Università degli studi di Milano

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE e NATURALI
CORSO DI LAUREA IN SCIENZE E TECNOLOGIE DELLA COMUNICAZIONE MUSICALE

"PLUGIN C# PER LA GESTIONE DI UN REPOSITORY IEEE 1599"

Matteo Radice

Elaborato finale

Relatore:
Prof. Luca Andrea Ludovico

Correlatore:
Dott. Adriano Baratè

Anno accademico 2008/2009

Indice

Introduzione....5
Obbiettivi dell'elaborato....5
Capitolo 1....7
1.1 Lo standard IEEE 1599-2008....7
1.1.1 Panoramica generale sullo standard IEEE 1599-2008....7
1.1.2 L'importanza dei metadati nella conservazione dell'informazione....10
1.2 Il Repository IEEE 1599....13
1.2.1 Logica e struttura del Repository IEEE 1599....13
1.2.2 Obbiettivi dei plugin di gestione del Repository....14
Capitolo 2....17
2.1 .NET Framework 3.5....17
2.1.1 Panoramica generale sulla tecnologia .NET....17
2.1.2 Struttura a layer della tecnologia e del Framework .NET....18
2.1.3 Common Language Runtime....19
2.1.4 Il linguaggio C#....20
2.1.5 Manipolazione di documenti XML in C#....21
2.2 ADO.NET ODBC....23
2.2.1 Panoramica generale su ADO.NET....23
2.2.2 La connessione ODBC....25
2.2.3 PSQLODBC 27
2.3 PostgreSQL....28
2.3.1 Panoramica generale su PostgreSQL....28
2.3.2 Generazione di numeri seriali in sequenze....29
2.3.3 Multi Version Concurrency Control....30
2.4 XPATH....33
2.4.1 Panoramica generale su Xpath....33
2.4.2 Query XPath in PostgreSQL....34
2.4.3 Query XPath in C#....35
Capitolo 3....37
3.1 Uploader....37
3.1.1 Connessione al database....38
3.1.2 Salvataggio dei profili utente....39
3.1.3 Integrazione di PluginUpload/Uploader nel Framework....45
3.1.4 Caricamento dei documenti IEEE 1599-2008 nel repository....48
3.2 Downloader....52
3.2.1 Interrogazione del repository per la ricerca dei documenti IEEE 1599....52

3.2.3 Visualizzazione tabulare dei risultati delle interrogazioni....62
3.2.4 Download dei documenti IEEE 1599-2008....64
3.2.5 Integrazione di PluginDownload / Downloader nel Framework....67
3.3 Deleter....69
3.3.1 Eliminazione di un documento IEEE 1599 dal repository....69
3.3.2 Integrazione di PluginDelete/Deleter nel Framework....72
3.4 Updater....73
3.4.1 Sostituzione di un documento IEEE 1599 nel repository....74
3.4.2 Integrazione di PluginUpdate/Updater nel Framework....76
Conclusioni....77
Risultati ottenuti e sviluppi futuri....77
Bibliografia....78
Sitografia....79
Ringraziamenti....81

Indice delle illustrazioni

Fig. 1: Layer di un documento IEEE 1599.....9
Fig. 2: Struttura del Repository IEEE 1599.....18
Fig. 3: Struttura a livelli dell'ambiente .NET.....21
Fig. 4: Struttura a livelli del Framework .NET.....22
Fig. 5: Connessioni ADO e ADO.NET a basi di dati.....26
Fig. 6: Connessione ODBC a una base di dati.....28
Fig. 7: Tabella dei livelli di isolamento di una transazione SQL.....33
Fig. 8: Maschera di connessione e caricamento dei documenti in Uploader.....39
Fig. 9: Maschera di ricerca dei documenti in Downloader.....55
Fig. 10: Visualizzazione tabulare dei risultati della ricerca in Downloader.....66
Fig. 11: Visualizzazione tabulare dei risultati della ricerca durante l'eliminazione di un file in Deleter.....72
Fig. 12: Confronto fra documento conservato nel Repository e nuovo documento aperto nel Framework in Updater.....75

Introduzione

Obbiettivi dell'elaborato

La generale affermazione del formato IEEE 1599 per la descrizione multistrato dell'informazione musicale e l'espansione del numero di tesi e progetti che fanno affidamento su questa nuova tecnologia hanno posto un problema relativo agli stessi documenti IEEE 1599: come conservarli e renderli fruibili a chi li deve utilizzare per lo sviluppo di nuovi progetti?
Da questa necessità è nato un progetto suddiviso in tre parti per la creazione di un Repository IEEE 1599 che contenesse e catalogasse automaticamente i documenti inseriti al suo interno, gestito utilizzando una serie di plugin sviluppati per il Framework IEEE 1599 (ambiente di scrittura ed elaborazione dei documenti IEEE 1599) e consultabile dal sito ufficiale del formato (www.mx.unimi.dico.it).
Questo elaborato tratta della progettazione e dello sviluppo di questo set di plugin, programmati in C#. Lo stage relativo a questo progetto è stato svolto presso il Laboratorio Informatica Musicale (LIM) dell'Università degli Studi di Milano. Proprio le ricerche effettuate nel corso degli ultimi anni dal LIM hanno dato origine alla creazione del formato MX, ora conosciuto come IEEE 1599 dopo la sua standardizzazione.
Durante questa esperienza e il relativo sviluppo di questi software è stato possibile entrare in contatto con diversi temi di importanza centrale nella progettazione di soluzioni a problemi reali come la gestione delle informazioni, la manipolazione dell'XML e l'interazione con basi di dati. Inoltre, è stato possibile approfondire il funzionamento del Framework IEEE 1599 e la struttura dei documenti IEEE 1599.

Il progetto ha visto principalmente due fasi: una iniziale di definizione delle necessità degli utenti (in primis dei ricercatori del LIM) nella ricerca, importazione ed esportazione dei documenti in funzione delle tecnologie scelte per la realizzazione e delle specifiche della base di dati con la quale interagire; una seconda fase di sviluppo vero e proprio che ha portato alla realizzazione di 4 plugin (Uploader, Downloader, Updater, Deleter) testati e funzionanti.

Nel primo capitolo di questo elaborato si è cercato di evidenziare l'importanza generale dei metadati ai fini di conservazione e localizzazione dell'informazione, in particolare quella musicale, e di come il sistema di metadati presente all'interno degli strati di un documento IEEE 1599 possa essere sfruttato dalla base di dati e di conseguenza dal sistema di interrogazioni implementato nei plugin per fornire all'utente un modo completo per ritrovare ed utilizzare i documenti inseriti. Ciò è stato reso possibile da una sorta di mappatura automatica dei contenuti di un documento IEEE 1599 sulle tabelle che costituiscono la struttura della base di dati, che trasforma quindi informazioni contenute in file XML in dati SQL per un database relazionale.

Nell'intero progetto è stato fatto largo uso di metodi di visita delle strutture XML dei documenti IEEE 1599 e più in generale di schemi XML utilizzati per risolvere alcuni problemi (come la gestione dei profili di connessione salvati localmente sulla macchina dell'utente). In particolare, la tecnologia più utilizzata a questi scopi è stato il sistema XPath 1.0, del quale vengono spiegate caratteristiche e funzionalità nel secondo capitolo, insieme a visioni d'insieme delle altre tecnologie informatiche impiegate, cioè l'ambiente Microsoft .NET 3.5, la connessione ad una base di dati tramite ADO.NET e driver ODBC e le caratteristiche di una base di dati realizzata con PostgreSQL.

Nel capitolo finale viene illustrata nei dettagli la realizzazione pratica dei quattro plugin, con spiegazioni esaustive sulle scelte fatte sia in termini di raggiungimento degli obbiettivi fondamentali di ogni programma sia riguardanti le implementazioni all'interno del già esistente Framework IEEE 1599 sviluppato dal LIM.

Capitolo 1

1.1 Lo standard IEEE 1599-2008

1.1.1 Panoramica generale sullo standard IEEE 1599-2008

La musica, come forma d'arte, è rappresentabile in diversi modi. Questo polimorfismo rispecchia l'eterogeneo bacino d'utenza che la musica può avere: dall'ascoltatore semplice a quello con una educazione musicale, dal ricercatore al musicologo, dallo studente al musicista. Queste figure hanno diverse esigenze e spesso necessitano di poter usufruire dell'informazione musicale attingendo contemporaneamente da più rappresentazioni. Ciò porta ad avere diversi oggetti multimediali, con la conseguente necessità di disporre di diversi metodi di descrizione delle informazioni.
Tutti i documenti in qualche modo connessi ad un brano musicale contribuiscono a descriverne le caratteristiche in maggiore dettaglio: le campagne di digitalizzazione delle fonti autografe e in generale dei media fisici relativi alla musica sono attualmente una realtà consolidata costituendo una grande quantità di dati dai quali attingere.
L'informazione musicale necessita quindi di un metodo che racchiuda in sé la sua più completa descrizione, sia per quanto riguarda il contenuto strettamente musicale sia per quello riguardante i metadati che lo caratterizzano. Deve quindi poter includere la rappresentazione semantica dei testi e delle partiture, immagini digitali non solo degli spartiti ma di tutto il materiale iconografico utile alla caratterizzazione del brano, rappresentazioni in file audio e video di performance di quel brano musicale. I formati di file atti a queste rappresentazioni singole esistono già. Il formato che permette alle diverse codifiche di interagire fra loro offrendo una visione panoramica sincronizzata di queste rappresentazione di un brano è lo standard IEEE 1599-2008, interamente basato sul linguaggio XML.

Lo standard è stato creato dal Laboratorio di Informatica Musica dell'Università degli Studi di Milano e conosciuto fino alla fase finale della sua standardizzazione IEEE (iniziata nel 2001 e conclusasi a luglio del 2008) con il nome di MX o MAX, "Musical application using XML". [1]

Le ragioni per le quali il formato è stato sviluppato in XML sono molteplici e in linea con altri progetti che nel corso degli anni hanno dimostrato come il linguaggio XML sia adatto a questo tipo di applicazione, ma possono essere ridotte alle tre caratteristiche più importanti: leggibilità, estensibilità e durabilità.

Rappresenta l'informazione in maniera modulare e gerarchica, in un modo nel quale le relazioni interne fra i vari moduli sono esplicite.

Lo standard IEEE 1599 stratifica quindi la rappresentazione di un brano su sei diversi layer. Al loro interno, vengono accolte tutte le rappresentazioni accennate precedentemente e viene fornito il supporto per la loro sincronizzazione.

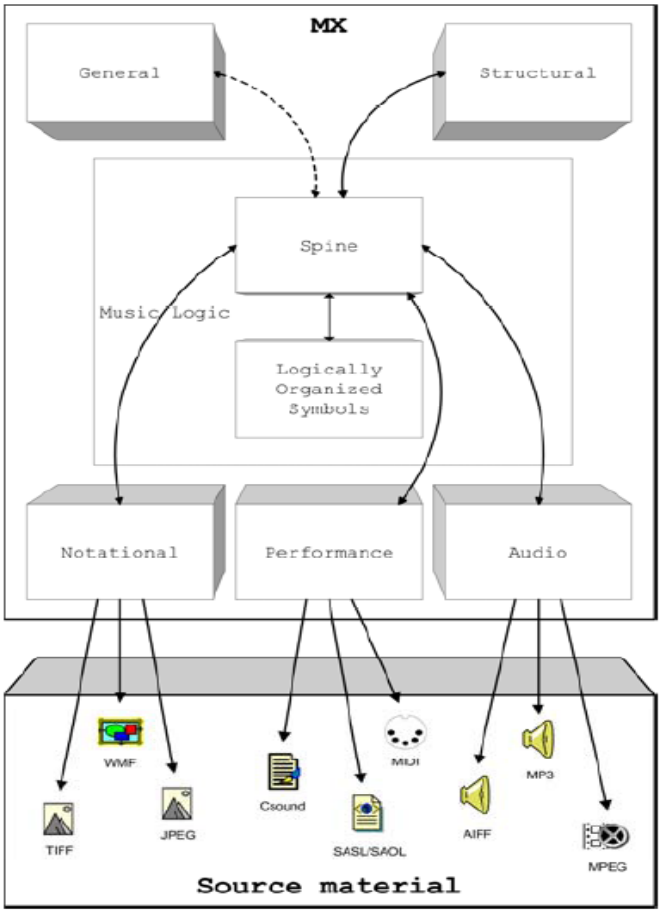

Fig. 1: Layer di un documento IEEE 1599

Il layer General contiene metadati relativi al brano, all'opera, all'autore, informazioni quindi fondamentali per l'identificazione del brano, anche se non è un layer obbligatorio per avere un documento IEEE1599 considerato valido.
Il layer Logic è in realtà composto da tre strutture fondamentali, "spine", "los", e "layout". Lo "spine" è la struttura attraverso cui tutte le risorse di un documento IEEE1599 sono sincronizzate fra loro, è una mappatura spazio-temporale. Il "los" descrive la partitura descrivendone gli oggetti musicali che la compongono mentre "layout" definisce l'impaginazione della partitura. E' un layer fondamentale.
Il layer Structural raccoglie le segmentazioni e le rappresentazioni del brano dal punto di vista dell'analisi musicologica, come quella basata su un modello a Reti di Petri.
Il layer Notational raccoglie le istanze grafiche del brano rappresentato dal documento, sia in formato notazionale, come può essere un file NIFF, sia in formato puramente grafico, cioè scannerizzazioni di partiture cartacee in formato jpg per esempio.
Il layer Performance è dedicato alle rappresentazioni che permettono ad un elaboratore di dare una performance in tempo reale sintetica del brano, come per esempio può fare un file MIDI, SASL/SAOL, Csound.
Infine, il layer Audio raccoglie tutte i file audio, compressi e non compressi, che di fatto rappresentano una registrazione del brano rappresentato dall'intero documento[1]. A queste registrazioni sono connessi tutti i relativi metadati, che insieme a quelli descritti nei nodi figli del tag "description" del layer General, costituiscono la fonte principale di dati di catalogazione all'interno dei documenti IEEE 1599.
Non solo lo standard, attraverso l'elasticità fornita dal linguaggio XML, permette una minore o maggiore ricchezza di informazioni considerando gran parte dei layer e del loro contenuto facoltativi: la presenza di più istanze appartenenti allo stesso layer (si pensi a più registrazioni nel layer Audio, o più istanze grafiche in quello Notational) è ugualmente supportata e incoraggiata. [1]
I file multimediali che costituiscono le istanze grafiche e notazionali del layer Notational e le rappresentazioni del layer Audio e Performance sono ovviamente contenute in file singoli esterni al documento XML IEEE1599. Vengono però linkate allo stesso, come risorse esterne da usare nei modi indicati dal documento stesso.

Considerando solo il documento XML IEEE1599 in sé quindi, si evince che la sua estrema portabilità e flessibilità sono molto adatte alla diffusione in rete di questo tipo di documenti. Diventa quindi necessario poterli conservare e renderli disponibili in maniera agevole, sfruttando al meglio le possibilità del formato per applicazioni concrete riguardanti la conservazione di

informazione musicale a scopo di studio. Ciò è realizzabile attraverso un progetto di un Repository di documenti IEEE1599 accessibile via web e gestibile attraverso plugin realizzati per il Framework IEEE 1599.

1.1.2 L'importanza dei metadati nella conservazione dell'informazione

Un metadato è una informazione che descrive le caratteristiche di un insieme di dati o di un oggetto multimediale, specificandone per esempio il contenuto, caratteristiche tecniche, autore e data di creazione: fornisce quindi dei criteri secondo i quali un oggetto può essere conservato e catalogato. Estendendo il concetto, è realistico affermare che i metadati sono gli elementi fondamentali per garantire la completa conservazione e accessibilità nel tempo ad una informazione di qualunque genere: sono insiemi di informazioni strutturate che non solo spiegano il contenuto di una risorsa, ma la localizzano e ne permettono la consultazione [2]. Esistono diverse concezioni di tipologie di metadati e dei loro compiti specifici.
Secondo le indicazioni NISO, è possibile identificare tre tipi principali di metadati: i metadati descrittivi, i metadati strutturali e i metadati amministrativi. I metadati descrittivi caratterizzano la risorsa per il suo contenuto allo scopo di identificare la risorsa (titolo, autore, argomento trattato ecc ecc) e di renderla accessibile in seguito ad una ricerca mirata ai contenuti [2]. I metadati strutturali invece descrivono come sono organizzate risorse composte da più elementi: ne caratterizzano quindi la struttura interna.
I metadati amministrativi infine assolvono lo scopo di definire i metodi di gestione di quella risorsa, con dati tecnici riguardanti tipo di file, tecnologie coinvolte, data di creazione. Questa categoria può avere due ulteriori specializzazioni: metadati di gestione dei diritti, relativi quindi al diritto d'autore della proprietà intellettuale della risorsa; metadati di preservazione, che contengono dati tecnici per la corretta conservazione della risorsa. [2]
Un buon sistema di metadati deve quindi facilitare la localizzazione di risorse cercate secondo criteri significativi, l'identificazione univoca di tali risorse ma anche il collegamento fra risorse con elementi comuni. L'uso dei metadati è di importanza chiave alla preservazione delle stesse risorse digitali, che vengono percepite come fragili al continuo evolversi e cambiare delle tecnologie.[2]

Se i metadati sono di così vitale importanza su singole risorse, assumono proporzioni ben più grandi se si pensa all'ammontare di informazioni elencate nel precedente paragrafo riguardo le molteplici rappresentazioni di un singolo brano musicale. Se poi si considera il caso di un documento IEEE 1599, è necessario tenere in considerazione sia i possibili metadati relativi al puro contenuto

musicale sia quelli relativi alle sue istanze notazionali, grafiche e audio, per esempio. Partendo dal layer General, il suo scopo è proprio quello di definire una serie di semplici metadati descrittivi come l'autore, il titolo del brano, l'opera o la collezione di brani alla quale appartiene, i generi musicali coinvolti. Tutti questi metadati descrittivi sono raccolti nella parte Description del layer General. Per quanto riguarda gli oggetti multimediali collegati a un documento IEEE1599 e linkati nei layer Audio e Notational, vengono forniti sia metadati descrittivi, che nel caso del layer Audio si estendono agli esecutori della registrazione, al genere musicale, all'album a cui la registrazione appartiene e alle etichette discografiche che pubblicano quella registrazione, sia a quelli amministrativi di gestione dei diritti. Per il layer Notational è prevista la sola descrizione della casa editrice proprietaria della partitura, oltre al già citato metadato relativo al diritto d'autore. I criteri di ricerca atti alla localizzazione di un documento IEEE 1599 possono essere in realtà molto più estesi, prevedendo anche interrogazioni al repository in base, per esempio, al formato file degli oggetti multimediali relativi al layer Audio conservati nella base di dati. Tali interrogazioni sono sicuramente meno frequenti ma ugualmente utili soprattutto agli sviluppatori di plugin per il Framework IEEE1599.

```
<general>
  <description>
    <main_title>Gottes Macht und Vorsehung</main_title>
    <author type="composer">Beethoven, Ludwig van</author>
    <author type="poet">Christian Fürchtegott Gellert</author>
    <number>5</number>
    <work_title>6 Geistliche Lieder Von Gellert</work_title>
    <work_number>op. 48</work_number>
    <genres>
      <genre name="Vocal music" />
      <genre name="Romanticism" />
    </genres>
  </description>
</general>
```

Tornando all'esempio del layer General, è importante notare quale sia l'effettivo scopo dell'elemento Description: la sua struttura è composta in modo da dare una descrizione generale del brano, nel modo in cui il compositore originale l'ha concepito. [3] Nel layer Audio è possibile trovare una simile struttura, che può assumere ovviamente contenuti differenti rispetto a quelli descritti in General proprio perchè si riferiscono a una particolare interpretazione registrata del brano in questione. [3] E' dunque possibile fare una distinzione fra metadati ridondanti e quindi relativi alla composizione nella sua accezione originale che pertanto non vengono rieptuti e tra metadati relativi alla versione registrata del brano; fra questi è possibile ritrovare il genere musicale, per adattarsi per esempio a quelle reinterpretazioni di un pezzo musicale eseguite secondo stili differenti

dall'originale. Questa parziale differenziazione fra i metadati attribuiti al layer Audio e gli elementi di Description non è che una prima discrepanza in uno schema che se da una parte è assolutamente completo dall'altra sembra prestarsi maggiormente alla rappresentazione di composizioni più tradizionali (si pensi alla terminologia Work e Movement per esempio) che a musica pop contemporanea e si basa su descrizioni, inserite nell'elemento "genre", dettate dal libero arbitrio più che da una convenzione. Questo elemento potrebbe rifarsi a diverse interpretazioni personali di ciò che il concetto di genere musicale comporta, portando a descrizioni riguardanti comuni classificazioni, classificazioni musicologiche, considerazioni riguardanti il periodo storico o l'area geografica dal quale proviene il brano. Se da una parte un uso libero di questo spazio per la descrizione del genere musicale del brano (abbinato ai pesi percentuali di ciascuna descrizione) arrichisce l'ammontare di informazioni legate al brano stesso, dall'altra genera difficoltà nell'interrogazione della base di dati per la ricerca di brani musicali in base a un criterio di ricerca che verte su tale parametro. [3]

Nonostante queste limitazioni della struttura di metadati offerta da un documento IEEE 1599 si è ritenuto comunque sufficiente per implementare attraverso plugin per il Framework IEEE1599 funzioni di ricerca dei documenti conservati nel repository utilizzando interrogazioni che vertono in gran parte sui metadati descrittivi contenuti principalmente nel layer General e nel layer Audio.

1.2 Il Repository IEEE 1599

1.2.1 Logica e struttura del Repository IEEE 1599

Parte centrale di un progetto che mira alla realizzazione di un sistema di conservazione e ricerca di documenti IEEE 1599 è ovviamente la base di dati nella quale i documenti vengono inseriti. Lo scopo di questo Repository non è solo conservare i documenti, ma estrapolarne dati significativi ai fini di utilizzarli come criteri di ricerca: vengono presi dati rappresentati in strutture Xml ed esportati verso uno schema relazionale scritto in SQL.

