UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

Tesi di Laurea in

INGEGNERIA DELL'INFORMAZIONE

Progettazione ed implementazione in Java di un sistema di gestione di immagini per una biblioteca digitale personalizzata

Laureando: Dario De Giovanni
Relatore: Prof. Giorgio Maria Di Nunzio

Anno Accademico 2011/2012

Indice

Elenco delle figure

Elenco dei listati codice

Capitolo 1
Introduzione

La costante crescita del mercato delle fotocamere digitali, l'utilizzo di smartphone come fotocamere e la crescita dei social network e della condivisione delle emozioni hanno cambiato profondamente le nostre abitudini nella gestione delle immagini; i computer risultano inondati da immagini e risulta difficile organizzarle e associare testi che ne descrivano il contenuto e le emozioni che avevano regalato.

Il mercato offre già soluzioni per questo problema, come Picasa[1] o Flickr[2], ma sono applicazioni che richiedono di essere collegati a Internet, e hanno bisogno della registrazione e del caricamento online delle immagini.
L'obiettivo di questa tesi è quello di progettare e realizzare un servizio di gestione di immagini digitali personali che possa essere utilizzato offline.

Per questo ho deciso di sviluppare un'applicazione che consentisse di organizzare le immagini in categorie e di associare un testo a ogni immagine e che avesse anche altre potenzialità, come dare un titolo all'immagine e poter visualizzare i metadata di un'immagine, se si è interessati.

Il progetto si può dire diviso in tre sezioni, i dati, l'applicazione per l'organizzazione delle immagini e quella per la visualizzazione.
I dati delle immagini sono salvati in un file XML [3]; l'XML è un linguaggio che permette di aggiungere mediante marcatori, *tag*, informazioni a un documento; nel file ogni immagine è rappresentata da un tag, chiamato *immagine,* questo contiene i tag per il percorso del file, il titolo, le categorie di cui fa parte l'immagine, la descrizione e i metadati.

Per creare e modificare l file XML delle informazioni delle immagini ho realizzato un programma in Java[4] che consente anche di visualizzare le immagini mentre si effettuano le modifiche.

Per la visualizzazione finale delle immagini suddivise in categorie ho deciso di usare pagine web HTML[5] basate sulle informazioni del file XML; a questo scopo mi sono serviti anche CSS, XSL e JavaScript[6]; i fogli stile CSS permettono di modificare lo stile di presentazione delle pagine HTML, i fogli stile XSL permettono di modificare lo stile di presentazione dei file XML nelle pagine web e JavaScript permette alle pagine HTML di interagire con i file XML.

1 http://picasa.google.com
2 http://www.fickr.com
3 http://www.w3.org/standards/xml
4 http://java.sun.com
5 http://www.w3.org/standards/webdesign/htmlcss
6 http://www.w3.org/standards/webdesign/script

Ho scelto di usare Java e tecnologie client-side per la visualizzazione per rendere l'applicazione indipendente dalla piattaforma utente, mentre se si fossero usate tecnologie client-side come JSP o PHP si sarebbe perso questo vantaggio.

1.1 Obbiettivi e Requisiti

L'obbiettivo di questo progetto è lo sviluppo di un'applicazione per l'organizzazione di immagini in categorie, a cui si possa associare anche una descrizione multilingua.

La progettazione dell'applicazione ha tenuto conto dei seguenti requisiti:

-**Soluzione *client side***: in modo che l'applicazione sia indipendente dalla piattaforma utente, e facilita un eventuale condivisione via Internet, perchè l'uso di tecnologie client side rende slegati dal provider di spazio web.

-**Soluzione multipiattaforma**: ho scelto una soluzione multipiattaforma scegliendo Java, un applicazione indipendente dalla piattaforma, e usando Firefox per la visualizzazione visto che è disponibile sia per Windows che per Linux.

-**Soluzione multilingua:** ho scelto una soluzione multilingua per consentire l'uso dell'applicazione anche all'estero, ma l'inserimento degli elementi multilingua viene fatto manualmente per ogni lingua; sarebbe più utile l'inserimento automatico ma il problema di traduzione automatica non è di facile soluzione e richiederebbe un lungo studio.

Capitolo 2
Tecnologie utilizzate

Per portare a termine il progetto ho usato diverse tecnologie come l'XML per i file dei dati, Java per la modifica di questi file e XHTML, CSS e JavaScript per la visualizzazione.

2.1 Struttura dei documenti

2.1.1 Struttura documenti XML

L'XML (eXtensible Markup Language) è un linguaggio di markup e costituisce un modo standard di descrivere i dati invece di utilizzare i formati proprietari dei vari DBMS. Un linguaggio di markup o linguaggio di contrassegno è un linguaggio che permette di aggiungere mediante marcatori (tag) informazioni a un documento; il documento risulta costituito da contenuto e da sistema di contrassegno.

Un documento XML è un documento di testo con estensione .xml, per scriverlo basta un semplice editor di testo; tutti i documenti XML devono essere ben formati, cioè devono rispettare alcune regole sintattiche:
-tutti i tag devono essere chiusi (<tag> ... </tag>); i tag vuoti (senza contenuto) possono essere chiusi inserendo la barra alla fine dell'elemento stesso (<tag .../>);
-i tag devono essere nidificati correttamente (i tag aperti per primi devono essere chiusi per ultimi);
-i valori degli attributi devono essere racchiusi tra virgolette.

L'XML è case sensitive, cioè è importante la differenza tra minuscole e maiuscole.

I caratteri speciali, come le vocali accentate, si possono inserire nella forma &#n, dove n è il codice Unicode del carattere, quindi ad esempio à si scrive con à ma si possono inserire anche senza codificarle, salvando il documento con la codifica UTF-8.

Un documento XML inizia con un prologo che contiene almeno la dichiarazione xml che identifica la versione della specifica XML a cui è conforme il documento (*<?xml version="1.0"?>*).
La dichiarazione può avere degli attributi come *encoding* che indica il tipo di codifica, il valore di default è UTF-8, e *stand alone* che impostato a *yes* indica che non ci sono riferimenti a entità esterne, cioè il documento è autonomo.

I documenti sono costituiti da markup e da dati di tipo carattere; i markup conferiscono la struttura al documento (comprendono i tag, ma anche i commenti, le dichiarazioni di tipo documento, le istruzioni di elaborazione ecc.), mentre i dati di tipo carattere sono compresi tra i tag.

Le dichiarazione di tipo di documento specificano una o più DTD da utilizzare e si indicano con l'elemento *<!DOCTYPE ... >*; una DTD indica quali sono i tag presenti nel documento XML e in che ordine si trovano. Le istruzioni di elaborazione sono direttive per il parser e sono racchiuse tra <? e ?>.

Dopo il prologo c'è l'elemento radice, la parte fondamentale del documento con il contenuto vero e proprio.
Ogni documento ha una radice che contiene tutti gli altri elementi.
Sono considerati caratteri vuoti gli spazi, le tabulazioni e i caratteri CR e LF; il parser passa tutti i caratteri all'applicazione che usa il documento, questa può considerare i caratteri vuoti come significativi e preservarli, o come insignificanti e ridurli a uno solo o rimuoverli (normalizzazione).

Nei dati di carattere non si possono usare i caratteri &, <, >, ' e " perché sono riservati al linguaggio XML.
Per usare questi caratteri nel contenuto o negli attributi degli elementi bisogna usare i riferimenti a entità: < < > > & & ' ' " ".
Gli attributi sono coppie *nome="valore"*; il valore dell'attributo deve essere racchiuso tra virgolette; può essere racchiuso tra apici se le virgolette fanno parte del valore.
In un elemento si può usare l'attributo generale xml:lang per specificare la lingua usata nell'elemento e nel valore degli attributi, per facilitare la ricerca con i motori di ricerca.

2.1.2 Strutturazione di file XML con DTD

L'XML permette di specificare un linguaggio di markup mediante una DTD; questa definisce i tag del linguaggio, gli attributi, e il modo in cui si possono utilizzare.
Un documento che rispetta le regole di una DTD si dice valido rispetto alla DTD; un documento può anche non utilizzare una DTD ma deve essere ben formato (well formed), cioè deve rispettare le regole sintattiche dei documenti XML.

Se si creano delle applicazioni che elaborano un particolare tipo di documento XML, definendo la DTD è possibile specificare come deve essere fatto il documento per poter essere elaborato. La DTD specifica il vocabolario XML (o la classe di documenti) per cui le applicazioni sono realizzate; in tal senso definisce l'applicazione.

