Esame di Programmazione 2

Giacomo Bianchi, Fabio Caironi, Manuel Luigi Trezzi

1 Introduzione

Il progetto ha lo scopo di implementare diverse classi utili alla simulazione di un motore di ricerca. Il progetto si compone di 10 classi che verranno descritte e commentate nel corso della relazione. Rispetto alla consegna sono stati implementati ulteriori metodi e classi al fine di espandere il software. Inoltre talune segnature di metodi indicati sono state modificate per migliorarne la coerenza all'interno del nostro progetto.

2 Le classi e i loro metodi

2.1 URL

Le istanze della classe rappresentano un URL ovvero una sequenza di caratteri che identifica univocamente l'indirizzo di una risorsa in Internet. La classe URL consiste dei seguenti tre attributi:

- *String protocollo* - Rappresenta il protocollo utilizzato dall'URL;
- *String hostname* - Rappresenta l'hostname dell'URL;
- *ArrayList<String> path* - Rappresenta il path dell'URL, abbiamo scelto di utilizzare un ArrayList<String> e non una String per questo attributo in quanto pensiamo al *path* come ad una successione di stringhe che all'occorrenza possiamo accorciare o allungare.

Oltre a questi vi sono due attributi statici:

- *final static public ArrayList<String> protocolli* - Rappresenta l'insieme dei protocolli ammissibili dai nostri URL;
- *final static public ArrayList<String> domini* - Rappresenta la lista dei domini ammissibili per i vari *hostname*;

La motivazione, abbastanza ovvia, per la quale abbiamo scelto di rendere questi due attributi statici è che vogliamo che queste liste siano accessibili ed in comune ad ogni URL. Si è scelto inoltre di renderli *final* per evitare che possano essere modificati nel corso dell'esecuzione andando ad invalidare URL che erano stati in precedenza approvati.

Per la classe URL abbiamo deciso di implementare i seguenti due costruttori:

- *URL(String protocollo, String hostname, String path)* - Questo costruttore prende in ingresso i tre attributi di un URL e si accerta della loro validità. In particolare esso lancerà delle eccezioni nei seguenti casi:
  * *ProtocolloNonValidoException* - Se la stringa *protocollo* non rientra nei valori ammissibili dichiarati nell'attributo statico *protocolli*;

    * *HostnameNonValidoException* - Questa eccezione viene lanciata nei seguenti casi: *hostname* contenga al suo interno degli spazi, inizi con un carattere che non è nè un numero nè una lettera o contenga al suo interno un carattere '/' (indice di un path);
    * *DominioNonValidoException* - Se l'hostname non termina con uno dei domini ammissibili in *domini*;
    * *PathNonValidoException* - Se la stringa *path* non inizia col carattere '/'.

    Oltre a verificare la correttezza dei parametri ricevuti e lanciare eccezioni il costruttore converte i caratteri di *hostname* in minuscolo e nel caso che quest'ultimo inizi con "*www.*" rimuove questa dicitura (affinchè *www.google.it* e *google.it* abbiano lo stesso *hostname*).

- *URL(String url)* - A differenza del precedente questo costruttore riceve in ingresso una stringa e si occupa di decomporla nei tre attributi necessari. Il costruttore ricerca la presenza di una sottostringa "*://*" splittando la stringa di partenza col metodo *split* e suppone che la prima sottostringa ricavata sia il protocollo (controllandolo come nel costruttore precedente), in assenza di "*://*" verrà lanciata l'eccezione *UrlNonValidoException*. Dopodichè elimina gli eccessivi / iniziali (in modo che *http://google.it* sia uguale a *http:///google.it*) ed invoca *split('/')* per isolare *hostname* e *path* eseguendo controlli analoghi (e lanciandone relative eccezioni) a quelli del costruttore. precedente

Analizziamo ora i metodi di questa classe:

- *boolean stessoHost(URL u)* - Verifica che l'URL su cui è invocato il metodo e quello passato come argomento abbiano il medesimo *hostname* (si osservi che se non fosse stata fatta la scelta di rimuovere "*www.*" il metodo con *www.google.it* e *google.it* avrebbe restituito *false*);
- *String toString()* - Override del metodo *toString()*, stampa l'URL nel formato *protocollo + "://" + hostneam + path*;
- *boolean equals(Object obj)* - Override del metodo *equals(Object obj)*. Due istanze della classe URL sono uguali se hanno il medesimo *toString()*;
- *int hashCode()* - Override del metodo *hashCode()*: è implementata una condizione necessaria all'*equals*. Abbiamo scelto un'implementazione usuale;
- *String getProtocollo()*;
- *String getHostname()*;
- *ArrayList<String> getPath()*;

Gli ultimi tre dall'ovvio significato.