L'estrapolazione ed elaborazione di tali dati deve essere necessariamente automatica per garantire la massima coerenza tra ciò che è stato inserito nelle varie tabelle e ciò che effettivamente è contenuto nel documento. Questa automazione è svolta attraverso procedure automatiche scatenate da comandi SQL, cioè i trigger. Una analoga procedura provvede alla corretta eliminazione dei dati derivati da un documento cancellato.

La "mappatura" dei contenuti del documento non è comunque completa: piuttosto sono stati scelti gli elementi di maggiore utilità e interesse per il riconoscimento del documento stesso. Il Repository IEEE 1599 rispecchia nella sua struttura parte dell'organizzazione a layer di un documento IEEE 1599. Nella sua parte centrale ricrea l'elemento "description" del layer General; il documento in sè è l'elemento centrale di tutto il database, ed è contenuto nel campo "codice_xml" nella tabella "TFile_Mx". Questa tabella contiene gli elementi di General che hanno una occorrenza univoca e strettamente legati al brano, come "main_title", "other_title", "notes". Vengono aggiunti due metadati amministrativi esterni al documento IEEE 1599: "upload_date" e "filename", rispettivamente la data di caricamento nel Repository e il nome del file attribuito al documento caricato. Come in tutte le altre tabelle, come chiave primaria viene dato un numero generato automaticamente dal database, un ID.

I metadati contenuti in Description vengono organizzati in relazioni, creando quindi altre tabelle: “TAuthor”, che contiene il nome degli autori, “TAuthor_type”, che specifica il ruolo di quell'autore in quel particolare brano prendendo il suo valore dall'attributo "type" del tag "author"; "TFile_Work" e "TWork" rispecchiano gli elementi "work", "work number" e "number", indicando rispettivamente l'opera a cui il brano appartiene, il codice di catalogo dell'opera e la posizione occupata dal brano nella collezione alla quale appartiene; "TDate" specifica la data e l'evento

collegato ad essa (in funzione del brano) prendendolo dall'elemento "date" e dal suo attributo "type"; infine, la tabella "TGenre" contiene tutte i nomi inseriti nell'attributo "name" dell'elemento "genre" trovate nell'analisi dei documenti IEEE 1599 inseriti nel Repository e "TGenre_File" collega tali nomi al file nel quale sono usate con i relativi pesi percentuali (l'attributo "weight") e descrizione (l'attributo "description").
Oltre ai metadati contenuti in General, l'automazione del Repository provvede a raccogliere in tabelle anche i metadati ritenuti significativi contenuti negli altri layer. In generale, i metadati amministrativi degli oggetti multimediali presenti nel layer Performance riguardanti i file MIDI, nel layer Audio, nelle istanze grafiche del layer Notational; vengono cioè esplicitati i nomi e i formati di questi file, il path in cui risiedono, le codifiche utilizzate per la digitalizzazione.
"TMidi_Instance" e "Tgraphic_Instance" rappresentano rispettivamente gli oggetti MIDI contenuti nel layer Performance e le istanze grafiche del layer Notational: la rappresentazione nel Repository del layer Audio è invece molto più estesa. La sua suddivisione in 6 tabelle (più una di raccordo) è dovuta alla necessità di gestire sia uno schema di metadati che specifica, oltre ai già citati metadati amministrativi, gli artisti esecutori, il loro ruolo (in un modo analogo a quello dell'autore del brano nel layer General) in funzione di quella interpretazione registrata, l'album al quale appartiene (descritto a sua volta da una serie di metadati quali il titolo, la data di pubblicazione, il numero di catalogo, il numero di tracce), le etichette che pubblicano quella registrazione.

1.2.2 Obbiettivi dei plugin di gestione del Repository

L'interazione di un utente con la base dati descritta nel paragrafo precedente può avvenire sia via web, attraverso il sito ufficiale www.mx.dico.unimi.it nella sezione "Scores", sia direttamente nel Framework IEEE 1599. Si è scelto di dotare il Framework di plugin che permettessero a chi sviluppa progetti multimediali attraverso la scrittura di documenti IEEE 1599 di accedere alla base di dati e di gestirla direttamente dallo stesso ambiente.

La gestione della base di dati offerta da questi quattro plugin fornisce funzioni di caricamento, download, sostituzione ed eliminazione dei documenti IEEE 1599.

Queste funzioni quindi garantiscono un'importazione e una esportazione di documenti IEEE 1599 direttamente dal Framework IEEE 1599 verso il database previa connessione autenticata con la base dati.

La finestra di connessione deve dare modo all'utente di specificare tutti i parametri in modo da poter raggiungere il Repository. Tali parametri vengono salvati e gestiti in un file di profili in modo da essere richiamabili facilmente.

Per la ricerca deve essere implementata una maschera di ricerca che permetta all'utente di specificare criteri anche complessi e che attraversino tutti i layer analizzati dal trigger di autoinserimento del Repository IEEE 1599: i risultati di tali ricerche "componibili" devono essere visualizzati in una tabella che abbia come colonne i criteri scelti dall'utente e che permetta di intraprendere l'azione desiderata (download, eliminazione, sostituzione) sul documento IEEE 1599 in maniera veloce, semplicemente cliccando su bottoni posti alla fine di ogni riga.

L'interazione dei plugin con il Repository deve necessariamente usufruire dei trigger che si occupano dell'autoinserimento e dell'eliminazione: devono quindi essere in grado di scatenare tali procedure.

Sebbene siano presenti molte parti in comune (la finestra di connessione, la gestione dei profili, la finestra di ricerca e la base della visualizzazione) ogni plugin è perfettamente separato dagli altri e quindi in grado di funzionare nel Framework IEEE 1599 anche senza la presenza degli altri plugin di gestione. In termini di codice ciò significa che le parti comuni effettivamente sono ripetute in ciascuno dei plugin.

Il plugin di Upload dei documenti non influisce in alcun modo sui metadati ricavati dall'analisi del documento stesso, ma crea automaticamente il nome del file e la data di caricamento inserendoli nella tabella "Tfile_MX", come metadati amministrativi utili alla gestione dei documenti archiviati.

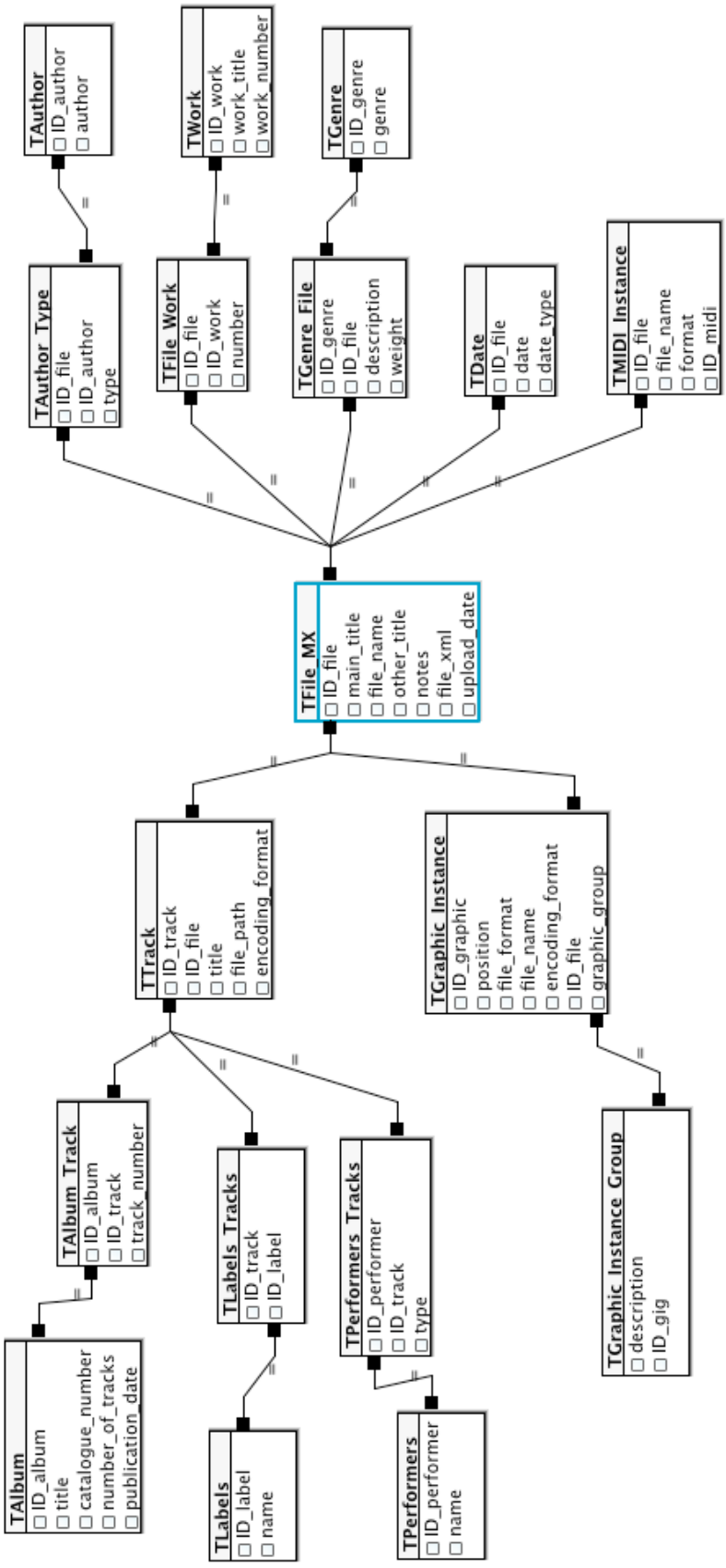

Fig. 2: Struttura del Repository IEEE 1599

Capitolo 2

In questo capitolo viene data una panoramica delle tecnologie informatiche utilizzate per la realizzazione dei plugin di gestione del repository. Tali plugin sono stati programmati usando il linguaggio C# all'interno dell'ambiente di sviluppo Microsft Visual Studio 2008. Sono applicazioni .NET framework 3.5 che si connettono a un database remoto realizzato con PostgreSQL tramite driver ODBC dedicati. I documenti XML IEEE 1599 vengono visitati dai plugin utlizzando le funzioni XPath 1.0, sulle quali si basa anche il trigger interno allo stesso repository.

2.1 .NET Framework 3.5

2.1.1 Panoramica generale sulla tecnologia .NET

Nel luglio del 2000 Microsoft annuncia il progetto .NET, una nuova tecnologia capace di integrare vari aspetti di programmazione emersi nel corso degli anni '90 e incentrata principalmente sul facilitare lo sviluppo di applicazioni e servizi per sistemi operativi Windows e le relative API. La piattaforma offre tra le altre feature un set di linguaggi, come C#, un ambiente di sviluppo applicazioni (Visual Studio .NET) e un motore di esecuzione degli oggetti costruiti all'interno del framework .NET, chiamato Common Language Runtime. [4]
I problemi che la tecnologia .NET risolve sono diversi ma accomunati dallo scopo di semplificare la programmazione e di ampliarne le possibilità, integrando aspetti differenti di vari metodi usati. Per esempio, la programmazione in C per servizi Windows presentava la difficoltà di dover gestire manualmente la memoria, forzava il programmatore all'utilizzo di scomodi puntatori e di strutture sintattiche complesse, al posto di un approccio ad oggetti più pratico; anche C++ non costituisce che un "layer" ad oggetti basato su C e quindi soffre delle stesse problematiche.[5]
Le caratteristiche della piattaforma .NET sono quindi l'alta interoperabilità con codice già esistente, la totale integrazione fra vari tipi di linguaggi, la condivisione di un singolo runtime tra tutti i linguaggi .NET, una vasta scelta di API e di classi di oggetti già predisposti per l'uso, la generale

facilità nella gestione delle DLL e della programmazione in generale [5], una maggiore sicurezza delle applicazioni sviluppate e la loro agilità e interoperabilità con altri software nelle reti locali e Internet [4].
I vantaggi di una programmazione ad oggetti sono chiari: al programmatore vengono fornite classi che rispondono a diverse esigenze, anche particolari, in modo da avere una maniera facile per implementare funzioni complesse basate su oggetti già esistenti con relativi attributi e metodi. [4]

2.1.2 Struttura a layer della tecnologia e del Framework .NET

La struttura della tecnologia .NET è rappresentabile a tre livelli. Al livello inferiore sono rappresentati i sistemi operativi Windows, sia per PC, sia per altri dispositivi; in diretto contatto col sistema operativo della macchina ci sono tre componenti, tutti allo stesso livello, a seconda dello scopo per il quale l'applicazione è stata sviluppata: sono il .NET Enterprise Servers, .NET Building Block Services e l'oggetto sul quale si vuole focalizzare l'attenzione, ovvero il .NET Framework. Al livello soprastante si trova l'ambiente di sviluppo dedicato alla tecnologia .NET, Visual Studio. [4]

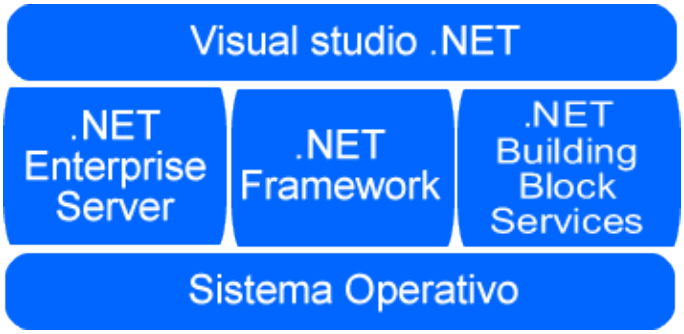

Fig. 3: Struttura a livelli dell'ambiente .NET

Lo stesso Framework .NET è rappresentabile a livelli: una tale rappresentazione permette di comprenderne il funzionamento attraverso la descrizione delle unità fondamentali, come la già citata CLR (Common Language Runtime).
Dal punto di vista di un programmatore infatti, l'intero Framework .NET è descrivibile come un ambiente di runtime unito ad una vasta libreria di classi base [5].
Il layer di runtime, CLR, si occupa principalmente di localizzare, caricare e gestire i tipi di oggetti .NET richiamati nell'applicazione, insieme alla completa gestione della memoria e dei controlli di sicurezza; comprende quindi sia funzioni primitive (come quelle di I/O per esempio) sia

servizi richiesti nella maggior parte delle applicazioni realizzate al giorno d'oggi [5].
Il blocco del Framework .NET è dunque composto da un livello inferiore, in diretto contatto con il sistema operativo, che è CLR; sopra di esso c'è la Base Class Library, la libreria di classi base contente i tipi di oggetti e di funzioni più comuni richiamabili all'interno del Framework. Al di sopra di questa libreria si trovano i set di classi che si occupano della gestione dei dati, della comunicazione con database e della manipolazione dell'Xml: rientrano in questo layer le classi per le interrogazioni SQL, il set di classi ADO.NET e le funzioni per la manipolazione, creazione e ricerca di dati Xml (comprendente quindi anche le funzioni di ricerca Xpath e le funzioni XSLT). Il livello superiore è infine composto da tre tipi differenti di classi: Web Forms, Web Service e Windows Forms[4]. Quest'ultima classe è stata usata per realizzare i plugin di gestione del repository.

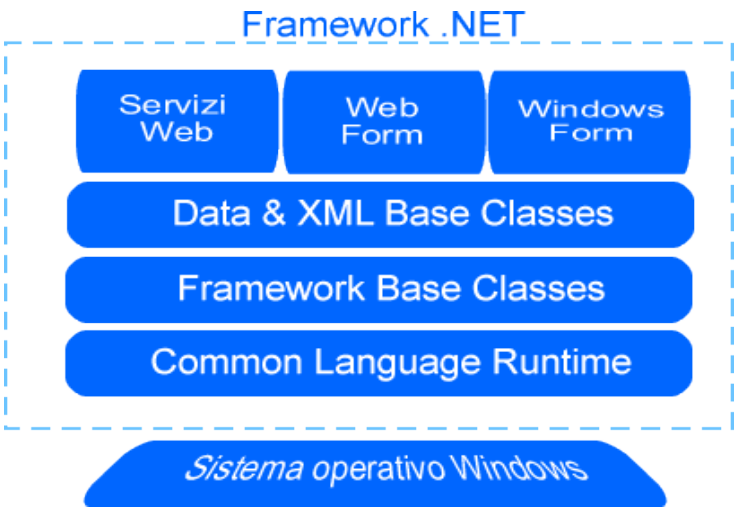

Fig. 4: Struttura a livelli del Framework .NET

2.1.3 Common Language Runtime

Il termine runtime indica un insieme di servizi esterni richiesti per la corretta esecuzione una unità di codice compilato. Lo stesso Common Language Runtime è un blocco composto da altre unità. All'interno di CLR si trova il Common Type System e all'interno di quest'ultimo il Common Language Specification[5].
Il Common Type System costituisce la parte del CLR che descrive nella loro completezza tutti i possibili tipi di dato e di costrutti di programmazione che l'ambiente di runtime supporta. Descrive

come queste diverse entità vengono rappresentate nel formato di metadati .NET e come le entità possono interagire fra di loro [5]. Il CTS rende quindi possibile l'interoperabilità tra i linguaggi supportati da .NET [4]. A questo scopo è adibito anche il Common Language Specification, all'interno del CTS. Il CLS specifica infatti una serie di regole da seguire per permettere una corretta integrazione dei linguaggi [4]: in altre parole, viene definito un sottoinsieme di tipi di dato e strutture di programmazione comuni in modo che siano adatte a tutti i linguaggi .NET. Quindi, dichiarando dati che cadono in questo sottoinsieme, l'interoperabilità dei vari linguaggi è garantita [5]. Sono stati citati i metadati .NET. Questi vengono usati per caratterizzare ed identificare i tipi di dato, strutture, oggetti, classi nel momento del runtime. Inoltre, questo sistema è richiesto per permettere la concreta descrizione del tipo ai fini dell'interoperabilità, dei servizi di compilazione offerti da Visual Studio e della tecnologia IntelliSense, il sistema dell'autocompilazione del codice all'interno dell'editor.[5]

In breve, l'esecuzione di CLR è per certi versi simile a quella di una Java Virtual Machine [4]. Entrambi i sistemi sono ambienti di runtime che gestiscono le differenze delle piattaforme sulle quali il programma compilato viene eseguito, garantendone alta compatibilità, a patto che, nel caso per esempio di .NET, CLR sia installato. Esistono distribuzioni (come Mono o Rotor) che creano sistemi di runtime CLR su sistemi operativi non Windows. Le differenze però tra JVM e CLR sono essenziali: CLR supporta molti linguaggi diversi, mentre JVM è dedicato solo a Java; benchè il bytecode di Java si possa prestare al supporto di altri linguaggi, viene interpretato, a differenza del linguaggio di basso livello usato da CLR, ovvero Common Intermediate Language, che viene compilato tramite JIT ("Just in Time"). Questa procedura fa ampio uso dei metadati .NET spiegati precedentemente.[4]

2.1.4 Il linguaggio C#

Il linguaggio C# si presenta come una ulteriore evoluzione dei linguaggi appartenenti alla famiglia di C.[5] La sua caratteristica più nota è la semplicità e la versatilità derivata da un approccio sintattico molto simile a quello di Java. C# integra inoltre funzionalità presenti in Visual Basic 6 e in C++, come la possibilità di dichiarare metodi con un numero variabile di argomenti, utilizzando array di parametri. Alcuni costrutti sono stati derivati da linguaggi funzionali come LISP e Haskell; con l'evoluzione della tecnologia Framework .NET nuove funzionalità sono state aggiunte al linguaggio (come il sistema LINQ), ampliando ulteriormente le sue caratteristiche di pulizia di sintassi, semplicità, flessibilità e versatilità nel programmare applicazioni di tipo Windows Form

senza le difficoltà riscontrate precendentemente con C++.[5]
L'impostazione generale di C# quindi è basata su un approccio ad oggetti. [4]
La predisposizione di C# ad operare su dati in formato XML e a gestire accessi e interrogazioni a basi di dati remote [6] è stata fondamentale per la scelta di utilizzare questo linguaggio per realizzare i plugin di gestione del Repository IEEE 1599 da integrare nel Framework IEEE 1599 (a sua volta programmato in C#). La realizzazione è stata notevolmente facilitata dalla tecnologia Intellisense presente nell'IDE Visual Studio 2008.

2.1.5 Manipolazione di documenti XML in C#

La rappresentazione di dati in formato XML è parte integrante del funzionamento interno di alcuni fra i componenti più importanti dell'ambiente .NET. Il Framework .NET fornisce un intero namespace, System.Xml, per lavorare con i documenti Xml. E' fornito il supporto alla rappresentazione DOM (Document Object Model), alla trasformazione XSLT, alle query XPath, alla validazione del DTD di un documento.
Le classi base di lettura e scrittura fornite da questo namespace sono le due classi astratte XmlReader e XmlWriter, le loro estensioni XmlTextReader e XmlTextWriter; le rappresentazioni di nodo XmlNode, una classe astratta la cui estensione XmlDocument permette la rappresentazione in memoria dell'intero tree di un documento Xml; la classe XmlResolver e la sua estensione XmlUrlResolver, usate per gestire le risorse esterne linkate a un documento Xml, come il DTD [7].

