Università degli Studi di Milano

Facoltà di Scienze e Tecnologie

Corso di Laurea in Informatica

SPAZI-UNIMI: PROGETTAZIONE E IMPLEMENTAZIONE DELL'INTEGRAZIONE E VALIDAZIONE DELLE DIVERSE FONTI DI DATI EDILIZI

Relatore: Prof. Carlo Bellettini

Correlatore: Dr. Matteo Camilli

Tesi di:
Paolo Venturi
Matricola: 775021

Anno Accademico 2013-2014

dedicato a ...

Prefazione

hkjafgyruet.

Organizzazione della tesi

La tesi è organizzata come segue:

- nel Capitolo 1 ....

Ringraziamenti

asdjhgftry.

Indice

Capitolo 1

Il progetto Spazi-Unimi

1.1 Introduzione al progetto

Il progetto Spazi-Unimi nasce dall'esigenza degli utenti (studenti, professori, etc.) dell'Università degli Studi di Milano di cercare in modo facile e veloce la posizione delle aule di loro interesse. Vista la dislocazione delle sedi universitarie in varie aree della città (e della regione) un nuovo studente o un visitatore può avere serie difficoltà nell'orientarsi: da qui l'idea di creare una App che semplifichi la ricerca degli edifici universitari e delle loro stanze.

Spazi-Unimi è stato ideato nell'ambito del progetto Campus Sostenibile, una collaborazione tra il Politecnico di Milano e l'Università degli Studi di Milano, che si propone di trasformare il quartiere Città Studi in un modello per quanto riguarda la qualità della vita e la sostenibilità. Dalla proposta del Prof. Carlo Bellettini è quindi partito lo sviluppo del progetto che è stato portato avanti con altri due studenti del dipartimento di Informatica: Samuel Gomes Brandao e Diego Costantino.

I file da cui estrarre le informazioni utili alla creazione dell'applicazione sono stati forniti da due diverse fonti:

- la Divisione Manutenzione edilizia e impiantistica che ha concesso le piantine delle sedi universitarie e le informazioni sulle aule didattiche;
- la Divisione sistemi informativi che ha concesso le informazioni sulle aule presenti sul sistema EasyRoom.

Durante le 18 settimane di stage interno il lavoro effettuato ha riguardato principalmente lo sviluppo della parte back-end che si propone di fornire agli addetti delle diverse fonti di dati un modo semplice e immediato per aggiornare le informazioni. La

parte su cui più si è incentrato il mio lavoro è stata l'unione dei dati provenienti dalle diverse fonti cercando di rendere disponibili all'utente finale le migliori informazioni per quanto riguarda completezza e qualità.

Nell'ultima parte dello stage invece ci si è concentrati sulla definizione di un'interfaccia REST API utile alle necessità della futura applicazione multipiattaforma scaricabile dagli utenti dell'università.

1.2 Il problema e i dati forniti per risolverlo

La prima attività svolta è stato uno studio di fattibilità: sapendo che sul mercato non era presente nessuna applicazione/tool simile a quella che si voleva sviluppare ci si è concentrati più sulla ricerca di possibili librerie utili all'analisi dei file forniti dalle varie fonti. Sia l'edilizia che i sistemi informativi hanno fornito dati sulle aule organizzati in fogli elettronici e scritti in formato XLS, per quanto riguarda le mappe messe a disposizione dall'edilizia invece le informazioni sono su file di tipo AutoCAD DWG.

I file XLS essendo per loro natura in formato tabellare risultano di non difficile lettura ma ancora più semplice risulta quella dei file CSV (Comma-separated values) un formato basato su file di testo ricavabile senza sforzo da fogli elettronici o da database.

Il formato AutoCAD DWG invece è risultato molto più complicato da analizzare in quanto risulta essere un file binario diviso in diverse sezioni la cui codifica è molto complessa. Vista l'impossibilità di ottenere dati in modo semplice si è cercato un formato più adatto ai nostri scopi in cui esportare la collezione di 606 file DWG forniti. La scelta è ricaduta sull'altro formato AutoCAD cioè il DXF: questo standard utilizza un file ASCII diviso in sezioni (HEADER, CLASSES, TABLES, ENTITIES, OBJECTS, THUMBNAILIMAGE ed END OF FILE) risultando abbastanza leggibile a chiunque. La sezione di maggior interesse per in nostri scopi è risultata ENTITIES che contiene tutti gli oggetti disegnati nel file con le loro caratteristiche.

Dimostrata la fattibilità del progetto partendo da questi formati di dati ci si è concentrati sulla ricerca degli scenari d'uso per l'applicazione:

- trovare le sedi universitarie vicine alla propria posizione;
- trovare le stanze di una certa categoria (biblioteche, aule, punti ristoro, etc...) più vicine;
- cercare le stanze per nome mostrando una lista in caso di ambiguità;
- mostrare le mappe interne degli edifici rendendole interattive;
- segnalare errori e problematiche con un apposito form in modo da rendere le informazioni disponibili sempre più corrette e affidabili.

1.3 Scelta di tool, tecnologie e tecniche di sviluppo

La fase successiva del progetto ha riguardato la scelta delle tecnologie da utilizzare durante lo sviluppo delle funzionalità: prima fra tutte la scelta del linguaggio di programmazione.

Dopo un'esplorazione delle varie possibilità è stato deciso di utilizzare Python nella sua ultima versione (la 3). Python è un linguaggio ad alto livello multi paradigma: è orientato agli oggetti ma possiede caratteristiche dei linguaggi funzionali che rendono molto più semplice e leggibile l'implementazione di alcuni pezzi di codice.

In Python le variabili non sono tipizzate (quindi ogni variabile è un puntatore ad un oggetto): il controllo dei tipi è comunque molto forte e viene fatto tramite tipizzazione dinamica. Per quanto riguarda la leggibilità del codice in Python viene utilizzata l'indentazione per dividere i programmi in blocchi: ciò porta il codice ad essere molto più elegante rispetto ad altri linguaggi e lo rende molto leggibile anche per chi non conosce il linguaggio.

