Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

PROGETTAZIONE DI UN'APPLICAZIONE E CONSEGUENZE DI UN APPROCCIO "CONTINUOS REFACTORING"

Relatore: Prof. Carlo Bellettini

Tesi di:
Samuel Gomes Brandão
Matricola: 803939

Anno Accademico 2013-2014

Dedica

Ai miei genitori Mônica e Zilmar
Aos meus pais Mônica e Zilmar

Prefazione

Ciao! Ci vuole scrivere una prefazione!

Organizzazione della tesi

La tesi è organizzata come segue:

- nel Capitolo 1 ....

Ringraziamenti

Vorrei ringraziare Paolo Venturi, Diego Costantino, e il prof. Dr. Carlo Bellettini.

Indice

Capitolo 1

Introduzione

Spazi Unimi è il nome dato a un progetto per l'ottenimento di dati sugli spazi dell'Università degli Studi di Milano, sviluppato durante il Tirocinio Interno per la laurea triennale in Informatica all'UNIMI, da Samuel Gomes Brandão, Diego Costantino e Palo Venturi. L'idea nasce a partire dalle proposte del progetto Campus Sostenibile, promosso dall'UNIMI e dal Politecnico di Milano, e si sviluppa posteriormente in autonomia, sotto l'orientamento del Prof. Dr. Carlo Bellettini.

Il progetto parte con lo sviluppo di un'applicazione software con lo scopo principale di estrarre, validare e correggere dati ottenuti da diverse sorgenti, procedendo in seguito alla loro integrazione. I dati vengono mantenuti su un database da venir utilizzato per futuri progetti attraverso l'uso di una specifica Application Programming Interface (API) con architettura REST (Representational State Transfer).

Le informazioni integrate riguardano la topologia e la destinazione d'uso degli edifici universitari e i loro spazi, con particolare importanza alla elaborazione e presentazione delle piante interne e localizzazione di stanze precise. In questo modo, siamo in grado di fornire accuratamente informazioni sulla localizzazione di palazzi, aule, o stanze a secondo della loro tipologia d'uso (bagni, biblioteca, sale studio, ecc).

In definitiva, tre sono stati i principali compiti del progetto:

- Estrarre la topologia interna degli edifici universitari a partire dalle piantine edili.

- Integrare le diverse informazioni testuali (fogli elettronici, file CSV[1]) fornite dall'università e validarle.
- Associare le informazioni testuali alla topologia dei palazzi, identificando la localizzazione di stanze rilevanti e la categoria funzionale delle altre stanze.

Su questa relazione descrivo il processo di sviluppo della suddetta applicazione da zero, con particolare rilievo alle sfide per la buona progettazione e all'utilizzo di un concetto che ho chiamato "continuos refactoring" [2]. Per quanto riguarda i compiti sopra, per motivi di brevità mi concentrerò di più sugli aspetti di estrazione dati dalle piantine edili.

[1]Comma separated values - formato testuale in cui le informazioni vengono inserite utilizzando le virgole come separatori

[2]Si veda il capitolo 3

Capitolo 2

Attività Preliminari

> Experience is simply the name we give our mistakes
>
> Oscar Wilde

La prima sfida per lo sviluppo di un progetto è quella di capirlo, e può richiedere tempo e dedicazione considerevole. Molto spesso però il capire avviene - e lo può soltanto - durante lo sviluppo stesso.

Il primo passo è stato quello di pensare ai casi d'uso che volevamo coprire con la nostra applicazione. Da questo punto abbiamo proceduto verso la comprensione dei dati e delle informazioni disponibili e la scelta delle tecnologie più adeguate per la loro elaborazione. Le sorgenti dati che dovevamo integrare erano due principali, a loro volta suddivise in più tipologie di informazione e formati di provenienza.

Dal dipartimento di Edilizia dell'UNIMI abbiamo ottenuto le piante architettoniche dei palazzi utilizzati dall'Università, in formato DWG - un formato file proprietario, e una serie di fogli elettronici con informazioni dettagliate sui palazzi e sulle loro stanze, in speciale quelle utilizzate per scopi didattici.