XmlReader e XmlWriter (e le loro derivazioni) sono scritture dati Xml che operano su stream di informazioni, non hanno quindi la possibilità di navigare agilmente nella struttura di un documento. Hanno però il vantaggio di essere molto veloci e di richiedere poca memoria. Nei plugin di gestione del Repository sono stati generalmente utilizzate queste classi quando si hanno a disposizione documenti già completi da importare o esportare. [7]

```
XmlTextWriter xmlWriter = new XmlTextWriter(fileProfile,
System.Text.Encoding.UTF8);
xmlWriter.Formatting = Formatting.Indented;
xmlWriter.WriteProcessingInstruction("xml", "version='1.0' encoding='UTF-8'");
xmlWriter.WriteStartElement("IEEE1599_repository_connection_profiles");
xmlWriter.Close();
```

Completamente differente è invece l'approccio basato su DOM della classe XmlDocument. Questa classe rappresenta l'implementazione della rappresentazione DOM di un documento

nell'ambiente .NET. DOM è una convenzione cross-platform e indipendente dal linguaggio usata per la rappresentazione e l'accesso a strutture dati XML (anche HTML e XHTML)[7]. Questo approccio risulta utile, per esempio, nella creazione di un nuovo documento XML specificandone dettagliatamente lo schema creandolo nodo per nodo.

```
XmlElement profileName = xmlDoc.CreateElement("profile_name");
XmlText profileNameNode = xmlDoc.CreateTextNode(profilesCombo.Text);
profileName.AppendChild(profileNameNode);
profile.AppendChild(profileName);
```

2.2 ADO.NET ODBC

2.2.1 Panoramica generale su ADO.NET

ADO.NET è un modello di accesso relazionale specificatamente creato per accedere a delle fonti di dati all'interno di applicazioni sviluppate in .NET. Costitusce una radicale evoluzione dal precedente modello ADO ("ActiveX Data Object"), basato sulla precedente tecnologia COM, giudicata un'architettura inadatta per applicazioni sempre più improntate verso il web [8]. E' un modello che rispecchia l'approccio .NET di programmazione ad oggetti e si basa fortemente sulla rappresentazione dei dati in formato XML nella fase di trasporto dei dati stessi. Invece di rappresentarli in fase di trasporto in formato binario, la rappresentazione sotto forma di XML permette di trattare dati "intelligenti" che preservano in maniera sicura schemi ed attributi, e passano più facilmente i firewall.[8]

La rappresentazione di questi dati non avviene strettamente in XML, bensì usando la tecnologia XSD (Xml Schema Definition). Questa tecnologia permette non solo di rappresentare i dati, ma anche di rappresentarne schemi e strutture in cui sono inseriti, nello stesso modo nel quale si potrebbero descrivere le tabelle SQL di un database. [9]

La connessione ad una DataSource avviene tramite i servizi .NET Data Providers. Questi servizi connettono l'applicazione alla fonte dati indicata. Esistono due tipi di servizi .NET Data Providers: i cosiddetti "bridge providers" e i "native providers". I "bridge providers" sono modelli di accesso che si basano su modelli precedenti la tecnologia .NET, come per esempio i driver ODBC (e di conseguenza i driver ODBC utilizzati nella realizzazione dei plugin di gestione del Repository IEEE 1599). Questo tipo di accesso prevede quindi un layer in più nella rappresentazione dello schema di connessione: il bridge provider fa da tramite tra l'ambiente .NET e il driver ODBC (passando attraverso il driver manager delle connessioni remote del sistema operativo), che a sua volta si connette con la sorgente dati (il Repository).[9]

I native providers invece eliminano un livello di collegamento, riuscendo direttamente a dialogare con i database, senza bisogno di librerie intermediarie.[8]

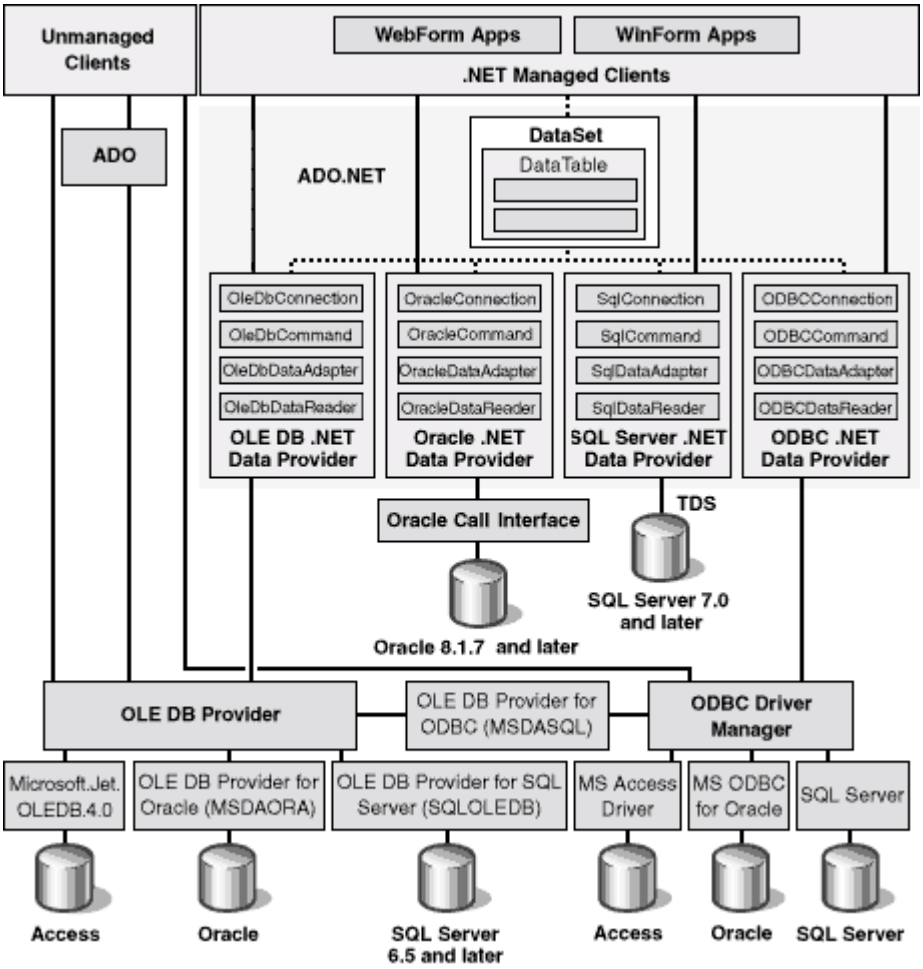

Fig. 5: Connessioni ADO e ADO.NET a basi di dati

I 4 oggetti chiave per l'accesso e la manipolazione dei dati sono Connection, Command, DataReader e DataAdapter.Connection stabilisce una connessione con la fonte dati indicata dai parametri inseriti nella stringa di connessione; un oggetto Command invece contiene in formato stringa istruzioni in formato SQL e punta ad un determinato oggetto Connection, provvedendo ad inviare ed eseguire il comando inserito sulla base di dati che risponde a quella connessione. Per fare questo vengono usati, a seconda del tipo di istruzione SQL, i metodi forniti dalla classe. [8]
DataReader è l'oggetto che fornisce la possibilità di leggere i risultati di una query eseguita tramite un oggetto Command; legge quindi uno stream di righe dalla sorgente dati. [9] Infine, DataAdapter, gestisce un set di oggetti Command per aggiungere, aggiornare e cancellare righe su una sorgente dati connessa tramite un oggetto Connection.[8]

2.2.2 La connessione ODBC

ODBC è un'interfaccia di accesso a database relazionali sviluppato nel 1992 sulla base delle specifiche Call Level Interface. Queste specifiche definiscono come un'applicazione dovrebbe inviare comandi e interrogazioni SQL al DataBase Management System e come le tuple restituite dal DBMS devono essere trattate dall'applicazione. Lo scopo di ODBC è di fornire un metodo di connessione e interazione con un DBMS indipendentemente dal sistema operativo utilizzato o dal database utilizzato, a patto che entrambi siano predisposti a lavorare con i driver ODBC. Si propone quindi come interfaccia universale. [14]

La comunicazione tramite tecnologia ODBC può essere schematicamente basata su tre elementi fondamentali: il Client ODBC, il Driver ODBC e il Server ODBC. Il Client usa una serie di comandi per interagire con il Server, o con il DBMS. Questi comandi però non possono essere compresi se non passano attraverso il Driver ODBC specifico per quel DBMS (nel caso di PostgreSql, PSQLODBC). Il Driver ODBC è quindi posto sulla macchina "lato client": trasforma i comandi inviati dal Client in una forma che il Server può accettare; viceversa, trasforma la risposta del Server in dati leggibili per il Client. [15]
Volendo ampliare e rendere più realistica questa visione, è possibile introdurre un quarto elemento, il Driver Manager: un gestore dei diversi driver ODBC da usare a seconda del DBMS, o più genericamente, della "Data Source" a cui si è connessi. [14]

L'architettura reale di ODBC prevede quindi 4 componenti: l'Applicazione, che si occupa della chiamata delle funzioni ODBC per l'interazione con la fonte dei dati; il Driver Manager, che carica un particolare Driver a seconda della richiesta fatta dall'applicazione e passa le chiamate delle funzioni ODBC al Driver caricato; il Driver elabora le chiamate delle funzioni, passa le istruzioni SQL alla Data Source alla quale si sta accedendo in un formato adatto e successivamente restituisce i risultati all'applicazione; Data Source rappresenta l'entità che fornisce i dati richiesti attraverso query SQL, per esempio un DBMS. [15]

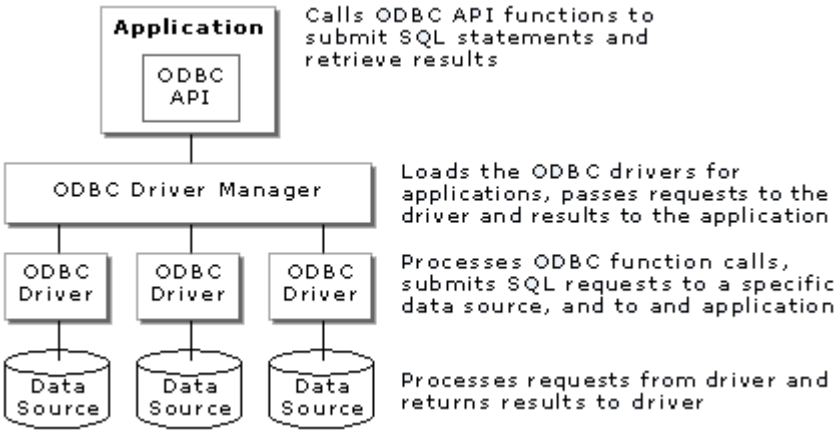

Fig. 6: Connessione ODBC a una base di dati

E' importante definire meglio il ruolo dei Driver. La loro funzione di librerie che implementano le funzioni richiamate nelle API ODBC è in pratica quella di esporre e rendere utilizzabili le capacità del DBMS per il quale sono state progettate. L'architettura dei Driver può essere essenzialmente di due tipi: File-Based e DBMS-Based. I Driver File-Based accedono direttamente alla sorgente dati, senza quindi utilizzare alcun database. Non solo eseguono i comandi ODBC, ma si occupano anche delle istruzioni SQL. [15]

Il tipo DBMS-Based al contrario interagisce con un database stand-alone, occupandosi quindi della sola esecuzione dei comandi ODBC, lasciando al motore del database l'esecuzione delle istruzioni SQL. Come già accennato, il Driver è associabile al ruolo di client mentre il DBMS è associabile a quello di server, anche se non bisogna dimenticare che è comunque possibile che entrambe le parti risiedano sulla stessa macchina (così come su macchine differenti tra loro collegate o con la sorgente dati posta dietro un gateway). [15]

I compiti fondamentali di un Driver ODBC sono quindi: la connessione e disconnessione dalla sorgente dati; la verifica di eventuali errori nelle funzioni non riscontrati dai controlli effettuati dal Driver Manager; la gestione delle transazioni di dati, in maniera trasparente all'applicazione; la trasmissione delle istruzioni SQL verso la sorgente dati (nel caso di DBMS-Based al database), eventualmente correggendole in maniera tale che si adattino alla sintassi pretesa da quel particolare motore sql; l'invio dei dati ricevuti attraverso il driver manager dall'applicazione e restituire i risultati della sorgente dati all'applicazione in un formato consono ad essa; la mappatura dei messaggi e dei codici di errore specifici di un certo DBMS sui codici ODBC SQLSTATE. I codici SQLSTATE infatti servono ad identificare un codice di errore o di un messaggio d'avviso restituito da uno specifico DBMS in modo che questo venga reso leggibile anche al di fuori del DBMS stesso quando questo viene interrogato attraverso una connessione ODBC. [15]

2.2.3 PSQLODBC

Il driver ODBC ufficiale dedicato a PostgreSQL è PSQLODBC (utilizzato nel progetto oggetto di questa tesi in versione Unicode UTF-8). Nella progettazione della parte di connessione al repository dei plugin di gestione del Repository Ieee 1599 si è deciso di dare la possibilità di specificare all'interno della stringa di connessione ODBC anche il nome del driver stesso, permettendo all'utente di poter scegliere quale driver installare nel proprio sistema operativo per dialogare con PostgreSql. Questo driver è parte integrante della distribuzione del DBMS PostgreSQL.
Per la connessione a server PostgreSQL esiste attualmente anche un .NET Data Provider, chiamato Npgsql e sviluppato interamente in C#.

2.3 PostgreSQL

2.3.1 Panoramica generale su PostgreSQL

PostgreSQL è un sistema di database object-relational (ORDBMS) basato sul precedente POSTGRES v.4.2 sviluppato al dipartimento di informatica dell'Università della California di Berkeley. Il progetto ebbe una sostanziale evoluzione tra il 1995 e il 1996 quando si passò dal linguaggio di interrogazione QUEL (usato fin da INGRES, progetto di ricerca di Berkeley per i decenni '70 e '80) alla totale adozione del linguaggio SQL. [10]

Un ORDBMS ha molte caratteristiche in comune ad un normale database relazionale: l'approccio è però quello di un modello di database object-oriented. Gli oggetti e le classi sono direttamente richiamabili all'interno delle query, permettendo all'utente di integrare tipi di dato e metodi creati autonomamente all'interno del DBMS. Questo tipo di database fungono da ponte di collegamento tra l'approccio relazionale più comune (RDBMS), nel quale i dati sono conservati nel database e vengono manipolati attraverso il linguaggio SQL, e l'approccio Object Oriented DBMS (OODBMS), nel quale i dati sono rappresentati sotto forma di oggetti e il database stesso non è che una collezione persistente di oggetti software. PostgreSQL è un software libero e open-source. [10]

PostgreSQL permette la programmazione di funzioni e di funzioni trigger, procedure attivate a seguire un determinato evento su una tabella. Le funzioni possono essere programmate in diversi linguaggi, sia procedurali sia compilati. Il DBMS stesso fornisce un linguaggio procedurale molto simile a quello trovato in sistemi come Oracle, cioè Pl/pgSQL. Questo linguaggio permette l'inclusione di loop e di costrutti di controllo ed è stato usato per programmare i trigger presenti nel Repository Ieee 1599. Altri linguaggi di scripting (Pl/Perl, Pl/Php) o compilati (C, C++, Pl/Java) possono essere inclusi nelle funzioni. [10]

I trigger in PostgreSQL sono quindi eventi scatenati da comandi SQL come INSERT o UPDATE. Vengono supportati solo sulle tabelle (e non sulle viste, cioè una query salvata che crea dinamicamente una tabella virtuale risultato dell'interrogazione) e questo supporto subisce l'ulteriore limitazione di non poter specificare un particolare campo ma solamente l'intera tabella [8]. Un'altra importante caratteristica è una grande varietà di tipo di dato a disposizione, dalle primitive geometriche ai vettori a lunghezza variabile. Il tipo di dato più importante ai fini del

progetto illustrato in questa tesi è XML. PostgreSQL supporta infatti campi di formato XML e fornisce una funzione di query Xpath.
L'utente può inoltre definire i propri data types, insieme ad altri oggetti come funzioni, indici, operatori, conversioni e cast. [10]
La creazione delle query, delle tabelle, dei trigger e in generale l'interazione con il DBMS può avvenire tramite PgAdmin, tool di gestione, sviluppo e amministrazione più usato per PostgreSQL. E' un software libero e open-source.

2.3.2 Generazione di numeri seriali in sequenze

Per la generazione di chiavi primarie "artificiali", cioè di numeri seriali sui record delle tabelle, PostgreSQL si serve di un oggetto chiamato "sequence". Non avendo un comando come "AUTO_INCREMENT" presente in altri DBMS, per ottenere un numero seriale è necessario definire un campo di tipo serial (come bigserial per esempio) ed automaticamente viene creata una sequenza relativa a quella tabella. La sequenza non è altro che una tabella speciale con una singola riga, creata dalla funzione CREATE SEQUENCE. L'utilizzo quindi di un campo di tipo "serial" indica che la generazione di un nuovo valore per il campo sarà effettuata solo dopo la consultazione della sequenza relativa, oggetto creato alla creazione del campo serial nella tabella e che verrà eliminato automaticamente all'eliminazione del campo al quale fa riferimento. Considerato il metodo col quale questo seriale viene generato, la clausola Not_Null è inserita di default [10]. Sono riscontrabili dei "buchi" nei numeri generati. Infatti, la gestione delle sequenze non ha lo scopo di generare valori strettamente sequenziali fra loro, bensì quello di generare ID univoci: una qualsiasi operazione che crea una nuova riga in una determinata tabella dove è presente un campo serial comporta la creazione di un numero di sequenza; se l'operazione non dovesse andare a buon fine (e quindi la riga non fosse stata inserita) quel numero di sequenza risulterebbe comunque già usato, e il record inserito subito dopo partirebbe dal numero successivo, proprio per garantire l'univocità di un campo solitamente usato per essere chiave primaria [10]. Un altro esempio può servire ad introdurre una importante caratteristica di PostgreSQL, cioè Concurrency Control: se due differenti client cercano di ottenere, usando la funzione nextval(), un nuovo serial da una certa sequenza verranno dati loro due valori diversi; se uno dei due interromperà la transazione SQL non utilizzando quindi il valore ottenuto dalla sequenza, quel numero seriale non verrà più utilizzato, creando un "gap", un "buco". Per ottimizzare l'impiego delle risorse del database è comunque possibile attuare delle strategie "gapless". [10]

2.3.3 Multi Version Concurrency Control

PostgreSQL è fornito di Concurrency Control; cioè è in grado di gestire thread di servizi che avvengono contemporaneamente da diversi client. Un thread di servizio è in generale una qualsiasi operazione di lettura (reader) o scrittura (writer) sui dati conservati nel database PostgreSQL. Il concetto generale è di dare ad ogni utente una visione del database che non viene cambiata finchè effettivamente non viene conclusa un'operazione SQL sui dati. Ciò significa che i thread reader e writer non si bloccano a vicenda e permettono ad ogni utente di continuare l'uso contemporaneo del database. Un processo di scrittura può bloccarne un altro solo se stanno agendo sullo stesso dato. La logica di funzionamento qui propone due possibilità: il thread writer legge i nuovi valori e poi continua nella sua operazione di scrittura, oppure l'operazione di scrittura viene annullata automaticamente se durante la transazione quel valore viene modificato da un altro thread. [16]. Questa spiegazione in breve del Concurrency Control può essere ulteriormente ampliata. Innanzitutto, il nome completo di questa funzionalità è Multi Version Concurrency Control. Multi Version sta infatti ad indicare che PostgreSQL è in grando di mostrare più "versioni" dello stesso database, cioè istantanee scattate in diversi momenti della base di dati. Gli eventi che MVCC previene sono tre fenomeni indesiderati e definiti come tali nelle specifiche dello standard SQL. Si tratta di "dirty read", "nonrepeatable read" e "phantom read". [10]

Questi tre fenomeni costituiscono tre casi possibili di malfunzionamento causate da una interazione non isolata fra due transazioni.