Se un documento utilizza una DTD può usare solo i tag definiti nella DTD e solo nel modo definito dalla DTD quindi il parser, oltre a controllare se il documento è ben formato, controlla se il documento è valido, cioè se è conforme alla DTD.
Un parser che controlla se il documento è valido rispetto alla DTD viene detto parser validante; quando si utilizza un parser si può impostare in modo che sia validante e quindi richieda necessariamente la DTD per il documento o che non sia validante e quindi ignori la DTD anche se presente.

La DTD può essere definita internamente o esternamente.
La DTD interna viene definita con l'elemento
<!DOCTYPE nomeRadice [definizione della DTD]>.
La DTD esterna viene definita in un file con estensione .dtd; nel documento XML si fa riferimento alla DTD esterna con l'elemento
<!DOCTYPE nomeRadice SYSTEM "nomeFile.dtd">;
dove *nomeRadice* indica il nome del tag radice nel file XML; se si usano contemporaneamente una DTD esterna e una DTD interna le definizioni interne prevalgono.

La DTD contiene dichiarazioni di elementi che contengono il nome dell'elemento e la descrizione del contenuto.
Un elemento può contenere:
-altri elementi:
<!ELEMENT tag (elenco elementi)>;
-un contenuto (dati di carattere analizzabili, cioè testo che può contenere markup):
<!ELEMENT tag (#PCDATA)>;

Per specificare la struttura di una dichiarazione di elementi si usano i simboli:
-() per racchiudere una sequenza (l'elemento deve contenere la sequenza);
-, per separare gli elementi e identificare l'ordine (l'elemento deve contenere gli elementi nell'ordine specificato);
-? per indicare che un elemento deve apparire una sola volta o mai;
-* per indicare che un elemento può comparire quante volte si desidera (l'elemento è opzionale; può non apparire, o apparire una o più volte);
-+ per indicare che l'elemento deve essere visualizzato una o più volte (è necessario ma può apparire più volte).

Agli elementi si possono associare degli attributi; per dichiarare un attributo nella DTD si usa la sintassi *<!ATTLIST nomeElemento NomeAttributo Tipo Default>*, che può comparire in qualunque punto della DTD.
Il tipo può essere:
-CDATA: il valore dell'attributo è costituito da dati di tipo carattere;
-NMTOKEN: il valore dell'attributo è costituito da una combinazione di lettere, numeri, punti, trattini, due punti e trattini di sottolineatura.

2.1.3 Parsing e creazione di file XML in Java

Un'applicazione accede al contenuto di un documento XML sfruttando le API (Application Program Interface) messe a disposizione da un parser XML.

Java supporta due API complementari per l'elaborazione di un documento XML:
-SAX (Simple API for XML);
-DOM (Document Object Model for XML).

L'accesso a un parser si ottiene attraverso le classi del package *javax.xml.parsers.*
Il package *javax.xml.parsers* definisce le classi di supporto per l'elaborazione di documenti XML. Tutte le classi sono astratte perché le API sono state progettate per permettere l'utilizzo di parser differenti.

Il parser analizza il documento XML definendo caratteristiche e proprietà che controllano il modo in cui avviene l'elaborazione dei documenti XML e la restituzione delle informazioni all'applicazione.

Una caratteristica è un'opzione che può essere attivata o meno come il supporto ai namespace e la validazione; viene impostata tramite un valore *boolean* (per default sono *false);* un parser validante verifica che al documento sia associata una DTD e che il documento sia conforme alla DTD.
Una proprietà è un'opzione il cui valore è rappresentato da un oggetto (di solito *String*); alcune proprietà possono essere impostate per influenzare il funzionamento del parser, altre forniscono informazioni sul processo di parsing.

Con l'API DOM il file XML è visto come un albero di oggetti; l'elemento più in alto nella gerarchia è il documento, un oggetto di tipo *Document;* ogni elemento del documento è un nodo, un oggetto di tipo *Node*, anche il testo di un elemento costituisce un nodo, l'insieme dei figli di un nodo è una *NodeList,* mentre l'insieme degli attributi di un nodo è una *NamedNodeMap.*

Un parser DOM costruisce in memoria l'albero e lo restituisce all'applicazione come oggetto Document.

I package fondamentali per usare l'API DOM sono:
-javax.xml.parsers: classi di base che permettono di ottenere l'accesso a un parser;
-org.w3c.dom: classi e interfacce fondamentali dell'API DOM;
-*org.xml.sax*: classi e interfacce fondamentali dell'API SAX; l'API SAX viene usata dal parser DOM per creare la struttura ad albero in memoria; durante il processo di parsing il parser DOM solleva eccezioni SAX.

Per elaborare un file XML si usano le classi *DocumentBuilder* e *DocumentBuilderFactory* del package *javax.xml.parsers*; per prima cosa si crea un oggetto di tipo *DocumentBuilderFactory*:
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance().
La classe *DocumentBuilderFactory* mette a disposizione i metodi per le caratteristiche del parser:
-*void setIgnoringElementContentWhitespace(boolean whitespace)*
fa in modo che l'oggetto generi un parser che ignori gli spazi nel contenuto degli elementi;
-*void setValidating(boolean validating)*
fa in modo che l'oggetto generi un parser in grado di effettuare la validazione;

Tutte le caratteristiche sono di default *false,* se si vogliono cambiare bisogna farlo prima di creare l'oggetto di tipo *DocumentBuilder;* perché si possa leggere un documento scritto con indentazioni bisogna che il parser sia validante e ignori gli spazi bianchi.

Una volta settate le caratteristiche si crea un ogetto di tipo *DocumentBuilder*:
DocumentBuilder domParser = dbf.newDocumentBuilder().

Per effettuare il parsing del file si usa il metodo *parse(File f)* di *DocumentBuilder,* che restituisce un oggetto di tipo *Document* che rappresenta l'albero costruito in memoria.
Il metodo *parse()* genera delle eccezioni:
-*IllegalArgumentException* se si passa un valore *null*;
-*IOException* se si verifica un errore di I/O;
-*SAXException* se si verifica un errore durante il parsing.
Le ultime due devono essere intercettate quindi il metodo parse va usato dentro un blocco *try/catch*.

Una volta creato l'oggetto di tipo *Document* si può esplorare l'albero a partire dalla radice; per ottenere la radice si usa il metodo *getDocumentElement()* di *Document* che restituisce come tipo *Element*, da qui per accedere ai figli di un nodo, compresa la radice, si usa il metodo *getChildNodes()* che restituisce una *NodeList*, mentre per accedere agli attributi si usa il metodo *getAttributs()* che restituisce una *NamedNodeMap*.
I metodi principali per l'esplorazione dell'albero e l'interazione con i nodi sono quelli di *Node, Element*, *NodeList*, *NamedNodeMap.*

Per creare un file XML si incomincia creando un oggetto *Document* vuoto:
Document documento = domParser.newDocument();
Dopo di che si creano i nuovi nodi con i metodi di *Document*: *createElement* per creare un nuovo tag, *createAttribute* per creare un nuovo attributo e *createTextNode* per creare un nodo di testo; per inserire i nuovi nodi nel documento o per modificare nodi esistenti si usano i metodi dell'interfaccia

Node, mentre per aggiungere, cancellare o modificare gli attributi di un elemento si usano i metodi di *NamedNodeMap* o i metodi di *Element.*
Infine si scrive l'albero su di un file con estensione .xml con:

```
try{
DOMSource domSource = new DOMSource(nome oggetto Document);
        StreamResult streamResult = new StreamResult(new FileOutputStream("nome file"));
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer serializer = tf.newTransformer();
        serializer.setOutputProperty(OutputKeys.METHOD,"xml");
        serializer.setOutputProperty(OutputKeys.ENCODING,"UTF-8");
        serializer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,"nome file DTD");
        serializer.setOutputProperty(OutputKeys.INDENT,"yes");
        serializer.transform(domSource, streamResult);
}
catch(java.io.FileNotFoundException e){}
catch(javax.xml.transform.TransformerConfigurationException  e){}
catch(javax.xml.transform.TransformerException e){};
```

Codice 1: Metodo per scrivere un file XML in Java

In questo modo il file avrà associata una DTD e sarà scritto con i ritorni a capo dopo ogni tag chiuso.

2.2 Interfaccia grafica

2.2.1 Interfaccie grafiche in Java

Per creare interfacce grafiche in Java esistono due package *java.awt* e *javax.swing*.
Per creare le finestre ho usato *javax.swing*, perché più recente; per creare una finestra basta estendere la classe *JFrame* di questo package, facendo questo si crea una finestra vuota, dopo di che con i metodi *setSize*, *setTitle*, *setVisible(true)*, della classe *JFrame,* si impostano le dimensioni della finestra, il nome della finestra e la si rende visibile.