Abbiamo convertito questi file binari in formato DXF (formato testuale "CAD-compatible" ma comunque incomprensibile all'essere umano) utilizzando uno strumento gratuito, dato che era il formato richiesto dalla libreria dxfgrabber [1] che abbiamo utilizzato. La dimensione dei file arrivava facilmente all'ordine dei megabyte,

[1] Libreria *opensource*, sorgente disponibile in https://bitbucket.org/mozman/dxfgrabber

con decine di migliaia di righe su ognuno, il che è ragionevole, se consideriamo che ogni file contiene, oltre al disegno delle stanze, una marea di informazioni imprescindibili al lavoro edile (misure, tubature, passaggio di elettricità, porte e finestre, scale, ascensori, sezioni, ecc).

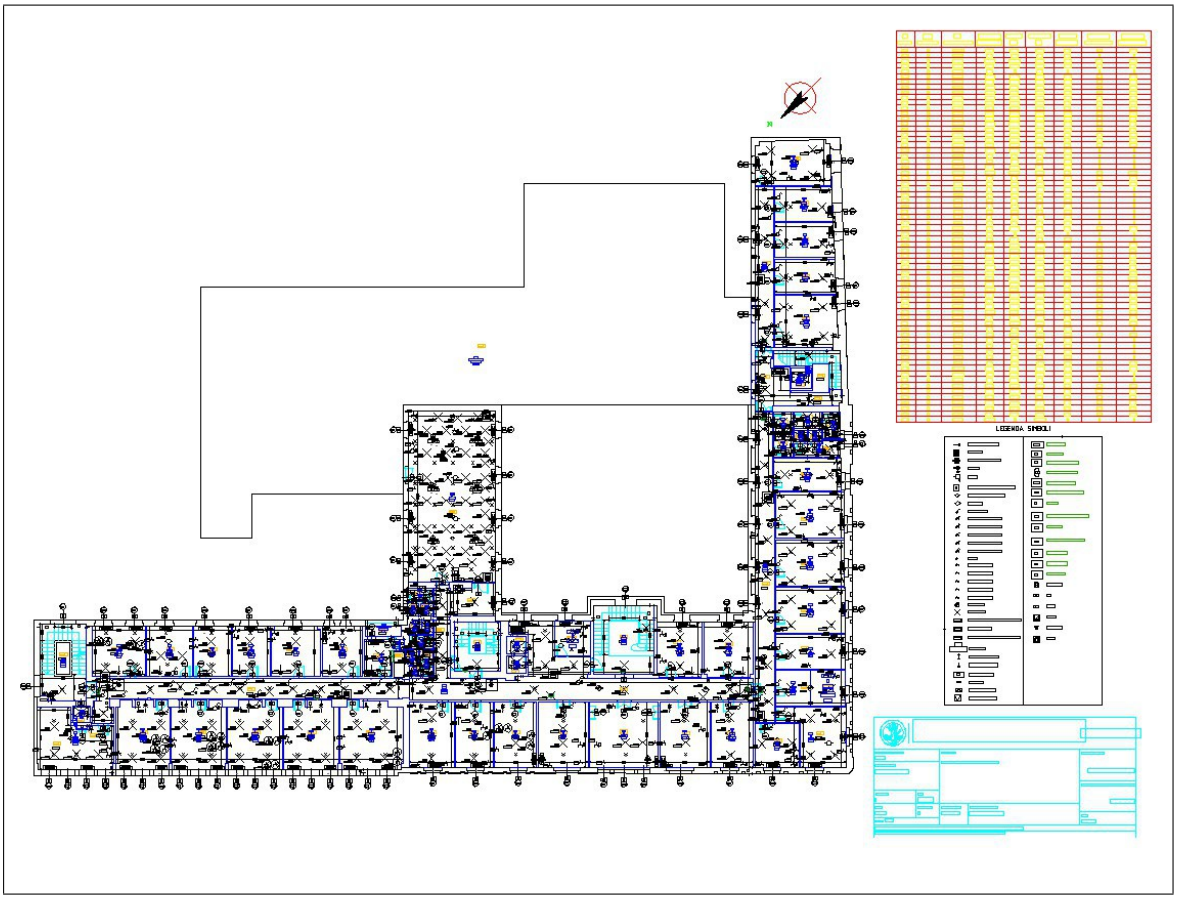

Figura 1: Esempio del contenuto di un file DWG di media complessità (Dipartimento di Informatica, 1° piano). La quantità di informazioni presente rende difficile la comprensione del contenuto informativo del file manualmente.

Con l'aiuto anche della divisione di Sistemi Informativi, rappresentata in speciale da Vincenzo Pupillo, abbiamo ottenuto in formato testuale CSV (comma separated values) le informazioni utilizzate dal sistema Easyrooms, che mira a fornire informazioni rispetto all'uso didattico degli spazi (eventi, lezioni, capienza delle aule, lauree, eccetera).

Entrambi i dipartimenti ci hanno aiutato con totale disponibilità e trasparenza, e senza il loro aiuto il nostro progetto non sarebbe mai stato portato a buon fine.

Con queste informazioni in mano ne abbiamo incominciato l'analisi, cercando di capire non solo le loro criticità, ma anche dove si sovrapponevano, completavano o fossero ridondanti. Al primo contatto ci sembravano perfette: avevamo informazioni di destinazione delle stanze, la loro capienza, accessibilità a disabili, e le potevamo localizzare sulle piante architettoniche utilizzando il loro codice identificativo. Dalle piante ottenevamo anche la localizzazione di aree di interesse come biblioteche, sale studio, bagni, spazi di ristorazione, eccetera.

2.1 La consistenza dei dati