2.2 PaginaWeb

Questa classe rappresenta una pagina web attraverso il suo URL, il suo contenuto in forma di testo e una lista di altri URL a cui punta. Da cui in poi quando scriveremo che la pagina p punta all'URL u intenderemo che nella lista dei link è contenuto l'URL u. Questa classe è composta da tre attributi:

- *URL url* - L'URL della PaginaWeb;
- *String contenuto* - Il contenuto testuale della PaginaWeb;
- *ArrayList<URL> link* - Lista degli URL a cui la PaginaWeb punta.

Per questa classe abbiamo implementato i seguenti costrutti:

- *PaginaWeb(URL u, String c, ArrayList<URL> l)* - Riceve in ingresso i tre attributi di PaginaWeb ed inoltre verifica che nè u nè nessun URL in l assuma il valore *null*, nel caso ciò avvenga lancia l'eccezione *UrlNonValidoException*. Inoltre fa in modo che la lista dei link non contenga più volte lo stesse URL;
- *PaginaWeb(URL u, String testogrezzo)* - Riceve in ingresso l'URL della pagina (verificando che non sia *null* analogamente al precedente) ed una stringa *testogrezzo* che rappresenta il contenuto della PaginaWeb in formato HTML. Il costruttore deve quindi occuparsi di ripulire *testogrezzo* dai vari tag HTML ed estrarre gli URL a cui la PaginaWeb deve puntare. Per una miglior implementazione di questo costruttore si utilizza una libreria esterna, per maggiori dettagli si legga la sezione 4 di questa relazionel. Anche in questo costruttore si unicizza la lista dei link;
- *PaginaWeb(URL u)* - Costruisce una PaginaWeb avente come URL quello passato come argomento (sempre verificando che non sia *null*) con contenuto vuoto e che non punta a nessun altro URL.

La classe possiede diversi metodi:

- *boolean puntaA(URL u)* - Verifica che la pagina su cui è invocato il metodo punti all'URL passato come argomento;
- *boolean puntaAHost(URL u)* - Verifica che la pagina su cui è invocato il metodo punti ad un URL avente lo stesso *hostname* dell'URL passato come argomento. Viene richiamato il metodo *stessoHost* della classe URL;
- *contiene(String x)* - Verifica che il *contenuto* della PaginaWeb abbia come sottostringa la stringa passata come argomento. Per una migliore efficacia del metodo si è scelto di convertire tutti i caratteri maiuscoli delle stringhe in esame in minuscoli, cioè la ricerca non è case sensitive (in tal modo se il contenuto della pagina fosse "Ciao, come stai?" e cercassimo "ciao" il metodo restituisce *true*);

- *int contieneQuanti(String x)* - Conta quante volte la stringa x compaia all'interno del *contenuto* della PaginaWeb su cui è invocato il metodo dopo aver verificato, per mezzo del metodo precedente, che x compaia almeno una volta. Anche per questo metodo si è scelto di convertire tutti i caratteri in minuscolo;

- *static PaginaWeb leggi(Scanner s)* - Il metodo procede come segue: legge una stringa dallo scanner s mediante il metodo *nextLine()* e tramite il metodo *split* la divide ogni qual volta incontri un TAB, dopodichè:

  * Se non ci sono TAB restituisce il valore *null*;
  * Se ne trova uno divide la stringa in due parti ed assume che la prima parte sia la stringa che rappresenta l'URL della PaginaWeb e la seconda il *testogrezzo* (richiamando quindi gli opportuni costruttori di URL e PaginaWeb);
  * Se ne trovasse più di uno assume che la parte prima del primo TAB sia l'URL della pagina, tra il primo e il secondo TAB ci sia il contenuto della pagina e gli altri pezzi siano gli URL a cui punta la PaginaWeb che restituiremo come risultato del metodo.

