituenti un nominativo contenente un nome specifico; d'altronde la progettazione del pattern è stata effettuata mirando ad identificare il più accuratamente possibile le sequenze da anonimizzare.

Analizzando attentamente il problema, si può fare il seguente ragionamento: per ridurre il più possibile il numero di parole da processare a ogni iterazione di ricerca del pattern di un nome, è possibile individuare preliminarmente tutte le sequenze di parole del documento che *potrebbero essere* dei nominativi e *potrebbero* fare match.

L'operazione di pre-filtraggio si applica processando il documento con un *pattern* che presenta un'espressione più generale e permissiva delle *regex* definite per i nomi del dizionario. Si osserva che una qualunque successione di caratteri *alfabetici* iniziante con un carattere maiuscolo *potrebbe* rappresentare un nome. Per realizzare un'espressione adatta si parte dalla definizione della *regular expression* già data ai nomi del dizionario:

```
<regex>:=(<second_name>\s)?(<nome>)\s<cognome>|
```

```
<cognome>\s(<nome>)(\s<second_name>)?
```

Nell'uso fatto finora di questa espressione, al tag <nome> viene attribuito, ciclo dopo ciclo, una parola del dizionario diversa. È possibile definire una variante più generale dell'espressione appena presentata:

```
<nome_pre>:= [A-Z][a-zA-ZÀ-ÖØ-öø-ÿ<extra>]+
```

```
<regex_pre>:=(<second_name>\s)?(<nome_pre>)\s<cognome>|
```

```
<cognome>\s(<nome_pre>)(\s<second_name>)?
```

Con questa *regex* si identifica una qualunque sequenza che rappresenta un nominativo, ma allo stesso tempo anche, ad esempio, tutte le parole consecutive inizianti con una maiuscola. È importante sottolineare che questo pattern presenta tutte le caratteristiche già ampiamente discusse sul formato delle *regex* per riconoscimenti automatici (gestione dei delimitatori, posizione relativa tra i nomi ed il cognome, etc.). Il processamento di una *regex* generale inoltre sarà inevitabilmente più lento di quello di *regex* aventi i nomi specificati, in quanto molte più sequenze faranno match essendo l'espressione più permissiva. La mancanza di accuratezza va rapportata all'impiego della *regular expression*: fornendo essa un semplice pre-filtraggio del documento non risulta necessario nè che individui esclusivamente nominativi nè che sia in grado di distinguerli gli uni dagli altri. Il leggero ritardo introdotto viene valutato sperimentalmente, applicando lo stesso l'algoritmo già usato in precedenza ed elaborando gli stessi documenti, sempre a parità di piattaforma.

Partendo dall'elaborazione di un documento contenente 10 nominativi ed in totale 5.000 parole, si misura che tempo il processamento medio della *regex* di pre-filtraggio è di circa 36 ms; si ricorda che per lo stesso documento il tempo di esecuzione medio per il processamento di un nominativo non presente è di circa 9 ms, se le occorrenze salgono a 10, il tempo è in media di circa 10 ms. Sebbene l'elaborazione di questa *regex* sia circa 4 volte più lenta dell'elaborazione di *regex* più specifiche, un ritardo introdotto dell'ordine dei millisecondi è trascurabile rispetto al tempo di processamento complessivo (stimato di circa 145 second). Si osserva, per inciso, che nell'elaborazione di questo particolare documento sono stati individuati altre 21 sequenze che rispettavano il pattern che non costituivano dei nominativi.

Terminata l'esecuzione dell'operazione di pre-filtraggio, occorre sfruttare opportunamente le informazioni ottenute: in particolare, si possono elaborare con le *regex* dei nomi le sole sotto-stringhe che hanno fatto match con la *regular expression* nell'operazione preliminare. In altri termini, con questo metodo si effettua una sola elaborazione del testo completo e molteplici processamenti delle sotto-stringhe che *potrebbero* contenere nominativi. Si osserva inoltre che, conoscendo con certezza gli indici di inizio e fine delle sotto-stringhe che *potenzialmente* contengono nominativi, è possibile "alleggerire" la *regex* associata ai nomi del dizionario: si possono rimuovere i costrutti *lookahead* e *lookbehind* in quanto la sotto-stringa è stata già rimossa dal contesto e quindi rappresenta esclusivamente i caratteri individuati dalla *regular expression* di pre-filtraggio.

A questo punto occorre valutare sperimentalmente l'ottimizzazione ottenuta. Si misura che sono sufficienti 2 ms di elaborazione in media per nome, mentre, nel caso raro in cui tutti i 10 nominativi presenti si riconducano ad un unico nome, il tempo medio di esecuzione tende verso i 3 ms. Si osserva che questo è possibile perché le parole nelle sotto-stringhe sono poco meno di 50, ossia si è ridotta di oltre il 99% la dimensione complessiva di dati da processare! Valutando l'ottimizzazione complessiva in termini di tempo si ha che, processando un dizionario di circa 14.500 termini, sono necessari in media 29 secondi.

Ovviamente questo dato è fortemente dipendente dal contenuto del documento. Vanno effettuate opportune misurazioni nel caso in cui la percentuale di nominativi presenti rispetto alle parole complessive nel documento sia maggiore. Si considera un documento di 5.000 parole contenente 200 sequenze riconosciute dall'algoritmo di pre-filtraggio delle quali 100 costituiscono dei nominativi. Si misura che il tempo di esecuzione medio è di 6 ms nel caso

generale del processamento di un nome con poche occorrenze e tenderà verso i 10 ms nel caso sporadico in cui le occorrenze del nome saranno proprio 100 o poco meno. Il tempo di esecuzione del pre-filtraggio è di circa 48 ms. Per inciso, si osserva che la ragione per cui il tempo necessario al pre-filtraggio non aumenta esageratamente è dovuta al pattern della *regex*. La *regular expression* impiegata infatti, facendo principalmente uso di caratteri espressi in un range unitamente al quantificatore "+", risulta particolarmente efficiente da elaborare. Nel caso di documenti più "ricchi" di nomi si stima l'elaborazione media complessiva di circa 1 minuto e 27 secondi.

Facendo il punto della situazione, si è ridotto di circa 5 volte il tempo di esecuzione complessivo nel caso di documenti "scarsamente popolati" da nominativi e quasi dimezzato il tempo necessario nell'elaborazione di documenti contenenti molte coppie nome-cognome. Occorre proseguire nello studio per migliorare l'ottimizzazione, considerando dove l'algoritmo di ricerca e l'impiego del dizionario possono essere migliorati.

4.2.2 Ordinamento del dizionario

Viene posta una semplice domanda sulle modalità di ricerca dei nominativi nell'insieme delle sotto-stringhe del documento: è computazionalmente equivalente prendere una ad una le sotto-stringhe e confrontarle con le *regex* dei nomi rispetto che prendere una ad una le *regex* dei nomi e confrontarle con le sotto-stringhe?

Si fornisce una spiegazione formale alla domanda.

Si definiscono due insiemi:

- $A = \{a_1, a_2, \dots a_n\}$
- $B = \{b_1, b_2, \dots b_m\}$

Gli elementi dell'insieme A rappresentano le sotto-stringhe (cardinalità N), mentre gli elementi dell'insieme di B indicano le *regex* dei nomi (cardinalità M). È importante considerare che per il secondo insieme è definita una relazione di unicità che lega il valore degli elementi: per ogni b_i appartenente a B non esiste un indice j , j compreso tra 1 ed M, tale che il testo del nodo i sia uguale al testo del nodo j (tranne se $i = j$); in altre parole gli elementi rappresentanti le *regex* (b_i) per definizione sono posti univoci in B.