Gli aspetti funzionali più importanti e più utili alla programmazione presenti in Python sono:

- le list comprehension, costruttori di liste che utilizzano una modalità di creazione molto intuitiva e matematica;

Listing 1.1: Esempio di List Comprehension

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- i generatori, simili alle list comprehension non occupano però memoria;

Listing 1.2: Esempio di Generatore

```
>>> g = ((x, y) for x in [1,2,3] for y in [3,1,4] if x != y)
>>> g
<generator object <genexpr> at 0x7fca9fb91240>
>>> list(g)
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- la parola chiave lambda, utilizzata per definire piccole funzioni utilizzate solo in certe zone del codice senza dover definire una funzione che non verrà più richiamata.

Listing 1.3: Esempio di utilizzo della lambda

```
>>> g = lambda x: x**2
>>> print(g(8))
64
```

Un'altra caratteristica che ci ha portato a scegliere Python è il fatto che sia un linguaggio interpretato e che fornisca un interprete da riga di comando avanzato (bpython) con il quale provare il codice risulta molto semplice e veloce.

Per quanto riguarda le performace Pyhton risulta migliore di altri linguaggi interpretati come PHP e Ruby e nonostante non sia paragonabile al C risulta abbastanza simile a linguaggi compilati (Java).

Ultimo vantaggio dell'usare Python ma dall'elevata importanza è il fatto che possieda una vasta libreria standard ed esista uno svariato numero di librerie importabili che possono svolgere e implementare molte funzionalità e algoritmi. Inoltre è presente un framework di testing unitario (unittest) utile a testare i vari metodi presenti nelle classi implementate: ciò ha reso possibile l'applicazione di modelli di sviluppo software come il TDD (Test Driven Development).

La seconda scelta da effettuare in ambito tecnologico è stata la tipologia di database: vista la grande mole di dati e la loro non completezza si è deciso di utilizzare MongoDB. MongoDB è il database NoSQL più diffuso a livello mondiale, la sua tipologia non è relazionale come per i database classici ma si basa sugli oggetti. La memorizzazione dei dati viene effettuata su una tipologia particolare di file JSON detti BSON con uno schema dinamico così che i campi che risulterebbero vuoti in un DB SQL qui non esistono.

Altro vantaggio della scelta di MongoDB è la possibilità di specificare query avanzate al database che in SQL non sarebbero possibili; grazie all'utilizzo di JavaScript inoltre si possono definire funzioni ad hoc da applicare ai risultati. Le prestazioni in lettura e scrittura di Mongo inoltre sono molto buone per dati di grosse dimensioni come quelli su cui abbiamo lavorato. Ciò è anche dovuto al fatto che in MongoDB esiste la possibilità di creare degli indici che velocizzino la ricerca per campi/documenti usati molto spesso nelle query. Tra gli indici creabili ve ne è uno in particolare che ci è sembrato molto utile per lo sviluppo del nostro progetto: l'indice geospaziale. Dovendo lavorare, tra le altre cose, anche sulle coordinate degli edifici universitari la possibilità di creare un indice per poter effettuare query specifiche basate sulle posizioni geospaziali è sembrata un enorme vantaggio.

Per fare lavorare al meglio il linguaggio (Python 3) con il database (MongoDB) abbiamo inoltre esplorato le librerie disponibili: la scelta è ricaduta su PyMongo che permette in modo facile e veloce di connettersi ad un database MongoDB da un codice Python. PyMongo inoltre fornisce la possibilità di effettuare semplicemente

ogni operazione possibile su di un DB: query, inserimenti, creazioni di collection e di indici.

Come per ogni progetto di una certa complessità sorge la necessità di utilizzare un sistema di versioning: la scelta è ricaduta su git uno dei più famosi ed utilizzati software di controllo di versione distribuito. Oltre alla possibilità di tenere traccia delle modifiche fatte ai file del progetto git permette di dividere il lavoro in branch separati così da poter sviluppare diverse funzionalità in autonomia e senza modificare il progetto centrale (nel branch 'master'). I vari branch sono poi unificabili grazie alla procedura di merge. La repository git così creata è stata poi caricata ad un servizio web di hosting specializzato: GitHub; in questo modo i vari elementi del team di sviluppo hanno potuto sempre avere il codice aggiornato alle ultime modifiche.

A questo punto l'esplorazione si è concentrata sui possibili tool/applicazioni utili all'organizzazione del lavoro e alla raccolta di appunti e note. Per quanto riguarda l'organizzazione è risultato molto utile e semplice da utilizzare Trello: si tratta di un'applicazione basata su cartelloni (board). Ogni board è diviso in liste ed ogni lista contiene delle tessere (card) ordinabili: indicando un'attività o una funzionalità da implementare su ogni card si ha un quadro generale molto chiaro dello stato del progetto. Durante lo sviluppo abbiamo ordinato le liste in base alle priorità che secondo noi possedeva ogni tessera; grazie alla possibilità di assegnare le tessere ai membri del team inoltre il lavoro di organizzazione è risultato facilitato.

L'ultima applicazione utilizzata è stata Evernote: utile alla raccolta di note ed appunti è risultata molto utile per salvare resoconti e insiemi di informazioni molto più grandi rispetto a quelli di Trello.

Come tecniche di sviluppo del software abbiamo cercato di applicare gli aspetti dell'Extreme Programming (XP) utili al nostro caso. XP è una metodologia di sviluppo del software appartenente alla famiglia delle metodologie agili ed è definita da 12 aspetti/attività principali:

1. Planning game
2. Brevi cicli di rilascio
3. Uso di una metafora
4. Semplicità di progetto
5. Testing
6. Refactoring
7. Programmazione a coppie
8. Proprietà collettiva

9. Integrazione continua
10. Settimana di 40 ore
11. Cliente sul posto
12. Standard di codifica

Il planning game cioè la definizione delle funzionalità da implementare viste le priorità, le stime dei costi e altre valutazione tecniche è stato messo in atta anche grazie all'utilizzo di Trello. Non sempre però è stato possibile definire dei test in quanto certi pezzi di codice dipendevano totalmente da librerie esterne o definivano algoritmi procedurali il cui funzionamento era difficilmente testabile in maniera efficace.

Il testing è risultato un'importante risorsa nel corso dello sviluppo del progetto: oltre a dare maggiore sicurezza sulla correttezza del codice scritto in molte occasioni è stato utilizzato il TDD. Il TDD (Test Driven Development) è una tecnica utile sia ad aumentare la copertura dei test sia a scrivere il codice nella maniera più semplice ed efficace. Lo sviluppo guidato dai test si divide in tre fasi fondamentali:

- scrittura del test, il test viene scritto in base alla funzionalità che si vuole implementare e deve fallire in quanto tale funzionalità non esiste ancora;
- definizione della funzionalità, viene implementato il codice e si controlla tramite il test la sua correttezza;
- refactoring, viene modificato il codice in modo da renderlo più efficiente, leggibile e riusabile.

Oltre ad essere presente nell'approccio TDD il refactoring è stato utilizzato largamente durante lo sviluppo del progetto soprattutto per rendere il codice, non sempre di facile comprensione, il più leggibile ed elegante possibile.

La programmazione a coppie suggerita da Extreme Programming è stata applicata per gran parte del processo di sviluppo nonostante il team fosse composto da 3 elementi. Per ovviare a questo 'problema' la composizione della coppia che portava avanti l'implementazione è stata decisa a rotazione: due persone scrivevano codice e una studiava nuove funzionalità o migliorava la propria conoscenza delle tecnologie utilizzate. Soprattutto nella prima parte di sviluppo questo approccio è risultato molto produttivo in quanto gli tutte le persone del progetto erano alle prime esperienze per quanto riguarda l'uso del linguaggio di programmazione scelto (Python), del database (MongoDB) ma anche per il lavoro in team. Nella seconda parte del progetto invece l'elemento che restava da solo avendo ormai acquisito sufficiente padronanza delle tecnologie scelte poteva sviluppare autonomamente nuove funzionalità.

Lo standard di codifica è stato scelto sempre in funzione della leggibilità e dell'eleganza del codice; il pair programmming inoltre ha aiutato enormemente nel rispettarlo. Per quanto riguarda la documentazione invece si è cercato di aggiungere dei commenti all'inizio di ogni metodo complesso che specificassero la funzione, i parametri in ingresso e i risultati ritornati. Altri commenti invece sono stati inseriti nella parti di codice considerate poco leggibili in modo da aiutare anche un membro esterno al team a capirlo.

1.4 Estrazione dei dati dai file DXF

La prima parte del progetto Spazi-Unimi ad essere stata sviluppata è quella riguardante la lettura e l'elaborazione delle informazioni estraibili dalle piantine delle sedi universitarie. Come già detto il formato DWG non risultando idoneo ad essere analizzato è stato convertito in un formato DXF grazie all'applicazione free: Teigha® File Converter. Successivamente si è passati all'esplorazione delle librerie Python in grado di leggere tale formato: la scelta è ricaduta sulla libreria dxfgrabber. Grazie alla funzione 'readfile' di tale libreria si può leggere il contenuto di un file DXF (in qualsiasi versione) e salvarlo in una variabile.

Listing 1.4: Lettura di un file DXF con dxfgrabber