  Questo metodo non gestisce l'eccezione *UrlSbagliatoException*, in cui si incorre se il formato degli URL in input non è corretto, ma la solleva. Ugualmente solleva l'eccezione *NoSuchElementException* quando non viene trovata una prossima riga da leggere (per esempio se si è giunti alla fine da un file).

- *String stringPreview(String s)* - Verifica la presenza della stringa s all'interno del contenuto della PaginaWeb su cui è invocato il metodo, dopodichè restituisce un'anteprima di s all'interno del contenuto della PaginaWeb (ad esempio se il contenuto fosse: "Oggi sono andato al mare, era proprio una bella giornata" e cercando "Mare" si otterrebbe: "...andato al mare, era prop..."). Nel caso s non sia presente restituisce la stringa vuota "", nel caso compaia più volte si considera solo la prima volta che appare;

- *URL getURL()*;

- *String getContenuto()*;

- *ArrayList<URL> getLink()*;

- *String toString()* - Override del metodo *toString()*, stampa in successione, *url*, *contenuto* e gli url delle pagine a cui punta racchiusi tra quadre;

- *boolean equals(Object obj)* - Override del metodo *equals(Object obj)*. Due istanze della classe PaginaWeb sono uguali se hanno il medesimo *url* secondo l'equals della classe URL;

2.3 MotoreDiRicerca

Questa classe possiede un unico attributo: *ArrayList<PaginaWeb>* *store* che rappresenta l'insieme delle PagineWeb "a disposizione" di un oggetto della classe MotoreDiRicerca.

L'unico costruttore a disposizione per questa classe è *MotoreDiRicerca()* il quale crea un nuovo *ArrayList<PaginaWeb>* (quindi per il momento vuoto) e lo assegna all'attributo *store*. Analizziamo ora i suoi metodi:

- *boolean aggiungiPagina(PaginaWeb p)* - Controlla se l'attributo *store* contiene p se non presente la aggiunge restituendo *true* altrimenti la aggiorna restituendo *false*, se fosse p fosse *null* restituisce false;
- *boolean aggiungiPagine(ArrayList<PaginaWeb> a)* - Opera come il metodo precedente ma con più pagine. Restituisce *true* se e solo se almeno una delle pagine in a non sono presenti in *store*;
- *int leggiPagine(Scanner s)* - Aggiunge pagine web a questo motore di ricerca sfruttando il metodo *PaginaWeb.leggi(s)*. Il metodo funziona iterativamente: legge dallo scanner s righe singole e termina alla prima occorrenza di una riga di formato inappropriato, oppure al primo sollevamento di *NoSuchElementException* (per esempio, alla fine di un file). Solleva *URLNonValidoException* se almeno un URL in input non è nel formato corretto.
- *PaginaWeb cercaPag(URL u)* - Controlla che l'attributo *store* contenga una pagina avente come URL quello passato come argomento e nel caso la restituisce. Se nessuna pagina viene trovata il metodo restituisce *null*;
- *boolean presente(URL u)* - Similmente al metodo precedente restituisce *true* se viene trovata una pagina con tale URL;
- *int indegree(URL u)* - Conta quante pagine all'interno dell'attributo *store* puntino all'URL u. L'intero restituito è detto *indegree* ed è un numero non negativo;
- *int hostIndegree(URL u)* - Similmente al metodo precedente, restituisce il numero di pagine che puntano ad almeno un URL con lo stesso *hostname* di u;
- *ArrayList<PaginaWeb> queryAll(String x)* - Cerca all'interno delle pagine contenuto nello store del motore di ricerca quelle che contengono la stringa passata come argomento e le restituisce;
- *ArrayList<PaginaWeb> querySorted(String x)* - Esegue una ricerca della stringa x in questo motore di ricerca. Restituisce un ArrayList di pagine web contenenti x, ordinate secondo la seguente gerarchia di confronto:

  1. Una pagina web che contiene la stringa x è maggiore di un'altra pagina web che contiene la stringa x se la prima ha indegree maggiore della seconda;

2. Una pagina web che contiene la stringa x è maggiore di un'altra pagina web che contiene la stringa x se le due pagine hanno lo stesso indegree e nella prima ci sono più occorrenze di x rispetto alla seconda.