Una transazione è una sequenza di operazioni che può concludersi con successo o insuccesso. E' schematizzabile in tre fasi. La prima è composta da un'istruzione che dà inizio all'intera sequenza; la seconda procede alla modifica dei dati; la terza verifica il risultato ottenuto: se sono state riscontrate anomalie si esegue un'struzione di "rollback" per abortire la procedura, mentre se l'operazione è andata a buon fine si esegue "commit", che funziona da conferma per le operazioni effettuate sui dati. [16]

"Dirty Read" avviene quando una transazione legge dei dati che sono stati scritti da una transazione estranea (proveniente da un altro client) ma che ancora non hanno ricevuto una istruzione di "commit". [10]

"Nonrepeatable Read" avviene quando una transazione rilegge dei dati precedentemente letti e nel confronto si "accorge" che sono stati modificati da una transizione estranea (che ha eseguito commit). [10]

"Phantom Read" infine si verifica quando una transazione riesegue un'interrogazione che ha precedentemente resituito una serie di righe che ora alla riesecuzione della query risultano variate da una transazione estranea che ha recentemente eseguito su di loro operazioni e infine commit.
Lo standard SQL definisce quattro livelli di isolamento delle transazioni. Questi livelli specificano quali fra i tre fenomeni sopracitati non si devono assolutamente verificare a seconda del livello di isolamento adottato. I quattro livelli sono: Read Committed, Read Uncommitted, Repeatable Read e Serializable. [10]

Livello di isolamento	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
Read Uncommitted	Possibile	Possibile	Possibile
Read Commited	Non possibile	Possibile	Possibile
Repeatable Read	Non possibile	Non possibile	Possibile
Serializable	Non possibile	Non possibile	Non possibile

Fig. 7: Tabella dei livelli di isolamento di una transazione SQL

PostgreSQL mappa questi 4 livelli possibili di isolamento su due metodi principali: Read Committed e Serializable. La ragione per questa riduzione (che non viola le linee guida dello standard SQL) è che è l'unico modo per avere un corretto funzionamento del Multi Version Concurrency Control. [10]
Questi due metodi di fatto costituiscono le due possibili soluzioni, introdotte all'inizio del paragrafo, al problema di interazione simultanea di due client sullo stesso dato.
Il livello di isolamento Read Committed è il settaggio di default di PostgreSQL. Quando una transazione usa questo livello, una semplice SELECT vede solo dati che hanno ricevuto commit prima dell'inizio della query ignorando quindi sia i dati non ancora confermati, sia i cambiamenti sui dati che hanno ricevuto un commit ma solamente dopo il momento di inizio della query. Read Committed fa quindi vedere a una transazione un'istantanea del database scattata al momento di inizio della transazione. Nel caso di una singola transazione con più SELECT al suo interno il comportamento di questo livello cambia leggermente: è infatti possibile che le SELECT successive vedano dati diversi dalle precedenti se una transazione concorrente ha dato il commit tra l'esecuzione della prima SELECT e l'effettivo inizio della seconda. [10]
Il funzionamento di Read Commited diventa più complicato su istruzioni come UPDATE e DELETE. La regola base è ovviamente la stessa, vengono cioè trovate righe che hanno ricevuto un

commit prima dell'inizio della transazione. Se invece tra l'inizio della transazione "A" e l'effettivo ritrovamento delle righe che soddisfano la condizione questi dati sono stati aggiornati o modificati (o eliminati, o comunque bloccati) da una transazione "B" ma senza ancora aver ricevuto commit, la transazione "A" attende che la concorrente arrivi alla sua fase finale: se l'esito è roll back, tutto può procedere normalmente; se invece è commit, la transazione "A" riprova ad applicare le sue operazioni ai dati aggiornati, verificando che le sue condizioni di ricerca siano ancora verificate, o, nel caso la transazione "B" avesse operato una DELETE, ignorando le righe eliminate. Questo comportamento rende Read Committed un metodo rischioso nel caso di stringhe di ricerca particolarmente complesse, perché l'istantanea del database che una transazione di update vedrebbe includerebbe le operazioni di aggiornamento operate da transazione concorrenti solo se sono effettuate sulle sue medesime righe, ignorando cambiamenti operati in altre parti del database. Si rivela quindi indicato nel caso le operazioni e lo schema del database siano semplici. [10]

Il metodo Serializable è ben più rigido nel livello di isolamento che instaura. La sua logica base è emulare una serie di transazioni come se anche quelle concorrenti fossero in realtà una serie ordinata di operazioni, eseguite una dopo l'altra. Una transazione in livello Serializable vede solo i dati che hanno ricevuto commit prima dell'inizio dell'intera transazione. La differenza sostanziale con Read Committed è che nel caso di una transazione che comprenda più query una dopo l'altra non esiste differenza fra la versione di database visto dalle varie SELECT. Non vengono inclusi in questa visione i cambiamenti effettuati (e già confermati) tra una query e l'altra. Anche nel caso di UPDATE e DELETE (e di SELECT FOR SHARE e SELECT FOR UPDATE) le transazioni vedono righe che sono state cambiate e alle quali è stato dato commit fino all'inizio della transazione stessa. Il caso particolare però è lo stesso riscontrato con Read Committed: prima che una riga venga trovata per la transazione "A" è già stata iniziata una transazione "B" che le ha variate. Come illustrato precedentemente,si attende la fase finale dell'operazione concorrente: se questa transazione concorrente dà istruzione di roll back, il problema non si pone; se viene dato il commit e la riga non è stata solamente bloccata, ma un'operazione concreta di UPDATE o DELETE è stata portata a termine, il livello di isolamento blocca la transazione "A" con un messaggio di errore. Una transazione serializzabile non può infatti modificare o bloccare righe che sono state modificate dopo il suo inizio. Questo vale solo per le transazioni di aggiornamento, non quelle di sola lettura, che non vengono mai bloccate e abortite. [10]

Una volta bloccata la transazione, questa viene rieseguita dall'inizio: avendo a disposizione un'istantanea del database che include già le modifiche che l'hanno bloccata precedentemente a questo punto può procedere senza problemi. Il comando SET TRANSACTION permette all'utente di configurare il livello di isolamento delle transazioni. [10]

2.4 XPATH

2.4.1 Panoramica generale su Xpath

XPath è un linguaggio di query specificatamente studiato per operare sui file XML: il suo nome completo è infatti XML Path Language. Lo scopo di questo linguaggio è semplificare la ricerca di contenuti all'interno di file XML. Xpath funziona su espressioni che specificano un percorso nella struttura ad albreo di un documento e che puntano quindi al contenuto di un particolare nodo. [11]
XPath costituisce quindi uno strumento essenziale per un accesso intuitivo e supportato da molti sistemi alla struttura di un documento XML.
La visione che questo linguaggio ha di un documento XML funziona a livello di rappresentazione ad albero. Un processore XPath riconosce i nodi e attribuisce loro un'importanza gerarchica a seconda del loro posizionamento fisico all'interno del documento. Ad ogni livello gerarchico riconosce sotto elementi collegati che contribuiscono alla ramificazione dell'albero, insieme ad elementi che non costituiscono di per sè un livello vero e proprio, cioè gli attributi. La rappresentazione quindi della struttura vista da un processore Xpath è riconducibile a un albero i cui nodi sono costituiti da scatole contenti dei dati.[11]

Un'espressione XPath non si serve della sintassi XML, ma assume una serie di forme e di elementi che permettono sia la navigazione sia l'esecuzione di alcuni semplici funzioni. La sintassi principale è uguale a quella usata negli URI(Uniform Resource Indicator), o nei filename di UNIX: è un path i cui passi vengono separati da slash fino a a raggiungere il nodo desiderato [12]. Questa è la forma più semplice di espressione. In realtà, è possibile identificare una qualsiasi espressione di questo tipo come composta da sottoelementi, cioè location steps. Insieme, formano un location path. [11]
Ogni location step è a sua volta scomponibile in tre parti differenti: axis, node test e i predicati opzionali. L'axis serve a specificare la relazione alla quale accedere partendo dal nodo specificato: seleziona i nodi figlio del nodo indicato, o tutti i legami gerarchici presenti nel tree in questione. Di default (non indicando alcun axis quindi) è settato sul nodo figlio del nodo di contesto [12]. E' importante notare come i vari tipi di axis (child::, parent::, descendant::, ancestor:: per esempio) adottino una visione schematica "familiare" nella navigazione di un tree, usando termini di

parentela per esplicitare le strutture gerarchiche di un documento e la direzione in cui la query XPath si deve muovere. [11]
Node test è la parte che indica il nome del nodo da selezionare in base al contesto e il tipo di nodo (se per esempio si tratta di un attributo). Infine, i predicati sono ulteriori restrizioni alla ricerca da effettuare e possono contenere ulteriori espressioni XPath. [12] Sono utilizzabili in funzione di predicato i normali operatori booleani
Sono inoltre a disposizione una serie di funzioni richiamabili all'interno di una espressione, come "count", che, avendo come argomento un location path, restituisce il numero di nodi selezionati da quella query XPath. [12]
E' possibile operare una ulteriore distinzione fra le stringhe di location path: come per URI, possono essere percorsi assoluti o relativi. I percorsi relativi partono da un nodo di contesto, un parametro implicito per l'espressione XPath. Un percorso assoluto invece parte dal nodo root del documento. [12] E' possibile fare questa distinzione osservando come inizia l'espressione XPath presa ad esempio: se il primo location step inizia già con uno slash, si tratta del nodo root, dunque facente parte di un location path assoluto. In caso contrario, si ha un location path relativo. [11]
La differenza che esiste fra un percorso URI e una espressione XPath è che se un percorso URI identifica univocamente un solo oggetto, mentre XPath lavora sempre su set di nodi; ogni location step è infatti applicato al set di nodi restituito dallo step precedente. In questa sua caratteristica XPath assomiglia maggiormente a SQL, un altro linguaggio che lavora su gruppi di elementi, anzichè set di nodi gruppi di tuple. [12]

2.4.2 Query XPath in PostgreSQL

I trigger di completamento automatico delle tabella all'inserimento di un documento IEEE 1599 nel repository si affidano ad una serie di query XPath che vanno ad analizzare i contenuti della struttura del documento. A partire dalla versione 8.4 di PostgreSQL gran parte delle funzioni di estrapolazione di contenuto XML allo scopo di trasformarli in dati SQL sono state dichiarate deprecated perchè le API che le implementavano non davano certezza di compatibilità su tutti i sistemi. E' rimasta quindi la sola funzione "xpath", che deve essere richiamata all'interno di un comando SQL, come per esempio SELECT. Questa funzione processa il documento XML (o il campo di una tabella contente XML) secondo una espressione XPath passata come argomento. [13]

```
xpath(xpath, xml[, nsarray])
```

La funzione ha tre argomenti: il primo deve essere una espressione XPath 1.0 racchiusa fra apici singoli; il secondo deve essere un documento XML ben formato (deve avere un solo elemento root, tutti i tag devono essere chiusi, la nidificazione deve essere corretta, i valori degli attributi devono essere chiusi fra apici, i nomi dei tag devono coincidere quando ripetuti essendo XML case sensitive); il terzo, facoltativo, deve essere un array bidimensionale per l'indicazione del namespace XML, con il primo elemento che indica il nome del namespace e il secondo il suo URI. [13]
Un esempio di XPath inserita in una SELECT può essere questo:

```
SELECT xpath ('/ieee1599/general/description/author/text()', file_xml)
FROM "TFile_MX"
WHERE Id_file = 447;
```

Questa interrogazione restituisce un array contenente le occorrenze trovate del tag author nel file indicato. I risultati quindi vengono restituiti all'interno di un vettore, anche se si tratta di un solo riscontro. [13]

2.4.3 Query XPath in C#

L'ambiente .NET fornisce il namespace System.Xml.Xpath per effettuare query XPath su documenti XML. Lo scopo di questa implementazione è avere un metodo facile e veloce di accedere alle risorse di un documento. Viene fornito un set di classi: XPathDocument, XPathNavigator, XPathNodeIterator, XPathExpression e XPathException. [7]

XPathDocument fornisce una classe per oggetti simili a XmlDocument ma non dedicati all'editing del documento bensì solo a una sua veloce lettura tramite espressioni XPath. Ha una grande flessibilità di overload, può infatti aprire documenti XML da file indicandone il path, da stream, da TextReader e da XmlReader. [7]
XPath Navigator fornisce una serie di metodi per muovere l'asse dell'espressione: ogni metodo permette di percorrere un livello di parentela del nodo contestuale (MoveToFirstChild(), MoveToParent(), ecc ecc). A questi sono affiancati dei metodi analoghi di selezione dei nodi (SelectChildren() per esempio), che restituiscono oggetti di tipo XPathNodeIterator. [7]
Questo oggetto è l'equivalente delle classi NodeList o NodeSet ed è utile a fare operazioni dirette sul nodo indicato, come Clone o Count.

XPathExpression contiene effettivamente la stringa di espressione Xpath, compilata con il metodo Compile e quindi utilizzabile come oggetto in altre funzioni.

```
XPathExpression expression = XPathExpression.Compile
("/IEEE1599_repository_connection_profiles/profile/profile_name/text()");
```

L'uso è differente da quello proposto dal metodo SelectNodes (System.Xml.XmlNodeList) che invece usa come argomento una semplice stringa che contiene l'espressione XPath.

```
XmlNodeList nodelist =
listalogin.SelectNodes("/IEEE1599_repository_connection_profiles/profile/profile
_name/text()");
```

XPathException definisce un'eccezione riscontrata in seguito ad un errore nell'esecuzione di una espressione XPath.

Capitolo 3

In questo capitolo si descrivono le strutture, la logica di funzionamento, il codice sorgente nel dettaglio e l'integrazione nel Framework IEEE 1599 dei 4 plugin di gestione del repository. Le parti di connessione, gestione dei profili di connessioni degli utenti e il motore di ricerca sono comuni e vengono illustrate una sola volta.

3.1 Uploader

Il plugin Uploader si occupa del caricamento di un documento aperto nel Framework IEEE 1599 all'interno del repository. Il suo scopo principale è quello di scatenare il trigger di autocompilazione delle tabelle all'interno del repository stesso, evento che si verifica all'inserimento di dati nel campo xml della tabella Tfile_Mx. Il plugin deve inoltre completare i campi estranei al codice xml, cioè quelli che forniscono informazioni aggiuntive sui file caricati, ovvero un file name generato automaticamente dal plugin in sede di caricamento del file e la data del caricamento del documento.

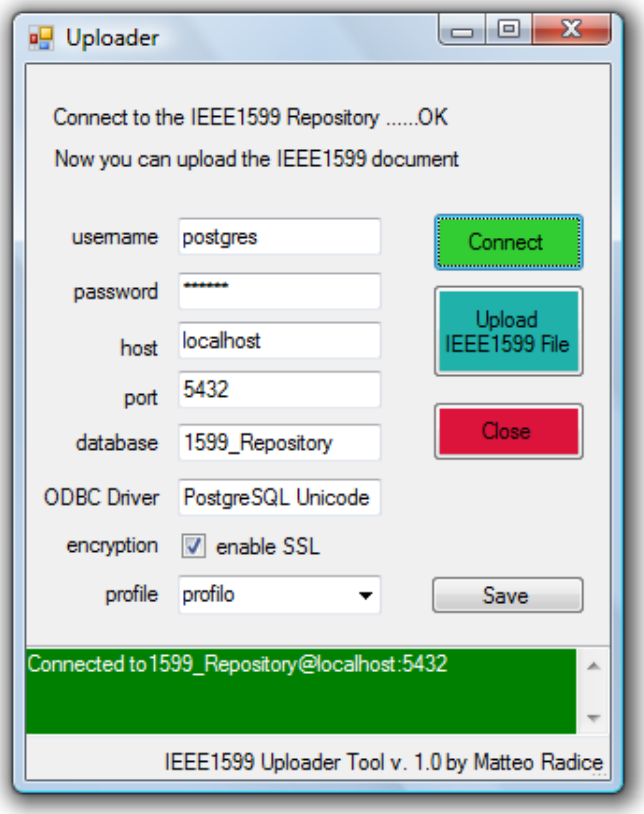

Fig. 8: Maschera di connessione e caricamento dei documenti in Uploader

3.1.1 Connessione al database

L'interfaccia di connessione al database è composta da 6 textbox che identificano i parametri di connessione Username, Password, Host del repository, porta di comunicazione dell'host, nome del database e il driver odbc utilizzato. A questo viene aggiunta una checkbox per l'abilitazione del protocollo crittografico di connessione Secure Sockets Layer (SSL). La combobox sottostante permette la selezione di un profilo di connessione precedentemente salvato o la nominazione di un nuovo profilo di connessione da salvare cliccando il bottone "Save". Il bottone connect compone la stringa di connessione e lancia il comando odbc al database seguendo il percorso indicato dall'utente attraverso l'interfaccia.
La stringa di connessione viene formata concatenando fra loro i campi di testo delle textbox. Il formato della stringa ODBC è il più esteso possibile[9]:

```
DRIVER={odbcDriver};SERVER=server;PORT=port;DATABASE=dbname;UID=username;
                    PWD=password;ENCRYPT=YES | NO;
```

Nella struttura try-catch viene innanzitutto creata una nuovo oggetto OdbcConnection di nome connessione, che è dichiarato come variabile private della classe Uploader; gli viene assegnato come argomento la stringa di connessione completa. La connessione viene poi aperta con il metodo Open() [9]. L'utilizzo di try-catch permette la gestione di errori durante questa sequenza di istruzioni, nel caso per esempio l'autenticazione non vada a buon fine o il server non risponda. Nel caso quindi di una qualsiasi exception, viene stampato su sfondo rosso nella textbox utilizzata come status bar l'errore riscontrato. Nel caso opposto, la status bar diventata verde e comunica l'avvenuta connessione al server e al database specificato.

```
namespace PluginUpload
{
    public partial class Uploader : Form
    {
        private System.Data.Odbc.OdbcConnection connessione;
        private String ssl = "NO";
```

La funzione “Connect_Click” si occupa effetivamente quindi di stabilire una connessione con il repository, utilizzando i parametri specificati.

```
     private void Connect_Click(object sender, EventArgs e)
            {
                String conn = "DRIVER={" + odbcDriver.Text + "};SERVER=" +
                        host.Text + ";PORT=" +
```

```
                port.Text + ";DATABASE=" +
                database.Text + ";UID=" +
                userID.Text + ";PWD=" +
                password.Text + ";Encrypt="+ ssl +";";
            try
            {
                connessione = new System.Data.Odbc.OdbcConnection(conn);
                connessione.Open();
                upload.Enabled = true;
                status.BackColor = Color.Green;
                status.Text = "Connected to" + database.Text.ToString()
                            + "@" + host.Text.ToString() + ":"
                            + port.Text.ToString() + "";
                connectlabel.Text += "......OK";
                selectlabel.Enabled = true;

            }
            catch (Exception err)
            {
                status.BackColor = Color.Red;
                status.Text = "Connection failed: " + err.Message;
                dialogresult = DialogResult.Cancel;
            }
        }
```

La gestione della checkbox di attivazione della cifratura invece si appoggia su un'altra funzione, che verifica lo stato (Checked o Unchecked, tralasciando Indeterminate) del controllo e assegna alla stringa globale privata "ssl" valore "YES" oppure "NO", in modo da concatenarlo correttamente alla stringa dei parametri di connessione.

```
private void encryptionCheck_CheckedChanged(object sender, EventArgs e)
    {
        if (encryptionCheck.CheckState == CheckState.Checked)
            ssl = "YES";
        else
            ssl = "NO";
    }
```

Considerando la capacità di questa interfaccia di connettersi a diversi database su diversi host a seconda dei parametri inseriti, si è pensato di permettere all'utente di salvare diversi profili di connessione.

3.1.2 Salvataggio dei profili utente

Come già accennato nel paragrafo precedente, la combobox "profile" e il pulsante "Save" servono per selezionare un profilo già creato o per salvarne uno nuovo. Tali profili vengono scritti in un file XML creato automaticamente dalla funzione nella cartella dei documenti personali nel sistema

operativo in uso all'apertura del plugin nel framework. Vengono dichiarate due variabili globali, una stringa contente il path del file xml nel quale saranno salvati i profili e un'istanza XmlDocument per caricare temporaneamente il contenuto del file xml per il suo utilizzo. La classe Environment fornisce informazioni e funzioni per gestire l'ambiente in cui il codice del programma funziona. [7]

```
private XmlDocument listalogin = new XmlDocument();
private String fileProfile =
Environment.GetFolderPath(Environment.SpecialFolder.Personal) +
"\\IEEE1599Framework\\loginProfile.xml";
```

Il tree del file xml "loginProfile.xml" è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<IEEE1599_repository_connection_profiles>
  <profile name="profilo">
    <profile_name>profilo</profile_name>
    <user_name>postgres</user_name>
    <host>localhost</host>
    <port>5432</port>
    <database>1599_Repository</database>
    <ODBC_driver>PostgreSQL Unicode</ODBC_driver>
    <encryption>YES</encryption>
  </profile>
</IEEE1599_repository_connection_profiles>
```

Si può notare la ripetizione del nome del profilo sia come attributo del tag padre <profile> sia come nodo figlio <profile_name>. Questa ridondanza è stata pensata per gestire più comodamente le stringhe Xpath delle funzioni di caricamento del profilo selezionato e di aggiunta dei profili presenti nel file come elementi del menù a tendina della combobox. La ridondanza torna immediatamente utile sul load generale del form (sia esso Uploader, Downloader, Updater o Deleter).

```
private void Uploader_Load(object sender, EventArgs e)
        {
            refreshProfileList();
        }
```

Sul caricamento del form uploader, per esempio, viene richiamata la funzione "refreshProfileList()".

```
private void refreshProfileList()
    {
            try
            {
                listalogin.Load(fileProfile);
                XmlNodeList nodelist = listalogin.SelectNodes
    ("/IEEE1599_repository_connection_profiles/profile/profile_name/text()");

                foreach (XmlNode profile_name in nodelist)
```

```
                {
                    profilesCombo.Items.Add(profile_name.InnerText);
                }

            }
            catch
            {
                profilesCombo.Items.Add("(none)");
            }

    }
```

Viene tentato il caricamento nell'istanza globale di classe XmlDocument "listalogin" del documento indicato nella variabile globale "fileProfile". Nel caso il file non venga trovato, nella lista viene scritto l'elemento "(none)", a indicare che la lista è vuota perchè non sono stati caricati profili. Nel caso il caricamento vada a buon fine, si procede ad accedere ai nodi selezionati con la funzione SelectNodes, il cui argomento è una stringa Xpath [7]. Questa stringa percorre la struttura del file e sfrutta la ridondanza citata prima riguardante il nome del profilo. Vengono infatti cercati tramite Xpath tutti i nodi <profile_name> e aggiunge il loro contenuto testuale in un elenco di nodi di tipo XmlNodeList. Usando un ciclo foreach, vengono selezionati singolarmente tutti gli elementi di tipo Xmlnode all'interno della lista, aggiungendoli uno a uno all'elenco degli elementi della combobox di selezione dei profili. Il motivo per cui non si può utilizzare l'attributo "name" del nodo <profile> è che selezionandolo con Xpath la ricerca restituisce in realtà tutti nodi suoi figli.
La funzione fondamentale per la creazione dei profili è quella di registrazione dei parametri dell'utente all'interno del file xml "loginProfile.xml". Nel caso questo non esista, la funzione provvede a crearne uno nuovo. Questa funzione viene richiamata cliccando sul pulsante "Save".