```
>>> dxf = dxfgrabber.readfile("File.dxf")
>>> dxf
<dxfgrabber.drawing.Drawing object at 0x7f72d5eef898>
```

Tale oggetto rispecchia la struttura dei file DXF quindi a noi è bastato leggere la sezione ENTITIES la quale contiene gli oggetti disegnati in tale file.

Listing 1.5: Salvataggio delle entità di un file DXF

```
>>> entities = dxf.entities
>>> entities
<dxfgrabber.entitysection.EntitySection object at 0x7f72d608b550>
>>> len(entities)
7850
```

Come si può notare dal codice preso come esempio ogni piantina contiene migliaia di entità e non tutte sono utili agli scopi del progetto. La scrematura di tali entità è stata fatta grazie ad una caratteristica comune dei disegni CAD: la suddivisione di tali entità in gruppi detti layer. Ogni layer dei file DXF forniti rappresenta un tipo di oggetto ben distinto; dopo un'attenta analisi di tali layer sono stati isolati i più utili:

- 'RM$', in esso sono contenuti i contorni delle stanze salvati come Polyline (o nella variante LWPolyline);

Listing 1.6: Analisi del layer 'RM$'

```
>>> rm = [e for e  in entities if e.layer == "RM$"]
>>> rm[0]
```

```
<dxfgrabber.entities.Polyline object at 0x7f72d346f9e8>
```

- 'NLOCALI', contiene le etichette con il codice identificativo di ogni stanza memorizzate come Text (o MText) e linee (per il contorno dei testi);

Listing 1.7: Analisi del layer 'NLOCALI'

```
>>> nloc = [e for e  in entities if e.layer == "NLOCALI"
            and e.dxftype in ["TEXT", "MTEXT"]]
>>> nloc[0]
<dxfgrabber.entities.Text object at 0x7f72d4afa198>
```

- 'RM$TXT', contiene le etichette con il codice identificativo di ogni stanza, la categoria di appartenenza di tale stanza e altre info utili alla Divisione Manutenzione edilizia e impiantistica come la metratura, in questo caso le etichette sono solo di tipo Text (o MText).

Listing 1.8: Analisi del layer 'RM$TXT'