Sulla finestra si possono inserire dei componenti come pulsanti, caselle, aree di testo, ecc; per questi componenti ci sono delle classi già predefinite, sempre nel package javax.swing, e sono: *JButton* per i pulsanti, *JTextField* per le caselle, *JTextArea* per le aree di testo.

I componenti vanno inseriti in contenitori, oggetti di tipo *Container*; il contenitore principale è lo sfondo della finestra che si ottiene con il metodo *getContentPane* della classe *JFrame;* dopo si possono creare nuovi contenitori denominati pannelli, oggetti di tipo *JPanel.*

Per aggiungere un componente o un contenitore in un altro contenitore si usa il metodo *add:*
contenitore.add(componente);
contenitore1.add(contenitore2), il *contenitore2* viene aggiunto al *contenitore1.*

I componenti sono aggiunti in base al layout del contenitore che può essere:
-*BorderLayout*, divide il contenitore in cinque regioni, *North, South, East, West e Center*;
-*GridLayout,* divide il contenitore in una griglia, si può decidere il numero di righe e di colonne;
-*FlowLayout*, dispone i componenti di seguito con una grandezza standard;
-null, si possono disporre i componenti come si vuole e farli della grandezza che si vuole.

Alcuni componenti possono generare degli eventi, gli eventi sono oggetti descritti da sottoclassi di *EventObject*, alcuni esempi sono i pulsanti e le caselle.

Per poter compiere delle azioni quando questi eventi si verificano bisogna creare delle classi che riescano a intercettare questi eventi; per intercettare un evento la classe deve implementare l'interfaccia dell'ascoltatore di eventi desiderati, per esempio per intercettare gli eventi dei pulsanti l'interfaccia da implementare è *ActionListener;* una volta creata questa classe bisogna aggiungere un oggetto di questa classe agli ascoltatori degli eventi generati dal componente:
componente.addActionListener(new GestoreEvento()).

Anche la finestra genera degli eventi, se si vuole intercettare quelli di chiusura della finestra bisogna cambiare la politica della finestra riguardo a questo evento:
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
in questo modo l'evento viene intercettato dalla classe da noi creata; altre politiche sono:
EXIT_ON_CLOSE definita nella classe *JFrame,* esce dall'applicazione usando il metodo *System.exit(0)*;
DISPOSE_ON_CLOSE definita nella classe *WindowConstants,* chiude ed elimina la finestra (ma non chiude l'applicazione);
HIDE_ON_CLOSE definita nella classe *WindowConstants,* nasconde la finestra, è l'azione di default.

Il package *javax.swing* mette a disposizione anche delle finestre predefinite, le finestre di dialogo, oggetti di tipo *JOptionPane;* per usarle basta richiamare i metodi statici di *JOptionPane*:
-*static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)*: mostra un messaggio all'utente;

-*static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType, int messageType)*: propone una finestra di conferma; restituisce un valore intero che indica l'opzione selezionata.

Altre finestre predefinite sono fornite da *JFileChooser* che fornisce le finestre per navigare tra le cartelle quando si vuole salvare o aprire un file; a queste finestre si possono aggiungere dei filtri in modo che visualizzino solo un certo tipo di file.

2.2.2 Gestione delle immagini in Java

Per interagire con le immagini si usano i package *java.awt.image*, *java.io, javax.imageio*, *javax.imageio.stream.*
Per aprire un'immagine si usa il metodo *ImageIO.read(new File(path del file))* questo metodo restituisce un oggetto di tipo *Image.*

Per visualizzare un'immagine su di un pannello bisogna che questo ridefinisca il suo metodo *paint* inserendovi le istruzioni:
```
super.paint(g);
Graphics2D g2 = (Graphics2D)g;
g2.drawImage(image,0,0,this);
```
dove *g* è un oggetto di tipo *Graphics* passato al metodo *paint* dalla finestra.

L'immagine potrebbe essere troppo grande per essere visualizzata e quindi bisogna ridimensionarla con il metodo
getScaledInstance(new width, new height, Image.SCALE_DEFAULT)
che restituisce un nuovo oggetto *Image* dell'immagine ridimensionata.

Per salvare un immagine bisogna creare un file in cui salvarla:
File out = new File(path e nome del file);
dopo di che si usano le seguenti istruzioni:
```
FileImageOutputStream outIm = new FileImageOutputStream(out);
ImageIO.write(immagine,estensione,outIm);
```
dove *immagine* è un oggetto di tipo *BufferedImage*; *Image* estende *BufferedImage, estensione* è la sigla di un'estensione di immagini (jpeg, gif, png, ecc.) e va racchiusa tra virgolette.

Se l'immagine è ridimensionata questo non funziona perché il metodo *getScaledInstance* non restituisce un oggetto di tipo *BufferedImage* quindi bisogna crearlo:
```
BufferedImage newIm = new BufferedImage(oldIm.getWidth(null), oldIm.getHeight(null),
BufferedImage.TYPE_INT_RGB);
newIm.createGraphics().drawImage(oldIm, null, null);
```
il primo metodo crea un immagine *BufferedImage* vuota con le dimensioni di *oldIm*, il secondo metodo disegna l'immagine *oldIm* sull'immagine *newIm*.

Alcune immagini potrebbero lanciare l'eccezione *java.lang.OutOfMemoryError* perché troppo grandi, per ovviare a questo problema bisogna fornire più memoria alla JVM con l'istruzione *-XmsMemoriaMin -XmxMemoriaMax* inserita da prompt dopo la parola chiave *java* e prima del nome della classe; dove *MemoriaMin* e *MemoriaMax* sono valori con unità di misura, *m* per MegaByte.

2.2.3 Gestione dei metadati in Java

Per accedere ai metadati di un'immagine bisogna scaricare dei package esterni perché non presenti in quelli già forniti.

I metadati di un immagine, o dati exif, sono dati incorporati nell'immagine che indicano risoluzione, data in cui è stata creata, se è una foto, con che macchina fotografica è stata scattata e che impostazioni aveva, e altre informazioni.

Una volta scaricato il package bisogna modificare la variabile d'ambiente *CLASSPATH* in modo che la JVM sappia che questo nuovo package è stato aggiunto.
Questo può essere fatto da prompt inserendo dopo la parola chiave *java* il seguente comando:
-classpath path del package;.; così però la modifica resta valida solo per l'applicazione lanciata, dopo la variabile si resetta.

I metodi per interagire con i metadata sono:
-*readMetadata(File file)* di *ImageMetadataReader* che restituisce un oggetto di tipo *Metadata;*
-*getDirectories* di *Metadata* che restituisce una lista di oggetti *Directory;*
-*getTags* di *Directory* che restituisce una lista di oggetti di tipo *Tag* che corrispondono al singolo dato.

Per accedere al nome e al valore del dato si usano i metodi *getTagName* e *getDescription* di *Tag.*

2.3 Web

2.3.1 Struttura documenti HTML

Le pagine Web sono documenti in formato testo che usano il linguaggio HTML (Hyper Text Markup Language) o XHTML (eXtensible HTML). Questi linguaggi definiscono dei codici speciali detti tag che permettono di stabilire di che tipo sono gli elementi che compongono il documento. L'inserimento dei tag nel documento è definito fase di contrassegno.

La differenza tra HTML e XHTML sta nel fatto che XHTML è un documento XML e quindi deve essere ben formato, cioè deve rispettare le regole dell'XML.

Per cambiare lo stile in cui vengono presentate le informazioni si possono usare i fogli stile CSS.

I documenti XHTML, essendo documenti XML, possono essere validati, cioè controllati rispetto alla DTD, usando un servizio di validazione. La DTD usata è specificata nella dichiarazione !DOCTYPE all'inizio del documento, questa corrisponde alla versione di XHTML usata, e indica ai servizi di validazione quali controlli fare per validare il documento e ai browser come interpretare i tag.

Esistono tre versioni di XHTML: *transitional, strict* e *frameset.*
Transitional è la più vicina all'HTML tradizionale, permette di usare markup di presentazione e elementi o attributi deprecati (per esempio l'attributo target per i link).
Con la versione *Transitional* si possono usare le tabelle per posizionare gli elementi della pagina.
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">.
Strict è la versione più rigida; per esempio non consente di usare l'attributo target per i link; bisogna usare *JavaScript* per simularne il comportamento.
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">.
Frameset è l'unica versione che consente di usare i frame.
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">.