L'algoritmo di ordinamento utilizzato è quello di:

Collections.sort(List<T>, Comparator<? super T>)

dove il Comparator utilizzato è PaginaWebComparator (vedi sotto);

- *PaginaWeb queryOne(String x)* - Sfruttando il metodo precedente restituisce il primo elemento della lista restituita da *querySorted* (se non vuota) altrimenti *null*;
- *ArrayList<PaginaWeb> getStore()*
- *boolean equals (Object obj)* - Confronta due motori di ricerca e stabilisce se sono uguali, ovvero se contengono le stesse pagine (le quali sono confrontabili tramite il solo URL), indipendentemente dall'ordine in cui sono inserite nei due motori;
- *String toString()*;
- *boolean ricerca (Scanner sc, PrintStream pr, String x, boolean miSentoFortunato, String stopSignal)* - Questo metodo è ideato per un I/O da tastiera: infatti verrà utilizzato nel *main* proprio a questo scopo. Questo metodo ha due corpontamenti diversi a seconda del valore assunto dalla variabile *misentofortunato*. Se esso fosse *true* allora il metodo ricerca all'interno delle pagine la stringa x e stampa il *toString()* della pagina contenente x con *indegree* maggiore restituendo *true*. Se non fosse trovata alcuna pagina il metodo restituisce *false* dopo aver informato l'utente dell'assenza di risultati. Diverso il caso in cui *misentofortunato* sia *false*: il metodo, come prima, ricerca tutte le pagine contenenti x e le stampa a gruppi di 5 (questo numero può essere cambiato all'interno del metodo). L'utente ha la possibilità di scegliere se visualizzare una pagina, fermare la ricerca tramite l'opportuno segnale di stop *stopSignal* o visualizzare il successivo gruppo di 5 pagine (se non si è arrivati alla fine dei risultati). Nel metodo sono stati inseriti diversi controlli per assicurarsi che l'utente abbia inserito un valore accettabile. Se esiste almeno una pagina contenente x il metodo restituisce *true* altrimenti *false*.

All'interno di questa classe son definite ulteriori classi le quali implementano l'interfaccia *Comparator<T>* utili a confrontare oggetti della classe PaginaWeb. Queste classi, eccetto *PaginaWebComparator* estendono MotoreDiRicerca perchè per eseguire un confronto basato sull'indegree è necessaria copiare l'informazione contenuta in *store*. In altre parole, queste classi rappresentano dei motori di ricerca fittizi appositamente creati per eseguire il paragone. Esse sono:

- *UrlIndegreeComparator* - È un *Comparator* con metodo *compare(URL u0, URL u1)* che restituisce un intero negativo, nullo o positivo a seconda che l'indegree di *u0* sia minore, uguale o maggiore di quello di *u1*;

- *PaginaWebIndegreeComparator* - Opera come la classe precedente sfruttando l'URL di una PaginaWeb;
- *PaginaWebFrequenzaComparator* -Ha come attributo la stringa rispetto alla quale verrà eseguito il confronto; ha il metodo *compare(PaginaWeb p0, PaginaWeb p1)* che restituisce un intero negativo, nullo o positivo a seconda che *p1* contenga meno, stesse o più occorrenze della stringa attributo ricercata rispetto a *p2*;
- *PaginaWebComparator* - Implementa un comparatore che utilizza le due gerarchie di comparazione di PaginaWeb precedenti, dando precedenza alla gerarchia sull'indegree: nel caso di indegree diversi prevale quello maggiore, nel caso di indigree uguali prevale il maggior numero di occorenze della stringa x.

2.4 SitoWeb

Questa classe rappresenta un sito web, nonchè un insieme di pagine web aventi lo stesso host. Estende MotoreDiRicerca poichè contiene una collezione di pagine web sulle quali è possibile eseguire tutti i metodi di ricerca e sorting, ad eccezione di alcuni metodi di setting di cui è fatto opportunamente override. In altre parole, ogni sito web e' un motore di ricerca ristretto alle pagine in esso contenute, che hanno la proprietà di condividere lo stesso hostname. Il dynamic binding per istanze di SitoWeb è sconsigliato, anche se e' opportunamente gestito nei metodi.