```
>>> txt = [e for e  in entities if e.layer == "RM$TXT"]
>>> txt[0]
<dxfgrabber.entities.Text object at 0x7f72d345b400>
```

Una volta capito come analizzare e salvare le entità di nostro interesse è stato necessario creare delle classi (nel package 'model') che rappresentassero queste entità e che fornissero dei metodi utili per trasformarle. La prima classe definita è stata la base per le altre e trattandosi di rappresentare oggetti disegnabili non poteva che essere Point. Un oggetto Point è creabile passando due numeri (interi o float) che rappresentano le coordinate sul piano cartesiano di tale punto. In tale classe sono stati poi implementati dei metodi di trasformazione utili a dei punti come: traslazione e applicazione di un fattore di scala.

Il passo successivo è consistito nel definire altre tipologie di oggetti che rispecchiassero i dati utili letti dalle entità dei DXF; le classi create sono state:

- Polygon, definibile come una collezione di punti ordinata che collegati formano un poligono;
- Text, un testo con un punto che indica la posizione in cui deve essere posto.

Analizzando la struttura di tali classi ci si è resi conto che si sarebbero potute rappresentare come oggetti con un punto di ancoraggio: per il testo il punto di posizione mentre per il poligono l'origine. Scegliendo però un punto che non fosse l'origine e relativizzando i Point che definiscono un Polygon si possono effettuare certe trasformazioni in maniera molto più semplice. Si è quindi definito un anchor_point per ogni poligono calcolabile come il punto più in basso e più a sinistra del più piccolo rettangolo contenete il poligono (bounding box). In questo modo la traslazione di un Polygon si applica solo traslando il suo anchor_point senza andare a modificare tutti i punti che lo compongono. L'evoluzione naturale è stata la definizione di una classe Anchorable dalla quale Polygon e Text ereditano: in questo modo tutti i metodi che implementano le trasformazioni sugli anchor_point di un oggetto sono definiti nello stesso luogo.

Definite le classi per gli oggetti base si è passati all'implementazione di classi che contenessero tali oggeti: la prima e basilare è Room. Un oggetto di tipo Room ha il compito di astrarre ciò che definisce una stanza quindi sarà composto da:

- un oggetto di tipo Polygon che rappresenti il contorno di tale stanza;
- uno o più oggetti di tipo Text che indichino il contenuto delle etichette associate a tale stanza;
- un Point che rappresenta il punto di ancoraggio di tale Room in modo da relativizzare tutti gli altri punti della stanza.

Giunti a questo punto però ci si è resi conto che Polygon e Room rispetto agli altri tipi di oggetti definiti avevano una particolarità: essendo bidimensionali erano gli unici a poter essere disegnati. Il passo successivo è stato definire la classe Drawable da cui i poligoni e le stanze potessero ereditare metodi comuni come le trasformazioni e il calcolo del bounding_box. I metodi di trasformazione sugli oggetti Drawable non vengono effettuati direttamente: lasciando la responsabilità alle singole classi viene richiamata la trasformazione corrispondente su ogni entità appartenente agli oggetti da modificare.

L'ultima classe del package 'model' definita prima dell'effettiva lettura dei file DXF è stata Floor. Un'istanza di tale classe rappresenta un piano di un edificio e come tale oltre ad essere una collezione di Room deve possedere anche informazioni sull'edificio a cui appartiene e sul piano che rappresenta.

Tutte le classi definite in 'model' possiedono due metodi particolari:

- to_serializable(), ritorna un oggetto di tipo json (un dizionario) che rappresenta l'oggetto su cui è stato richiamato il metodo;
- from_serializeble(json), riceve un oggetto di tipo json e da esso crea un'istanza della classe usando le informazioni contenute.

Il passo successivo è stato definire una classe dxf_reader in grado di leggere i dati da un file DXF e creare un Floor contenente tutti i dati utili. Dopo aver letto il file con dxfgrabber come visto in precedenza viene richiamata una funzione (_extract_entities()) che scrorre la lista di entità lette e controlla se appartengono alla tipologia e ai layer di nostro interesse. I testi trovati vengono salvati in un'apposita lista mentre per quanto riguarda le polyline delle stanze vengono sottoposte ad alcune operazioni prima di venire immagazzinate:

- viene creato un oggetto di tipo Polygon con i punti della polyline trovata;
- viene controllato che il poligono sia chiuso, se non lo è viene scartato;
- vengono uniti i punti troppo vicini in un unico punto così da semplificare il disegno finale;
- viene controllato tramite un apposito algoritmo che il poligono non sia di forme problematiche (si controlla che due segmenti non adiacenti non si intersechino) in caso contrario viene scartato;
- viene istanziato un oggetto di tipo Room semplicemente passando al costruttore il poligono controllato e semplificato.

L'oggetto di tipo Room risultante viene salvato nella lista di stanze letta e passato al costruttore del Floor. La lista di testi letti invece deve ancora essere elaborata: ogni testo deve venire associato alla stanza che contiene il suo punto di ancoraggio, se non viene associato a nessuna allora è scratato. Il metodo associate_room_texts(texts) della classe Floor scorre i testi e controlla a quale Room associare un testo utilizzando un algoritmo che indica se un punto è contenuto in un poligono. Tale algoritmo non è risultato banale quindi dopo un'esplorazione di procedure già esistenti si è deciso di utilizzare il 'Ray casting algorithm' [1]: l'idea di base è quella che se un punto è all'interno di un poligono una retta che parte da lui intersecherà un numero dispari di volte il perimetro del poligono. Nel caso in cui il punto interseca un numero pari di volte invece si può dire che non è contenuto nel poligono a meno di casi particolari come le intersezioni con i vertici.

Listing 1.9: Ray casting algorithm in pseudocodice

```
count = 0
# Per ogni lato del poligono
for side in polygon:
# Controlla se la retta partente da P interseca con il lato
```

[1] http://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm

```
    if ray_intersects_segment(P,side) then
        count = count + 1
# Se il numero di intersezioni e' pari il punto e' all'interno
if is_odd(count) then
    return inside
else
    return outside
```

A questo punto il nostro oggetto di tipo Floor contiene una lista di Room che a loro volta contengono i testi compresi nel loro Polygon. Vi è però una criticità dovuta a come sono state disegnate le piantine degli edifici: ogni disegno non si posiziona mai nello stesso punto degli altri rispetto all'origine e la scala con cui gli elementi sono stati disegnati non risulta essere sempre simile. Per questo motivo la classe Floor implementa un metodo normalize() da richiamare dopo aver aggiunto gli elementi al piano. Questo metodo di occupa di traslare tutti gli oggetti del piano in modo da avere il punto minimo (sia per l'asse x che per quella y) del bounding box generale nell'origine. Il piano viene quindi ridimensionato in modo che abbiano tutti dimensioni simili: per fare ciò viene calcolato un fattore di scala da applicare per fare in modo che le dimensioni massime siano 1024 * 1024.

Il nostro Floor si può considerare completo a meno delle informazioni sulla sua identificazione: il codice del palazzo a cui appartiene e il codice che identifica di che piano si tratti. Il codice del palazzo definito per semplicità come building_id (abbreviato b_id) viene estratto in maniera semplice dal nome del file DXF processato: applicando una espressione regolare si estrae la prima parte del nome che secondo convenzione è proprio il nostro b_id. Per quanto riguarda l'identificazione del piano (floor_id o f_id) invece il processo è più complicato in quanto la convenzione di farlo seguire al b_id nel nome del file non è stata sempre seguita e molti identificatori risultano errati. Per evitare una correzione manuale di tutti gli errori presenti nei file a disposizione la soluzione adottata ha riguardato l'implementazione di una classe apposita 'floor_inference' con lo scopo di implementare una procedura che cerchi di ricavare il corretto f_id. La procedura si occupa di ricavare dal DXF letto i dati del layer 'CARTIGLIO' nel quale sono contenute le informazioni riguardanti ciò che è stato disegnato nella cartina (nome del palazzo, disegnatore, data...) tra cui il nome del piano. Controllando quali testi contenessero la parola 'PIANO' o quali fossero vicini a tale parola è stata ricavata per ogni file una lista non troppo lunga di possibili nomi di piano. Il passo successivo è stata la creazione di un file floor_inference.json che contenesse un dizionario in cui:

- le chiavi sono gli identificatori da noi creati per i piani salvati come stringhe;

- nella chiave floor_name sono indicati i nomi dei piani uniformati;
- nella chiave suffix_regex sono riportate le espressioni regolari per i suffissi dei nomi dei file che corrispondono al piano;
- nella chiave name_regex sono presenti sempre espressioni regolari ma da applicare alla lista di stringhe ricavate dal cartiglio.

Listing 1.10: Esempio del dizionario in 'floor_inference.json'

```
"03" : {
      "floor_name": "Rialzato Piano Terra",
      "suffix_regex": ["^r[ap\\.]?$"],
      "name_regexes": [ "^rialzato$" ]
   },
   "05" : {
      "floor_name": "Mezzanino Piano Terra",
      "suffix_regex": ["^a[pr]$", "^arp?$"],
      "name_regexes": [ "^ammezzato$", "^ammezzato rialzato$", "^ammezz
   },
   "10" : {
      "floor_name": "Primo Piano",
      "suffix_regex": ["^p?1[ap]?\\.?$"],
      "name_regexes": ["^primo$", "^(piano\\s+)primo", "^primo(\\s+pian
   },
```

Applicando le espressioni regolari corrispondenti ai suffissi nel nome e alle stringhe nel cartiglio la procedura ricava nella maggior parte dei casi due floor_id anche se in alcuni casi (doppio cartiglio) può ricavarne di più. I controlli effettuati sui floor_id sono i seguenti:

- se solo un f_id è stato trovato utilizza quello;
- se sono stati trovati 2 f_id corrispondenti utilizza uno dei due indifferentemente;
- se i due f_id trovati sono diversi stampa un messaggio per avvisare l'utente ma utilizza quello ricavato dal cartiglio dato che è ritenuto più affidabile;

- se sono presenti più cartigli con f_id diversi utilizza quello uguale al f_id ricavato dal suffisso del nome stampando un messaggio;
- se i f_id ricavati dai cartigli non corrispondono con quello ricavato dal nome (o quest'ultimo non è presente) stampa un messaggio di errore e scarta il piano;
- se non sono stati trovati f_id in nessuna fonte stampa un errore e scarta il piano.

In questo si riesce a ricavare sia il building_id che il floor_id dal file DXF: questi dati vengono perciò passati al costruttore del Floor insieme alla lista di Room andando a creare così un oggetto contenete tutti i dati letti dalla piantina.

1.5 Estrazione dei dati dai file CSV

I dati ricevuti sulle aule come scritto in precedenza erano in formato XLS che per comodità si è deciso di convertire in CSV: una tipologia di file di testo rappresentante una tabella. In questo formato nella prima linea sono indicati i nomi delle colonne mentre in quelle successive sono inseriti i dati. Le varie colonne sono separate da un carattere specifico che viene deciso in fase di codifica: nel nostro caso si trattava di una virgola (','').

Per la lettura di un file CSV in Python è bastato importare la libreria 'csv' e dopo aver aperto il file lo si può leggere partendo dal primo byte (seek(0))con la funzione 'reader': il risultato deve essere trasformato in un iteratore che fatto scorrere conterrà il file letto riga per riga.

Listing 1.11: Lettura di un file CSV

```
>>> csvfile = open("file.csv")
>>> csvfile
<_io.TextIOWrapper name='file.csv' mode='r' encoding='UTF-8'>
>>> csvfile.seek(0)
0
>>> reader = csv.reader(csvfile, "excel")
>>> reader
<_csv.reader object at 0x7fc86bd0bcf8>
>>> it = iter(reader)
>>> it
<_csv.reader object at 0x7fc86bd0bcf8>
>>> next(it)
['header_0', 'header_1', 'header_2', 'header_3', 'header_4']
>>> next(it)
['line_1-0', 'line_1-1', 'line_1-2', 'line_1-3', 'line_1-4']
```

In questo modo basta leggere il contenuto fino alla fine dell'iteratore per ricavare tutti i dati del file. Per rendere le informazioni lette meglio manipolabili in un futuro sono state memorizzate in una lista di dizionari in cui le chiavi corrispondono ai nomi delle colonne lette (header).

Listing 1.12: Memorizzazione dei dati in un dizionario