```
private void saveProfileButton_Click(object sender, EventArgs e)
    {
        if (profilesCombo.Text != "")
        {
            try
            {
                XmlDocument xmlDoc = new XmlDocument();
                try
                {
                    xmlDoc.Load(fileProfile);
                }
                catch (System.IO.FileNotFoundException)
                {
                    XmlTextWriter xmlWriter = new XmlTextWriter
                    (fileProfile, System.Text.Encoding.UTF8);

                    xmlWriter.Formatting = Formatting.Indented;

                    xmlWriter.WriteProcessingInstruction("xml",
                    "version='1.0' encoding='UTF-8'");
```

```
        xmlWriter.WriteStartElement
        ("IEEE1599_repository_connection_profiles");
        xmlWriter.Close();
        xmlDoc.Load(fileProfile);
    }

    XmlNode root = xmlDoc.DocumentElement;
    XmlElement profile = xmlDoc.CreateElement("profile");
    profile.SetAttribute("name", profilesCombo.Text);
    root.AppendChild(profile);
    //creao nuovo elemento profilo

    XmlElement profileName =
    xmlDoc.CreateElement("profile_name");

    XmlText profileNameNode =
    xmlDoc.CreateTextNode(profilesCombo.Text);

    profileName.AppendChild(profileNameNode);
    profile.AppendChild(profileName);

    XmlElement username = xmlDoc.CreateElement("user_name");
    XmlText usernameNode = xmlDoc.CreateTextNode(userID.Text);
    username.AppendChild(usernameNode);
    profile.AppendChild(username);
    //creato nuovo elemento username

    XmlElement server = xmlDoc.CreateElement("host");
    XmlText serverNode = xmlDoc.CreateTextNode(host.Text);
    server.AppendChild(serverNode);
    profile.AppendChild(server);
    //creato nuovo elemento host

    XmlElement serverPort = xmlDoc.CreateElement("port");
    XmlText serverPortNode = xmlDoc.CreateTextNode(port.Text);
    serverPort.AppendChild(serverPortNode);
    profile.AppendChild(serverPort);
    //creato nuovo elemento port

    XmlElement databaseName = xmlDoc.CreateElement("database");
    XmlText databaseNameNode =
    xmlDoc.CreateTextNode(database.Text);

    databaseName.AppendChild(databaseNameNode);
    profile.AppendChild(databaseName);
    //creato nuovo elemento database

    XmlElement driver = xmlDoc.CreateElement("ODBC_driver");
    XmlText driverNode = xmlDoc.CreateTextNode(odbcDriver.Text);
    driver.AppendChild(driverNode);
    profile.AppendChild(driver);
    //creato nuovo elemento server driver


    XmlElement encryption = xmlDoc.CreateElement("encryption");
    XmlText encryptionNode = xmlDoc.CreateTextNode(ssl);
    encryption.AppendChild(encryptionNode);
    profile.AppendChild(encryption);
    //creato nuovo elemento server encryption
```

```
                xmlDoc.Save(fileProfile);
                profilesCombo.Items.Add(profilesCombo.Text);
                MessageBox.Show
              ("Your login settings (except password) have been saved.",
              "Profile Saved", MessageBoxButtons.OK, MessageBoxIcon.None);
            }
            catch (Exception ex)
            {
                MessageBox.Show("ERROR: " + ex.Message, "ERROR",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
        else
        {
            MessageBox.Show("You have to write a profile name in the profile
            combobox in order to save your login settings", "Unable to save
            profile name", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }

    }
```

Utilizzando due strutture try-catch annidate si gestisce sia il caso nel quale non si dovesse trovare (perchè spostato, rimosso o non ancora creato) il file contente i profili, sia il caso in cui il file sia stato correttamente trovato e caricato e si possa procedere all'aggiunta di un nuovo profilo. In entrambi i casi, la creazione di un nuovo profilo è necessaria. La struttura quindi prevede che tutto sia inserito in una try generale con relativo catch di errore (per gestire gli errori di creazione di un nuovo profilo). All'interno di questa vi è un'altra struttura di controllo try-catch attraverso la quale viene tentato il caricamento del file xml contenente i profili all'interno di una istanza xmlDocument. Se il file non viene trovato, ne viene immediatamente creato uno nuovo. Viene creato un oggetto XmlTextwriter che scriva sul percorso indicato dalla variabile globale "fileprofile" e in codifica UTF8, viene specificata la formattazione indentata, viene inserita la dichiarazione di versione xml 1.0 e la codifica. Con la funzione WriteStartElement si crea l'elemento radice di tutto il documento (l'elemento viene anche chiuso automaticamente). XmlTextWriter viene chiuso, il file appena scritto viene caricato nell'istanza XmlDocument.

Creato quindi il file se non trovato (con relativa eccezione specifica "System.IO.FileNotFoundException" [5]), si procede alla scrittura del nuovo profilo. Il metodo utilizzato è differente rispetto a prima, si è preferito utilizzare le funzioni fornite dalla classe XmlDocument. Il meccanismo consiste innanzitutto nel posizionarsi sul nodo radice del documento, cioè "IEEE1599_repository_connection_profiles". Viene creato successivamente un oggetto XmlElement utilizzando il metodo CreateElement con il nome del tag del nodo, mentre il contenuto testuale del nodo viene realizzato con il metodo CreateTextElement e prendendo la stringa dalla textbox corrispondente [7]. Avendo quindi il nodo xml e il suo contenuto testuale, si collegano tra

loro con una sintassi del tipo "xmlNode.AppendChild(xmltextNode)" e successivamente si utilizza la medesima forma per collegare il nodo xml al nodo padre. Con il metodo SetAttribute, che ha come argomenti il nome dell'attributo e il valore dell'attributo, è possibile creare un attribute al nodo xml a cui viene applicato. Una volta completata e riempita, la struttura xml del nuovo profilo si trova ancora nell'istanza della classe XmlDocument.[7] Deve essere salvata nel file indicato dalla variabile globale "fileProfile" e la voce del nuovo profilo viene aggiunta (con "profilesCombo.Items.Add(profilesCombo.Text)")alla lista degli elementi elencati dalla combobox "profilesCombo".

Questa voce immediatamente richiamabile suggerisce la necessità di ricaricare anche nell'istanza XmlDocument globale "listalogin" la nuova versione del documento. Tale operazione si rende necessaria nel caso in cui venga chiamata la funzione "profilesCombo_SelectedIndexChanged", cioè la funzione che gestisce il cambio di elemento selezionato (quindi di indice) nell'elenco elementi della combobox.

```
private void profilesCombo_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (profilesCombo.SelectedItem != "(none)")
```

Prima di assegnare a ogni textbox il relativo parametro registrato nel profilo è necessario controllare che la voce selezionata non sia "(none)", nel caso già citato che non sia stato mai creato alcun profilo, o il file di impostazioni non esista. Se venisse infatti selezionato questo elemento non esistente, l'intera funzione andrebbe in exception.

```
listalogin.Load(fileProfile);
userID.Text =
listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\"" +
profilesCombo.SelectedItem.ToString() + "\"]/user_name/text()").InnerText;

host.Text =
listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\"" +
profilesCombo.SelectedItem.ToString() + "\"]/host/text()").InnerText;

port.Text =
listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\"" +
profilesCombo.SelectedItem.ToString() + "\"]/port/text()").InnerText;

database.Text =
listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\"" +
profilesCombo.SelectedItem.ToString() + "\"]/database/text()").InnerText;

odbcDriver.Text =
listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\"" +
profilesCombo.SelectedItem.ToString() + "\"]/ODBC_driver/text()").InnerText;
```

```
if
((listalogin.SelectSingleNode("/IEEE1599_repository_connection_profiles/profile[@name=\""
+ profilesCombo.SelectedItem.ToString() + "\"]/encryption/text()").InnerText) == "YES")
       {
                     encryptionCheck.CheckState = CheckState.Checked;
       }
```

In questa funzione di selezione torna nuovamente utile la ridondanza del nome del profilo. Se prima veniva utilizzato il nodo <profile_name> per caricare i nomi dei profili creati, è possibile individuare in una sola stringa tutti i nodi figlio di un nodo <profile> padre con attributo "name" uguale all'elemento selezionato nell'elenco.
Questa stringa viene usata come argomento del metodo SelectSingleNode, e con InnerText viene selezionato il testo contenuto nel nodo individuato.
Un controllo differente deve essere applicato alla checkbox che abilita la cifratura della connessione: se è stata registrata con valore YES (ricordando che nella stringa ODBC la voce "encryption" può avere valore YES o NO), richiamando il profilo la checkbox deve apprire nello stato "Checked".
Come si può notare, per ovvi motivi di sicurezza, la password è l'unico parametro che non viene mai salvato.

3.1.3 Integrazione di PluginUpload/Uploader nel Framework

Prima di vedere la logica di funzionamento del caricamento dei file IEEE 1599-2008, è necessario fare alcune considerazioni sul metodo utilizzato per integrare i plugin di gestione del repository. Abitualmente i plugin per il Framework IEEE 1599 vengono integrati in modo tale che appaiano aperti in schede nella finestra principale, come programmi che agiscono sul documento in maniera parallela. Sono richiamabili da menù che rispecchiano il layer sul quale operano e che si attivano solo nel caso in cui ci sia effettivamente un file aperto nell'ambiente. E' evidente la differenza tra questo tipo di applicazioni e quelle relative al repository. Di fatto, una funzione di upload da un computer locale verso un repository (sia anch'esso locale o molto più probabilmente remoto) è un'esportazione di un documento, così come è un'importazione una funzione che permette di scaricare direttamente nel Framework un file conservato nel repository. Sono quindi implementati nel menù "File" alle voci Import e Export. Export è abilitato dal Framework solo nel caso ci sia un documento aperto.
L'integrazione quindi non può essere fatta a schede, ma deve piuttosto far comparire una finestra con priorità su quella dell'ambiente generale.

Il metodo con il quale questi 4 plugin sono stati implementati è strettamente legato al loro primo sviluppo come applicazioni stand alone. Il concetto è quello di avere un Windows Form programmato nella struttura dei plugin per il Framework IEEE 1599-2008, che richiama una istanza del form che effettivamente contiene la parte centrale del plugin, passandogli come argomento il documento xml aperto nel Framework.

Nel caso del plugin di upload, abbiamo PluginUpload come effettivo plugin richiamato dal Framework, e il form Uploader richiamato da PluginUpload che effettua il caricamento del documento nel repository.

```
public override bool DllExec(IEEE1599Document document)
{
    if (document != null)
    {
        try
        {
            String xml = getFormattedXml(document);
            xml = xml.Replace("'", "''");
            String filename;
            try
            {
                filename =document.Xml.SelectSingleNode
                ("/ieee1599/general/description/author[1]/text()")
                .InnerText.ToString().Trim();
            }
            catch
            {
                filename = "Unknown_author";
            }
            try
            {
                filename += " - " + document.Xml.SelectSingleNode
                ("/ieee1599/general/description/main_title/text()")
                .InnerText.ToString().Trim() + ".xml";
            }
            catch
            {
                filename += " - untitled.xml";
            }

            Uploader uploader = new Uploader(xml, filename);
            DialogResult dl = uploader.ShowDialog();

            if (uploader.dialogresult == DialogResult.Cancel)
            {
                return false;
            }

            else
            {
               return true;
            }
        }
        catch (Exception a)
```

```
            {
              MessageBox.Show("ERROR: " + a.Message,"ERROR",
              MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
        else
        {
            return false;
        }
        return false;
    }
```

Questa funzione funge da metodo main all'esecuzione della .dll richiamata. Viene effettuato un controllo sul contenuto del documento (controllo ridondante, dato che PluginUpload è accessibile dal menù Export che si attiva solo con un documento IEEE 1599 aperto nel Framework) e successivamente questo viene passato a una funzione esterna, "getformattedxml", che accetta come argomento un'istanza di classe IEEE1599 Document. Questa funzione, che verrà analizzata successivamente, ha lo scopo di restituire una stringa che rappresenti l'intero file aperto nel framework, mantenendo però l'identazione dell'xml originale. Questa stringa deve essere però controllata nuovamente con il metodo "Replace" per inserire un carattere di escape fondamentale per non generare un errore nella stringa sql che verrà passata successivamente al repository: la presenza infatti del carattere "apice singolo" costituisce per PostgreSQL la chiusura dell'argomento nella stringa SQL di insert. Tale carattere può essere presente nelle parti testuali dell'xml. Viene sostituito con il carattere "doppio apice" che viene interpretato come sequenza di escape dell'apice singolo.

La stringa xml che rappresenta il documento da caricare e che verrà inserita nel comando SQL di insert è quindi pronta.

Viene generato ora automaticamente il file name. Si è scelto di implementare una procedura a questo scopo perchè il nome del file aperto nel Framework non viene passato come parametro ai plugin. Per questo si sfrutta nuovamente Xpath per generare un nome file composto dalla prima occorrenza del tag <author> trovato nel documento IEEE 1599-2008 seguito dal contenuto del tag <main_title>. La presenza di file name abbinati ai record potrà tornare utile per future applicazioni.

Attraverso una struttura try-catch vengono gestiti i casi nei quali i tag <author> o <main_title> non siano stati compilati nel documento. Al risultato restituito dal metodo "SelectSingleNode" viene letta la proprietà "InnerText" cioè il testo compreso fra l'apertura e la chiusura del nodo specificato); al testo restituito viene applicato il metodo "Trim()" per eliminare eventuali spazi bianchi all'inizio e alla fine della stringa.

Avendo ora tutti i parametri necessari all'inserimento nel repository, viene creata un'istanza della classe form Uploader, e attraverso il metodo "ShowDialog" viene creata la finestra di dialogo con

cui l'utente potrà finalmente interagire, connettendosi e, cliccando su "Upload IEEE1599 Document", effettuare il caricamento nel repository.
La precedentemente citata funzione "getFormattedXml" invece sopperisce a un problema riscontrato utilizzando per lo stesso scopo (cioè trasformare l'intero documento in una sola stringa) più semplici metodi di lettura completa di un testo xml (come "OuterXml()" [7]). La formattazione del testo xml non viene più considerata e l'identazione è di conseguenza distrutta.

```
public static string getFormattedXml(IEEE1599Document doc)
{
    StringWriter sw = new StringWriter();

    XmlTextWriter xtw = new XmlTextWriter(sw);

    xtw.Formatting = System.Xml.Formatting.Indented;

    doc.Xml.WriteTo(xtw);

    return sw.ToString();
}
```

E' quindi necessario dichiarare due istanze, una di classe StringWriter, cioè uno stream buffer leggibile come stringa, e una XmlTextWriter che scriverà direttamente nello stream [7]. Passato come parametro della funzione il documento IEEE1599 aperto nel Framework e settato XmlTextWriter per scrivere in maniera indentata, viene utilizzata la funzione WriteTo per caricare il contenuto del documento all'interno dell'istanza XmlTextWriter; successivamente viene dato in output il contenuto a stringa dello stream buffer.

3.1.4 Caricamento dei documenti IEEE 1599-2008 nel repository

Il caricamento del documento aperto nel Framework all'interno del repository avviene in conseguenza a un click dell'utente sul pulsante "Upload IEEE1599 Document". Come già spiegato in precedenza, il comando sql insert sulla tabella centrale del repository "TFile_Mx" scatena il trigger di riempimento automatico delle tabelle interno al repository. Tale trigger effettua la lettura del codice e la conseguente compilazione delle tabelle associate ai vari attributi attraverso funzioni Xpath.

```
private void upload_Click(object sender, EventArgs e)
{
    Connect.Enabled = false;
    upload.Enabled = false;
    close.Enabled = false;
```

Cliccando quindi sul pulsante vengono disabilitati gli altri controlli. L'utente potrà interagire con l'applicazione solo quando il caricamento avrà dato esito, positivo o negativo che sia. Vengono create delle stringhe contenti i comandi Sql da passare al repository, e le relative istanze di classe OdbcCommand che costituiranno i comandi veri e propri passati al database attraverso il driver Odbc sulla connessione stabilita.

```
try
  {
        String insert = "INSERT INTO \"TFile_MX\" (file_xml) VALUES
        (XMLPARSE(document(E'" + xml + "')));";
        String name = "UPDATE \"TFile_MX\" SET file_name=\'" + filename +
        "\' WHERE \"ID_file\" = currval(\'\"TFile_MX_ID_file_seq\"\');";
        String data = "UPDATE \"TFile_MX\" SET upload_date=\'" +
        DateTime.Now.ToString("yyyy/MM/dd") + "\' WHERE \"ID_file\" =
        currval(\'\"TFile_MX_ID_file_seq\"\');";

        System.Data.Odbc.OdbcCommand command = new
        System.Data.Odbc.OdbcCommand(insert, connessione);
        System.Data.Odbc.OdbcCommand updatefilename = new
        System.Data.Odbc.OdbcCommand(name, connessione);
        System.Data.Odbc.OdbcCommand upload_date = new
        System.Data.Odbc.OdbcCommand(data, connessione);
```

Il codice Sql scritto nella stringa "insert" rappresenta l'inserimento fondamentale per scatenare il trigger. Attraverso la concatenzione di stringhe viene incluso nel comando la stringa "xml" contenente l'intero documento IEEE 1599 passatogli dalla classe PluginUpload che ha richiamato il form Uploader.
Il comando XMLPARSE serve a PostgreSQL a trasformare dati da inserire all'interno di un campo di tipo "xml" partendo da dati in formato carattere. Specificando il parametro "document" della funzione XMLPARSE si indica al dbms che si sta passando un documento già completo di dichiarazione di versione xml e di DTD. La stringa contenente il codice xml viene preceduta dalla lettera "E" per gestire i caratteri da trattare con sequenze di escape contenute nel documento da inserire.[16]

Gli altri comandi SQL da inviare riguardano il filename creato in PluginUpload e passato ad Uploader e la data in cui viene caricato il documento.
Per aggiungere queste due informazioni è necessario usare un comando di UPDATE: una generica INSERT sulla tabella "Tfile_Mx" richiamerebbe la procedura trigger utilizzata per l'autocompletamento delle tabelle, generando quindi un errore. Vengono fatte delle UPDATE sui campi della tabella gestendo la posizione di inserimento con l'utilizzo della funzione "currval" fornita da PostgreSQL. Dando come argomento la sequenza di dati Bigserial legata alla tabella (in

questo caso "TFile_MX_ID_file_seq") viene restituito l'ultimo ID creato per quella tabella. La data da inserire viene invece creata fornendo un formato coerente con quello del campo "upload_date" nella tabella del repository.

Creati i comandi ODBC, vanno eseguiti usando il metodo "ExecuteNonQuery"[6]. La gestione della buona riuscita e degli eventuali errori di trasmissione o esecuzione sono gestiti tramite una struttura try-catch. Viene dato un valore alla variabile dialogresult, che determina il valore da restituire al Framework da parte di PluginUpload ("return true" o "return false").

```
try
   {
      status.Text += "\nInserting IEEE1599 Document in the
      Repository...";
      status.BackColor = Color.Black;
      status.Update();
      command.ExecuteNonQuery();

      status.Text += "\nInserting filename...";
      status.BackColor = Color.Black;
      status.Update();
      updatefilename.ExecuteNonQuery();

      status.Text += "\nInserting upload date...";
      status.BackColor = Color.Black;
      status.Update();
      upload_date.ExecuteNonQuery();


      status.BackColor = Color.Green;
      status.Text = "File " + filename + " inserted successfully";

      MessageBox.Show("IEEE1599 document: " + filename + "\nuploaded
      successfully.", "File Uploaded Successfully", MessageBoxButtons.OK,
      MessageBoxIcon.None);
      dialogresult = DialogResult.OK;
      command.Dispose();
      updatefilename.Dispose();
      upload_date.Dispose();
      connessione.Close();
      this.Dispose();

   }
 catch (Exception a)
   {
    status.BackColor = Color.Red;
    status.Text = "Insert failed: " + a.Message;

    command.Dispose();
    updatefilename.Dispose();
    upload_date.Dispose();
    Connect.Enabled = true;
    upload.Enabled = true;
    close.Enabled = true;
```

```
    dialogresult = DialogResult.Cancel;
    }
 }
 catch (Exception b)
  {
   dialogresult = DialogResult.Cancel;
   status.BackColor = Color.Red;
   status.Text = b.Message;
  }

  Connect.Enabled = true;
  upload.Enabled = true;
  close.Enabled = true;
  connessione.Close();
}
```

3.2 Downloader

Il plugin Downloader si occupa di cercare un documento IEEE 1599 in base ai parametri inseriti dall'utente. I risultati della ricerca vengono mostrati in una tabella dalla quale è possibile scaricare direttamente all'interno del Framework il documento selezionato, aprendolo sia con la validazione via web del DTD sia senza validazione, usando la stessa metodologia presente nel Framework per la normale apertura di un documento in locale. Per questo, Downloader può essere considerato come un plugin che permette l'apertura di documenti remoti ed è inserito nel menù File > Import.
Dal punto di vista della struttura del programma, Downloader introduce (oltre alla connessione e alla gestione dei profili) diverse funzioni presenti anche nei plugin suoi derivati, Updater e Deleter. Tali funzioni costituiscono la logica di ricerca del documento e di presentazione tabulare dei risultati.
Inoltre, a differenza di Uploader, Downloader è organizzato su tre pannelli differenti, che vengono mostrati e nascosti a seconda delle azioni dell'utente. I pannelli rispecchiano le fasi della ricerca di un documento: connessione al database (con eventuale richiamo del profilo salvato o salvataggio di uno nuovo), input dei criteri della ricerca, visualizzazione dei risultati. La gestione grafica dei pannelli avviene tramite i metodi "Hide()" e "Show()". [5]

3.2.1 Interrogazione del repository per la ricerca dei documenti IEEE 1599

La ricerca dei documenti avviene per mezzo di una query scritta dinamicamente in base a quanto inserito nel pannello di ricerca. Il pannello è organizzato in 28 campi testuali di diverso tipo: la tipologia di controllo più frequente fra questi campi è di tipo "textbox". Selezionando la checkbox relativa a un certo parametro di ricerca, la textbox compare e viene attivata. Lasciando il campo vuoto, la query è comunque valida (a parte l'eccezione di "upload_date"): equivale infatti a scrivere un asterisco, anche se Postgresql sembra preferire la forma di due apici singoli senza contenuto. Selezionato quindi un campo si attiva il pulsante "Execute Query" ed è possibile inviare la query appena creata al repository.
I campi presentati in questo pannello rappresentano tutti i dati significativi selezionati fra i vari layer di un documento IEEE 1599 e di conseguenza rappresentati nel repository. La logica di funzionamento di questo pannello è andare a verificare ogni volta che l'utente cambia lo stato di una checkbox se ci sono le condizioni valide per eseguire la query.

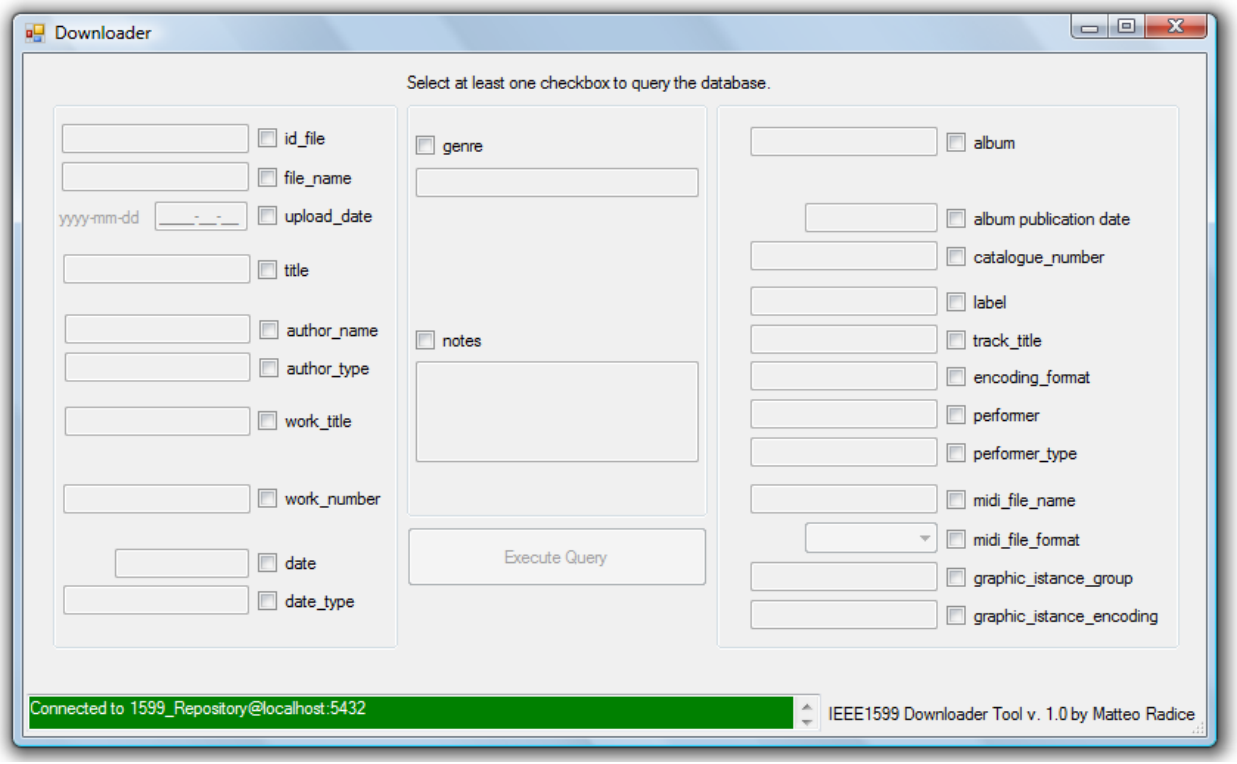

Fig. 9: Maschera di ricerca dei documenti in Downloader

Su un parametro di tipo "semplice", come può essere il campo "title", il codice si presenta in questo modo:

```
private void titleCheck_CheckedChanged(object sender, EventArgs e)
{
    boxCheck(titleCheck, title);
    executeCheck();
}
```

Vengono richiamate due funzioni, scritte per evitare di ripetere sequenze di codice sempre uguali: la maggioranza dei campi sui quali l'utente può agire è costituita come già detto da semplici textbox attivate dalla checkbox posta alla loro destra. Quando lo stato di questa checkbox viene cambiato, viene richiamata la funzione "boxCheck", che ha come argomenti il nome della checkbox sulla quale l'utente ha cliccato e il nome del relativo campo testo.