Si presti attenzione al fatto che l'insieme delle sotto-stringhe presenta N elementi distinti tra loro in identità ("id nodo"), ma il cui contenuto testuale può essere equivalente a quello di altre sotto-stringhe. L'equivalenza tra due sotto-stringhe sussiste, in questo particolare contesto, qualora i contenuti testuali di entrambe facciano match con una stessa *regex* presente nell'insieme B . L'insieme delle *regex*, invece, presentano elementi i cui valori testuali sono sempre univoci, in quanto ottenuti da nomi sempre diversi.

Definiti gli insiemi si procede alla formulazione della risposta alla domanda precedente. Per chiarezza espositiva indicheremo con " $A \rightarrow B$ " la procedura con cui si opera il confronto prendendo una ad una le sotto-stringhe per poi confrontarle con le *regex*, mentre indicheremo con " $B \rightarrow A$ " la procedura con cui si opera il confronto prendendo una ad una le *regex* dei nomi per poi confrontarle con le sotto-stringhe.

Nel caso in cui l'insieme delle sotto-stringhe non contenga alcun nominativo, si ha una complessità computazionale equivalente per " $A \rightarrow B$ " e per " $B \rightarrow A$ ". Viene mostrato nei calcoli in figura che il numero di confronti necessari C_1 è dato dal prodotto delle cardinalità degli insiemi, ossia $C_1 = N \cdot M$.

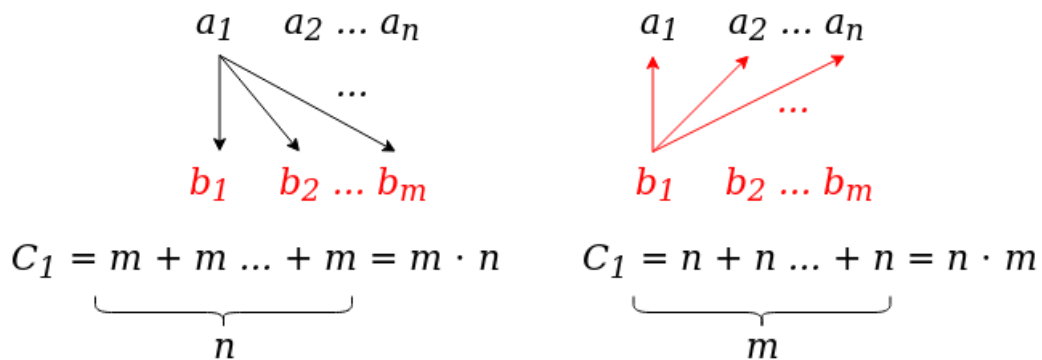
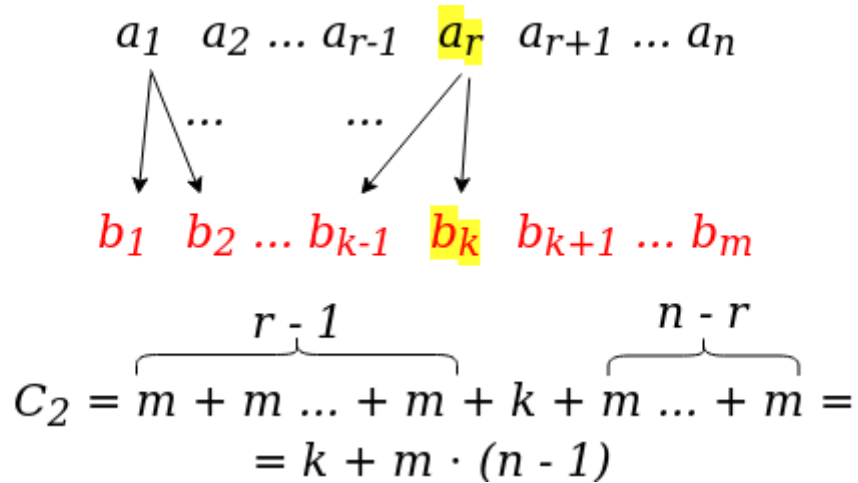


Figura 6. Calcolo di C_1



$$\begin{array}{ccccccc}
 a_1 & a_2 & \dots & a_{r-1} & a_r & a_{r+1} & \dots a_n \\
 \swarrow & \downarrow & \dots & \downarrow & \searrow & \downarrow & \\
 b_1 & b_2 & \dots & b_{k-1} & b_k & b_{k+1} & \dots b_m
 \end{array}$$

$$\begin{aligned}
 C_2 &= \overbrace{m + m \dots + m}^{r-1} + k + \overbrace{m \dots + m}^{n-r} = \\
 &= k + m \cdot (n-1)
 \end{aligned}$$

Figura 7. Calcolo di C_2

Si suppone ora che il testo della sotto-stringa di posizione r faccia match con la *regex* in posizione k ; si applica "A->B". Come si mostra attraverso i calcoli in figura, il numero totale di confronti C_2 sarà inferiore a C_1 poiché, quando la sotto-stringa in posizione r risulta equivalente alla *regex* in posizione k si può direttamente passare alla valutazione della stringa in posizione $r + 1$ e non svolgere i rimanenti $m - k$ confronti per la stringa in posizione r . Si calcola che $C_2 = k + m \cdot (n - 1)$.

Supponendo sempre che il testo della sotto-stringa di posizione r faccia match con la *regex* in posizione k , si applica "B->A". Similmente al caso precedente, quando la sotto-stringa in posizione r risulta equivalente alla *regex* in posizione k si può direttamente passare alla valutazione della *regex* in posizione $k + 1$ e non svolgere i rimanenti $n - r$ confronti per la *regex* in posizione k . Si osserva inoltre che è inutile confrontare le rimanenti $m - k$ *regex* con la sotto-stringa di posizione r : essa è risultata già equivalente ad una *regex* e, essendo le *regex* poste per definizione diverse tra loro, è di conseguenza impossibile che soddisfi un'altra uguaglianza. Per inciso si osserva che nel calcolo di C_2 un ragionamento di questo genere non è applicabile: ogni sotto-stringa può potenzialmente fare match con una qualsiasi *regex*. Il numero totale di confronti C_3 viene calcolato come mostrato, si ottiene $C_3 = k + m \cdot (n - 1) - n + r$.

$$\begin{aligned}
C_3 &= \overbrace{n + n \dots + n}^{k-1} + r + \overbrace{(n-1) \dots + (n-1)}^{m-k} = \\
&= n \cdot (k-1) + r + (n-1) \cdot (m-k) = \\
&= k + m \cdot (n-1) - n + r
\end{aligned}$$

Figura 8. Calcolo di C_3

Si osserva che $C_3 = C_2 - n + r$ e, essendo r la posizione della sotto-stringa coinvolta nel match, si ha che C_3 è sempre minore di C_2 qualunque sia la posizione della sotto-stringa. L'unico caso in cui $C_3 = C_2$ si verifica quando la $r = n$, ossia se la sotto-stringa è l'ultima dell'insieme.