Subito dopo il *doctype* c'è l'elemento <html> con la dichiarazione del *namespace* usato:
<html xmlns:"http://www.w3.org/1999/xhtml" xml:lang="it" lang ="it">.
L'attributo *lang* specifica che il documento è scritto in italiano; l'attributo *xml:lang="it"* specifica che anche i tag XML sono in italiano.

Bisogna anche dichiarare il tipo di codifica dei caratteri; si può specificare nella dichiarazione XML opzionale, o prologo XML, che precede il *doctype* (valori: UTF-8 per Unicode o ISO-8859-1 per ASCII):
<?xml version="1.0" encoding="UTF-8" ?>.

Per validare il documento si può usare il Validation Service del W3C (http://validator.w3.org).

2.3.2 Formattazione file HTML con CSS

I fogli stile CSS (Cascading Style Sheets) permettono di applicare caratteristiche di formattazione agli elementi HTML.

L'HTML definisce la struttura del documento, mentre i fogli stile permettono di definire lo stile di presentazione. I browser che non supportano gli stili ignorano le informazioni di stile e il documento viene visualizzato senza applicare gli stili; i browser non sempre implementano allo stesso modo le specifiche CSS quindi i risultati possono essere diversi.

I tipi di fogli stile possibili sono: in linea, incorporati e esterni.
I fogli in linea permettono di applicare la formattazione ad un elemento specifico; per applicare uno stile in linea si aggiunge l'attributo *style* al tag dell'elemento.
I fogli incorporati permettono di applicare la formattazione a gruppi di elementi nella pagina; per fare questo si inseriscono le informazioni di stile in un tag <style> inserito dopo il tag <html> e prima del tag <body>.
La sintassi per l'applicazione degli stile ad un tag HTML è:
tagHTML {attributoDiStile1: valore 1; ... attributoDiStileN: valoreN}

I fogli esterni permettono di applicare la formattazione a gruppi di elementi in pagine diverse. Il foglio stile viene definito in un documento con estensione .css; questo documento contiene le definizioni di stile (senza i tag <style>), con la stessa sintassi degli stili incorporati. Le pagine che utilizzano questi fogli devono usare la seguente istruzione:
<link rel="stylesheet" type="text/css" href="fileStile.css">.

Se presenti tutti e tre vengono applicati primi gli stili in linea, poi quelli incorporati ed infine quelli esterni.

Gli attributi *id* e *class* nei tag HTML permettono di aumentare la granularità del controllo sugli elementi.
L'attributo *id* assegna ad un elemento un nome in modo univoco, il nome non può contenere spazi o trattini; l'attributo *class* invece permette di raggruppare gli elementi in classi, gli elementi con lo stesso valore dell'attributo *class* appartengono alla stessa classe.
Questo permette di formattare solo gli elementi di una classe lasciando invariati gli altri.

2.3.3 Formattazione file XML con XSL

Per visualizzare un documento XML si possono usare i fogli stile XSL; un foglio stile è un documento XML ben formato con estensione .xsl.

Il nodo radice deve essere *stylesheet.* Nell'elemento *stylesheet* viene usato l'attributo *xmlns* per fare riferimento allo spazio dei nomi del linguaggio XSL.

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">.

Per applicare direttamente un foglio stile ad un file XML si usa l'istruzione di elaborazione:
<?xml-stylesheet?> (<?xml-stylesheet type="text/xsl" href="file.xsl"?>);

Un foglio stile comprende uno o più modelli (template); i template sono le regole di trasformazione; specificano la struttura del documento che deve essere generato. Ogni template specifica mediante un pattern o query gli elementi del documento XML di origine a cui applicare la trasformazione.
Le query sono di tipo dichiarativo (non procedurale); specificano cosa deve essere cercato nel documento ma non come deve essere eseguita la ricerca.

I pattern possono essere costituiti dal nome di un elemento, dal nome di un attributo, preceduto dal simbolo @ o pattern più complicati specificati usando XPath.

I template si descrivono con l'elemento *xsl:template*; l'attributo *match* specifica il pattern; tutto ciò che è all'interno dell'elemento template costituisce la regola di trasformazione:
<xsl:template match="pattern">
...
</xsl:template>.

L'istruzione *xsl:sort* permette di ordinare gli elementi in base al valore di un pattern (elemento o attributo) specificato con l'attributo *select:* *<xsl:sort select="pattern" />*; l'attributo *order* permette di specificare l'ordinamento ascendente (ascending) o discendente (descending), l'attributo *data-type* permette di specificare se l'ordinamento deve essere eseguito considerando i dati di tipo numerico (number) o testuale (text).

Per recuperare i dati dal documento di origine si usa l'elemento *xsl:value-of* con l'attributo *select:*
<xsl:value-of select="pattern"/>; questa istruzione restituisce solo il primo elemento corrispondente al pattern; l'istruzione *xsl:value-of* non può essere usata nel valore di un attributo.
Per inserire i dati di un elemento del documento di origine nel valore di un attributo basta indicare il nome dell'elemento tra parentesi graffe.

Si può eseguire un ciclo con l'istruzione *xsl:for-each*:
<xsl:for-each select="pattern">
...
</xsl:for-each>
Questa istruzione scandisce i figli di un elemento ma non visualizza nulla, per visualizzare dei dati bisogna inserire all'interno del ciclo l'istruzione *xsl:value-of*.

Si può realizzare una struttura condizionale con l'istruzione *xsl:if*, l'espressione logica che costituisce la condizione è specificata nell'attributo *test:*

<xsl:if test="condizione">

...

</xsl:if>.

I pattern possono essere specificati usando XPath; XPath è un linguaggio che permette di indirizzare parti di un documento XML; questo viene considerato come una struttura ad albero in cui ogni parte del documento è un nodo.

C'è un solo nodo radice; tutti gli altri nodi hanno un genitore (parent) e possono avere dei figli (child); ogni elemento ha come figli gli elementi che contiene; i nodi di testo sono figli degli elementi che li contengono, mentre gli attributi hanno un parent (il nodo che li contiene) ma non sono figli di quel nodo.

Un'espressione XPath può dare come risultato un insieme di nodi oppure un valore booleano, numerico o stringa. La valutazione di una espressione avviene in un contesto; questo viene cambiato dalle istruzioni *xsl:template* e *xsl:for-each* in base al pattern specificato.

Nei pattern XPath si possono utilizzare simboli e test dei nodi; i simboli possibili sono:
-/ scende di un livello nell'albero;
-// scende di un numero qualsiasi di livelli nell'albero;
-. rappresenta l'elemento corrente, o nodo di contesto;
-.. rappresenta l'elemento genitore, cioè il nodo parent del nodo di contesto;
-@ permette di accedere agli attributi del nodo di contesto;
In particolare:
-@nome seleziona l'attributo specificato del nodo di contesto;
-@* seleziona ogni attributo del nodo di contesto;

In XSL si possono dichiarare e inizializzare variabili con l'elemento *xsl:variable*:

<xsl:variable name="nomeVar" select="pattern" />

<xsl:variable name="nomeVar">valore</xsl:variable>

Si può visualizzare il valore di una variabile con l'istruzione *xsl:value-of* indicando il nome della variabile preceduto dal simbolo *$* nell'attributo *select*:

<xsl:value-of select="$nomeVar" />

2.3.4 JavaScript, uso file XML in HTML

JavaScript è un linguaggio di scripting; i linguaggi di scripting sono linguaggi le cui istruzioni vengono inserite in un documento HTML e interpretate da un interprete interno al browser.

Le istruzioni JavaScript devono essere incluse in script all'interno di un documento HTML.
JavaScript ha un notazione formale come Java, e come Java è un linguaggio orientato agli oggetti.

Ho usato JavaScript per poter modificare le variabile dei fogli stile XSL, per leggere i file XML e per cambiare i nomi delle categorie nel menu a tendina in base alla lingua scelta.

Si può interagire con i file XML, e quindi anche con XSL, inserendo un parser in una pagina HTML e usando gli script per interagire con il documento.
Un documento XML è visto come un insieme di elementi annidati, strutturati in modo gerarchico ad albero; il parser analizza ed eventualmente convalida il documento e restituisce una struttura ad albero di tutti gli elementi inclusi nel documento.