```
>>> header = [ s.strip() for s in next(it) ]
>>> content = []
>>> for line in it:
        content.append(
            {
                c: l.strip() for c, l in zip(header, line)
            }
        )
[ {
    'header_0' : 'line_1-0',
    'header_1' : 'line_1-1',
    'header_2' : 'line_1-2',
    'header_3' : 'line_1-3',
    'header_4' : 'line_1-4'
} , {
    'header_0' : 'line_2-0',
    'header_1' : 'line_2-1',
    'header_2' : 'line_2-2',
    'header_3' : 'line_2-3',
    'header_4' : 'line_2-4'
} ]
```

Avendo a disposizione solo 5 tipologie di file CSV da cui leggere i dati si è deciso di non dover indicare ogni volta quale tipo di file si sta processando: la procedura di lettura infatti inferisce la tipologia analizzando l'header letto. Salvando nel file di configurazione general.json le intestazioni delle 5 tipologie di CSV si è poi andati a controllare quale di esse coincideva (o conteneva) con quella appena letta.

Listing 1.13: Tipologie di header memorizzate in 'general.json'

```
"csv_headers" : {
    "edilizia":{
        "buildings"       : ["l_b_id", "b_id", "address", "lat", "lon"]
```

```
        "rooms"           : ["r_id", "b_id", "room_name", "capacity",
                             "l_floor", "cat_name"],
        "room_categories" : ["cat_id", "cat_name"]
    },
    "easyroom":{
        "buildings"       : ["b_id", "address", "building_name",
                             "n_floors"],
        "rooms"           : ["r_id", "b_id", "room_name", "capacity",
                             "l_floor", "accessibility", "equipments" ]
    }
}
```

Dopo aver inferito la tipologia del file l'ultima cosa da fare è applicare un filtro sui dati letti: non tutti infatti sono risultati utili al progetto e perciò sono state selezionate solo le colonne ritenute importanti. La scrematura si basa ancora una volta sui dizionari di header in general.json: è bastato applicare al dizionario ottenuto dalla lettura del CSV una funzione che filtra le chiavi.

Listing 1.14: Definizione della funzione filter_keys