Per estendere il ragionamento, si prende la *regex* di posizione k e si indica con T il numero di sotto-stringhe che sono già risultate equivalenti ad una *regex* nelle elaborazioni precedenti. Risulta che, per il ragionamento spiegato nel calcolo di C_3 , potranno essere risparmiati almeno $N - T$ confronti per la *regex* k e per ogni *regex* in posizioni successive. In generale, maggiore è il numero di nominativi nel documento, maggiore sarebbe l'ottimizzazione della procedura.

Le conclusioni che si traggono sono due:

- indipendentemente dal numero o dalla posizione dei nominativi nelle sotto-stringhe, nell'algoritmo di minimizzazione è più efficiente prendere una ad una le *regex* dei nomi e con ognuna trattare l'insieme delle sotto-stringhe
- un efficace ordine di elaborazione delle *regex* può introdurre ottimizzazioni

Va eseguita ora una valutazione sperimentale dei tempi di esecuzione del processo di "minimizzazione" tenendo conto di questi elementi. Le *regex* che fanno match con i nominativi presenti nel testo, quindi, vengono trattate tra le prime. Risulta di interesse principalmente valutare i miglioramenti ottenibili nel caso di documenti *molto popolati* da

nominativi, poiché è questo il caso in cui l'ottimizzazione entra in gioco. Si esegue il test sullo stesso documento trattato precedentemente con la sola tecnica di pre-filtraggio. Esso è composto da 5.000 parole, dal pre-filtraggio sono individuate 200 sotto-stringhe delle quali 100 costituiscono dei nominativi. Nel test si è fatto riferimento a dati Istat [43] per stabilire, quanto più realisticamente possibile, il miglior ordine di elaborazione delle *regex* associate ai nomi; risulta che circa il 25% dei nominativi è individuato entro i primi 15 cicli di esecuzione; circa il 35% entro i primi 50 ed il 100% entro circa i primi 3000. Per inciso, si osserva che un altro documento può dare origine a tempi significativamente peggiori qualora presenti molti nominativi con nomi desueti e presenti in fondo al dizionario. Si misura che, dopo il "transitorio iniziale" (ossia dalla 51° *regex* in poi), il tempo medio è di 5 ms per l'elaborazione di una *regex*. Si osserva che, una volta che sono state elaborate circa le prime 3000 *regex*, il tempo medio di elaborazione di una *regex* scende a 4 ms. Per completezza, si riporta anche il tempo medio per la fase di "transitorio iniziale", che si misura di circa 8 ms; il tempo di pre-filtraggio, già precedentemente calcolato, risulta di 48 ms. Calcolando complessivamente la durata del processamento si ottiene che l'elaborazione media complessiva è di circa 61 secondi, si ottiene quindi una riduzione significativa rispetto al tempo di 1 minuto e 27 prima individuato.

Si considerano, in relazione alle conclusioni della discussione teorica prima formalizzata, le implicazioni realizzative: in primis che l'imposizione del verso con cui eseguire la "minimizzazione" non determina particolari problemi implementativi, in secundis che l'ordinamento efficiente dei nomi del dizionario può essere trattato attraverso una tecnica di *machine learning*.

La tecnica di apprendimento che si realizza si basa sull'estrapolazione di informazioni utili nei documenti inviati dagli utenti. Si tiene traccia, infatti, della *frequenza* con cui un nome appare in diversi processamenti. Alla fine di ogni "minimizzazione", dopo aver concluso il salvataggio del nuovo file *OOXML*, il programma Java eseguirà l'update sul database MySQL delle frequenze dei nomi trovate, incrementandole. Nell'interrogazione al database per l'ottenimento dell'insieme dei nomi, di conseguenza, si richiederà l'ordinamento per frequenza delle tuple restituite. In questo modo, più volte un nome compare nei documenti, più incrementi riceverà il suo campo frequenza, più avrà la probabilità di comparire in cima al dizionario. Si progetta inoltre l'inserimento di un campo "ultimo utilizzo", associato ai nomi sul database, che permette di avere un secondo criterio di ordinamento utile. Tale campo viene

impiegato anche per evitare una crescita esagerata del valore della frequenza; periodicamente, con *crond* ad esempio, si decrementano i valori del campo "frequenza" dei nomi il cui "ultimo utilizzo" è troppo vecchio. Si osserva che i valori di configurazione per la gestione di queste elementi dipendono dal carico medio del sistema, noto solo dopo la progettazione, andranno valutati con precisione in fasi future. Ovviamente l'aggiornamento del campo "ultimo utilizzo" sarà contestuale all'aggiornamento della *frequenza*. Si osserva, per inciso, che l'impiego di un dizionario di cognomi sarebbe stato problematico per l'aggiornamento delle frequenze. Per via del principio *Privacy by Design*, infatti, aggiornando la frequenza per la entry del cognome di una persona, indirettamente si sarebbe tenuto traccia di un dato sensibile, specialmente nel caso in cui il cognome risulti "raro". Una valutazione rilevante da fare è sulle implicazioni relative agli accessi concorrenti al dizionario, dovute all'introduzione dell'algoritmo appena spiegato, che esegue scrittura di "frequenza" e *ultimo utilizzo* sul database. Si osserva che non sono problematiche le "letture sporche", poiché, se si verificano, al più risulta alterato l'ordine di qualche nome; inoltre nemmeno un "lost update" risulta eccessivamente problematico, poiché l'ultimo utilizzo risulta sempre aggiornato e la perdita di un incremento delle frequenza di un nome non rappresenta in linea di massima un problema. Questi scenari presentati vanno valutati complessivamente, però, una volta che si realizza il reale carico del sistema: perdere molti update risulta critico per l'ottimizzazione del processamento. Conviene in linea precauzionale prevedere una semantica transazionale per l'interazione tra modulo Java e database. Per inciso, eseguire l'update finale con una transazione non influenza il tempo di attesa dell'utente, poiché il documento è stato già elaborato per intero.

Si fa infine un'ultima osservazione che consente di abbattere i tempi di attesa enormemente, a costo di perdere dell'accuratezza nel processamento automatico: ridurre il numero di nomi del dizionario processato, elaborando solo quelli in cima. Si osserva che questa soluzione è ancor più efficace se i nomi contenuti nel documento sono pochi. In questo caso, infatti, il processamento dei nomi durante la fase di "transitorio iniziale" richiede lo stesso tempo del processamento di nomi "a regime", ossia dopo l'avvenuta riduzione del numero di sotto-stringhe da elaborare. Nei documenti "ricchi" di nominativi, come già spiegato, ciò non avviene. Per concludere, si usano i risultati dei test sperimentali precedentemente misurati per stimare il tempo di processamento complessivo, supponendo di:

- applicare la tecnica di pre-filtraggio
- usare i primi 1.500 nomi di un dizionario ordinato per frequenza

Caso A:

- documento contenente 10 nominativi ed un totale di 5.000 parole
- processamento medio della *regex* di pre-filtraggio di circa 36 ms
- individuazione di 31 sotto-stringhe totali
- durata elaborazione in media per nome di circa 2 ms

Durata esecuzione senza troncamento: 29 secondi circa

Durata esecuzione con troncamento: 3 secondi circa

Caso B:

- documento contenente 100 nominativi ed un totale di 5.000 parole
- processamento medio della *regex* di pre-filtraggio di circa 48 ms
- individuazione di 200 sotto-stringhe totali
- durata elaborazione in media per nome di circa 8 ms per prime 50 iterazioni
- durata elaborazione in media per nome di circa 5 ms per successive

Durata esecuzione con solo pre-filtraggio: 1 minuto e 27 secondi

Durata esecuzione senza troncamento: 61 secondi circa

Durata esecuzione con troncamento: 7.7 secondi circa

I tempi a cui si giunge risultano soddisfacenti in entrambi i casi. Si prevede di lasciare all'utente la possibilità di scegliere di ricevere un servizio o più veloce o più accurato, in base alle sue esigenze.