Questa classe, oltre all'attributo proveniente da MotoreDiRicerca, possiede un attributo di tipo URL *urlBase* che rappresenta l'URL di base di tutte le pagine incluse in un SitoWeb. Abbiamo scelto di implentare i seguenti due costruttori:

- *SitoWeb()* - Imposta *null* come valore dell'attributo *urlBase*;
- *SitoWeb(URL u)* - Imposta come *urlBase* il valore di u privato del suo *path* (se presente);

Eredita i metodi di MotoreiRicerca e dei seguenti fa Override verificando la compatibilita dell'hostname:

- *boolean aggiungiPagina(PaginaWeb p)*
- *boolean aggiungiPagine(ArrayList<PaginaWeb> a)*
- *boolean leggiPagine(Scanner s)*

In più aggiunge i seguenti due metodi:

- *static int choiceList(Scanner sc, PrintStream pr, ArrayList<String> scelte, String stopSignal)* - Stampa una lista di scelte e acquisisce la risposta dell'utente verificandone la correttezza. Restituisce l'intero corrispondente alla scelta acquisita oppure -1 se l'utente digita *stopsignal*;

- *static SitoWeb websiteBuilder*(Scanner sc, PrintStream pr, String stopSignal) - Come ricerca di MotoreDiRicerca questo metodo é pensato per I/O da tastiera. Il metodo ha lo scopo di creare un sito web. Nel dettaglio, inizialmente chiede all'utente di scegliere uno dei protocolli disponibili e di inserire un *hostname* comprensivo di dominio, nel caso quest'ultimo sia assente chiede all'utente di selezionarne uno. Fatto ciò il metodo chiede all'utente di inserire il contenuto della homepage (ovvero la pagina avente come URL *urlBase*) ed gli URL delle pagine a cui essa punta. Una volta conclusa la procedura di creazione della home si procede con la costruzione dell'albero del sito ovvero si andranno a creare tutte le "sottopagine" della *homepage* in maniera iterativa. Similmente a prima per ciascuna di queste sottopagine viene chiesto all'utente di inserire prima il percorso, ovvero il *path* della pagina, poi il suo contenuto ed infine le pagine a cui punta. Durante questo processo il metodo si accerta che ogni volta l'utente inserisca i dati in maniera corretta avvisandolo ogni qual volta ciò non avvenga gestendo le eccezioni tranne *IllegalStateException* la quale viene sollevata in caso di chiusura dello scanner. Durante l'esecuzione l'utente ha sempre la possibilità di abortire il processo digitando l'opportuno *stopSignal*.

2.5 UrlNonValidoException

Questa classe, che estende la classe Exception, rappresenta un'eccezione che viene lanciata ogni qual volta il programma riscontra un URL non valido. In particolare si distinguono le "sottoeccezioni":

- *DominioNonValidoException;*
- *HostnameNonValidoException;*
- *PathNonValidoException;*
- *ProtocolloNonValidoException;*

Riteniamo che il nome di queste classi sia sufficientemente esplicativo per comprenderne il loro funzionamento.

3 Guida al main

Questo *main* differisce da ProvaMotoreDiRicerca, indicato nella consegna, perchè le opzioni: "Leggi pagine" e "Mi sento fortunato" sono eseguibili solo separatamente. Il programma però ovviamente estende le funzionalità della classe ProvaMotoreDiRicerca come si può dedurre dalla seguente spiegazione. Il metodo *main* è contenuto all'interno della classe *MotoreTreBiCa*. All'avvio il *main* stamperà all'utente il seguente menù in attesa che venga eseguita una determinata scelta:

Benvenuti nel motore di ricerca TreBiCa. Digitare una scelta:

1. *Aggiungi pagina web singola;*

2. *Aggiungi gruppo di pagine web;*

3. *Costruisci un sito web;*

4. *Esegui una ricerca;*

5. *Mi sento fortunato;*

6. *Tutte le pagine web di TreBiCa;*

7. *Esci.*

Una volta verificata la correttezza dell'inserimento il programma eseguirà le istruzioni richieste:

1. Si aggiunge una PaginaWeb allo *store* con la sintassi del metodo *PaginaWeb.leggi(s)*. All'utente viene richiesto se preferisce inserirla tramite tastiera o passando il percorso del file nel quale è contenuta la PaginaWeb;

2. In analogia al punto precedente, permette di aggiungere un gruppo di pagine dando sempre la possibilità di scegliere se inserirle da tastiera o da file;