Sotto uno sguardo più attento, però, l'immagine mentale che ci eravamo costruiti di quei dati incominciò a rilevare i suoi difetti: le fessure venivano come errori di battitura, l'utilizzo duplice di codici identificativi per i palazzi, piante architettoniche fuori scala o semplicemente disegno della stessa stanza più di una volta sullo stesso file. Spesso i disegni venivano ripetuti sulla stessa posizione, probabilmente frutto di un'operazione di copia e incolla interrotta a metà. All'occhio umano saltavano facilmente gli errori, e riuscivamo a correggerli riconoscendo dei pattern di riferimento, ma per un elaboratore no sarebbe stato così facile. Soltanto a un programmatore interessato ad estrarre informazioni in modo automatico con l'uso di un calcolatore questi errori avrebbero causato danni, e allora ci toccava gestirli.

Con buona probabilità questi errori sono stati accumulati lungo gli anni, ed è totalmente comprensibile che ci siano, se consideriamo ad esempio che la maggior parte delle piante architettoniche di cui disponevano sono state disegnate in formato cartaceo e solo posteriormente trasferite in formato digitale, quando la costruzione dell'edificio era già finita. Sono inoltre state fatte e raccolte lungo periodi significativi, create da persone diverse, e perciò è difficile mantenere degli *standard* nella loro rappresentazione digitale. Ovviamente nel processo di trasferimento da cartaceo a digitale dettagli vengono persi e errori vengono introdotti, e le versioni digitali non passano per la stessa meticolosa verifica di quelle utilizzate per la costruzione degli edifici.

Nessuna assunzione

La molle di dati era significativa, in speciale per i file DWG: più di 700, ognuno con dimensione media di 4.5Mb, e casi estremi di fino a 44Mb. Questi file contenevano tutte le informazioni edili: tubature, finestre, porte, scale, sezioni dei palazzi, disegno dei muri, cortili, terrazze, eccetera. Ci è stato poco tempo di analisi di quei dati per capire che l'unico modo di giudicarne la loro qualità sarebbe stato incominciando con la loro estrazione e imparando durante il processo. Queste caratteristiche hanno determinato tanti aspetti dello sviluppo, addirittura il nostro *workflow*, che frequentemente si è dimostrato "REPL Driven" o "Experimentation Driven"[2].