4.3 Tecnologie complementari

La redazione di questa tesi è avvenuta su Mediawiki, la più importante fra le piattaforme wiki essendo il motore di Wikipedia.

Le piattaforme wiki sono riconosciute come elemento caratterizzante dell'evoluzione Internet al Web 2.0, cioè al web nel quale gli utenti-navigatori, fino ad allora "*consumatori*" ("*consumers*") di contenuti si sono trasformati essi stessi in "*produttori*" ("*producers*"), e cioè in "*prosumers*".

Un *wiki* è fondamentalmente un sito web che contiene informazioni e che consente agli utenti di editare facilmente il suo contenuto.

Nel redigere una pagina wiki si utilizza ciò che è chiamato *wikitext* per definire capitoli, paragrafi, hyperlink, elementi di formattazione della pagina, etc.; sebbene il *wikitext* risulti non difficile da apprendere, quasi ogni piattaforma wiki è corredata da editor di tipo visuale, che non richiede alcuna conoscenza della sintassi del *wikitext*; tuttavia è esperienza comune che utilizzare direttamente il *wikitext* induce a concentrarsi maggiormente sui contenuti e non sulla formattazione. Il linguaggio *wikitext* è strettamente correlato con il linguaggio HTML, infatti nella scrittura è possibile utilizzare tag come h1, span, div e così via, oltre che la sintassi esclusiva di *wikitext*; simmetricamente una pagina prodotta da media wiki è costituita da HTML ben formato e direttamente importabile in ogni word processor.

I wiki presentano una funzionalità di grande utilità nella redazione di documenti articolati: la tracciatura delle successive revisioni ("*Cronologia*"). La possibilità di ripercorrere l'evoluzione del testo è davvero d'aiuto quando si fissano idee originali in uno scritto, cercando di definirle, precisarle, chiarirle, come è avvenuto in effetti, anche nella redazione di questa tesi. Per curiosità, si segnala che, pur avendo redatto la maggior parte del testo off-line, la pagina wiki di questa tesi conta oltre 50 revisioni.

Per inciso, si osserva che è stata molto utile la funzionalità di "ricerca nel codice" offerta dalla piattaforma di revisione online GitHub nella corretta comprensione della libreria Docx4j, il cui codice sorgente è ivi rilasciato.

Si conclude infine dicendo che per ottimizzare le procedure di debug del codice sono stati realizzati utili tool a riga di comando per Linux per la rapida estrazione-modifica-ricompattazione di documenti OOXML. Si propone in appendice un comando bash per questi scopi.

4.4 *Sviluppi futuri*

In questa sezione si fa cenno, senza poterle approfondire, ad alcune interessanti questioni emerse nello svolgimento della tesi.

4.4.1 Accesso al servizio web

In fase iniziale, il servizio web potrà essere reso accessibile in forma completamente aperta, senza richiedere cioè la registrazione dell'utente o l'autenticazione dell'utente già registrato. In questo modo, in effetti, si premia la facilità d'uso, ma si incorre nella nota problematica di un uso malevolo, costituito da accessi a raffica, finalizzati a saturare il server e generare una condizione di *DoS – Denial of Service*. Diverse sono le tecniche che possono essere adottate per contrastare questo tipo di accessi malevoli, come richiedere di superare "*captcha*" e/o introdurre ritardi crescenti ed eventualmente blocchi in risposta ad accessi prevenienti con alta frequenza dallo stesso indirizzo IP. È possibile anche pensare ad un accesso previa registrazione ed autenticazione dell'utente, penalizzando l'immediatezza di utilizzo del servizio, ma eventualmente ottenendo l'indirizzo email degli utenti che acconsentono a lasciarlo per successive finalità marketing. È possibile, infine, un utilizzo misto, contando in un cookie il numero di accessi eseguiti e richiedendo all'utente di registrarsi per continuare ad utilizzare il servizio, dopo qualche accesso.

4.4.2 Ampliamento dinamico del dizionario

Nel restituire il documento elaborato all'utente, gli si dà la possibilità di richiedere una nuova elaborazione, dopo aver indicato i nominativi che fossero sfuggiti; pare inesauribile, infatti, la "fantasia" dei genitori nel dare nome ai propri figlioli! I nominativi indicati dall'utente sono sfuggiti al servizio perché non presenti nel dizionario: facilmente viene in mente l'idea di arricchire il dizionario con i nominativi indicati dall'utente. Va però considerato il caso che l'utente in malafede indichi nominativi fasulli al fine di inquinare il dizionario. Il caso malevolo può essere affrontato attraverso una strategia che preveda non direttamente l'inserimento del nuovo nominativo nel dizionario, ma invece l'utilizzo di un concetto di "candidatura": il nuovo nominativo viene registrato, ma si attende di avere un certo numero di utenti che lo propongono prima di approvarne l'inserimento nel dizionario. Può anche essere utilizzato un concetto di "attendibilità" della candidatura di un nuovo nominativo, verificando ad esempio l'utente che lo propone scarichi effettivamente il file rielaborato. Nel caso l'accesso avvenga con autenticazione, e cioè identificando gli utenti, si apre la possibilità di individuare e bannare gli utenti malevoli.

4.4.3 *Natura del documento e ricorrenza del nominativo*

Si coglie facilmente la differenza di formato fra un documento in forma di elenco (es. una graduatoria) da un documento in forma di relazione (es. una sentenza giudiziaria). Differenze di questo tipo potrebbero essere utilizzate per escludere o ammettere la ripetizione dei nominativi. Per meglio dire: in un elenco la ripetizione di un nominativo è di fatto da escludersi o va trattata come omonimia. In una relazione, l'individuazione di un nominativo può essere utilizzata per cercarlo direttamente in altri punti del documento stesso.

4.4.4 *Altri dati personali*

Altri dati personali sono trattabili con le stesse tecniche analizzate in questa tesi: date e luoghi di nascita, codici fiscali, indirizzi, email, numeri di telefono, sesso etc. In qualche caso, come per i codici fiscali, l'individuazione del pattern da trattare è persino più semplice. Diversi studi [46] hanno dimostrato che, utilizzando set di dati personali parziali, è possibile la re-identificazione dei soggetti, pur in documenti pseudonimizzati. È questo un altro motivo per estendere i trattamenti descritti anche agli altri dati personali richiamati.

4.4.5 *Altri alfabeti*

È possibile estendere il sotto-insieme di caratteri Unicode utilizzati nelle *regex* per elaborare documenti scritti in altri alfabeti non latini.

Conclusioni

Questa tesi è partita da un problema basilare, quasi scolastico, dell'informatica: la ricerca di un testo in un documento, al fine della sua cancellazione o modifica.

Il problema si è subito discostato dalla sua formulazione basilare, principalmente perché, nei casi d'uso, il documento da ricercare non è un semplice testo ("*plain text*") ma invece è il prodotto di un *word-processor*: quindi è costituito da una struttura XML più o meno complessa, rappresentante formattazioni, riferimenti interni o esterni, note, etc. La ricerca deve avvenire nei soli nodi di puro testo, individuando ed escludendo dall'analisi lessicale gli elementi testuali di mark-up che non costituiscono contenuto informativo. Nell'approfondire le strutture ed i concetti XML ho potuto notare quanto essi siano ricorrenti in molti ambiti dell'informatica.