```
private void boxCheck(CheckBox check, TextBox text)
{
    if (check.CheckState.Equals(CheckState.Checked))
    {
        text.Enabled = true;
    }
    else
```

```
            {
                text.Enabled = false;
            }
        }
```

La funzione "boxCheck" non fa altro che occuparsi della abilitazione (o disabilitazione) della casella di testo relativa a una checkbox in base allo stato "Checked" oppure "Unchecked / Indeterminate".
Questa funzione, in ogni caso, non copre tutte i parametri impostabili dall'utente. Sono presenti infatti campi di tipo diverso da Textbox, come "upload_date" che è un controllo MaskedTextBox e si presenta con una maschera di tipo regex per l'input del formato di data corretto ("YYYY/MM/DD", anno in 4 cifre, mese in due, giorno in due); e ci sono casi nei quali i campi sono legati da una relazione logica "padre-figlio". L'esempio può essere il campo "album". Selezionando la sua checkbox non solo appare il campo testo per inserire il titolo dell'album, ma anche una seconda checkbox per attivare il campo testo per inserire il numero di traccia. Lo stesso tipo di soluzione è adottata anche per "Work" e il relativo attributo "number".

```
private void albumCheck_CheckedChanged(object sender, EventArgs e)
        {
            boxCheck(albumCheck, album);
            if (albumCheck.CheckState.Equals(CheckState.Checked))
            {

                trackNumberCheck.Enabled = true;
                trackNumberCheck.Visible = true;
                trackNumber.Visible = true;
            }
            else
            {
                trackNumberCheck.Enabled = false;
                trackNumberCheck.Visible = false;
                trackNumber.Visible = false;
            }
            executeCheck();
        }
```

Viene usata ancora "boxCheck" perchè i campi in questione sono ancora delle textbox e viene aggiunta la gestione di abilitazione e di visibilità dei controlli figlio "sbloccati" dalla selezione di un parametro padre ("album"). Il parametro figlio ("track_number") presenta invece la stessa soluzione già vista precedentemente.

```
    private void trackNumberCheck_CheckedChanged(object sender, EventArgs e)
            {
                boxCheck(trackNumberCheck, trackNumber);
                executeCheck();
            }
```

Un caso che combina sia la presenza di controlli differenti da una Textbox sia la gestione di parametri figlio è il campo "genre". Viene data la possibilità sia di specificare, nella casella di testo, la descrizione testuale del genere che si sta cercando, sia di specificarne il peso, compreso tra due valori inseriti con controlli di tipo NumericUpDown. Ultimo parametro particolare è la selezione del tipo di codifica Midi applicata a un file multimediale sincronizzato dal documento IEEE 1599 che si intende cercare. Come scritto sul manuale di mx, il tag <midi_file_format> può assumere tre valori, 0,1 e 2. Pertanto, la realizzazione più naturale prevede l'uso di una combobox, nella quale sono inseriti di default tali valori. All'attivazione del campo, deve essere richiamato automaticamente un primo valore, per esempio "0".

```
private void midiFileFormatCheck_CheckedChanged(object sender, EventArgs e)
        {
            if (midiFileFormatCheck.CheckState.Equals(CheckState.Checked))
            {
                midiFileFormat.Enabled = true;
                midiFileFormat.Text = "0";
            }
            else
            {
                midiFileFormat.Enabled = false;
            }
            executeCheck();
        }
```

La funzione "ExecuteCheck" si occupa invece di controllare la presenza di almeno una checkbox selezionata, in modo da abilitare il pulsante di esecuzione della query. E' costituita da un controllo su tutte le checkbox del pannello, utilizzando l'operatore OR.

```
private void executeCheck()
        {
            if (authorNameCheck.CheckState.Equals(CheckState.Checked)
                || authorTypeCheck.CheckState.Equals(CheckState.Checked)
                || fileIDcheck.CheckState.Equals(CheckState.Checked)
                || fileNameCheck.CheckState.Equals(CheckState.Checked)
                || titleCheck.CheckState.Equals(CheckState.Checked)
                || uploadDateCheck.CheckState.Equals(CheckState.Checked)
                || workTitleCheck.CheckState.Equals(CheckState.Checked)
                || workNumberCheck.CheckState.Equals(CheckState.Checked)
                || dateCheck.CheckState.Equals(CheckState.Checked)
                || dateTypeCheck.CheckState.Equals(CheckState.Checked)
                || genreCheck.CheckState.Equals(CheckState.Checked)
                || notesCheck.CheckState.Equals(CheckState.Checked)
                || albumCheck.CheckState.Equals(CheckState.Checked)
                || albumPublicationDateCheck.CheckState.Equals
                   (CheckState.Checked)
                || labelCheck.CheckState.Equals(CheckState.Checked)
                || catalogueNumberCheck.CheckState.Equals(CheckState.Checked)
```

```
                || trackTitleCheck.CheckState.Equals(CheckState.Checked)
                || performerCheck.CheckState.Equals(CheckState.Checked)
                || performerTypeCheck.CheckState.Equals(CheckState.Checked)
                || encodingFormatCheck.CheckState.Equals(CheckState.Checked)
                || midiFileFormatCheck.CheckState.Equals(CheckState.Checked)
                || midiFileNameCheck.CheckState.Equals(CheckState.Checked)
                ||graphicIstanceGroupCheck.CheckState.Equals
                  (CheckState.Checked)
                || graphicEncodingCheck.CheckState.Equals(CheckState.Checked))
            {
                DoQuery.Enabled = true;
            }
            else
            {
                DoQuery.Enabled = false;
            }
        }
```

Cliccando sul pulsante "DoQuery" (il cui testo, visibile all'utente, è in effetti "Execute Query"), la funzione procede a fare un controllo sequenziale su tutti i campi attivi raccogliendone i contenuti testuali e creando dinamicamente la query per il repository in base a quanto selezionato e inserito dall'utente. Per fare questo, è necessario lavorare su tre differenti stringhe, che identificano le tre parti fondamentali di una query sql: Select, From, Where. Viene gestita anche la clausola di ordinamento "Order By". La logica di funzionamento è quella di impostare le tre stringhe e andare a concatenare successivamente le informazioni inserite nelle varie caselle di testo.

```
private void DoQuery_Click(object sender, EventArgs e)
        {
            String select = "SELECT DISTINCT \"TFile_MX\".\"ID_file\",
            \"TFile_MX\".\"main_title\", \"TFile_MX\".\"other_title\" ";
            String from = " FROM \"TFile_MX\"";
            String where = " WHERE \"TFile_MX\".\"ID_file\"
                           = \"TFile_MX\".\"ID_file\"";
            String order_by = "";

            bool valid = false;
            bool tMidiJoin = false;
            bool tAlbumJoin = false;
            bool tTrackJoin = false;
            bool tPerformersJoin = false;
            bool tAuthorJoin = false;
            bool tWorkJoin = false;
            bool tDateJoin = false;
            bool tGraphicIstanceJoin = false;
            bool order = false;
```

Come base per la stringa di select vengono dati i campi che appartengono alla tabella centrale "TFile_MX" e che vengono sempre mostrati, qualunque sia la query inviata al repository. Questi campi sono la posizione in cui è stato inserito il file sul database (ID_file) e i titoli del brano. La selezione per ora avviene su una sola tabella (e quindi di conseguenza nella stringa "from" è presente solo "TFile_MX") e nella stringa "where" viene messa una condizione sempre verificata: la

stringa è impostata in questo modo per permettere con maggiore facilità le concatenazioni di clausole come verrà mostrato successivamente.
Il vero problema della generazione dinamica delle query sta però nelle join tra più tabelle. E' infatti necessaria una logica che aggiunga tabelle nelle clausole From, campi nella stringa di Select, e ovviamente criteri di ricerca nella stringa Where, in base a quanto selezionato dall'utente. Se l'utente fa una ricerca su campi relativi all'autore, la visualizzazione dei dati successiva dovrà mostrare solo i campi selezionati; di conseguenza la join fra le tabelle dovrà includere solo quelle effettivamente necessarie, in modo da non incorrere nel problema di una ricerca senza risultati a causa di corrispondenze mancanti.
Per come è strutturato il repository molti campi dipendono dalle medesime join fondamentali (e non potrebbe essere altrimenti, rispecchiando completamente la struttura a layer dei tag propria di un documento IEEE 1599-2008) alle quali vengono successivamente aggiunte altre tabelle. Un secondo rischio è quindi quello di ripetere erroneamente le medisime clausole, generando una query SQL non valida.
Vengono dichiarati una serie di valori booleani per il controllo della validità generale della query, per verificare che una determinata join non sia stata già effettuata e per gestire l'ordinamento dei campi nella visualizzazione dei risultati.
Il metodo con il quale viene creata la query da inviare al repository varia a seconda dei campi selezionati. Verranno quindi mostrati esempi per ogni caso differente.

Le operazioni relative a tag rappresentati nella tabella "TFile_Mx" sono le più semplici del sistema: come già fatto notare sopra, la base dell'intera query (formata dalle tre stringhe Select, From, Where) prevede già l'inclusione di questa tabella e dei campi "main_title", "other_title", "ID_file". Nella concatenazione finale delle tre stringhe, vengono aggunti in fondo i campi non presenti nel documento IEEE 1599 ma creati tramite l'inserimento del documento nel repository con il plugin di Upload, cioè "file_name" e "upload_date". Nel caso quindi di query su questi due campi, non è necessaria alcuna aggiunta alla clausola SELECT.
Fanno parte di quest'area i campi del pannello di ricerca "id_file", "file_name", "upload_date", "title" e "notes". Prendendo ad esempio il campo "title", la parte di codice che gestisce la sua selezione e inclusione nella query è questa:

```
if (titleCheck.CheckState.Equals(CheckState.Checked))
        {
            where += "AND (lower(\"TFile_MX\".\"main_title\") LIKE LOWER(\'%" +
            title.Text + "%\') OR lower(\"TFile_MX\".\"other_title\") LIKE
            LOWER(\'%" + title.Text + "%\')) ";
```

```
        if (order == false)
          {
            order_by = "ORDER BY \"TFile_MX\".\"main_title\" ASC,
            \"TFile_MX\".\"other_title\" ASC";
            order = true;
          }
        else
          {
            order_by += ", \"TFile_MX\".\"main_title\" ASC,
            \"TFile_MX\".\"other_title\" ASC";
          }
        valid = true;
    }
```

Viene fatto un controllo sulla checkbox relativa al campo testo "title". Se si trova nello stato "Checked", le relative stringhe di codice sql vengono concatenate a "select", "where", "from", "order". Nel caso di "title" non c'è bisogno di effettuare una join per aggiungere una tabella, perchè "Tfile_Mx" è sempre selezionata come tabella principale. Viene quindi fatta solo una semplice concatenazione alla clausola WHERE: la ricerca del testo inserito nella textbox "title" viene effettuata sia sul campo "main_title" che su quello "other_title", in modo da avere qualsiasi corrispondenza possibile tra i dati inseriti nel repository. Questa clausola WHERE di esempio mostra il motivo della clausola "WHERE \"TFile_MX\".\"ID_file\" = \"TFile_MX\".\"ID_file\"" inserita all'inizio nella stringa "where": è utile per avere delle concatenazioni semplici, concatenando stringhe che iniziano con "AND..." a cui segue la nuova clausola.

L'uso della forma "lower(nome campo) = LOWER(testo inserito dall'utente)" per gestire qualsiasi combinazione di lettere maiuscole e minuscole è combinato alla sintassi " nome campo LIKE %testo inserito%" al posto della semplice uguaglianza, che usando i caratteri jolly "%" permette di trovare la sequenza di caratteri inserita in qualsiasi posizione si trovi in una stringa, sia che siano presenti caratteri "estranei" che la precedono, sia che la seguano. Questa forma è applicata a tutti i campi testuali del pannello di ricerca.

La logica della clausola "ORDER BY" è ancora simile. Per non forzare in alcun modo l'ordine mettendo un ordinamento di default, viene fatto un controllo sulla variabile booleana "order". Se è in stato "false" significa che la clausola non è stata ancora aggiunta, e viene quindi dato alla stringa "order_by" un valore del tipo "ORDER BY (campi relativi a questa query)", costituendo il criterio principale; se invece un primo criterio di ordinamento è stato dato (valore di order quindi TRUE), ne viene concatenato un altro, iniziando la stringa con una virgola e elencando i campi aggiuntivi e il verso dell'ordinamento. L'ordine nel quale queste clausole vengono verificate però è necessariamente sequenziale, e di conseguenza lo è quello delle checkbox, elencato qui di seguito: campi relativi al file; agli autori; all'opera nel quale il brano è inserito; alle date significative per il

brano; alla descrizione del genere del brano; alle tracce audio (e quindi degli album e degli interpreti) per il quale è fornita una sincronizzazione; ai file midi relativi al brano musicale; alle rappresentazioni grafiche delle partiture del brano. La concatenazione di ulteriori clausole FROM e WHERE nel caso sia necessaria una NATURAL JOIN con altre tabelle è controllata tramite i valoro booleani specifici per ogni possibile join prevista. Per meglio gestire il sistema di concatenazione, non vengono effettuate le join con una sintassi del tipo "FROM nome tabella 1 NATURAL JOIN nome tabella 2" che necessiterebbe cambiamenti radicali della stringa "from", e non una semplice aggiunta di tabelle con stringhe del tipo ", nome_nuova_tabella". Le natural join vengono quindi esplicitate nella stringa where, facendo uguaglianze sui campi corrispondenti delle tabelle da collegare, "AND TWork.ID_work = TFile_Work.ID_work AND TFile_Work.ID_file = TFile_MX.ID_file". Nell'esempio del campo "work" è possibile inoltre osservare il funzionamento in questa fase di formazione della query dei campi testo "figlio" di un certo parametro, in questo caso "number" (cioè il numero del brano nella scaletta dell'opera).

```
if (workTitleCheck.CheckState.Equals(CheckState.Checked))
            {
                if (tWorkJoin == false)
                {
                    select += ", \"TWork\".\"work_title\",
                    \"TFile_Work\".\"number\", \"TWork\".\"work_number\"";
                    from += ", \"TWork\", \"TFile_Work\"";
                    where +=
                  "AND \"Twork\".\"ID_work\"=\"Tfile_Work\".\"ID_work\"
                  AND \"TFile_Work\".\"ID_file\"=\"TFile_MX\".\"ID_file\" ";
                    tWorkJoin = true;
                }

                where += "AND lower(\"TWork\".\"work_title\") LIKE
                LOWER(\'%" + workTitle.Text + "%\')";

                if (order == false)
                {
                    order_by = "ORDER BY \"TWork\".\"work_title\" ASC,
                    \"TFile_Work\".\"number\" ASC";
                    order = true;
                }
                else
                {
                    order_by += ", \"TWork\".\"work_title\" ASC,
                    \"TFile_Work\".\"number\" ASC";
                }

                if (numberCheck.CheckState.Equals(CheckState.Checked))
                {
                    where += "AND lower(\"TFile_Work\".\"number\") LIKE
                    LOWER(\'%" + number.Text + "%\')";
                }
                valid = true;
            }
```

Il primo controllo effettuato in questa funzione è la verifica del valore bool "tWorkJoin". Di default è falso, nel caso contrario significa che la join su queste tabelle ("TWork" e "TWork_number") è già stata aggiunta alla stringa generale. Nel caso in cui sia una condizione non verificata, viene effettuata la concatenazione sulle tre stringhe che compongono la query, selezionando i nuovi campi da visualizzare e creando l'unione fra le tabelle. Viene settato il valore bool "tWorkJoin" a "true". Viene concatenata alla stringa "where" la clausola che contiene il valore da cercare e viene gestito come nell'esempio precedente l'aggiunta di nuovi criteri di ordinamento. L'ultimo controllo è fatto sullo stato della checkbox "numberCheck", cioè il parametro "number" figlio di "work". Se si trova nello stato "checked", viene aggiunta la clausola WHERE relativa al contenuto del campo.
La gestione delle join diventa più complessa nell'area di parametri dipendenti contemporaneamente dalla tabella "TTrack" e da "TAlbum", con le relative tabelle di raccordo, e ovviamente da quella centrale al database "TFile_Mx". In questi casi sono due le join da gestire e verificare. Viene prima verificato lo stato della join sulla tabella "TTrack", in modo che sia collegata a "TFile_Mx". Successivamente viene verificata la natural join con "TAlbum" e "TAlbum_Tracks", ed eventualmente realizzata.

```
if (albumCheck.CheckState.Equals(CheckState.Checked))
            {
                if(tTrackJoin == false)
                {
                    from += ", \"TTrack\"";
                    where += " AND \"Tfile_MX\".\"ID_file\"
                  = \"TTrack\".\"ID_file\"";
                    tTrackJoin = true;
                }
                if (tAlbumJoin == false)
                {
                 select +=
                  ", \"TAlbum\".\"title\", \"Talbum_Track\".\"track_number\",
                   \"TAlbum\".\"number_of_tracks\",
                  \"Talbum\".\"catalogue_number\",
                   \"TAlbum\".\"publication_date\" ";

                 from += ", \"TAlbum\", \"TAlbum_Track\"";
                 where += "AND \"Talbum_Track\".\"ID_album\"
                  = \"TAlbum\".\"ID_album\"
                  AND \"TTrack\".\"ID_track\" = \"TAlbum_Track\".\"ID_track\"";
                 tAlbumJoin = true;
                }

                where += "AND lower(\"TAlbum\".\"title\") LIKE LOWER(\'%" +
                          album.Text + "%\')";

                if(trackNumberCheck.CheckState.Equals(CheckState.Checked))
                {
                    where += "AND lower(\"TAlbum_Track\".\"track_number\") LIKE
                    LOWER(\'%" + trackNumber.Text + "%\')";
                }
```

```
            if (order == false)
            {
                order_by = "ORDER BY \"TAlbum\".\"title\" ASC";
                order = true;
            }
            else
            {
                order_by += ", \"TAlbum\".\"title\" ASC";
            }

            valid = true;
        }
```

Alla fine quindi dell'intera verifica di tutti i campi textbox presentati nel pannello di ricerca, viene formata la stringa che costituisce la stringa SQL per l'interrogazione del database. L'intera sequenza di scrittura e invio del comando Odbc viene effettuata se la query è giudicata valida (tramite la variabile booleana "valid" alla quale viene dato valore nelle funzioni precedenti) . Le stringhe "select", "from", "where", "order by" vengono concatenate (aggiungendo come già specificato prima i campi "file_name" e "upload_date") e viene creato il comando SQL.

```
if (valid) //se almeno una checkbox è segnata
   {
     String query = select + ", \"TFile_MX\".\"file_name\",
      \"TFile_MX\".\"upload_date\"" + from + where + order_by + ";";

      System.Data.Odbc.OdbcCommand command =
      new System.Data.Odbc.OdbcCommand(query,connessione);
```

Usando una struttura try-catch viene inviato il comando Odbc contente la query al database. Con il metodo "ExecuteReader()" vengono letti i risultati della query e inseriti nell'istanza globale di classe OdbcDataReader "reader"[6]. Utilizzando la proprietà HasRows di questa classe, è possibile verificare se la query ha dato effettivamente dei risultati: nel caso siano presenti dei dati in output, viene visualizzato il numero di righe restituite [6] (tramite la conversione a stringa del valore int RecordsAffected) e viene caricato il pannello di visualizzazione dei risultati. In caso contrario viene notificata l'assenza di risultati.