Da queste analisi abbiamo concluso che potevamo fare poche o nessuna assunzione iniziale sulla qualità, formato, presenza o consistenza dei dati.

2.2 Le scelte tecnologiche a partire dai dati

Questa natura dei dati trasformava tante delle nostre richieste in *"Wicked Problems"*, problemi la cui comprensione potrebbe avvenire soltanto durante la loro risoluzione stessa e non era possibile *a priori*. Questo è stato uno dei primi motivi che ci ha fatto scegliere Python 3 come linguaggio di riferimento per l'estrazione e l'elaborazione: la capacità di scrivere velocemente degli script in grado di estrarre e produrre dettagliate analisi delle caratteristiche dei dati.

Oltre a questo aspetto, anche i seguenti punti hanno contribuito alla scelta di Python:

- La presenza delle *list comprehentions*, un potente strumento per l'esecuzione di operazioni di trasformazione e filtraggio dei dati, specialmente se associate a funzioni per l'aggregazione di dati e agli iteratori di Python 3.
- L'esistenza di una forte comunità di sviluppatori e entusiasti per il linguaggio, specialmente in Italia.
- Ampia presenza di librerie di supporto per le attività che dovevamo eseguire (lettura delle piante, elaborazioni geometriche, ecc).

[2]Più dettagli nel capitolo 3

- Essendo Python un nuovo linguaggio per tutti i partecipanti al progetto, rappresentava una positiva sfida didattica.
- La scelta di Python non sarebbe limitante per la continuazione del progetto in futuro da parte di altri studenti/tesisti, in quanto è anche insegnato all'università.

Per quanto riguarda la scelta del DBMS (Database Management System) è stata l'inconsistenza dei dati a guidare la scelta: le informazioni presenti sui diversi palazzi non erano omogenee in termini quantitativi ne qualitativi. Su qualche edifici disponevamo di più informazioni topologiche mentre su altri quasi nessuna. Anche le informazioni inerenti all'edificio stesso (come il suo nome rappresentativo o scopo d'utilizzo - ad esempio "Dipartimento di Informatica") non sempre erano presenti o validi, e ciò si ripeteva anche sugli altri dati. L'uso di un DBMS relazionale avrebbe comportato degli *schema* con considerevoli campi *null* e denormalizzato. Per questi motivi abbiamo scelto un DBMS che seguisse un modello di memorizzazione *schemaless*.

Per le note caratteristiche prestazionali, supporto nativo a calcoli su coordinate geografiche, presenza di forte comunità, documentazione chiara e completa e la diversità di librerie aggiuntive disponibili, abbiamo scelto MongoDB come DBMS di riferimento.

Capitolo 3

Svolgimento delle Attività

3.1 *La scelta di un workflow: partire con il TDD*

Per incominciare lo sviluppo avevamo l'obbiettivo di utilizzare il *TDD - Test Driven Development*, un consacrato *workflow* per lo sviluppo software noto per portare a soluzioni di design più semplici, pulite, disaccoppiate e testabili, tutte proprietà desiderabili da un software. Dato che sul TDD esiste già una svariata bibliografia online, lo riassumo in poche parole: consiste sostanzialmente nello scrivere i test delle nuove funzionalità prima di implementarle, seguendo un preciso flusso di lavoro (scrivo il test, lo eseguo e lo vedo fallire, implemento la funzionalità e rieseguo il test fino a farlo passare, faccio *refactoring* e ricomincio con la prossima funzionalità). In questo modo è permesso allo sviluppatore testare l'interfaccia stessa delle funzioni, metodi, moduli o oggetti che dovrà sviluppare, per poi andare a riempire i dettagli.

Con questo obbiettivo in mente abbiamo incominciato il progetto affrontando quello che ci sembrava l'aspetto più difficile e che allo stesso tempo più ci interessava, cioè l'estrazione dei dati delle piante architettoniche.