Nell'analisi delle specifiche, un'altra complicazione si è presto aggiunta: l'usabilità del servizio cresce drasticamente se si evita di chiedere all'utente, come si era pensato in un primo tempo di fare, quali siano le stringhe (i nominativi) da ricercare e trattare; è nata allora l'idea di reperire queste stringhe in un dizionario di nomi ed in un dizionario di cognomi, considerando anche le permutazioni dei lemmi reperiti. Ho così dovuto approfondire alcuni problemi tipici dei dizionari, come il loro ampliamento automatico con tecniche oggi comprese nel *machine-learning*. Non ho potuto "resistere" alla tentazione di *formalizzare la matematica* in base alla quale ho costruito le strategie di ricerca.

Spunto per la tesi è stata la recente legislazione in materia di dati personali, il GDPR. Esaminandone gli articoli pertinenti, ho maturato una riflessione generale: sempre più il software dovrà essere progettato e realizzato anche alla luce di altre discipline, non solo di quelle informatiche, come le discipline giuridiche ed il diritto.

Durante la fase iniziale della collaborazione con l'azienda mi sono dedicato alla messa a punto delle specifiche; ciò mi ha permesso di comprendere quanto questa fase sia importante e come completezza e precisione delle specifiche siano alla base di un progetto efficace.

Molto importanti sono state la analisi relative al tipo e formato dei documenti da assumere come input ed alla presentazione ottimale del servizio rispetto all'usabilità.

Centrale nel lavoro ed avvincente è stata la definizione della strategia per il riconoscimento dei lessemi attraverso la costruzione dinamica di *regex* idonee. In questa fase del lavoro, anche per passione personale, ho approfondito le questioni del riconoscimento "di testi nei testi" che legano, fin dalle origini, l'informatica e la linguistica.

La fase di definizione dell'architettura del servizio mi ha permesso di ripercorrere molte le materie studiate nei tre anni del Corso di Ingegneria Informatica, affrontando argomenti come il Single Responsibility Principle, il Dependency Inversion Principle, lo "stile" architetturale REST. Molto istruttiva è stata anche la riflessione sulla scomposizione del servizio in moduli software.

Come sempre accade nell'affrontare un progetto nuovo, sono nate molte nuove idee; ho così immaginato numerosi sviluppi futuri, sia a perfezionamento funzionale dell'accesso web al servizio, sia ad estensione delle specifiche per coprire casi applicativi contigui (es. quando il dato che si vuole trattare è un codice fiscale).

Bibliografia

- [1] Regolamento europeo in materia di protezione dei dati personali – pagina informativa. Url: <https://www.gpdp.it/web/guest/regolamentoue>
- [2] Regolamento europeo in materia di protezione dei dati personali . Url: https://eur-lex.europa.eu/legal-content/IT/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.ITA&toc=OJ:L:2016:119:TOC
- [3] Wikipedia: Ambiguity, Linguistic forms. Url: https://en.wikipedia.org/wiki/Ambiguity#Linguistic_forms
- [4] Steven L. Small; Garrison W Cottrell; Michael K Tanenhaus (22 October 2013). Lexical Ambiguity Resolution: Perspective from Psycholinguistics, Neuropsychology and Artificial Intelligence. Elsevier Science. ISBN 978-0-08-051013-2.
- [5] Treccani: Disambiguazione. Url: <http://www.treccani.it/vocabolario/disambiguazione/>
- [6] WordNet. Url: <https://wordnet.princeton.edu/>
- [7] R. Navigli, S. P. Ponzetto. BabelNet: Building a Very Large Multilingual Semantic Network. Proc. of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010), Uppsala, Sweden, July 11-16, 2010, pp. 216-225. <http://aclweb.org/anthology/P/P10/P10-1023.pdf>
- [8] Roventini A., Alonge A., Calzolari N., Magnini B., Bertagna F. (2000), "ItalWordNet: a Large Semantic Database for Italian", Proc. of the 2nd International Conference on Language Resources and Evaluation (LREC 2000), Athens, Greece, 2000, pp. 783-790.
- [9] E. Pianta, L. Bentivogli, C. Girardi. MultiWordNet: developing an aligned multilingual database, Proc. of the First International Conference on Global WordNet, Mysore, India, January 21-25, 2002. <http://multiwordnet.fbk.eu/paper/MWN-India-published.pdf>
- [10] R. Navigli, K. Litkowski, O. Hargraves. 2007. SemEval-2007 Task 07: Coarse-Grained English All-Words Task. Proc. of Semeval-2007 Workshop (SemEval), in the 45th Annual Meeting of the Association for Computational Linguistics (ACL 2007), Prague, Czech Republic, pp. 30–35 <http://www.aclweb.org/anthology/S/S07/S07-1006.pdf>

- [11] Wikipedia: Latin script in Unicode. Url: https://en.wikipedia.org/wiki/Latin_script_in_Unicode
- [12] Wikipedia: ISO/IEC 8859-1. Url: https://it.wikipedia.org/wiki/ISO/IEC_8859-1
- [13] Treccani: Uso delle maiuscole. Url: http://www.treccani.it/enciclopedia/uso-delle-maiuscole_%28La-grammatica-italiana%29/
- [14] DataWorld: Dataset nomi. Url: <https://data.world/axtscz/italian-first-names>
- [15] Simone Bertelegni, 2006: L'Italia è il regno dei cognomi. Url: https://www.corriere.it/Primo_Piano/Cronache/2006/09_Settembre/15/cognomi.shtml
- [16] Wikipedia: Rich Text Format. Url: https://en.wikipedia.org/wiki/Rich_Text_Format
- [17] Wikipedia: Office Open XML file formats. Url: https://en.wikipedia.org/wiki/Office_Open_XML_file_formats
- [18] Wikipedia: Office Open XML. Url: https://en.wikipedia.org/wiki/Office_Open_XML
- [19] Wikipedia: Standardization of Office Open XML. Url: https://en.wikipedia.org/wiki/Standardization_of_Office_Open_XML
- [20] Wikipedia: OpenDocument. Url: <https://en.wikipedia.org/wiki/OpenDocument>
- [21] Wikipedia: OpenDocument Standardization. Url: https://en.wikipedia.org/wiki/OpenDocument_standardization
- [22] Wikipedia: Comparison of Office Open XML software. Url: https://en.wikipedia.org/wiki/Comparison_of_Office_Open_XML_software
- [23] Wikipedia: Comparison of OpenDocument software. Url: https://en.wikipedia.org/wiki/Comparison_of_OpenDocument_software
- [24] Wikipedia: Comparison of word processors. Url: https://en.wikipedia.org/wiki/Comparison_of_word_processors
- [25] Wikipedia: Pages (word processor). Url: [https://en.wikipedia.org/wiki/Pages_\(word_processor\)](https://en.wikipedia.org/wiki/Pages_(word_processor))
- [26] Apple: Convert Pages documents to PDF, Microsoft Word, and more. Url: <https://support.apple.com/en-us/HT202227>