Il modello a oggetti dei documenti XML è descritto dallo standard DOM (Document Object Model) XML.
L'elemento al più alto livello della gerarchia è il documento *(Document);* ogni elemento del documento è un nodo *(Node);* anche il testo contenuto in un elemento è un nodo; i figli di un nodo costituiscono una *NodeList*; l'insieme degli attributi di un nodo costituisce una *NamedNodeMap.*

L'oggetto *Document* viene creato dal parser. Il modo in cui inserire una istanza di un parser in una pagina HTML e ottenere un oggetto *Document* dipende dal browser e dal parser utilizzato.

In Internet Explorer L'istruzione *new ActiveXObject("microsoft.xmldom")* crea un'istanza del parser *msxml* e restituisce un oggetto di tipo *Document.*
Il documento XML si carica con il metodo *load()* di *Document.* Si può controllare l'esito del caricamento verificando la proprietà *readyState* del documento; il valore 4 indica che il documento è stato caricato completamente:

```
documento.load("dati.xml");
while(!documento.readyState == "4");
```

In Firefox invece si usa l'istruzione

```
document.implementation.createDocument("", "", null)
```

che restituisce un oggetto di tipo *Document*, il documento XML si carica sempre con il metodo *load()*.

Ponendo a *false* la variabile *async* di *Document* si aspetta che il documento sia caricato prima di procedere con le altre istruzioni, questo è necessario in Firefox altrimenti ci potrebbero essere problemi.

Capitolo 3
Implementazione

Per l'implementazione oltre ai dati delle immagini ho organizzato anche le categorie e le lingue in file XML; l'applicazione Java visualizza le immagini che si possono aggiungere e permette di modificare alcune informazioni, come la descrizione e il titolo; infine si visualizzano le immagini organizzate tramite pagine web XHTML.

3.1 Organizzazione dati in file XML

Per l'organizzazione dei dati ho usati i file XML
-lingue.xml
-immagini.xml
-categorie.xml

3.1.1 lingue.xml

Le lingue possibili sono inserite nel file XML lingue.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE lingue[
<!ELEMENT lingue (lingua+)>
<!ELEMENT lingua (#PCDATA)>
]>
<lingue>
        <lingua>lingua1</lingua>
        ...
        <lingua>linguaN</lingua>
</lingue>
```

Codice 2: lingue.xml

3.1.2 immagini.xml

Le informazioni relative alle immagini sono salvate nel file XML immagini.xml; ogni immagine è identificata da un tag *immagine* con un attributo *iden*, un numero che identifica l'immagine e l'ordine di inserimento delle immagini.
Ogni *immagine* contiene i tag: *path*, *title*, *categorie*, *descr*, *exif*.

Il tag *path* indica il percorso assoluto dell'immagine.
Il tag *title* indica il titolo dell'immagine, che può essere diverso dal nome.
Il *tag categorie* contiene il tag *categoria*.
Il tag *categoria* contiene l'identificativo di una categoria presente nel file XML categorie.xml:

```
<categorie>
        <categoria>1</categoria>
        ...
        <categoria>n</categoria>
</categorie>.
```

Codice 3: Organizzazione tag categorie nel file immagini.xml

Il tag *descr* contiene i tag *descrizione* che contengono le descrizioni delle immagini in una lingua specifica; ogni tag *descrizione* ha come attributo la lingua in cui è scritta la descrizione.
Le righe della descrizione sono inserite nel tag *riga*:

```
<descr>
        <descrizione lingua=”lingua1”>
                <riga>testo riga 1</riga>
                ...
                <riga>testo riga k</riga>
        </descrizione>
        ...
        <descrizione lingua=”linguaN”>
                <riga>testo riga 1</riga>
                ...
                <riga>testo riga k</riga>
        </descrizione>
</descr>.
```

Codice 4: Organizzazione tag descr nel file immagini.xml

Il tag *exif* è opzionale, se l’immagine non possiede dati exif il tag non compare; questo tag contiene un tag *dato* per ogni metadato dell’immagine; ogni *dato* ha un attributo *name* che identifica il nome del metadato, e contiene il valore del metadato:

```
<exif>
        <dato name=”dato1”>valore dato1</dato>
        ...
        <dato name=”datoM”>valore datoM</dato>
</exif>.
```

Codice 5: Organizzazione tag exif nel file immagini.xml

I tag e gli attributi del file immagini.xml sono definiti dalla DTD DizionarioIm.dtd, in modo che l’applicazione Java possa leggere il file e scriverlo in maniera leggibile:

```
<!ELEMENT immagini (immagine*)>
<!ELEMENT immagine (path, title, categorie, descr, exif*)>
<!ATTLIST immagine iden CDATA #REQUIRED>
<!ELEMENT path (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

```
<!ELEMENT categorie (categoria*)>
<!ELEMENT categoria (#PCDATA)>
<!ELEMENT descr (descrizione+)>
<!ELEMENT descrizione (riga*)>
<!ATTLIST descrizione lingua CDATA #REQUIRED>
<!ELEMENT riga (#PCDATA)>
<!ELEMENT exif (dato+)>
<!ELEMENT dato (#PCDATA)>
<!ATTLIST dato name CDATA #REQUIRED>.
```

Codice 6: DizionarioIm.dtd

3.1.3 categorie.xml

Il file XML delle categorie, categorie.xml, contiene i nomi delle categorie in tutte le lingue.
Il tag radice è il tag *categorie* che contiene i tag *categoria*, almeno uno, per ogni categoria definita.

Ogni tag *categoria* ha un attributo *iden* che identifica l'ordine di inserimento delle categorie e viene utilizzato nel file immagini.xml per identificare la categoria in modo univoco, la categoria *tutte,* l'unica già presente, ha *iden* uguale a 0; il tag *categoria* contiene un tag *cat* per ogni lingua.

Ogni tag *cat* ha un attributo *lingua* che identifica in che lingua è scritto il nome della categoria, e contiene il nome della categoria nella lingua identificata dall'attributo:

```
<categoria iden="n">
        <cat lingua="lingua1">nome categoria n nella lingua1</cat>
        ...
        <cat lingua="linguaN">nome categoria n nella linguaN</cat>
</categoria>.
```

Codice 7: Organizzazione del file categorie.xml

I tag e gli attributi del file categorie.xml sono definiti dalla DTD DizionarioCa.dtd, in modo che l'applicazione java possa leggere il file e scriverlo in maniera leggibile:

```
<!ELEMENT categorie (categoria+)>
<!ELEMENT categoria (cat+)>
<!ATTLIST categoria iden CDATA #REQUIRED>
<!ELEMENT cat (#PCDATA)>
<!ATTLIST cat lingua CDATA #REQUIRED>.
```

Codice 8: DizionarioCa.dtd

3.2 Applicazione Java

L'applicazione Java serve per aggiornare il file XML delle informazioni relative alle immagini, aggiungendone di nuove o modificando quelle già presenti.

Per facilitare l'applicazione Java con i menu multilingua, i nomi sono inseriti nel file XML menu.xml.
Ogni finestra dell'applicazione è identificata da un tag *finestra* che ha un attributo *nome* per identificare la finestra specifica, e un tag *menu*, con l'attributo *lingua*, per ogni lingua.
In ogni *menu* ci sono i tag *pulsante*, *didascalia* e *messaggio*, ogni tag ha un attributo *nome* per identificare in maniera univoca il componente.
Il tag *pulsante* contiene la scritta che compare sul relativo pulsante della finestra.
Il tag *didascalia* contiene la scritta che compare sulla relativa etichetta della finestra.
Il tag *messaggio* contiene la scritta che compare sul relativo messaggio che compare per certe azioni compiute sulla finestra:

```
<finestra nome="finesrta1">
        <menu lingua="lingua1">
                <pulsante nome="pulsante1">scritta pulsante1</pulsante>
                ...
                <pulsante nome="pulsanteM">scritta pulsanteM</pulsante>
                <didascalia nome="didascalia1">scritta didascalia1</didascalia>
                ...
                <didascalia nome="didascaliaH">scritta didascaliaH</didascalia>
                <messaggio nome="messaggio1">scritta messagio1</messaggio>
                ...
                <messaggio nome="messaggioK">scritta messagioK</messaggio>
        </menu>
        ...
        <menu lingua="linguaN">
                ...
        </menu>
</finestra>
...
<finestra nome="finestraZ">
        ...
</finestra>
```