```
def filter_keys(d, valid_keys):
   return { k: d[k] for k in d if k in valid_keys }
```

Capitolo 2

Integrazione e validazione dei dati edilizi

2.1 Tecniche di definizione del DBMS del progetto Spazi-Unimi

La fase successiva del progetto ha riguardato la definizione di un DBMS (Database Management System) cioè l'insieme dei programmi software che hanno lo scopo di gestire l'organizzazione, la memorizzazione e il rendere disponibili i dati per quanto riguarda un database.

L'uso dei database e il loro sviluppo sono andati di pari passo con il progresso in ambito informatico: già negli anni 60 infatti cominciarono ad essere sviluppate e introdotte le prime tipologie di basi di dati cioè i database navigazionali. Questa tipologia si basava sulla navigazione dei dati in maniera manuale: ogni dato era connesso con un altro e per reperirlo bastava inserire il giusto percorso; i database navigazionali erano perciò divisibili in due categorie: quelli a rete e quelli gerarchici. Alla fine degli anni 60 cominciarono ad affermarsi i primi hard disk e ciò portò i database navigazionali, fin lì memorizzati su nastri magnetici, ad essere considerati inefficienti. Edgar F. Codd nel 1970 pubblicò A Relational Model of Data for Large Shared Data Banks un articolo in cui proponeva una nuova tipologia di database basata su tabelle il cui contenuto si poteva manipolare con un linguaggio dedicato. Lo sviluppo di queste idee portò negli anni a quello che tutt'oggi è il modello di database più utilizzato cioè il relazionale mentre il linguaggio che è stato sviluppato è l'SQL (Structured Query Language). Negli anni 80 si svilupparono i primi database

ad oggetti che memorizzano i dati in forma di oggetti proprio come nell'OOP (Object Oriented Programming). Questo continuo svilupparsi e crescere dei database è dovuto alla sempre crescente mole di dati che si vuole memorizzare cercando però di mantenere le prestazioni ottimali.

Il Data Menagement è la disciplina che si occupa di tutte le operazioni utili al funzionamento di un database: dall'inserimento dei dati fino alla loro rimozione passando per tutte le possibili trasformazioni intermedie. Nel caso del nostro progetto la fase più complessa è consistita proprio nell'inserimento dei dati nel DB cercando di uniformarli ed unirli tra loro. Le tecniche di Data Management che sono state utilizzate quindi sono state principalmente: il Data Filtering, il Data Validation e il Data Integration.

Il Data Filtering è una tecnica che consiste nel filtrare considerati i dati non utili e perciò da non memorizzare nel database. Per fare ciò bisogna innanzitutto analizzare i dati e capire quali scartare perché considerati inutili dal sistema e quali perché forniscono informazioni talmente errate da poter essere considerate dannose dalla percezione dell'utente finale.

Il Data Validation è il processo che si occupa di rendere i dati disponibili corretti e utilizzabili dalle applicazioni dipendenti da essi: per fare ciò si deve definire un livello minimo di qualità del dato da rispettare. Tale qualità non è sempre semplice da definire e nel nostro specifico caso può essere descritta tramite tre proprietà:

- l'accuratezza, cioè la una misura di quanto un dato si avvicina al valore considerato corretto;
- la completezza, indica la quantità dei dati disponibili rispetto al totale, può essere rappresentata perciò come una percentuale;
- la consistenza, cioè la proprietà di un dato di non entrare in conflitto con un altro, nel progetto si è avuta soprattutto per dati provenienti da fonti diverse.

Il Data Integration è il processo di Data Management che si occupa di manipolare i dati, provenienti anche da più fonti, così da uniformarli e prepararli alla memorizzazione nel database.

La teorizzazione dell'integrazione dei dati è iniziata negli anni 80 e prevedeva l'interoperabilità tra diversi database[1]. Bisogna aspettare però fino al 1991 per avere il primo sistema di Data Integration dipendente dai metadati: l'università del Minnesota infatti definì tale sistema per l'IPUMS (Integrated Public Use Microdata Series) cioè il database contenete i dati di ogni censimento degli Stati Uniti dal 1850 al 2000. Il Data Integration System implementato per l'IMPUMS si basa su un approccio ETL (Extract, Transform, Load) e fa confluire i dati finali in un database basato sul Data Warehouse.

[1]John Miles Smith (1982): Multibase: integrating heterogeneous distributed database systems

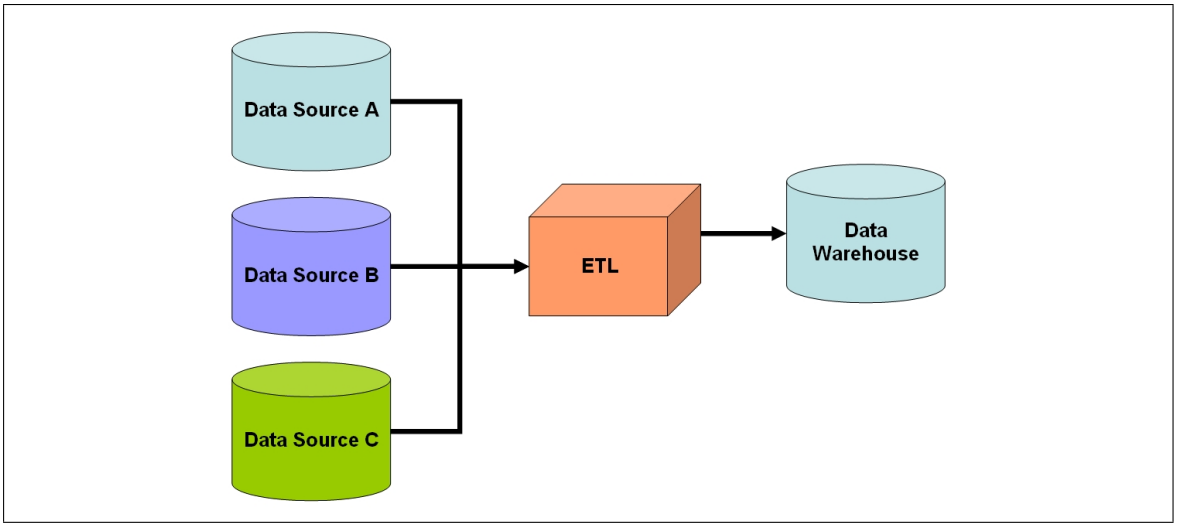

Figura 1: Sistema di Data Integration basato sull'approccio ETL con memorizzazione in Data Warehouse

L'approccio ETL è composto, come dice il nome, da tre fasi distinte:

- estrazione, consiste nel ricavare dalle varie fonti (file, database, etc..) i dati utili al processo, nel nostro caso si tratta delle informazioni estratte in precedenza dai file CSV e DXF;
- trasformazione, cioè l'applicazione di tutte le trasformazioni utili ai dati affinché possano essere utilizzati dall'utente finale, per il processo da noi sviluppato si tratta di applicare le procedure descritte in precedenza (Data Filtering e Data Validation) e di definire una nuova procedura per l'unificazione dei dati;
- caricamento, si intende la memorizzazione dei dati provenienti dalla fase di trasformazione in un database definibile per la maggior parte dei casi come un Data Warehouse.