- [27] Wikipedia: LAMP (software bundle). Url: [https://en.wikipedia.org/wiki/LAMP_\(software_bundle\)](https://en.wikipedia.org/wiki/LAMP_(software_bundle))
- [28] WhatIs: LAMP (Linux, Apache, MySQL, PHP). Url: <https://whatis.techtarget.com/definition/LAMP-Linux-Apache-MySQL-PHP>
- [29] IBM: LAMP stack. Url: <https://www.ibm.com/cloud/learn/lamp-stack-explained>
- [30] Matthias Gelbmann, October 2013: Debian/Ubuntu extend the dominance in the Linux web server market at the expense of Red Hat/CentOS. Url: https://w3techs.com/blog/entry/debian_ubuntu_extend_the_dominance_in_the_linux_web_server_market_at_the_expense_of_red_hat_centos
- [31] Netcraft, 2014: June 2014 Web Server Survey. Url: <https://news.netcraft.com/archives/2014/06/06/june-2014-web-server-survey.html>
- [32] Wikipedia: Single Responsibility Principle. Url: https://en.wikipedia.org/wiki/Single_responsibility_principle
- [33] Wikipedia: Dependency Inversion Principle. Url: https://en.wikipedia.org/wiki/Dependency_inversion_principle
- [34] Roy Thomas Fielding, 2000: Architectural Styles and the Design of Network-based Software Architectures. Url: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [35] Docx4j. Url: <https://www.docx4java.org/trac/docx4j>
- [36] Github: Docx4j. Url: <https://github.com/plutext/docx4j>
- [37] Oracle: Java String. Url: <https://docs.oracle.com/javase/9/docs/api/java/lang/String.html>
- [38] Github: session id php. Url: <https://github.com/php/php-src/blob/master/ext/session/session.c>
- [39] Standard ECMA-376. Url: <https://www.ecma-international.org/publications/standards/Ecma-376.htm>
- [40] Standard ISO/IEC 29500-1:2008. Url: <https://www.iso.org/standard/51463.html>
- [41] Standard ISO/IEC 29500-1:2016. Url: <https://www.iso.org/standard/71691.html>

- [42] Wikipedia: Open Packaging Conventions. Url:
https://en.wikipedia.org/wiki/Open_Packaging_Conventions
- [43] Istat: Quanti bambini si chiamano...?. Url:
<https://www.istat.it/it/dati-analisi-e-prodotti/contenuti-interattivi/contanomi>
- [44] Github: research of "getJAXBNodesViaXPath". Url:
<https://github.com/plutext/docx4j/search?q=getJAXBNodesViaXPath+in%3Afile&type=Code>
- [45] Xpath. Url: <https://www.w3.org/TR/xpath-30/>
- [46] Luc Rocher; Julien M. Hendrickx; Yves-Alexandre de Montjoye: Estimating the success of re-identifications in incomplete datasets using generative models. Url:
<https://imperialcollegelondon.app.box.com/s/lqqcugie51pllz26uixjvx0uio8qxgo5/file/493461282808>

Indice delle figure

Figura 1. Diffusione mondiale dell'alfabeto latino (https://en.wikipedia.org/wiki/Latin_script)

Figura 2. Diagramma di Venn (1)

Figura 3. Diagramma di Venn (2)

Figura 4. Diagramma di Venn (3)

Figura 5. UML - Diagramma di sequenza

Figura 6. Calcolo di C_1

Figura 7. Calcolo di C_2

Figura 8. Calcolo di C_3

Appendice

Vengono presentati in questa sezione i seguenti moduli java:

- App.java
- Elaborator.java
- Persona.java
- Testing.java

Ed il seguente script bash:

- docxedit.sh

App.java

```
Appri App.java Salva
1 //..librerie
2 public class App
3 {
4     //debug=true attiva alcune stampe utili
5     public static boolean debug = false;
6     //log4jPrints=false disattiva stampe di routine usate da docx4j
7     private static final boolean log4jPrints = false;
8     private static String inputFile;
9     private static String outputFile;
10    public static boolean applicaDizionario = false;
11    //elenco persone da pseudonimizzare
12    private static List<Persona> persone = new ArrayList<Persona>();
13
14
15    //java docxElaborator -i input -o output "Lorenzo:Mario;Amorosa" "Francesco:Nicola;De Rosa"
16    public static void main(String[] args) {
17        /*inizializzazione log4j
18         * la variabile "log4jPrints" a false disabilita stampe verbose di logging di routine di Docx4j
19         */
20        BasicConfigurator.configure();
21        /**
22         *
23         * disabilitazione del seguente warning dovuto a Docx4j che inquina stdout:
24         *
25         * WARNING: An illegal reflective access operation has occurred [...]
26         *
27         */
28        disableWarning();
29        if(!log4jPrints) {
30            @SuppressWarnings("unchecked")
31            List<Logger> loggers = Collections.<Logger>list(LogManager.getCurrentLoggers());
32            loggers.add(LogManager.getRootLogger());
33            for (Logger logger : loggers) {
34                logger.setLevel(Level.OFF);
35            }
36        }
37        parametersCheck(args);
38    }
39}
```

```
Appri App.java Salva
39
40    File doc = new File(inputFile);
41    WordprocessingMLPackage wordMLPackage = null;
42    try {
43        //carico il documento .docx
44        wordMLPackage = WordprocessingMLPackage.load(doc);
45    } catch (Docx4JException e) {
46        System.out.println("Eccezione in WordprocessingMLPackage.load: caricamento fallito di " + inputFile);
47        e.printStackTrace();
48        System.exit(2);
49    }
50    //ottengo document.xml che contiene i dati di interesse
51    MainDocumentPart mainDocumentPart = wordMLPackage.getMainDocumentPart();
52    String textNodesXPath = "//w:t";
53    List<Object> textNodes = null;
54    try {
55        //ottengo tutti i nodi contenenti il testo visibile nel documento
56        textNodes = mainDocumentPart.getJAXBNodesViaXPath(textNodesXPath, true);
57    } catch (XPathBinderAssociationIsPartialException | JAXBException e) {
58        System.out.println("Eccezione in mainDocumentPart.getJAXBNodesViaXPath, lettura fallita da " + inputFile);
59        e.printStackTrace();
60        System.exit(3);
61    }
62
63    //elaboro i singoli nodi contenuti in document.xml
64    Elaborator elab = new Elaborator(textNodes, persone, applicaDizionario);
65    elab.work();
66
67    //salvataggio nuovo file docx modificato
68    File exportFile = new File(outputFile);
69    try {
70        wordMLPackage.save(exportFile);
71    } catch (Docx4JException e) {
72        System.out.println("Eccezione in wordMLPackage.save, salvataggio fallito di " + outputFile);
73        e.printStackTrace();
74        System.exit(4);
75    }
76    //..aggiornamento del campo frequenza e ultimo aggiornamento per i nomi individuati
```



```
Apri  App.java  Salva  ~/Appendice
155     }
156
157     for (String argument : remainder)
158     {
159         if(debug)
160             System.out.print("\n" + argument + "\n ");
161         if(argument.matches(pattern)) {
162             String cognome;
163             String nomiString;
164             List<String> nomi = new ArrayList<String>();
165
166             nomiString = argument.split(Pattern.quote(";"))[0];
167             cognome = argument.split(Pattern.quote(";"))[1];
168             nomi = List.of(nomiString.split(Pattern.quote(":")));
169             if(nomi.size() > 10) {
170                 //possibile chiusura senza pe: salto la persona in questione e continuo
171                 ParseException pe = new ParseException("Attenzione: sono inseribili massimo 10 nomi per persona");
172                 throw pe;
173             }
174
175             //aggiungo la persona
176             if(debug) {
177                 System.out.println("\nLegenda caratteri espressi in range (formato Unicode):");
178                 System.out.println("\u00C0 == À");
179                 System.out.println("\u00D6 == Ö");
180                 System.out.println("\u00D8 == Ø");
181                 System.out.println("\u00F6 == ö");
182                 System.out.println("\u00F8 == ø");
183                 System.out.println("\u00FF == ÿ\n");
184             }
185             persone.add(new Persona(cognome, nomi, persone.size() + 1));
186         }
187         else {
188             ParseException pe = new ParseException(argument + ": input non ben formato");
189             throw pe;
190         }
191     }
192 }
```