La libreria dxfgrabber gestiva il compito di leggere i file da disco, interpretare le sue informazioni e darci un oggetto che rappresentasse il suo contenuto. Ci forniva la lista di entità grafiche/geometriche (linee, segmenti, cerchi, raggruppamenti di oggetti) e i testi, o etichette, tutto rispettando i *layers*[1] con cui il file veniva organizzato. La

[1] I *layers* sono strati diversi su cui gli elementi del file vengono collocati. Servono a raggruppare elementi diversi per scopo funzionale o semantico, come nell'esempio citato per le porte.

grande maggioranza di questi file seguiva uno standard preciso per il *naming* dei *layers*: c'era un *layer* "PORTE", un *layer* "RM$" che conteneva i poligoni delle stanze, un *layer* "FINESTRE", e così via. Ma in qualche file, Il layer "PORTE" potrebbe chiamarsi "PORTA", le finestre potrebbero essere state inserite insieme alle murature nel *layer* "MURI", mentre il *layer* "SCALE" era vuoto con le scale disegnate insieme alle stanze su "RM$". Sul *layer* delle porte, a volte trovavamo le porte disegnate come linee e archi sparsi, a volte raggruppate all'interno di oggetti composti (chiamati Insert, una specie di gruppo di oggetti), e lo stesso valeva per le finestre e le scale.

Con morfologie talmente eterogenee da trattare, in che modo avremmo potuto incominciare il lavoro su questi dati/file utilizzando il TDD? Una delle qualità del TDD sta nel fatto che molto spesso, quando non si sa come incominciare l'implementazione di una funzionalità, si è in grado di scrivere almeno i test che andranno a testarla. Nel nostro caso, anche se studiassimo i dati a lungo, che assunzioni potevamo veramente fare? E allora che tipologia di test avremmo potuto scrivere inizialmente per guidarci lo sviluppo? Con queste domande abbiamo concluso che l'utilizzo sistematico del TDD avrebbe portato all'utilizzo di almeno una delle seguenti strategie per la scrittura dei test iniziali:

- Basarsi sui file VERI, rischiando di ottenere implementazioni che funzionassero soltanto per casi simili, o test poco affidabili.
- Basarsi su una quantità enorme di dati fittizi, cercando di coprire molti casi, ma scrivendo dei test fragili e potenzialmente più numerosi di quanto fosse ragionevole, specialmente per la loro manutenzione.
- Basarsi su usi complessi di *mock* per limitare la quantità di dati fittizi e isolare piccoli casi da trattare. Poco pratico perché i piccoli casi da trattare sarebbero troppi da gestire esplicitamente.

Per evitarle dovevamo essere in grado di fare qualche assunzione, ma non riuscivamo a trovare proprietà che venissero rispettate nel 100% dei casi.

Dopo un naturale momento di disperazione, abbiamo capito un aspetto imprescindibile per un lavoro così complicato, su dati così eterogenei: non dobbiamo cercare il 100%. Non l'avremmo mai raggiunto. Ci serviva, invece, una soglia ammissibile

di errore, la possibilità di segnalarli all'utente o superarli attraverso qualche euristica particolare. Sorsero allora i seguenti obbiettivi, per ogni funzionalità / problema in considerazione:

- Definire la soglia di errore ammissibile.
- Superare la maggior parte possibile di errori, con uso di euristiche diverse per coprire più casi possibili.
- Rilevare errori non risolubili per rendere possibile all'utente la loro correzione manuale. In questo modo, anche se non riuscivamo a soddisfare la soglia di errore ammissibile, avremmo sviluppato un sistema che tende in futuro a migliorare.

In altre parole, non avremmo più partito dai test, ma da un'esplorazione sistematica del problema e dei dati. Tale approccio ci ha consentito gestire meglio l'incertezza, sostituendo le assunzioni per ipotesi da testare.