Codice 9: Organizzazione del file menu.xml

I tag di menu.xml sono definiti dalla DTD DizionarioMe.dtd:

```
<!ELEMENT finestre (finestra+)>
<!ELEMENT finestra (menu+)>
<!ATTLIST finestra nome CDATA #REQUIRED>
<!ELEMENT menu (pulsante*,didascalia*,messaggio*)>
<!ATTLIST menu lingua CDATA #REQUIRED>
<!ELEMENT pulsante (#PCDATA)>
<!ATTLIST pulsante nome CDATA #REQUIRED>
<!ELEMENT didascalia (#PCDATA)>
<!ATTLIST didascalia nome CDATA #REQUIRED>
<!ELEMENT messaggio (#PCDATA)>
<!ATTLIST messaggio nome CDATA #REQUIRED>.
```

Codice 10: DizionarioMe.dtd

3.2.1 La finestra iniziale

Per cominciare si lancia da prompt il *main* della classe *TestS* che crea un oggetto di tipo *Start:*

```
class TestS{
        public static void main(String[] args){
                Start st = new Start();
        }
}
```

Codice 11: TestS.java

La classe *Start* costruisce la finestra iniziale con i vari menu, un menu a tendina per le lingue e quattro pulsanti per le operazioni possibili:

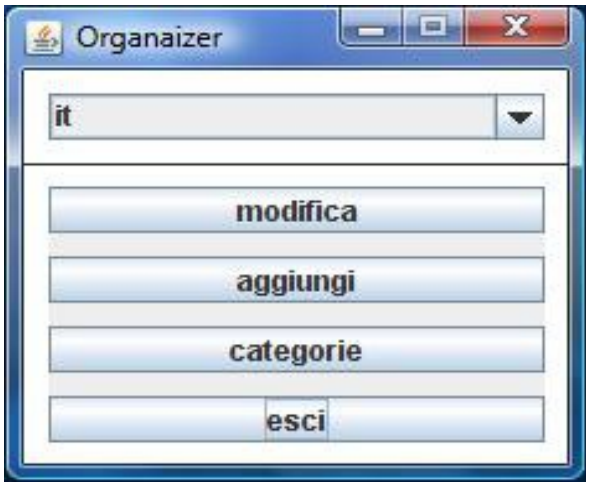

Figura 1: Finestra iniziale dell'applicazione Java

I nomi dei menu e delle lingue vengono salvati su una lista dal file XML relativo:

```
...
//lettura file lingue.xml e creazione relativa NodeList
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(true);
dbf.setIgnoringElementContentWhitespace(true);
try{
        DocumentBuilder domParser = dbf.newDocumentBuilder();
        documento = domParser.parse(new File("lingue.xml"));
        lingue=documento.getDocumentElement();
}
catch(SAXParseException e){
        System.out.println("Parsing error: "+e.getMessage());
        System.exit(1);
}
catch(FileNotFoundException e){}
catch(Exception e){
        e.printStackTrace();
}
...
```

Codice 12: Metodo per la letture dei file XML in Start.java

Le stesse istruzioni sono usate per i nomi dei menu sostituendo “lingue.xml” con “menu.xml”.

Le 4 operazioni possibili sono:
1-Modifica;
2-Aggiungi;
3-Categorie;
4-Esci;

4-Esci:

L’evento del pulsante *esci* viene intercettato dal relativo gestore che attiva la procedura di uscita dall’applicazione, che consiste nel confermare l’operazione, e a risposta positiva spegne la JVM:

```
//classe per la gestione dell'evento esci
 class GestoreEsc implements ActionListener{
        public void actionPerformed(ActionEvent ev){
                esci();
        }
}
```

```
//procedura d'uscita
public void esci(){
int c =
JOptionPane.showConfirmDialog(null,menuL.item(4).getFirstChild().getNodeValue(),
"Uscita",JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE);
   if(c==0){ System.exit(0); }
}
```

Codice 13: Gestore evento del pulsante esci e metodo esci() in Start.java

Le altre operazioni vengono gestite dal metodo *startOp* ma vengono sempre attivate dal evento lanciato dal relativo pulsante:

```
//inizio dell'interazione con l'utente
public void startOp(){
        while(!fine){
                //aspetta finchè non si registra un evento dei pulsanti
                while(!inizio){}

                inizio=false;
                st.setVisible(false);

                if(newc){
                //registrato evento crea nuova categoria
                        CreaCat ct = new CreaCat(lingue,lS);
                        while(!ct.done()){}
                        ct.dispose();
                }
                else{
//registrato evento modifica o aggiungi "op" definisce l'operazione dacompiere tra le due
                        Organaizer o = new Organaizer(op,lS,lingue);
                        while(!o.done()){}
                }
                newc=false;
                st.setVisible(true);
        }
}
```

Codice 14: Metodo sartOp di Sart.java

3-Categorie:

L'evento del pulsante *categorie* viene intercettato dal relativo gestore che imposta le variabile *newc* al valore *true* per indicare al metodo *startOp* che il pulsante *categorie* è stato premuto; questo fa si che un nuovo oggetto di tipo *CreaCat* venga creato, passandogli le variabili: *lS* (indica la lingua scelta), *lingue* (indica la testa della lista delle lingue).

1_2-Modifica_Aggiungi:

Entrambi i pulsanti portano alla creazione di un nuovo oggetto di tipo *Organaizer* passandogli le variabili:
op (indica l'operazione scelta 1 o 2), *IS* (indica la lingua scelta), *lingue* (indica la testa della lista delle lingue).

4.2.2 Creare nuove categorie

Il costruttore di questa classe crea la finestra per l'interazione dell'utente con il file XML delle categorie, la finestra mostra le categorie già presenti (parte sinistra), e consente di crearne di nuove (parte destra); per creare una nuova categoria bisogna inserire il nome della categoria in tutte le lingue, si sceglie la lingua dal menu a tendina e si digita il nome nella casella sottostante, cambiando lingua si salva il nome della categoria nella lingua presente al momento.

Figura 2: Finestra gestione categorie

Una volta inserito il nome della categoria in tutte le lingue si preme il pulsante *<<Aggiungi* che attiva la procedura di creazione di una nova categoria nel file XML; il metodo crea un nuovo tag *categoria* contenente i tag *cat* che contengono i nomi della categoria nelle varie lingue, ogni tag *cat* ha un attributo *lingua* che indica la lingua del nome; ogni *categoria* ha un attributo *iden* che corrisponde ad un identificativo usato nel file XML delle immagini per indicare di quale categoria fa parte un'immagine:

```
//creazione nuovo nodo
int lastName = new
Integer(categorie.getLastChild().getAttributes().item(0).getNodeValue());
lastName++;
Element categoria = documento.createElement("categoria");
Attr iden = documento.createAttribute("iden");
iden.setValue(""+lastName);
categoria.setAttributeNode(iden);
for(int j=0;j<lu;j++){
        Element cat = documento.createElement("cat");
        Attr ling = documento.createAttribute("lingua");
        ling.setValue(lin.item(j).getFirstChild().getNodeValue());
```

```
        cat.setAttributeNode(ling);
        cat.appendChild(documento.createTextNode(c[j]));
        categoria.appendChild(cat);
}
categorie.appendChild(categoria);
```

Codice 15: Metodo per la creazione di una nuova categoria in CreaCat.java

Dopo di che si salva sul file XML categorie.xml

3.2.3 L'organizzatore di immagini

Il costruttore della classe *Organaizer* crea la lista delle informazioni delle immagini dal file XML immagini.xml e crea un oggetto di tipo *ModifyVew* o *AddVew* in base all'operazione passata dal metodo *startOp* della classe *Start*.
Organaizer mette a disposizione i metodi per l'aggiunta, la modifica, e la rimozione di informazioni; l'aggiunta e la modifica inizialmente creano un nuovo tag *immagine* contenente gli altri tag delle informazioni relative all'immagine, l'aggiunta aggiunge questo tag al file, mentre la modifica sostituisce il vecchio tag con quello nuovo:

```
//aggiunge una nuova immagine alla NodeList
 public void aggiungiImmagine(String p,AbstractVew im,int nI,String t){
        Element newIm = creaImmagine(p,im,nI,t);
        radice.appendChild(newIm);
        salva();
 }

//Modifica un nodo già presente sostituendolo con il nuovo nodo con le nuove informazioni
public void modificaImmagine(int i,String p,AbstractVew im,int nI,String t){
        Element newIm = creaImmagine(p,im,nI,t);
        radice.replaceChild(newIm,radice.getChildNodes().item(i));
        salva();
 }
```

Codice 16: Metodi per l'aggiunta e la modifica di informazioni in AbstractVew.java

3.2.4 Aggiungere un immagine

Il costruttore di questa classe avvia la procedura di apertura di un immagine e settaggio dell'interfaccia utente; la scelta dell'immagine da aprire viene fatta tramite il metodo *showOpenDialog(this)* della classe *JFileChooser*:

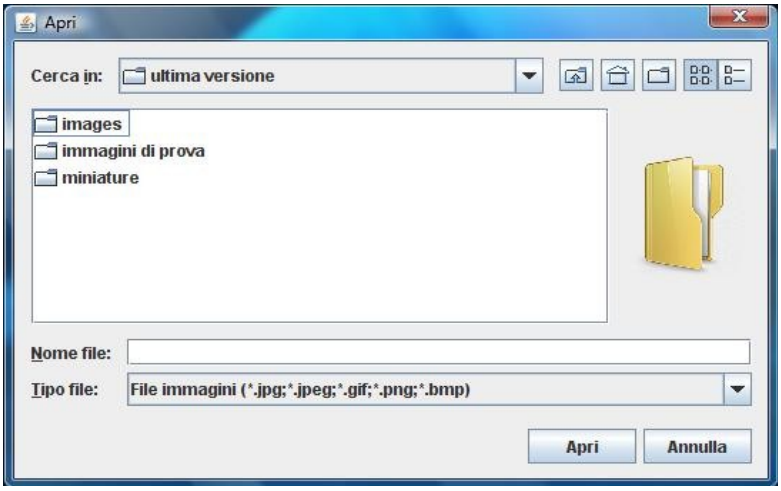

Figura 3: Finestra di navigazione dei file

Una volta scelta l'immagine si apre la finestra di interfaccia che permette di navigare tra le immagini della stessa cartella, cambiare cartella, dare un titolo all'immagine (inizializzato con il nome dell'immagine), aggiungere una descrizione e aggiungere le categorie di cui si vuole che l'immagine faccia parte.

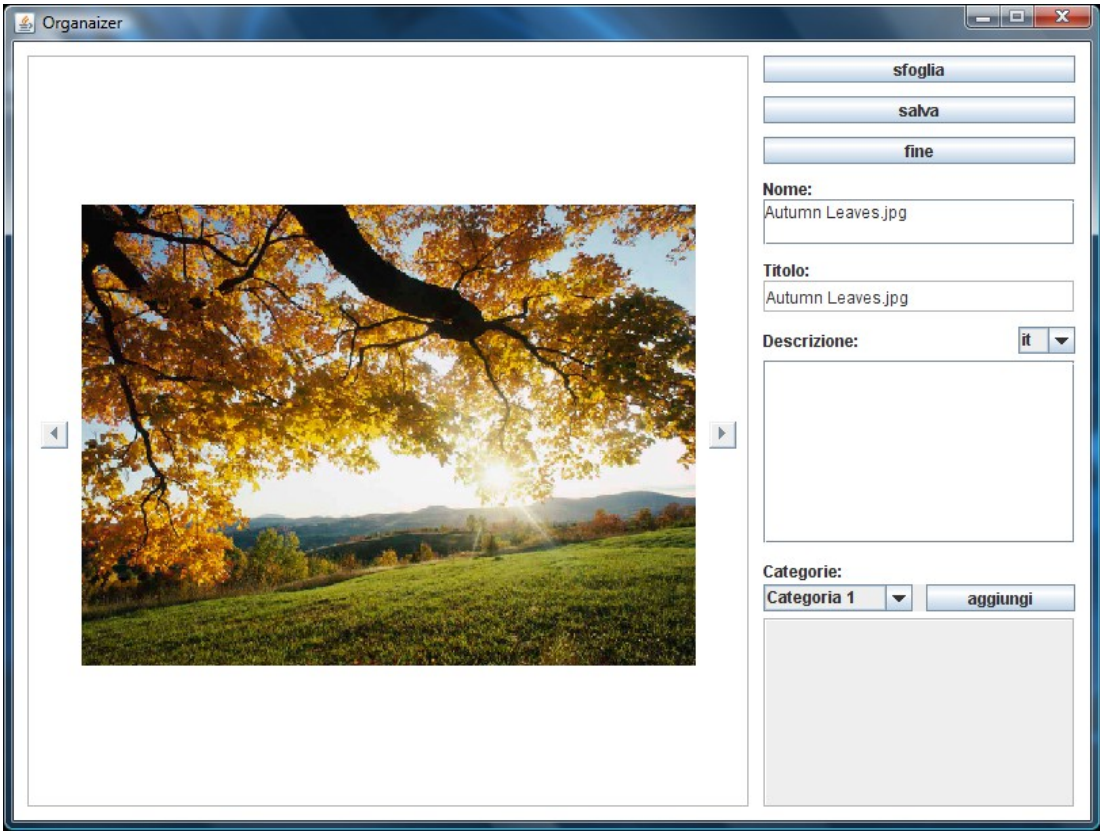

Figura 4: Finestra per aggiungere nuove immagini

Quando un'immagine viene salvata oltre a salvare le informazioni inserite, il percorso del file e i meta dati, si crea una miniatura dell'immagine in modo da creare un anteprima, la miniatura è un immagine .gif denominata con l'identificativo dell'immagine originale; l'identificativo è un numero progressivo per identificare immagini diverse, che potrebbero avere lo stesso nome, ogni nuova immagine prende come numero identificativo quello dell'ultima immagine salvata + 1.

3.2.5 Modificare le informazioni salvate

Questa classe apre l'interfaccia utente, che permette di navigare tra le immagini già salvate, modificare titolo e descrizioni, aggiungere o togliere categorie e rimuovere immagini (le informazioni nel file XML).

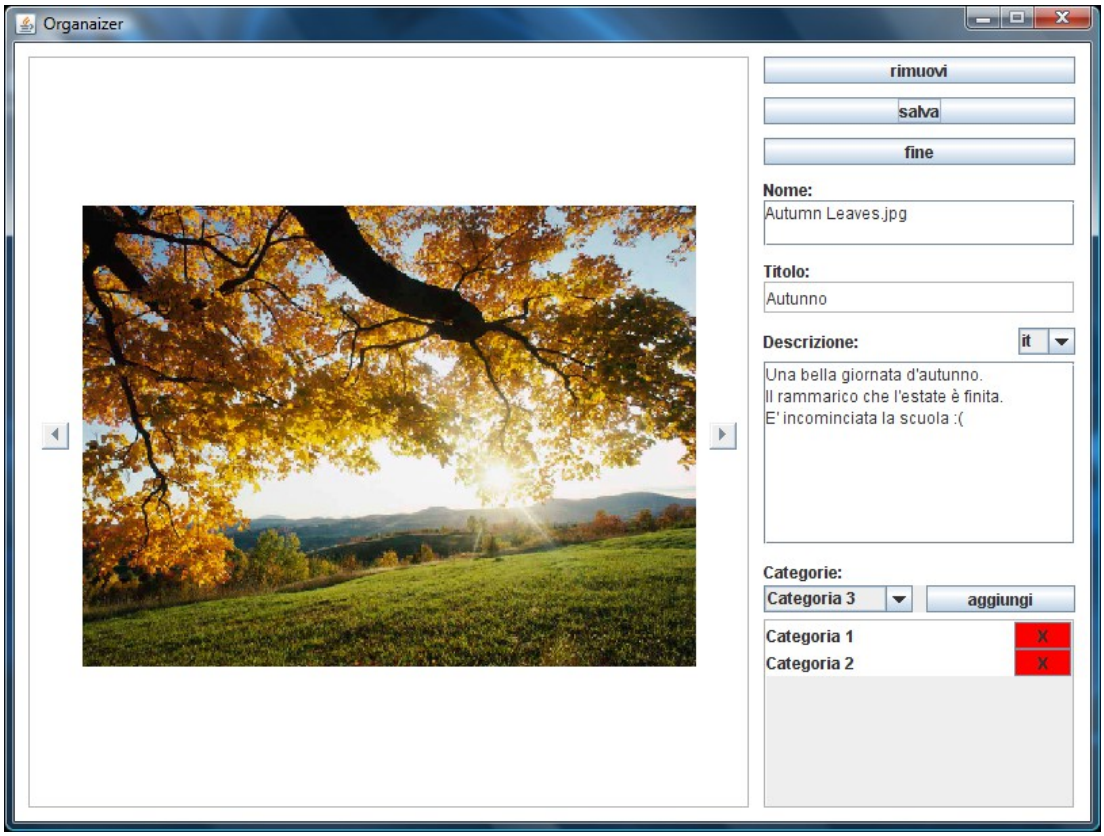

Figura 5: Finestra per modificare le informazioni

3.3 Visualizzare le immagini

La visualizzazione delle immagini, e delle relative informazioni, viene fatta tramite pagine HTML e fogli stile XSL.
I fogli stile XSL definiscono la posizione delle immagini e delle informazioni, mentre le pagine HTML controllano il contorno.