```
      if (reader.HasRows == true)
       {
         status.ForeColor = Color.White;
         status.BackColor = Color.Green;
         status.Text = "Query executed, " +
         reader.RecordsAffected.ToString() + " rows affected";

         //gestione pannelli, da pannello di query a quello di visualizzazione
```

```
this.panel2.Hide();
this.panel4.Show();
```

3.2.3 Visualizzazione tabulare dei risultati delle interrogazioni

I dati presenti nell'OdbcDataReader vengono letti singolarmente e inseriti in una tabella creata dinamicamente come oggetto di classe DataGridView, definendo attraverso dei cicli le righe, le colonne e il loro contenuto. In fondo alle colonne (create in base al numero di campi restituiti come risultato della query) in questo plugin sono presenti due pulsanti per ogni riga, per permettere il download del documento IEEE 1599 sia con validazione, sia senza.

L'oggetto di classe DatagridView viene inizializzato svuotandolo da risultati precedenti (nel caso per esempio si sia effettuata precedentemente una ricerca e si torni col tasto "Back" al pannello di ricerca per effettuarne una seconda). Vengono definite le istanze di classe DataGridViewCell, DataGridViewColumn e DataGridTextBoxColumn che costituiscono rispettivamente i template da seguire nella creazione delle celle, delle intestazioni delle colonne e del tipo di colonne (cioè contenenti testo) che verranno create [6]. Settando a True la proprietà "Capture" la tabella è predisposta a ricevere l'input del mouse, necessario a gestire il click per il download del file.

```
dataGridView1.Columns.Clear();
dataGridView1.Capture = true;
DataGridViewColumn colHead;
DataGridTextBoxColumn colStyle = new DataGridTextBoxColumn();
DataGridViewCell cell = new DataGridViewTextBoxCell();
```

Il codice di formazione della tabella conta il numero di campi presenti in "reader", crea in un ciclo un totale di colonne pari al numero di campi presenti nei record restituiti e le etichetta prendendo il nome del campo letto e lo stile di cella definito sopra. Usando il metodo "Add" vengono aggiunte nuove colonne all'oggetto "dataGridView1".[6]

```
int intDgvCount = reader.FieldCount;
int intDgvMin = 0;
while (intDgvMin >= 0 && intDgvMin < intDgvCount)
  {
       colHead = new DataGridViewColumn();
       colHead.Name = reader.GetName(intDgvMin) as string;
       colHead.CellTemplate = cell;
       dataGridView1.Columns.Add(colHead);
       intDgvMin++;
   }
```

Una volta definite le colonne che vanno a comporre la tabella, si aggiungono le due colonne composte di celle di tipo "button". La loro logica di funzionamento è esattamente la stessa delle

celle normali; non vengono utilizzate propriamente come bottoni, ne hanno solo l'aspetto. La gestione del click su uno di questi bottoni è realizzata in funzione del numero di cella su cui è stato fatto, in modo tale che se il numero corrisponde con la posizione della cella "dgBtnCol", venga effettuata l'azione predisposta logicamente per quel tasto (in questo caso il download del documento). Tale funzione verrà spiegata successivamente. Viene riportata qui sotto un esempio di dichiarazione di una di queste colonne.

```
DataGridViewButtonColumn dgBtnCol = new DataGridViewButtonColumn();
dgBtnCol.Name = "Download";
dgBtnCol.Text = "Download file";
DataGridViewButtonCell buttoncell = new DataGridViewButtonCell();
dgBtnCol.CellTemplate = buttoncell;
dataGridView1.Columns.Add(dgBtnCol);
dgBtnCol.UseColumnTextForButtonValue = true;
```

La struttura della tabella è stata creata completamente, si procede alla scrittura dei record leggendoli dai risultati restituiti dal repository. La lettura viene effettuata usando il metodo "Read" della classe OdbcDataReader, che costituisce anche la condizione del ciclo primo ciclo while: finchè ci sono record da leggere il ciclo continua. Nella routine interna invece, i campi dei risultati di una riga vengono letti e inseriti nel vettore di stringhe "holder". Una volta arrivato all'ultimo campo da inserire nel vettore, viene aggiunta l'intera riga completa alla tabella. La logica generale è quindi quella di avere prima la riga completa inserita in un vettore a stringhe di lunghezza pari al numero dei campi contati con il metodo "FieldCount()"; successivamente aggiungere l'intera riga trasferendola dal vettore all'oggetto "dataGridView1".

```
intDgvMin = 0;
string[] holder = new string[intDgvCount];
while (reader.Read())
   {
   while (intDgvMin >= 0 && intDgvMin < intDgvCount)
     {
      try
         {
           holder[intDgvMin] = reader.GetString(intDgvMin) as string;
         }
      catch
         {
           holder[intDgvMin] = "* tag not inserted *";
         }

        if (intDgvMin == (intDgvCount - 1))
           {
             dataGridView1.Rows.Add(holder);
           }
        intDgvMin++;
      }
      intDgvMin = 0;
    }
```

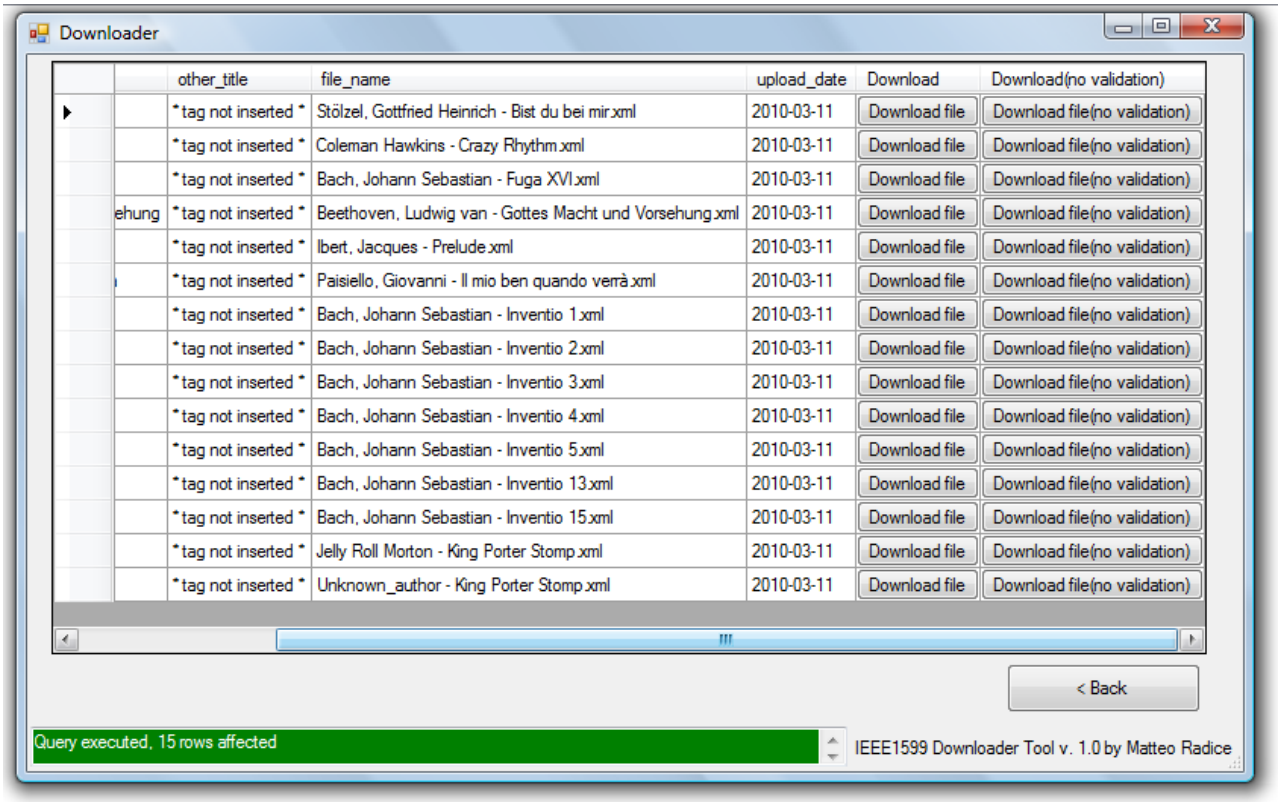

Fig. 10: Visualizzazione tabulare dei risultati della ricerca in Downloader

La gestione del ritorno dal pannello di visualizzazione dei risultati a quello di ricerca è affidata alla funzione che controlla il bottone "Back".

```
private void backbutton_Click(object sender, EventArgs e)
{
    this.panel4.Hide();
    this.panel2.Show();
    status.BackColor = Color.Black;
    status.ForeColor = Color.White;
    status.Text = "Ready.";
}
```

3.2.4 Download dei documenti IEEE 1599-2008

Il download dei documenti viene effettuato tramite una query al repository che seleziona il campo "file_xml" (contente il codice xml del documento IEEE 1599-2008) nella posizione ID indicata. Tale posizione deve essere presa automaticamente dal plugin in base alla riga della tabella sulla quale, tramite il bottone "Download" o "Download(no validation)", si sta cliccando. Il risultato deve inoltre subire un cast, effettuato all'interno della stessa query, per passare da un campo di tipo xml a

un risultato di tipo text. Questo cast può apparire inutile, ma, come verrà spiegato successivamente, è in realtà una necessità in conseguenza del metodo utilizzato nel comando di INSERT precedentemente spiegato.

Come già accennato, la gestione generale del funzionamento dei bottoni di download verte sul fatto che non funzionano come tali, ma ne hanno solo l'aspetto: sono celle normali, ed è necessario quindi un altro tipo di controllo. La funzione verifica un'azione di click innanzitutto su una generica cella della tabella. Viene poi verificata la posizione della cella sulla quale si è cliccato: contando il numero dei campi e sapendo che la numerazione parte da 0, le ultime due colonne, cioè quelle contenti i pulsanti di download, saranno sempre in posizione "dataGridView1.ColumnCount -1" e "dataGridView1.ColumnCount -2", dove "ColumnCount" è un metodo che restituisce il numero di colonne dell'istanza "dataGridView1". Se il click si verifica su queste colonne significa che è stato cliccato un bottone di scaricamento. Viene usata l'istanza "e" con relativa proprietà "ColumnIndex" per avere il numero di indice di colonna sulla quale si sta facendo l'azione di click. A seconda poi della posizione sulla quale è stata fatta, viene settata la variabile pubblica globale booleana "validation", che controlla la scelta di validazione o non validazione nel download del documento direttamente all'interno del Framework. Se il click viene effettuato su una qualsiasi cella che non sia nelle ultime due colonne, un messaggio di avvertimento viene visualizzato nella barra status.

```
private void dataGridView1_CellContentClick_1
      (object sender, DataGridViewCellEventArgs e)
        {
           if ((e.ColumnIndex == dataGridView1.ColumnCount - 1)||
            (e.ColumnIndex == dataGridView1.ColumnCount - 2))
            {
                if (e.ColumnIndex == dataGridView1.ColumnCount - 1)
                {
                    validation = false;
                }

                if (e.ColumnIndex == dataGridView1.ColumnCount - 2)
                {
                    validation = true;
                }
```

Gestita la scelta di validazione, si procede alla creazione della stringa sql che avrà come risultato il codice xml del documento selezionato. Come già accennato prima, è necessario un cast del campo "file_xml" da xml [] a text. Il motivo è che se questo cast mancasse il codice risultante sarebbe privo della riga di dichiarazione di versione xml (<?xml version="1.0" encoding="UTF-8"?>). Questo è legato al metodo di inserimento del codice xml nel campo "file_xml" del repository che fa uso del comando XMLPARSE. L'Id del documento desiderato viene ottenuto utilizzando ancora le proprietà dell'istanza "e", in questo caso prendendo il contenuto della cella di nome "ID_file" della

riga il cui indice è "e.RowIndex". Viene creata un'istanza OdbcCommand per inviare la query.

```
String id;
id = dataGridView1.Rows[e.RowIndex].Cells["ID_file"].Value.ToString();
String select = "SELECT CAST (\"TFile_MX\".\"file_xml\" as text)
FROM \"TFile_MX\" WHERE \"TFile_MX\".\"ID_file\" = " + id + ";";
status.Text = select;
System.Data.Odbc.OdbcCommand command_download = new
System.Data.Odbc.OdbcCommand(select, connessione);
```

Con una struttura try-catch viene eseguito il comando Odbc ed eventualmente gestiti errori di esecuzione. Il risultato presente in "reader" è solo uno, una stringa testuale che viene inserita nella posizione "0" del vettore di stringhe "file". Se l'operazione va a buon fine, l'intero form "Downloader" viene chiuso. I dati passano al form PluginDownload, che come nel caso di PluginUpload, è in diretto contatto con il Framework.

```
try
{
     reader = command_download.ExecuteReader();
     reader.Read();
     file[0] = reader.GetString(0);
     status.ForeColor = Color.Green;
     status.Text = "File imported";
     reader.Close();
     connessione.Close();
     dialogresult = DialogResult.OK;
     this.Dispose();
  }

 catch
{
     status.ForeColor = Color.White;
     status.BackColor = Color.Red;
     status.Text = "Failed to get desired IEEE1599 document from database";
}
```

Il documento scaricato è per ora nel vettore "file". Tale vettore è dichiarato come variabile privata del form "Downloader". Viene fornito un metodo pubblico richiamabile quindi da PluginDownload per ottenere il codice xml restituito dal repository.

```
public string[] getXmlFile()
{
    return file;
}
```

3.2.5 Integrazione di PluginDownload / Downloader nel Framework

L'integrazione di Downloader nel Framework è realizzata in maniera simile a quella di Uploader, anche se in un certo senso va nella direzione opposta. Se PluginUpload fornisce ad Uploader il documento da inserire favorendone quindi il passaggio dal Framework (col quale PluginUpload è in comunicazione), PluginDownload richiama Downloader per farsi restituire il documento IEEE 1599, per poi inviarlo direttamente al Framework e permetterne l'apertura. Viene creata quindi un'istanza del form Downloader e usando il metodo "ShowDialog()" viene visualizzato e messo a disposizione dell'utente (che si connetterà, farà la sua ricerca, visualizzerà i risultati e cliccherà sui pulsanti di download del documento che intende scaricare).
Controllando la buona riuscita delle operazioni con "dialogresult", se la sequenza di Downloader è stata completata (quindi dalla connessione all'effettivo download) viene caricato il codice xml scaricato da Downloader nel vettore "file". Questo caricamento viene effttuato richiamando il metodo pubblico "donwloader.getXmlFile()", che permette l'accesso alla vettore privato della classe "Downloader". Il codice viene trasferito nel vettore String "s".

```
public override bool DllExec(IEEE1599Document document)
{
    Downloader downloader = new Downloader();
    DialogResult dl = downloader.ShowDialog();

    if (downloader.dialogresult == DialogResult.Cancel)
    {
        return false;
    }
    else
    {
        String[] s = downloader.getXmlFile();
```

Viene controllata la scelta di validazione impostata dall'utente cliccando su "Download" o "Download(no validation)" verificando lo stato (true o false) della variabile pubblica "validation" della classe Downloader. Se "validation" è TRUE, il codice xml caricato nel vettore "s" viene semplicemente caricato con il metodo "LoadXml(String)" nell'istanza di classe IEEE1599Document "document". Questo caricamento è inserito in una struttura try-catch. Nel caso infatti l'utente non fosse connesso a internet (condizione necessaria affinchè la validazione vada a buon fine) si avrebbe una Exception: nel caso quindi non ci sia possibilità di validazione, la proprietà "XmlResolver" di "document" viene settata a "null" e un messaggio a video avvisa l'utente che a causa di un errore (il cui messaggio viene preso direttamente dall'Exception, qui indicata con "c") il file viene aperto senza la validazione. Un nuovo caricamento è quindi tentato e inserito in una

seconda try-catch per gestire eventuali errori nell'esecuzione del metodo "LoadXml", con relativi messaggi di errore.

Nel caso in cui l'utente ha scelto di non validare il documento, viene settato "XmlResolver" a null e tentato il caricamento, in una procedura analoga a quella descritta sopra.

```
if (downloader.validation) //apertura con validazione
  {
    try
      {
       document.Xml.LoadXml(s[0]);
       return true;
      }
    catch (Exception c)
      {
        document.Xml.XmlResolver = null;
        MessageBox.Show("Unable to validate the selected IEEE1599 Document: \n"
        + c.Message + "\n\nThe file will be loaded with no validation",
        "Unable to perform validation", MessageBoxButtons.OK,
        MessageBoxIcon.Exclamation);

        try
         {
           document.Xml.LoadXml(s[0]);
           return true;
         }
        catch (Exception a)
          {
             MessageBox.Show("Unable to load the selected IEEE1599 Document: \n"
            + a.Message, "ERROR", MessageBoxButtons.OK, MessageBoxIcon.Error);
             return false;
          }
       }

   }

else //senza validazione
 {
        document.Xml.XmlResolver = null;
        try
         {
          document.Xml.LoadXml(s[0]);
          return true;
         }
         catch (Exception b)
           {
             MessageBox.Show("Unable to load the selected IEEE1599 Document: \n"
             +b.Message, "ERROR", MessageBoxButtons.OK, MessageBoxIcon.Error);
              return false;
           }
  }
```

Come per PluginUpload, anche PluginDownload non ha effettivamente una propria veste grafica pur essendo il plugin effettivamente richiamato al caricamento dell'intero Framework.

3.3 Deleter

Il plugin di eliminazione dei documenti conservati nel repository "Deleter" è composto per la maggior parte da funzioni già presenti in Downloader, modificate per il nuovo scopo. Dopo la funzione di connessione, l'utente è posto di fronte a un pannello di ricerca uguale a quello di Downloader, e alla medesima logica di visualizzazione di risultati. Scopo del plugin è permettere di individuare con certezza assoluta il documento da cancellare. Dalla visualizzazione dei risultati è possibile selezionare il documento IEEE 1599 da eliminare con lo stesso meccanismo usato da Downloader per permettere il download dei file.
Questo plugin è fondamentale per la corretta gestione del repository, sebbene sia anomalo per il Framework, dato che non fa uso di documenti già aperti al suo interno nè ne apre di nuovi. Si è ritenuto comunque comodo la sua inclusione tra i form richiamabili dall'interno del Framework stesso per una totale gestione del repository senza dover operare direttamente su di esso.

3.3.1 Eliminazione di un documento IEEE 1599 dal repository

Il comando di eliminazione di un documento viene dato in maniera del tutto analoga a quello di scaricamento in locale su "Downloader". Avendo costituito la tabella di visualizzazione dei risultati di ricerca nello stesso modo, anche in "Deleter" è presente una colonna composta di celle di tipo "button" che, se cliccate, introducono l'utente alla procedura di eliminazione del documento selezionato. Il controllo è ancora il medesimo di "Downloader": viene verificata la posizione della cella sulla quale si è cliccato per identificare che sia effettivamente un bottone.

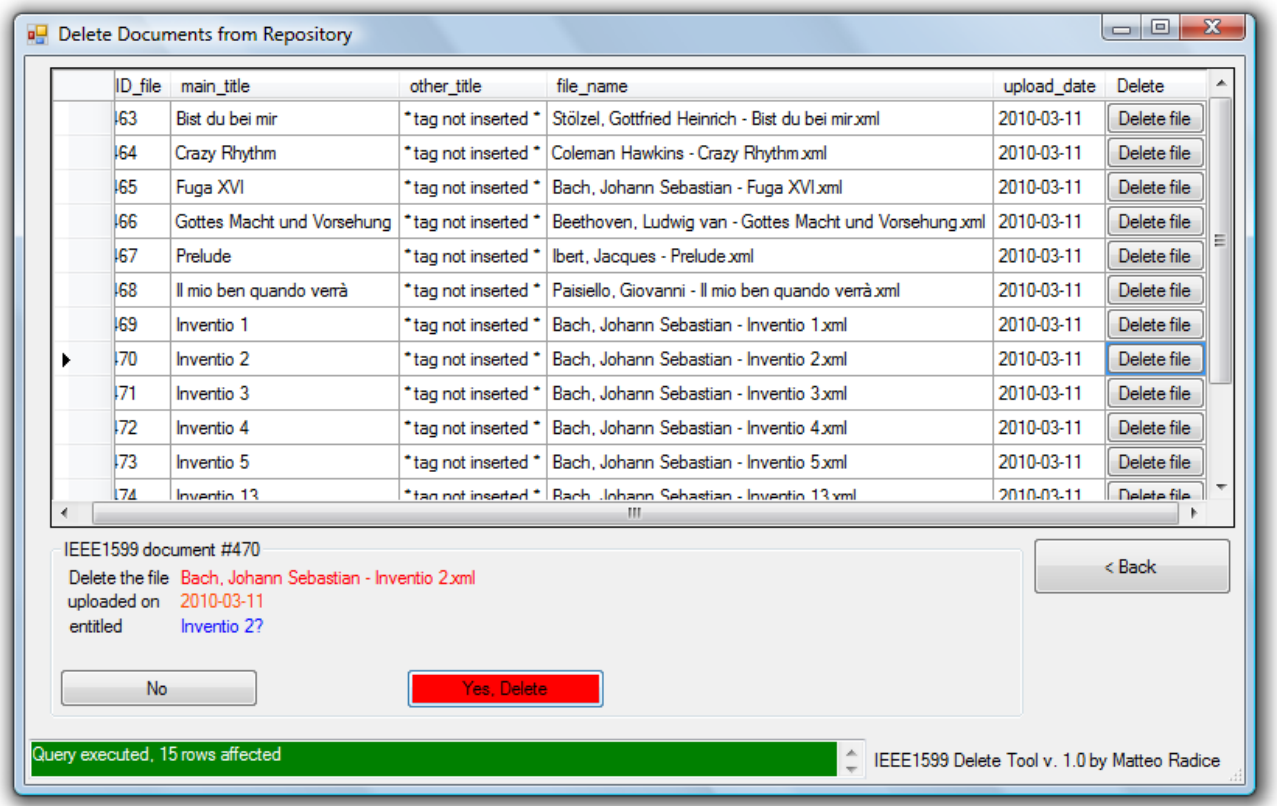

Fig. 11: Visualizzazione tabulare dei risultati della ricerca durante l'eliminazione di un file in Deleter

```
private void dataGridView1_CellContentClick_1(object sender,
DataGridViewCellEventArgs e)
    {
      if (e.ColumnIndex == dataGridView1.ColumnCount - 1)
        {
```

l click sul tasto "Delete File" fa comparire un pannello precedentemente nascosto. In questo pannello vengono assegnati a delle label predefinite il titolo del brano, la posizione in cui il file è stato inseito nel repository, il nome del file e la data di upload. Tali valori sono presi direttamente dalla tabella, utilizzando i puntatori di riga "e.RowIndex" e il nome della cella. Questo pannello ha come scopo quello di un doppio controllo: è una sorta di conferma della volontà dell'utente di voler eliminare il documento IEEE 1599 che ha selezionato. Cliccando su "Yes, Delete" viene avviata l'effettiva procedura, mentre "No" fa scomparire il pannello. Nelle variabili globali "id" e "rownumber" vengono inseriti i valori dell'ID file da eliminare e della riga corrispondente nella tabella "dataGridView1".