In uno speech nell'Agile UX NYC 2012, Joshua Seiden discorre sul rimpiazzare i *requirements* con ipotesi quando sviluppando per un ambiente dove non ce ne sono certezze: *"When youŕe in production, building to known standard, you want requirements. When youŕe in an environment of uncertainty, you want hypotheses."* (Seiden, 2012).

L'autore sottolinea che molto spesso i *requirements* nascondono forti assunzioni sugli effetti dei cambiamenti, e la tecnica suggerita di esprimerli in forma di ipotesi è utile nell'esplicitare l'incertezza. Un approccio simile abbiamo adottato nel gestire le assunzioni che facevamo sui nostri dati, attraverso la definizione di soglie di errore come delle ipotesi di soddisfacibilità dei nostri requisiti.

In seguito allo stabilire delle ipotesi sui dati, seguiva la fase della loro validazione, dalla quale volevamo ricavare un percentuale di soddisfacibilità (quale percentuale dei dati totali soddisfa positivamente o negativamente l'ipotesi in questione).

3.2 *Experimentation Driven Development*

Per *Experimentation Driven Development* intendo una modalità pratica di sviluppo del *software* che attraverso l'esplorazione pratica del dominio applicativo, dei problemi da risolvere, e della natura dei dati, si prefigge di:

1. Garantire una comprensione più approfondita del problema.
2. Capire caratteristiche importanti/comuni su una molle considerevole di dati.
3. Rivelare in che modo il linguaggio utilizzato, i suoi moduli o librerie esterne possono contribuire alla risoluzione del problema
4. Confrontare ipotetiche soluzioni con grandi quantità di input non fittizi per capirne la precisione e/o *recall*.
5. Permettere una naturale validazione e conseguente migrazione degli approcci di esplorazione utilizzati verso un'implementazione funzionante.

È imprescindibile che l'esplorazione avvenga nell'ambiente stesso su cui avverrà l'implementazione. Per questo motivo abbiamo utilizzato la REPL (*Read Eval Print Loop*), con l'interprete python bpython[2], che fornisce diverse funzionalità utili per un approccio del genere:

- Colorazione del codice per facilitarne la comprensione
- *Indenting* automatico, il che risparmia considerevole tempo scrivendo codice su Python, in cui l'indentazione è significativa.
- Introspezione sui metodi e funzioni utilizzate, esibendo direttamente sulla REPL dettagli sulla loro segnatura e anche documentazione (docstring).
- Possibilità di salvare il codice utilizzato direttamente sul filesystem
- Funzionalità *rewind*, che permette di disfare le istruzioni precedenti. Ci permetteva ad esempio di esplorare e trasformare i dati per vederne determinati risultati, tornare in dietro e riprovare con strategie diverse.

Il nostro *workflow* basico era composto dai seguenti passi, eseguiti in modo iterativo:

- Comprendere teoricamente la natura del problema
- Utilizzare la REPL per conoscere meglio i dati e le sfide implementative, analizzando la numerosità totale degli input e la quantità di casi che soddisfano qualche insieme di proprietà desiderate.
- Creare/salvare codice che permetta o la risoluzione del problema (in termini di soglie ammissibili) o l'esecuzione di test significativi sulle funzionalità da implementare.

[2] `http://www.bpython-interpreter.org/`

Ad ogni iterazione, le prove precedenti e codici ottenuti venivano riutilizzati e permettevano una più ampia comprensione, oltre a generare dei prototipi/bozze implementativi.

Il prototipo/bozza differisce da una versione finale principalmente per aspetti di completezza (può non gestire ad esempio casi limite) e qualità del codice e prestazioni (viene scritto il più veloce possibile in modo da non bloccare il *workflow*, ma non necessariamente è un'implementazione efficace o chiara).

Esempio pratico

Un esempio su cui abbiamo utilizzato questa strategia è l'estrazione delle stanze, e in seguito le etichette e testi associati a loro, in modo da poter identificare, ad esempio, dove si trovasse la "Aula Beta" sulla piantina, o quali stanze rappresentavano dei Bagni, studi o uffici, cioè le categorie delle stanze.