Java Larg. tab.: 8 Rg 103, Col 9 INS

Elaborator.java

```
Apri  Elaborator.java  Salva
1 //..librerie
2
3 public class Elaborator {
4     private List<Object> textNodes;
5     private List<Persona> persone;
6     private StringBuilder plainText;
7     private List<Integer> from, to;
8     boolean applicaDizionario;
9
10    public Elaborator(List<Object> textNodes, List<Persona> persone, boolean applicaDizionario) {
11        this.textNodes = textNodes;
12        this.persone = persone;
13        this.applicaDizionario = applicaDizionario;
14        plainText = new StringBuilder();
15        from = new ArrayList<>();
16        to = new ArrayList<>();
17        String toAppend;
18        Object obj, oldParent = null;
19
20        if(App.debug)
21            System.out.println("\nStampa contenuto nodi e loro indici");
22        /*logica per estrapolazione contenuto file ooxml*/
23        from.add(0, 0);
24        for (int i = 0; i < textNodes.size(); i++) {
25            obj = textNodes.get(i);
26            @SuppressWarnings("rawtypes")
27            Text text = (Text) ((javax.xml.bind.JAXBElement) obj).getValue();
28            //Verifico se aggiungere \n (omesso in quanto segna la fine del paragrafo e docx lo traduce con </w:p>)
29            //per verificare controllo che il testo corrente ed il testo precedente appartengono allo stesso paragrafo
30            if(App.debug && i > 0)
31                System.out.println(((org.docx4j.wml.R)text.getParent()).getParent().hashCode() + " == " + oldParent.hashCode());
32
33            if(i == 0 || ((org.docx4j.wml.R)text.getParent()).getParent().hashCode() == oldParent.hashCode())
34                toAppend = text.getValue();
35            else
36                toAppend = '\n' + text.getValue();
37            //..ulteriori tediose elaborazioni della stringa cosi' come spiegato nel capitolo sul formato OOXML
38        }
39    }
40
41    Java  Larg. tab.: 8  Rg 36, Col 124  INS
```

```
Apri  Elaborator.java  Salva
44
45    /*variabile che contiene l'intero testo "appiattito", per delimitare blocchi di testo appartenenti
46    a nodi "non semanticamente non connessi" basta usare un "carattere non alfabetico" */
47    plainText.append(toAppend);
48    oldParent = ((org.docx4j.wml.R)text.getParent()).getParent();
49    if(i != 0)
50        from.add(i, to.get(i - 1));
51    to.add(i, from.get(i) + toAppend.length());
52    if(App.debug)
53        System.out.println("Nodo " + i + ": |" + toAppend + "| ; inizio: " + from.get(i) + ", fine: " + to.get(i));
54    if(App.debug)
55        System.out.println();
56    }
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
255
```



```

1 //..librerie
2 public class Persona {
3     private String cognome;
4     private List<String> nomi = new ArrayList<String>();
5     private int id;
6     private String regex;
7
8     //strutture usate per la permutazione dei nomi
9     private List<String> comboIndexDuplicati;
10    private List<String> comboIndex;
11
12    public Persona(String cognome, List<String> nomi, int id) {
13        super();
14        if(Character.isUpperCase(cognome.charAt(0))
15            this.cognome = cognome.charAt(0) + "({?)" + cognome.substring(1, n.length) + ")";
16        else
17            this.cognome = "{?)" + cognome;
18        for(String n : nomi){
19            this.nomi.add(n.charAt(0) + "({?)" + n.substring(1, n.length) + ")");
20        }
21        this.id = id;
22        this.comboIndexDuplicati = new ArrayList<>();
23        this.comboIndex = new ArrayList<>();
24        calcolaRegex();
25    }
26
27    public Persona(String regex, int id) {
28        super();
29        this.id = id;
30        //regex individuata da dizionario
31        this.regex = regex;
32    }
33
34    @Override
35    public String toString() {
36        return "Persona [cognome=" + cognome + ", nomi=" + nomi + ", id=" + id + "]";
37    }
38 }

```

```

39  /*
40  * Determino la regex che identifica la persona nel documento se non fornita in input:
41  * - il cognome compare sempre, o all'inizio o in fondo
42  * - i nomi posso comparire opzionalmente in qualunque ordine
43  * - i singoli termini di un nominativo sono divisi tra loro e da altri elementi del testo da uno o piu:
44  *   \r \n \t e spazi (regex \s)
45  * - ogni nominativo e' necessariamente preceduto e seguito da un carattere non alfabetico e non accentato
46  *
47  * Esempio: Lorenzo,Mario;Amorosa
48  * - Lorenzo Mario Amorosa
49  * - Mario Lorenzo Amorosa
50  * - Amorosa Lorenzo Mario
51  * - Amorosa Mario Lorenzo
52  * - Lorenzo Amorosa
53  * - Mario Amorosa
54  * - Amorosa Lorenzo
55  * - Amorosa Mario
56  *
57  */
58
59 private void calcolaRegex() {
60     calcolaComboIndici();
61     //(?! regex case insensitive, (?<=REGEX) positive look behind, (?=REGEX) positive look ahead
62     String SEP = "\\s+";
63     String PRE = "(?<=[^A-Za-zÀ-Öö-ßÜüçÊëËîïóôùîíçğşwwwWWWYyööß|\\^])";
64     String POST = "(?=^[^A-Za-zÀ-Öö-ßÜüçÊëËîïóôùîíçğşwwwWWWYyööß|\\$]|)";
65     StringBuilder pattern = new StringBuilder("(?!(");
66     for(String sequenza : comboIndex) {
67         //antepongo cognome
68         pattern.append(PRE + cognome + SEP);
69         for(int i = 0; i < sequenza.length(); i++) {
70             if(i != (sequenza.length() - 1))
71                 pattern.append(nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + SEP);
72             else
73                 pattern.append(nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + POST);
74         }
75         //postpongo cognome

```

```

67      //antepongo cognome
68      pattern.append(PRE + cognome + SEP);
69      for(int i = 0; i < sequenza.length(); i++) {
70          if(i != (sequenza.length() - 1))
71              pattern.append(nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + SEP);
72          else
73              pattern.append(nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + POST);
74      }
75      //postpongo cognome
76      for(int i = 0; i < sequenza.length(); i++) {
77          if(i != 0)
78              pattern.append(nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + SEP);
79          else
80              pattern.append(PRE + nomi.get(Integer.parseInt(String.valueOf(sequenza.charAt(i)))) + SEP);
81      }
82      pattern.append(cognome + POST);
83  }
84  //gestisco il carattere pipe in fondo alla regex
85  pattern.substring(0, POST.length() - 1);
86  if(App.debug) {
87      System.out.println(toString());
88      System.out.println("Indici per permutazioni nomi: " + comboIndex);
89      System.out.println("Regex: " + StringEscapeUtils.escapeJava(pattern.toString()));
90      System.out.println();
91  }
92  //inizializzo la regex della persona
93  regex = pattern.toString();
94  }
95
96
97
98
99
100
101
102
103
104