3. Con questa scelta si permette all'utente di creare un sito web ovvero una successione di pagine aventi in comune lo stesso *hostname*. Come prima cosa sarà richiesto di creare la *homepage* dopodichè si inseriranno le altre pagine indicandone il *path*. L'aggiunta è possibile solo tramite tastiera. Si ha la possibilità di uscire in qualsiasi momento dalla creazione indicando se quello che è stato costruito fino a quel punto vada salvato, e aggiunto allo *store*, o scartato. Per ulteriori dettagli si consulti il metodo *websiteBuilder*;

4. In questa parte l'utente dovrà inserire una stringa da ricercare all'interno delle pagine contenute nello *store*. Verranno poi restituite, a gruppi di 5, tutte le pagine trovate ordinate secondo la gerarchia descritta nelle classi precedenti. L'utente può scegliere se visualizzare le successive pagine (nel caso non siano già state restituite tutte) o farsi stampare uno o più risultati della ricerca. Per ulteriori dettagli si consulti il metodo *ricerca*;

5. L'utente esegue un ricerca in modalità "Mi sento fortunato". In breve, stampa la prima pagina che si otterrebbe cercando la stringa digitata;

6. Visualizza tutte le pagine contenute nel MotoreDiRicerca in ordine di indegree;

7. Termina.

Qualsiasi scelta venga fatta il metodo si accerta che tutti gli inserimenti siano fatti correttamente gestendo le varie eccezioni. Ogni volta che finisce una di queste azioni deve ricomparire il menù. Inoltre in ogni momento l'utente può digitare STOP per abortire il processo corrente e tornare al menu principale. Il codice è stato modulizzato come segue: per l'implementazione delle scelte 1 e 2 abbiamo scritto il fulcro codice direttamente nel

main mentre per le restanti si richiamano i metodi *websiteBuilder* e *ricerca* descritti precedentemente. Questi metodi, essendo scritti in altre classi e resi pubblici, possono anche essere usati da altri programmi.

In allegato al progetto vi è anche un file di prova "Provaprogramma" contenente una lista di pagine web da leggere con l'opzione 2 (lettura da FILE)

4 Gestione delle eccezioni

Nel corso del progetto si è dovuto decidere quali eccezioni gestire e quali sollevare al chiamante, tra quelle sollevate dai metodi java utilizzati e quelle da noi appositamente create. In primo luogo, non volendo permettere che si potessero creare un URL o una PaginaWeb con attributi *null* oppure non corretti da un punto di vista sintattico (cfr. costruttori URL), abbiamo sollevato l'eccezione *UrlNonValidoException* nei costruttori e nei casi in questione. In questo modo si evita anche, nel corso del programma, di incappare nella noiosa eccezione *NullPointerException*. *UrlNonValidoException* passa anche attraverso i metodi *public static PaginaWeb leggi(Scanner s)* di PaginaWeb e *public int leggiPagine(Scanner s)* di MotoreDiRicerca, i quali la sollevano per comunicare al chiamante il motivo della non avvenuta creazione di una PaginaWeb (notare che leggi termina restituendo *null* se è invece il formato della stringa in input a non essere corretto). Ancora, il metodo leggi, utilizzando uno Scanner e il suo metodo *nextLine()*, solleva l'eccezione *NoSuchElementException* per comunicare che l'interruzione è stata causata dall'assenza di una prossima riga da leggere sullo Scanner, segnale utile per stabilire la fine di un file. Difatti, il metodo *leggiPagine* cattura quest'ultima eccezione e la usa come alternativa per terminare l'esecuzione del metodo. Per quanto riguarda i tratti di codice di I/O implementati nel main (scelte 1, 2) e i metodi *websiteBuilder* e *ricerca*, anch'essi per l'I/O, si è deciso di gestire (quasi) interamente le eccezioni. Il motivo è voler evitare la terminazione inaspettata del programma e iterare il dialogo con l'utente fino a che le scelte o le stringhe da lui/lei digitate non generino eccezioni. In particolare, si ha avuto a che fare con le eccezioni *NoSuchElementException*, *InputMismatchException*, *FileNotFoundException*, *IOException*. Le uniche eccezioni non gestite qui sono *UnsupportedEncodingException* e *IllegalStateException*, che si verificano rispettivamente se la codifica byte-char specificata non è supportata o se lo Scanner dovesse chiudersi inaspettatamente (casi rari). Infine si è reso necessario, sempre per il controllo e lo smistamento degli errori, specializzare l'eccezione *UrlNonValidoException* nelle sottoclassi indicate al punto 2.5: *websiteBuilder* utilizza questa distinzione per comunicare all'utente cosa non ha funzionato precisamente.