La lettura delle stanze è stata poco problematica, e con l'uso della REPL siamo riusciti a ottenere a piccoli passi i codici per estrarre le stanze e le etichette.

Le difficoltà maggiori sono sorte nell'associare le etichette alle relative stanze. L'unico vincolo fra le etichette di una stanza e il suo disegno era la loro relativa posizione, senza nessun vincolo a livello sintattico. Abbiamo allora utilizzato un algoritmo che, dato un poligono S (stanza) e un punto P nello spazio (punto di ancoraggio dell'etichetta testuale), rispondeva alla domanda "Il punto P è dentro il poligono S?". Si tratta di un algoritmo particolarmente interessante e concettualmente semplice, chiamato *Ray Casting Algorithm*[3].

La nostra ipotesi sui dati era che, essendo in grado di rispondere a questa domanda, saremmo riusciti ad associare etichette correttamente al 90% delle stanze[4].

In presenza di stanze piccole o strette (come corridoi ad esempio), poteva capitare che l'etichetta, per mancanza di spazio, venisse collocata all'interno di una stanza vicina. All'occhio umano era semplice capire a quale stanza tale etichetta appartenesse,

[3]Una chiara spiegazione è presente su Wikipedia: `http://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm`

[4]La scelta di questa soglia era giustificata dal fatto che in mezzo ai file di cui disponevamo c'erano anche altri file che non ci davano informazioni utili, o che si riferivano a palazzi non utilizzati per scopi didattici, in cui la percentuale di identificazione di stanze è naturalmente più bassa

ma un po' più complicato in termini algoritmici.

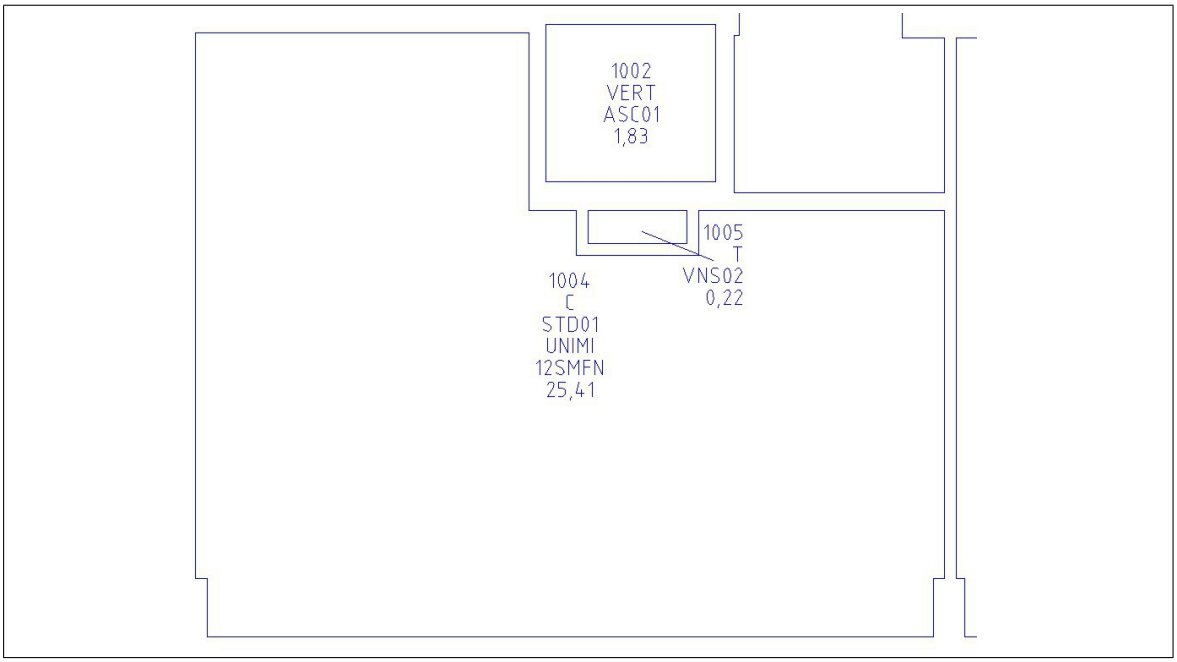

Figura 2: Esempio di stanza con più etichette al suo interno. Le etichette corrette sarebbero quelle indicate dal numero 1004. Quelle di numero 1005 sono riferite al piccolo rettangolo indicato (in questo particolare caso, ma non sempre) dalla linea uscente, e si tratta di un "Cavedio Porta Impianti", cioè uno spazio per il passaggio dei cavi