```
    selectedFileBox.Enabled = true;
```

```
selectedFileBox.Visible = true;

id = dataGridView1.Rows[e.RowIndex].Cells["ID_file"].Value.ToString();
rownumber = e.RowIndex;
selectedFileBox.Text = "IEEE1599 document #" + id;

fileNameLabel.Text =
   dataGridView1.Rows[e.RowIndex].Cells["file_name"].Value.ToString();
uploadDateDelete.Text =
   dataGridView1.Rows[e.RowIndex].Cells["upload_date"].Value.ToString();

 titleDeleteLabel.Text =
   dataGridView1.Rows[e.RowIndex].Cells["main_title"].Value.ToString() + "?";
```

La funzione del bottone "No" è una semplice gestione grafica del pannello.

```
private void noDeleteButton_Click(object sender, EventArgs e)
{
    selectedFileBox.Enabled = false;
    selectedFileBox.Visible = false;
}
```

La funzione che invece si occupa del caso nel quale l'utente abbia cliccato "Yes, Delete" contiene l'effettiva procedura di eliminazione. Tale procedura, per come è organizzato e ramificato il repository nell'adeguarsi alla struttura xml dei documenti IEEE 1599, deve essere necessariamente regolata ed effettuata da una procedura trigger all'interno del repository stesso.

Il meccanismo di eliminazione verte su un comando di Insert fatto sulla tabella "TDelete". Inserendo l'ID del documento da eliminare all'interno di questa tabella (che presenta un solo campo) viene fatto scattare un trigger che si occupa della cancellazione di quel documento. Questo ID viene salvato e immediatamente viene cancellato il documento IEEE 1599 identificato da questo numero dalla tabella "Tfile_MX". L'ID inserito in "TDelete" viene eliminato, seguito dalle procedure di "pulizia" delle tabelle regolate dal trigger stesso.

Viene pertanto creata un'istanza di OdbcCommand che si occupa di questa INSERT. Il comando viene eseguito con il metodo "ExecuteNonQuery". Se l'esecuzione va a buon fine viene rimossa la riga indicata dalla variabile globale "rownumber" viene eliminata, in modo che l'utente si accorga che il record è stato effettivamente eliminato. Un messaggio nella barra status conferma la buona riuscita dell'eliminazione.

```
private void yesDeleteButton_Click(object sender, EventArgs e)
        {
            String insert_delete = "INSERT INTO \"TDelete\" (\"ID_file\")
            VALUES(" + id + ");";
            System.Data.Odbc.OdbcCommand command_delete = new
            System.Data.Odbc.OdbcCommand(insert_delete, connessione);
            status.Text = "Please Wait...";
            status.BackColor = Color.Black;
            status.Update();
```

```
        try
        {
            command_delete.ExecuteNonQuery();
            command_delete.Dispose();
            dataGridView1.Rows.RemoveAt(rownumber);
            status.BackColor = Color.Green;
            status.Text = "File #"+ id +" deleted.";
        }
```

In una catch vengono gestiti gli eventuali errori di esecuzione del listato precedente.

```
        catch (Exception a)
        {
            status.ForeColor = Color.White;
            status.BackColor = Color.Red;
            status.Text = "Failed to delete desired IEEE1599 document from
            database: " + a.Message;
        }
        selectedFileBox.Enabled = false;
        selectedFileBox.Visible = false;
    }
```

3.3.2 Integrazione di PluginDelete/Deleter nel Framework

Introducendo il plugin di eliminazione dei documenti è stato messo in evidenza la sua differenza da ogni altro tipo di programmi sviluppati per lavorare nel Framework: non necessita l'apertura di un documento per poter funzionare, nè permette l'importazione di un documento IEEE 1599 esterno per una qualsiasi elaborazione. Queste caratteristiche vengono ampiamente riscontrate nell'integrazione tra il form "Deleter" che si occupa dell'effettiva eliminazione e il form in comunicazione col Framework "PluginDelete" Come già mostrato negli altri casi, "PluginDelete" crea un'istanza della classe "Deleter" con la quale l'utente interagisce e porta a termine le operazioni, in questo caso quelle di eliminazione.

```
    public override bool DllExec(IEEE1599Document document)
    {
        Deleter deleter = new Deleter();
        deleter.ShowDialog();
        return false;
    }
```

3.4 Updater

Il plugin di aggiornamento dei documenti IEEE 1599 già inseriti nel repository è "Updater". Il vero scopo di questo plugin è in realtà fornire all'utente un'interfaccia semplice e veloce per sostituire interi documenti (più che aggiornarli solo in determinate parti) conservati nel database con versioni più aggiornate dello stesso brano aperta nel Framework. L'interfaccia è strutturata come negli altri plugin, un form con un pannello di connessione, una maschera di ricerca e una visualizzazione tabulare dei risultati; nel caso si clicchi sul pulsante "Update", viene visualizzato un pannello che confronta il codice già presete nel repository con quello nuovo. Ciò implica che in "Updater" siano integrate tutte le funzioni viste precedentemente, riadattate a questo nuovo uso: è infatti presente il caricamento del documento aperto nel Framework all'interno del repository, il download del codice da sostituire per il confronto, l'eliminazione del documento considerato obsoleto.

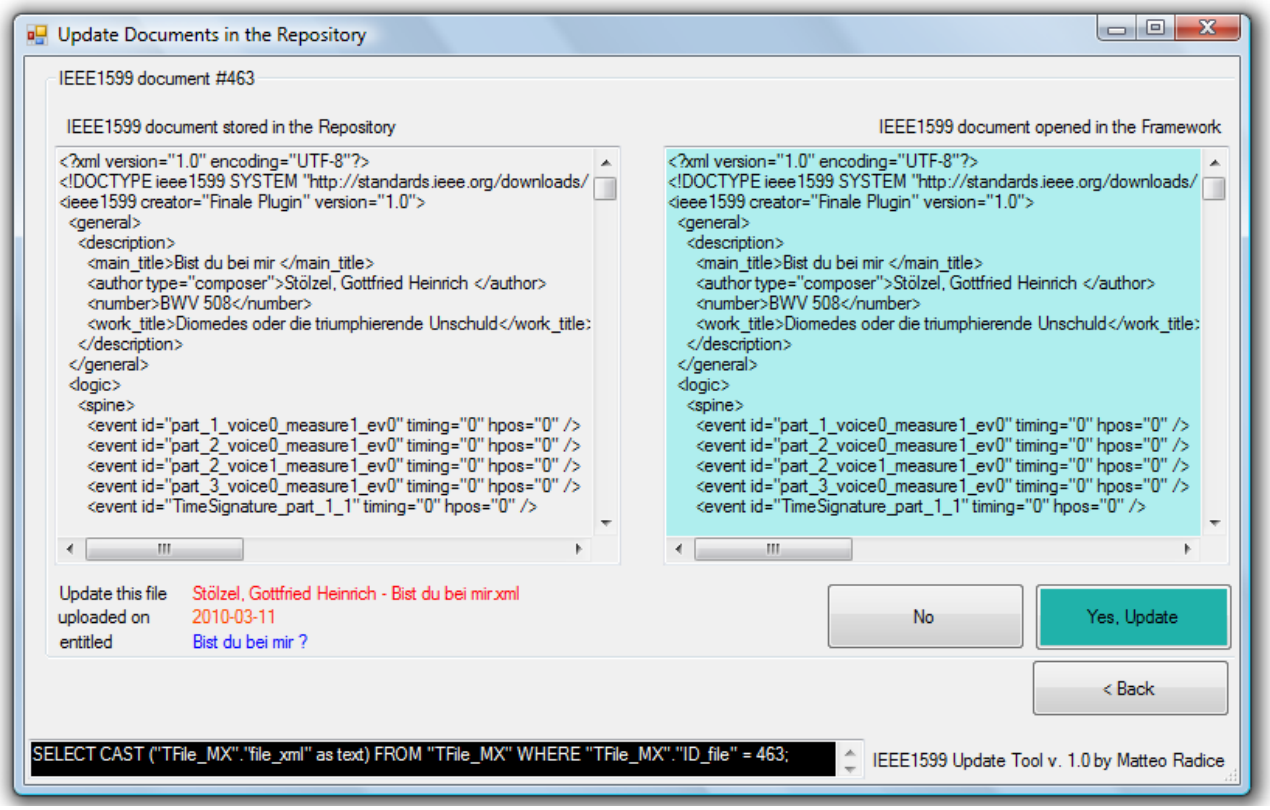

Fig. 12: Confronto fra documento conservato nel Repository e nuovo documento aperto nel Framework in Updater

3.4.1 Sostituzione di un documento IEEE 1599 nel repository

Come per il download e l'eliminazione, anche in "Updater" la procedura di aggiornamento su un certo documento è iniziata dall'utente con un click sul tasto "Update" nella visualizzazione tabulare dei risultati della ricerca. Il controllo è pertanto il medesimo (cioè sul numero di colonna per identificare la cella sulla quale si è cliccato come bottone e usando l'indice della riga sulla quale è stata fatta l'azione si trova il campo contrassegnato con "ID_File"). Viene mostrato e abilitato un pannello realizzato con il controllo windows form "GroupBox", che si sovrappone, nascondendola, alla tabella realizzata con "DataGridView". Questo pannello carica in una textbox il codice xml del documento già presente sul database recuperandolo con una select. Dalla tabella vengono recuperati invece il titolo, il nome del file e la data di caricamento, per identificare in maniera più chiara il file che si sta andando a sostituire. Prendendo il codice xml dalla variabile globale "xml", si carica all'interno dell'altra textbox il codice del nuovo documento IEEE 1599 che si intende inserire nel repository. L'uso del metodo "Update" sulle istanze di classe Textbox corregge un piccolo malfunzionamento che non permette la stampa a video del messaggio assegnato alla casella di testo ("status" per esempio) durante l'esecuzione di alcune funzioni come quelle di comunicazione via Odbc al database.

```
private void dataGridView1_CellContentClick_1(object sender,
DataGridViewCellEventArgs e)
   {
      if (e.ColumnIndex == dataGridView1.ColumnCount - 1)
       {
         status.BackColor = Color.Black;
         status.ForeColor = Color.White;
         status.Text = "Please wait...";
         status.Update();

         id = dataGridView1.Rows[e.RowIndex].Cells["ID_file"].Value.ToString();

         selectedFileBox.Text = "IEEE1599 document #" + id;
         fileNameLabel.Text =
         dataGridView1.Rows[e.RowIndex].Cells["file_name"].Value.ToString();
         uploadDateDelete.Text =
         dataGridView1.Rows[e.RowIndex].Cells["upload_date"].Value.ToString();
         titleDeleteLabel.Text =
         dataGridView1.Rows[e.RowIndex].Cells["main_title"].Value.ToString()
         + "?";

         String select = "SELECT CAST (\"TFile_MX\".\"file_xml\" as text)
         FROM \"TFile_MX\" WHERE \"TFile_MX\".\"ID_file\" = " + id + ";";
         status.Text = select;
         System.Data.Odbc.OdbcCommand command_download =
         new System.Data.Odbc.OdbcCommand(select, connessione);
```

```
        try
                {
                    reader = command_download.ExecuteReader();
                    reader.Read();
                    file[0] = reader.GetString(0);
                    oldFileTextBox.Text = file[0];
                    newFileTextBox.Text = xml;
                    selectedFileBox.Enabled = true;
                    selectedFileBox.Visible = true;
                }
                catch (Exception a)
                {
                    status.BackColor = Color.Black;
                    status.ForeColor = Color.White;
                    status.Text = "ERROR" +a.Message;
                    status.Update();
                    selectedFileBox.Enabled = false;
                    selectedFileBox.Visible = false;
                }
        }
```

Rispondendo in maniera affermativa alla richiesta di conferma che pone il pannello di sostituzione dei file, inizia la vera e propria procedura che prevede tramite comandi Odbc l'eliminazione del vecchio record e il caricamento nel repository del nuovo documento. Queste due operazioni si svolgono nella maniera già descritta per i plugin "Uploader" e "Deleter". Per quanto riguarda l'eliminazione del vecchio documento, come già mostrato in precedenza viene fatta una insert sulla tabella TDelete, in modo che il trigger interno al repository che si occupa dell'eliminazione scatti. Per l'upload viene fatta una INSERT che si occupa di caricare il codice xml nella tabella "TFile_Mx" per scatenare il trigger di inserimento, seguito da delle UPDATE per inserire il file name creato con una query xpath sul documento stesso e la data di caricamento. Viene inoltre fatta una SELECT per restituire all'utente la nuova posizione (cioè il nuovo valore "Id_file") in cui è stato inserito il documento appena caricato. E' necessario, poichè l'id del vecchio file e quelli del nuovo non coincidono. Ciò è dovuto alla gestione a sequenze di PostgreSQL per i campi di tipo "serial" o "big serial" usati per descrivere gli ID dei documenti.

```
private void yesUpdateButton_Click(object sender, EventArgs e)
   {
      String insert_delete = "INSERT INTO \"TDelete\" (\"ID_file\")
      VALUES(" + id + ");";
      System.Data.Odbc.OdbcCommand command_delete =
      new System.Data.Odbc.OdbcCommand(insert_delete, connessione);

      String insert = "INSERT INTO \"TFile_MX\" (file_xml)
      VALUES (XMLPARSE(document(E'" + xml + "')));";

      String name = "UPDATE \"TFile_MX\" SET file_name=\'" + filename + "\'
      WHERE \"ID_file\" = currval(\'\"TFile_MX_ID_file_seq\"\');";
```

```
String data = "UPDATE \"TFile_MX\" SET upload_date=\'" +
DateTime.Now.ToString("yyyy/MM/dd") + "\' WHERE \"ID_file\" =
currval(\'\"TFile_MX_ID_file_seq\"\');";

System.Data.Odbc.OdbcCommand command =
new System.Data.Odbc.OdbcCommand(insert, connessione);

System.Data.Odbc.OdbcCommand updatefilename =
new System.Data.Odbc.OdbcCommand(name, connessione);

System.Data.Odbc.OdbcCommand upload_date =
new System.Data.Odbc.OdbcCommand(data, connessione);

String newPosition = "SELECT \"TFile_MX\".\"ID_file\" FROM \"TFile_MX\"
WHERE \"ID_file\" = currval(\'\"TFile_MX_ID_file_seq\"\');";

System.Data.Odbc.OdbcCommand new_id = new
System.Data.Odbc.OdbcCommand(newPosition, connessione);
```

Avendo quindi già preparato tutti i comandi per l'eliminazione del vecchio documento, l'inserimento del nuovo codice passato direttamente dal Framework con il relativo nome file e la data di caricamento e la restituzione all'utente del nuovo ID del documento inserito si procede ad eseguire queste istanze OdbcCommand (e nel caso della nuova posizione OdbcDataReader) utilizzando i metodi “ExecuteNonQuery” e “Read”, inseriti all'interno di una struttura try-catch. Il funzionamento è quindi analogo ai casi precedentemente illustrati.

3.4.2 Integrazione di PluginUpdate/Updater nel Framework

L'integrazione di "Updater" attraverso il form "PluginUpdate" è esattamente la stessa già mostrata ed utilizzata su "Uploader". "PluginUpdate" si occupa infatti di estrapolare dal documento IEEE 1599 aperto nel Framework l'intero contenuto di codice xml in una stringa mantendone la formattazione indentata, utilizzando quindi la funzione "getFormattedXml". Il documento IEEE 1599 viene inoltre visitato con delle query xpath per ottenerne il primo autore e il titolo principale, utilizzati per formare il nome del file. Queste due stringhe vengono passate a un'istanza del form "Updater" che viene creata dallo stesso "PluginUpdate". Eventuali errori riscontrabili in queste operazioni sono gestite da una struttura try-catch.

Conclusioni

Risultati ottenuti e sviluppi futuri

Il progetto complessivo ha portato alla realizzazione di una base di dati automatizzata funzionante, utilizzabile direttamente attraverso il Framework IEEE 1599 e attraverso una pagina php che verrà inserita nel sito ufficiale del progetto MX. Le funzionalità dei plugin indicate negli obbiettivi elencati nel corso di questo elaborato sono stati raggiunti.

Tuttavia, l'implementazione nello schema della base di dati degli elementi contenuti in un documento IEEE 1599 non è completa: in futuro, la mappatura del documento potrebbe essere ulteriormente estesa e completata, comportando di conseguenza una estensione della funzione di composizione della query nelle parti di ricerca dei plugin. La stessa logica di ricerca potrebbe in futuro subire una ristrutturazione, in modo da offrire per esempio una navigazione meno rigida fra le informazioni contenute in un singolo documento e che stabilisca relazioni di similitudine o di qualche genere di attinenza fra più documenti IEEE 1599, permettendo un utilizzo più “agile” del Repository. Allo stato attuale, i plugin sono indipendenti l'uno dall'altro. Ciò comporta il vantaggio di non doverli includere per forza tutti in una ipotetica installazione per garantire il funzionamento di uno solo di loro. Una possibile evoluzione potrebbe però implementare tutte le funzioni di gestione in un'unica applicazione plugin per il Framework, aggiungendo nuove funzionalità di calcolo e inserimento di informazioni musicali quali tonalità utilizzate e time signature all'interno di specifiche relazioni create nel database. Una soluzione del genere darebbe possibilità di indicizzazione e ricerca per armatura in chiave del brano o per segnature di tempo (oppure, eseguendo altre elaborazioni, per BPM). Queste elaborazioni si sono rivelate impossibili da implementare dalla semplice analisi XPath effettuata dal trigger del Repository, mentre sono facilmente realizzabili in C#, e verrebbero inserite all'interno della procedura di caricamento del documento all'interno della base di dati.

Il progetto riguardante il Repository IEEE 1599 e i suoi plugin di gestione rimane dunque un inizio, un punto di partenza che potrà evolversi e dare origine a nuovi progetti di espansione paralleli alla diffusione dello standard per il quale è stato realizzato: le sue caratteristiche attuali rispondo alle esigenze degli utenti dei documenti IEEE 1599 riscontrate fino a questo punto e colmano la lacuna di un'archivio gestito direttamente nell'ambiente di sviluppo dedicato e che garantisce perfetta coerenza tra il contenuto del documento e i dati inseriti nelle tabelle, avviando così un primo processo di conservazione dei documenti sino ad oggi realizzati.

Bibliografia

1. Luca A. Ludovico, “Key concepts of the IEEE 1599 Standard”*, Proceedings of the IEEE CS Conference The Use of Symbols To Represent Music And Multimedia Objects*, IEEE CS, Lugano, Switzerland, 2008.

2. National Information Standards Organization (NISO), *Understanding Metadata,* NISO Press, 2001.

3. Silvana Castano, Alfio Ferrara, Goffredo Haus, Luca A. Ludovico, Stefano Montanelli, Giampaolo Racca, Giancarlo Vercellesi, “The MX Formalism for Semantic Web-Compatible Representation of Music Metadata”*, Advances in Multimedia Information Systems*, Lecture Notes in Computer Science, vol. 3665/2005, pp. 78-92, Springer Berlin / Heidelberg, 2005.

4. Thuan L. Thai, Hoang Lam, *.NET Framework Essentials*, O'Reilly Media, 2001.

5. Andrew W. Troelsen, *Pro C# 2008 and the .NET 3.5 Platform*, Apress, 2007.

6. Jason Price, *Mastering C# Database Programming*, Sybex, 2003.

7. Christian Nagel, Bill Evjen, Jay Glynn, and Morgan Skinner, *Professional C#*, Wrox, 2008.

8. William R. Vaughn, Peter Blackburn, *ADO.Net examples and best practices for C# programmers*, Apress, 2002 .

9. Bill Hamilton, *ADO.NET 3.5 Cookbook*, O'Reilly Media, 2008.

10. The PostgreSQL Global Development Group, *PostgreSQL 8.4 Official Documentation - Volume I. The SQL Language*, Fultus Corporation, 2009.

11. John E. Simpson, *Xpath and Xpointer: Locating Content in XML Documents*, O'Reilly Media, 2002.

12. Howard Katz, Don Chamberlin, Denise Draper, Mary Fernandez, *XQuery from the Experts: A Guide to the W3C XML Query Language,* Addison-Wesley Professional , 2003.

13. The PostgreSQL Global Development Group, *PostgreSQL 8.4 Official Documentation - Volume V. Internals and Appendixes,* Fultus Corporation, 2009.

Sitografia

14. Alex Mackman, Chris Brooks, Steve Busby, Ed Jezierski, Jason Hogg, Roberta Leibovitz, Colin Campbell, *.NET Data Access Architecture,* Microsoft Corp., 2001. [http://msdn.microsoft.com/en-us/library/ee817654.aspx]

15. *ODBC Programmer's Reference,* MSDN Library. [http://msdn.microsoft.com/en-us/library/ms714177%28VS.85%29.aspx]

16. Hao Chen, Heechul Lim, Jin Xiao, “Concept Architecture of PostgreSQL”, University of Waterloo David R. Cheriton School of Computer Science, 2002 [http://www.cs.uwaterloo.ca/~h8chen/course/798/ConceptArch.pdf]

Ringraziamenti

Ringrazio Simone Motta, i miei "colleghi" Giovanni Sala e Luca Trinchinetti , il professor Luca A. Ludovico e il dottor Adriano Baratè per il supporto tecnico che mi hanno dato nella realizzazione di questo progetto e nella stesura dell'elaborato che lo descrive; ringrazio inoltre Federica Grosso e i miei genitori, per il fondamentale supporto morale durante gli anni dell'Università.