Al caricamento della pagina viene attivata la funzione dello script interno *applica();* questa funzione controlla se ci sono parametri nella query-string e li salva in un array, *param-value;* dopo di che carica i file XML, carica gli alberi in memoria, e il file XSL con la funzione *caricaFileXML;* salvando i file in diverse variabili.

Dopo di che crea un collegamento alle variabili del foglio XSL per poterle modificare.
Adesso se c'erano parametri questi modificano le variabili del foglio XSL; dopo di che si caricano le lingue sul menu a tendina tramite le istruzioni:

```
for(i=0;i<lLingue.length;i++){          //lLingue è la lista dei tag <lingua> del file XML lingue.xml
        var nodo= lLingue[i];
        select = document.forms.modulo.elements.lingua;          //punta al menù a tendina
        var lingua = nodo.firstChild.nodeValue;
        select.options[i] = new Option(lingua, lingua);
                                //crea una nuova opzione con valore il nome della lingua
        if(lingua==linguas)                    //linguas indica la lingua selezionata
                select.options[i].selected=true;                    //selezionare linguas
}.
```

Codice 17: Metodo per aggiungere le lingue al menu a tendina su immagini.html

In fine, in base alla lingua selezionata, si caricano le categorie dal file XML categorie.xml e si inserisce il titolo letto dal file XML MenuWeb.xml tramite la funzione *changeL*, che modifica la variabile relativa alla lingua nel foglio XSL, richiamata anche quando si cambia lingua dal menu a tendina.

Fatto questo si crea l'elenco delle immagini da visualizzare che poi viene usato dallo script esterno per la visualizzazione in primo piano di sei immagini alla volta, questa funzione è richiamata anche quando si cambia lingua o categoria, per modificare la lista di immagini da visualizzare.

Alla fine si carica il foglio stile sulla pagina con la funzione *disegna*, richiamata anche quando si cambia lingua o categoria in modo da visualizzare i cambiamenti:

```
function disegna() {
        if (window.ActiveXObject) {
        //IE
                ex=documento.transformNode(fogliostile);
                document.getElementById("pagina").innerHTML=ex;
        }
        else if (document.implementation && document.implementation.createDocument) {
        // Firefox
                xsltProcessor=new XSLTProcessor();
                xsltProcessor.importStylesheet(fogliostile);
              resultDocument = xsltProcessor.transformToFragment(documento,document);

                var newPage = document.createElement("div");
                newPage.setAttribute("id", "pagina");
                newPage.appendChild(resultDocument);

                var oldPage = document.getElementById("pagina");
                var page = oldPage.parentNode;
                page.replaceChild(newPage, oldPage);
        }
}.
```

Codice 18: Metodo per visualizzare il foglio stile XSL in immagini.html

Il cambio della categoria invece richiama la funzione *changeC* che modifica la relativa variabile nel foglio XSL:

```
function changeC() {
        var categoria = fogliostile.createTextNode(document.modulo.cat.value);
        c=parseInt(document.modulo.cat.value);
        variabili[0].replaceChild(categoria, variabili[0].firstChild);
        setList();
        disegna();
}
```

Codice 19: Metodo per la gestione del cambio della categoria in immagini.html

La pagina iniziale appare vuota, senza immagini (*figura 7*), da qui si sceglie dai menu a tendina la lingua e la categoria (*figura 6*).
Una volta scelta la categoria il foglio stile XSL immagini.xsl dispone le miniature in fila andando a capo a fine pagina; in testa alla pagina viene visualizzato uno script che permette di visualizzare 6 foto alla volta come su una mensola (*figura 8*).
Per lo script ho dovuto scaricare una libreria esterna costituita dai file *jquery-1.6.4.min.js* e *script.js*, per fare in modo che le immagini scorressero mentre venivano cambiate.
Per caricare lo script si usano le seguenti istruzioni inserite nel *body* del file HTML:

<script>!window.jQuery && document.write(unescape('%3Cscript src="jquery-1.6.4.min.js"%3E%3C/script%3E'))</script>
<script src="script.js"></script>

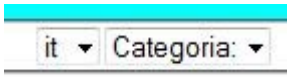

Figura 6: Menu

Figura 7: Finestra iniziale della visualizzazione

Figura 8: Finestra della visualizzazione dopo aver scelto la categoria

Ogni miniatura corrisponde ad un link che punta alla pagina dove vengono visualizzate l'immagine in dimensioni maggiori con le relative informazioni, a questa pagina vengono passate, attraverso la query-string:
-*l'iden* dell'immagine;
-la lingua selezionata;
-la categoria selezionata.

La pagina appena si carica salva queste variabili in un array, *param-value*, dopo di che carica i file XML e XSL con la funzione *caricaFileXML,* uguale a quella della pagina precedente, dopo di che modifica le variabili del foglio XSL secondo i valori di *param-value* e dei nomi delle didascalie letti dal file MenuWeb.xml; a questo punto crea un array con gli *iden* delle immagini appartenenti alla stessa categoria selezionata, e inizializza un puntatore alla posizione dell'immagine da visualizzare; infine scrive gli indicatori di immagine precedente, immagine successiva e il titolo nella lingua selezionata, e prepara il link di *home* in modo da passare alla pagina iniziale i valori di lingua e categoria scelti.
Ed infine richiama la funzione *disegna*, uguale a quella della pagina iniziale, per caricare il foglio stile sulla pagina.

Figura 9: Finestra per visualizzare le informazioni dell'immagine

Gli indicatori di immagine precedente e immagine successiva se premuti avviano le funzioni corrispettive per aggiornare il puntatore dell'array delle immagini e quindi passare ad un'altra immagine.

Capitolo 4
Conclusioni

Il lavoro svolto in questa tesi riguarda lo sviluppo di un'applicazione per l'organizzazione in categorie di immagini con relative descrizioni multilingua, e la visualizzazione di queste tramite pagine web.
Per fare questo ho usato l'XML per salvare le informazioni delle immagini, Java per lo sviluppo dell'applicazione per la scelta delle immagini e l'inserimento o la modifica delle informazioni, e XHTML, CSS, XSL per la visualizzazione con pagine web.
Durante lo sviluppo dell'applicazione Java ho dovuto studiare come scrivere e leggere dei file XML, perchè Java di default riesce a leggere e scrivere i file senza indentazione, ma per rendere leggibili i file anche da parte dell'utente ho dovuto trovare come scriverli con indentazioni e ho dovuto aggiungere le DTD perchè Java potesse leggerli senza problemi; come salvare immagini ridimensionate, perchè per salvare un immagine questa deve essere un oggetto del tipo *BufferedImage* mentre il metodo per il ridimensionamento non restituiva questo tipo, e l'estrazione dei metadata di un'immagine, per cui ho dovuto scaricare una libreria esterna e trovare della documentazione.
Utilizzando tecnologie client side, che quindi vengono interpretate in modo diverso da browser differenti, ho dovuto risolvere alcuni problemi di compatibilità.
Ho cercato di mantenere la compatibilità tra Firefox e Internet Explorer, ma mi sono concentrato soprattutto su Firefox per fare in modo che l'applicazione fosse il più possibile multipiattaforma, visto che Firefox è disponibile sia per Windows che per Linux.
Il progetto si può anche ampliare, infatti adesso funziona correttamente solo su postazioni locali, ma bastano poche modifiche per consentire anche la visualizzazione in Internet, siccome si usano tecnologie client side e quindi si è slegati dal provider di spazio web.
Sarebbe utile poter fare l'inserimento delle categorie e delle descrizioni multilingua in maniera automatica, invece di doverle inserire a mano, ma il problema di traduzione automatica non è di facile soluzione e richiederebbe un lungo studio.

Bibliografia

[1] Scorzoni F.: Internet e il WWW, 2^ ed., Loescher, (2008).

[2] Scorzoni F.: La programmazione in Java, Loesher, (2009).

[3] METADATA EXTRACTOR, http://www.drewnoakes.com/code/exif

[4] API METADATA EXTRACTOR, http://metadata-extractor.googlecode.com/svn/trunk/Javadoc/index.html

[5] HTML.it, http://www.html.it/

[6] http://java.sun.com

[7] API Java, http://docs.oracle.com/javase/6/docs/api

[8] http://www.w3.org