5 Perchè la scelta di una libreria esterna

Per l'implementazione del costruttore *PaginaWeb(URL u, String testogrezzo)* si è scelto di ricorrere ad una libreria esterna *Jsoup*. Il fine di questo costruttore è, presi in ingresso l'URL della pagina e una stringa, ripulire il contenuto di *testogrezzo* dai tag HTML ed

estrarre gli URL a cui la nostra PaginaWeb in costruzione punta. Ad esempio, data in ingresso la stringa:

This is a sample page that contains some link and some other link.

Il costruttore dovrà restituire la stringa:

This is a sample page that contains some link and some other link

ed i link:

[http://www.corriere.it/,https://foo.bar/div/index.html]

Vediamo come, sfruttando i metodi della libreria standard, si potrebbe procedere alla rimozione dei tag HTML. Un primo tentativo potrebbe essere il seguente:

```
int i = testogrezzo.indexOf('<');
while( i!=-1) {
    int f = testogrezzo.indexOf('>');
    testogrezzo=testogrezzo.substring(0,i)+testogrezzo.substring(f+1);
    i = testogrezzo.indexOf('<');
}
this.contenuto=testogrezzo;
```

Effettivamente questo algoritmo, ricevendo in ingresso la stringa che abbiamo scelto prima come esempio, restituisce la stringa desiderata. Vi sono però diverse problematiche con questa scelta, consideriamo ad esempio:

I know that 5<7.
 I love math

Vorremmo che il nostro costruttore ci restituisca la stringa:

I know that 5<7. I love math

ma, senza grosse sorprese, esso fornirà come contenuto della PaginaWeb la stringa:

I know that 5 I love math

Un'altra questione aperta è la seguente: come si comporta il costruttore in caso trovi un '<' non seguito da un '>'? Tutti questi problemi sono risolvibili: possiamo fare in modo che il nostro costruttore, una volta rilevata la posizione di un carattere '<', prima di procedere all'eliminazione del tag verifichi che esista un successivo '>' e che tra i due non sia presente un carattere '<' nel mezzo (nel caso che le seguenti condizioni non siano rispettate il costruttore non deve rimuovere nulla). Questo lascia ancora aperte delle questioni: consideriamo la stringa:

I know that 5<7 and also 8>7

vorremmo che il costruttore restituisca la stringa invariata ma, tralasciando i dettagli esso restituirà:

I know that 57

ovvero ha considerato la sottostringa *<7 and also 8>* come un tag HTML. Ancora una volta possiamo risolvere questa situazione: una volta individuato un candidato tag HTML potremmo controllare che il primo carattere dopo il '>' non sia un numero (in tal caso non sarà considerato tag HTML e il costruttore non rimuoverà nulla) oppure fare in modo che il costruttore abbia a disposizione la lista di tutti i possibili tag HTML e di volta in volta controlli che ci sia una corrispondenza tra il candidato tag trovato ed uno nella lista. La prima scelta porterebbe ad inserire diverse regole grammaticali per verificare se si è di fronte a un tag HTML oppure no (ma rischieremmo comunque di rimuovere parti di stringa che non sono dei veri tag). La seconda comporterebbe che ogni volta si debba scandire una lunga lista di elementi fino a trovare una corrispondenza cosa che vorremmo evitare. Per questi motivi abbiamo deciso di ricorrere ad uno strumento che avesse giìmplementato al suo interno dei metodi per riconoscere i tag HTML.

Per completare il discorso vediamo come potremmo estrarre un link HTML mediante l'uso delle funzioni della libreria standard. Supponiamo di aver individuato una coppia di $(<, >)$ di posizione (i, f) e di aver già verificato di esser di fronte ad un effettivo tag HTML. Si potrebbe procedere nella maniera seguente:

```
j = testogrezzo.indexOf("a href=",i);
if ((j>i)&&(j<f)) {
    int j1 = testogrezzo.indexOf('\"',j);
    int j2 = testogrezzo.indexOf('\"',j1+1);
    link.add(new URL(testogrezzo.substring(j1+1,j2));
}
```