```

```

215  /*
216  * INPUT [0,1] ; OUPUT [0, 1, 10, 01]
217  * INPUT [0,1,2] ; OUPUT [0, 1, 2, 10, 01, 21, 20, 12, 02, 012, 021, 120, 102, 210, 201]
218  */
219  private void calcolaComboIndici() {
220      StringBuilder s = new StringBuilder();
221      for(int i = 0; i < nomi.size(); i++)
222          s.append(i);
223      for (int i = 0; i < s.toString().length(); i++) {
224          comboIndexDuplicati.add(String.valueOf(s.toString().charAt(i)));
225          recurse(s.toString(), "" + s.toString().charAt(i));
226      }
227      //Elimino le stringhe che contengono caratteri uguali (ES. 001, 221, ect.)
228      for(String ind : comboIndexDuplicati) {
229          if(!hasCharsDuplicated(ind))
230              comboIndex.add(ind);
231      }
232      //Ordino le stringhe per lunghezza: antepongo quelle con piu' nomi, altrimenti sbaglio pseudonimizzazione
233      Collections.sort(comboIndex, new LengthComparator());
234  }
235
236  private boolean hasCharsDuplicated(String s) {
237      for(int i = 0; i < s.length(); i++)
238          for(int j = i; j < s.length(); j++) {
239              if(i != j && s.charAt(i) == s.charAt(j))
240                  return true;
241          }
242      return false; }
243
244
245  private void recurse(String inp, String s) {
246      if (s.length() == inp.length())
247          return;
248      for (int i = 0; i < inp.length(); i++) {
249          comboIndexDuplicati.add(s + inp.charAt(i));
250          recurse(inp, s + inp.charAt(i));
251      }
252      return; }

```

Testing.java

```
Testing.java
~/.Appendice
Salva
Persona.java x Testing.java x
1 private static void tempoRegex() {
2     final String regex = //..regex
3     final String string = //..testo
4     final Pattern pattern = Pattern.compile(regex, Pattern.MULTILINE);
5     final Matcher matcher = pattern.matcher(string);
6     long start, total;
7     start = System.currentTimeMillis();
8     while (matcher.find()) {
9         System.out.println("Full match: " + matcher.group(0));
10        for (int i = 1; i <= matcher.groupCount(); i++) {
11            System.out.println("Group " + i + ": " + matcher.group(i));
12        }
13    }
14    total = System.currentTimeMillis() - start;
15    System.out.println("Total: " + total + " ms");
16    System.exit(0);
17 }
18
19 /* routine per debug, manipolazione indici, pattern complesso */
20 List<String> nomi = new ArrayList<String>();
21 nomi.add("Lorenzo");
22 nomi.add("Mario");
23 Persona p = new Persona("Renzi", nomi, 1);
24 from = new ArrayList<>();
25 to = new ArrayList<>();
26 from.add(0, 0); from.add(1, 11); from.add(2, 21);
27 to.add(0, 11); to.add(1, 21); to.add(2, 31);
28 plainText = "oo renzi Lorenzo Renzi!oooooooo";
29 pseudo = p.pseudonimizza(plainText, from, to);
30 System.out.println("PlainText: " + plainText + "\nRES: " + pseudo);
31 System.out.println("from, aspetto [0,11,12]: " + from);
32 System.out.println("to, aspetto [11,12,21]: " + to);
33 System.out.println("Nodo 0: |" + pseudo.substring(from.get(0), to.get(0)) + "|");
34 System.out.println("Nodo 1: |" + pseudo.substring(from.get(1), to.get(1)) + "|");
35 System.out.println("Nodo 2: |" + pseudo.substring(from.get(2), to.get(2)) + "|");
36
Java Larg. tab.: 8 Rg 1, Col 1 INS
```

```
Testing.java
~/.Appendice
Salva
Persona.java x Testing.java x
55
56 /* funzione utile per leggere db nomi formato JSON presente su Data World */
57 private static void leggiNomi() {
58     String maschiJSON, femmineJSON;
59     List<String> maschi = new ArrayList<>(), femmine = new ArrayList<>();
60
61     try {
62         BufferedReader bf = new BufferedReader(new FileReader("/home/lorenzo/Scaricati/ITGivenMale.json"));
63         maschiJSON = bf.readLine();
64         maschi = Arrays.asList(maschiJSON.split(Pattern.quote("\\", "gender":\\"M\\", "culture":\\"IT\\"},{\\"name\\":\\""}));
65         bf.close();
66     } catch (IOException e) {
67         e.printStackTrace();
68     }
69
70     try {
71         BufferedReader bf = new BufferedReader(new FileReader("/home/lorenzo/Scaricati/ITGivenFemale.json"));
72         femmineJSON = bf.readLine();
73         femmine = Arrays.asList(femmineJSON.split(Pattern.quote("\\", "gender":\\"F\\", "culture":\\"IT\\"},{\\"name\\":\\""}));
74         bf.close();
75     } catch (IOException e) {
76         e.printStackTrace();
77     }
78
79     for(String s : maschi)
80         System.out.print(s + ' ');
81     for(String s : femmine)
82         System.out.print(s + ' ');
83
84     System.exit(0);
85 }
86
87
88
89
90
Java Larg. tab.: 8 Rg 1, Col 1 INS
```

Docxedit.sh

```
1 # Funzione per aprire il file word/document.xml contenuto in un docx, modificarlo, poi aggiornare il docx e aprirlo con libreoffice
2 function docxedit(){
3     if [[ "$#" -ne 1 ]] ! -e "$1" ]] ! "$1" =~ docx$ ]]; then
4         if [[ "$#" -ne 1 ]]; then
5             echo "Passare un solo argomento in input"
6         fi
7         if ! [[ -e "$1" ]]; then
8             echo "Il file $1 non esiste"
9         fi
10        if ! [[ "$1" =~ docx$ ]]; then
11            echo "Il file $1 non ha estensione docx"
12        fi
13        echo "Usage: docxedit file.docx"
14    else
15        #variabile usata per tornare a fine script nella cartella di provenienza
16        curdir=$(pwd)
17        #DOCX restera' uguale all'argomento passato se si lancia docxedit nella cartella che contiene il documento docx
18        DOCX=$1
19        #P restera' uguale al path corrente se si lancia docxedit nella cartella che contiene il documento docx
20        P=$(pwd)
21        #il seguente blocco serve per trattare docx passati con path
22        if [[ $DOCX =~ \ / ]]; then
23            P=$(echo $DOCX | awk 'BEGIN{FS=OFS="/"}{NF--; print}')
24            DOCX=$(echo $DOCX | awk -F '/' '{print $NF}')
25            echo "P: $P"
26            echo "DOCX: $DOCX"
27            cd $P
28        fi
29        unzip "$DOCX" "word/document.xml" -d /tmp
30        cd /tmp
31        gedit "word/document.xml" && zip --update "$P/$DOCX" "word/document.xml"
32        # remove this line to just keep overwriting files in /tmp
33        rm -f "word/document.xml" # or remove -f if you want to confirm
34        rmdir "word"
35        cd "$curdir"
36        libreoffice -o "$P/$DOCX"
37    fi
38 }
```