Con il termine Data Warehouse si vuole definire un archivio di dati informatico (database) il quale memorizza le informazioni inerenti a un'azienda o ad un'organizzazione. L'approccio del Data Integration basato su ETL e Data Warehouse pur essendo molto intuitivo e di vecchia concezione risulta tutt'oggi adatto ad essere implementato in sistemi che non richiedono un aggiornamento frequente dei dati.

Negli ultimi anni la tendenza per quanto riguarda i meccanismi di Data Integration è cambiata: grazie alle nuove tecnologie gli accessi a database, anche remoti, risultano molto più veloci e le applicazioni moderne si basano per lo più su dati in continua evoluzione e sempre aggiornati. Ciò viene definito architettura con schema globale

ed ha portato ad un minor impiego della logica ETL e ad un maggior utilizzo di interrogazioni (query) combinate di più database. In questo modo si viene e creare un database virtuale i cui dati sono gli stessi dei database reali: tali informazioni vengono reperite ed elaborate in tempo reale tramite wrapper (involucro).

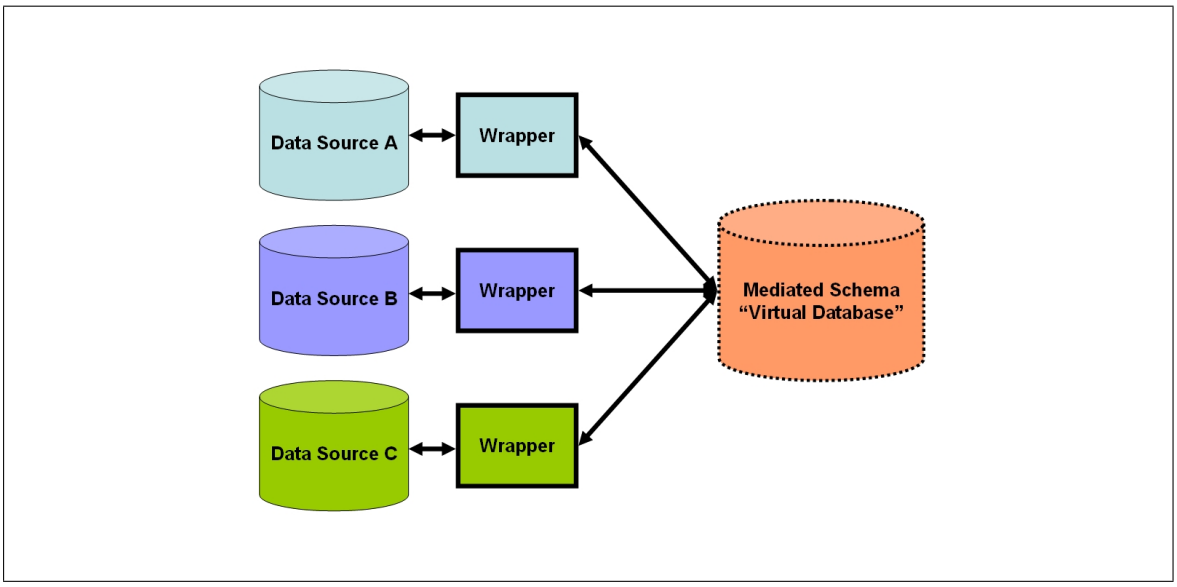

Figura 2: Architettura con schema globale: Data Integration basato su un approccio di Database virtuale e wrapper

Il ruolo del wrapper in questo tipo di sistemi è riconducibile all'Adapter Pattern: il suo scopo è quello di rendere possibile la comunicazione tra due componenti le cui interfacce non corrispondono.

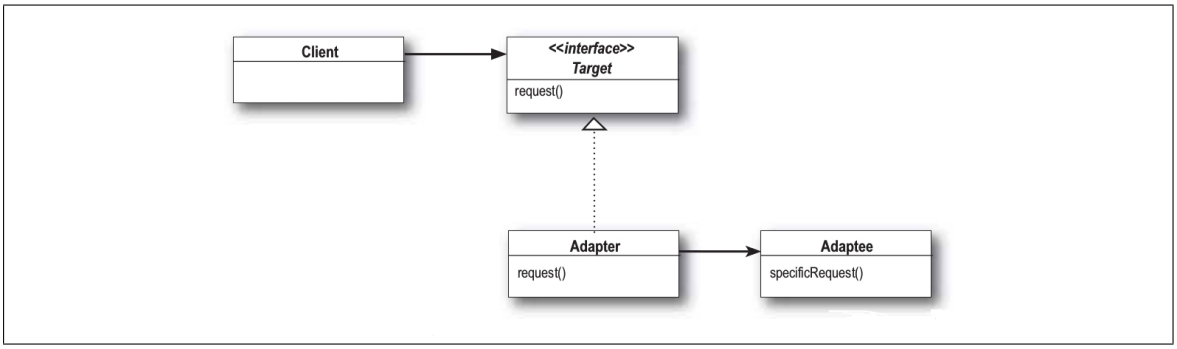

Figura 3: Class Diagram UML dell'Adapter Pattern

L'architettura con schema globale può essere definita in due modalità base:

- GAV (Global As View), in cui lo schema globale vien espresso in base agli schemi delle sorgenti;
- LAV (Local As View), dove lo schema globale è indipendente dalle sorgenti e la relazione con ognuna di esse è definita tramite delle view.

L'approccio GAV è di più immediata realizzazione ma in caso di cambiamenti nelle fonti, come l'aggiunta di una nuova sorgente, risultano complesse le modifiche da apportare. Viceversa il LAV favorisce le possibili estensioni rispetto a nuove sorgenti ma risulta più complicato definire le query utili alla sua definizione.

Per GLAV (Global and Local As View) si intende un ibrido tra le due strategie appena descritte: in questo caso la relazione tra le sorgenti e lo schema globale è stabilita attraverso delle view, alcune definite sullo schema globale e alcune sulle sorgenti.

Dato che il progetto Spazi-Unimi prevede di operare su dati edilizi i quali risultano statici a meno di rare variazioni (ad esempio un cambio di sede per un dipartimento) si è deciso di implementare un sistema ETL che definisce un database di tipo Data Warehouse.

2.2 Analisi dei dati (da DXF e CSV)

2.3 Salvataggio dei dati in MongoDB

Una volta definite le classi con lo scopo di leggere i dati dalle due tipologie di file a disposizione si è deciso di implementare un programma principale ('main') dal quale richiamare con gli appositi comandi il processamento dei file. I comandi implementati sono stati:

- dxf file.dxf, al comando dxf si passa uno o più file DXF da processare;
- csv file.csv, il comando csv come l'altro può ricevere uno o più file di tipologia corretta.

Tali comandi dopo avere controllato l'estensione e la presenza dei file indicati istanziano un apposito 'task': il compito di tali classi (DXFTask e CSVTask) è quello di richiamare tutte le operazioni da effettuare sui file. La prima operazione da effettuare è la lettura e per fare ciò viene richiamata la classe corrispondente (DXFReader o CSVReader). Una volta letti i dati dai file non resta che salvarli nel database; per fare ciò bisogna prima definire una struttura per i file BSON che andremo ad inserire i MongoDB. Si è deciso di salvare le informazioni in una collection Building in cui usare il building_id come identificatore per i documenti.

2.4 Strategie di merging dei dati adottate

2.5 Definizione di un sistema non dipendente dall'ordine di esecuzione

2.6 Possibili miglioramenti

2.7 Reporting degli errori e delle criticità

2.8 Indirizzo ben formato: teoria e possibile utilizzo

2.9 Definizione di un DBAnalysis per avere statistiche specifiche sul Merge

Capitolo 3

Considerazioni finali e risultati

3.1 Definizione di API

3.2 Statistiche sui tempi di calcolo e di risposta del DB

3.3 Considerazioni sullo sviluppo del progetto

3.4 Miglioramenti/crescita dal lato personale

Bibliografia

[1] M. Gotti, I linguaggi specialistici, Firenze, La Nuova Italia, 1991.

[2] R. Wellek, A. Warren, Theory of Literature , 3rd edition, New York, Harcourt, 1962.

[3] A. Canziani et al., Come comunica il teatro: dal testo alla scena. Milano, Il Formichiere, 1978.

[4] Ministry of Defence, Great Britain, Author and Subject Catalogues of the Naval Library, London, Ministry of Defence, HMSO, 1967.

[5] H. Heine, Pensieri e ghiribizzi. A cura di A. Meozzi. Lanciano, Carabba, 1923.

[6] L. Basso, "Capitalismo monopolistico e strategia operaia", Problemi del socialismo, vol. 8, n. 5, pp. 585-612, 1962.

[7] L. Avirovic, J. Dodds (a cura di), Atti del Convegno internazionale Umberto Eco, Claudio Magris. Autori e traduttori a confronto (Trieste, 27-28 novembre 1989), Udine, Campanotto, 1993.

[8] E.L. Gans, The Discovery of Illusion: Flaubert's Early Works, 1835-1837, unpublished Ph.D. Dissertation, Johns Hopkins University, 1967.

[9] R. Harrison, Bibliography of planned languages (excluding Esperanto). `http://www.vor.nu/langlab/bibliog.html`, 1992, agg. 1997.