In seguito la REPL ci ha permesso capire quanto buona effettivamente fosse la nostra soluzione. L'algoritmo riusciva ad associare etichette con un *recall* del 94% delle stanze, con una *precision* sul 88%. Avevamo in mano, dall'uso della REPL, una soluzione che sembrava quasi soddisfacente. Non avevamo il 90% desiderato ma eravamo abbastanza vicini.

Abbiamo allora lanciato una nuova ipotesi: se scartiamo qualche tipologie specifiche di stanze che ai nostri scopi sono di minore importanza, riusciremo ad aumentare la precisione al 90% desiderato, al patto di ridurre il percentuale di *recall*.

Con l'uso della REPL abbiamo eseguito delle prove veloci manipolando i codici già esistenti in modo che scartassero le etichette di tipo "cavedio" e infatti siamo arrivati a un' *recall* del 93% con 90% di precisione stimata.

Infatti, durante lo sviluppo, l'uso della REPL per l'esplorazione di questi dati prima dell'effettiva sintesi degli algoritmi e scrittura del codice ci ha permesso di

affrontare i problemi in modo più efficiente. Ci permetteva velocemente di capire caratteristiche comuni o analizzare attributi dei dati e provare alternative, per poi migrare quelle conclusioni verso un'implementazione effettiva.

A volte ottenevamo dalla REPL un modulo già semilavorato, a volte semplicemente una media campionaria che ci permetteva di avere più confidenza nelle scelte prese. Quando una precisione assoluta e deterministica non era possibile, l'analisi delle qualità dei dati giustificava assunzioni che ci permettevano di procedere a un'estrazione **sufficientemente o statisticamente buona**, superando le difficoltà dell'analisi e la difficoltà di correggere errori umani. In altre parole, ci permetteva di continuare anche quando in termini assoluti sarebbe stato sbagliato farlo.

L'ultimo vantaggio è quello di garantire un rapido *feedback* per le strategie adottate. Di certo non tanto rapido quanto una suite di test, ma nel contesto in cui la scrittura dei test sarebbe stata problematica, l'uso della REPL ha agevolato significativamente la valutazione manuale e l'ottenimento di statistiche.

3.3 La naturale migrazione verso un approccio prototipale

Molti componenti dell'applicazione finale sono nati a partire da questi esperimenti con la REPL, incentivando un approccio alla progettazione di tipo prototipale.

Il principale vantaggio era quello di poter conoscere la natura dei problemi o i limiti e le potenzialità delle nostre scelte algoritmiche, senza dover pagare un prezzo troppo alto se mai fosse necessario tornare in dietro. Fino a questo punto non si trattava di design di software, ma di sola esplorazione e comprensione. A misura che riuscivamo ad avere implementazioni soddisfacenti per ogni funzionalità, le incorporavamo al *codebase* principale e proseguivamo verso l'esplorazione di ulteriori limiti e possibilità.

Da queste versioni prototipali ottenute dalla sperimentazione, ottenevamo versioni più elaborate eseguendo delle continue trasformazioni e operazioni di *refactoring*. Perché funzionasse correttamente, le prove sulla REPL dovevano essere estremamente veloci, e allora le attività di design e *testing* non erano la priorità. In questa fase di *refactoring* dovevamo sanare queste eventuali deficienze.

Una volta capito meglio il *wicked problem* in questione, lo sviluppo proseguiva in una di due possibili strade: scartare il prototipo per rifarlo da capo, o rielaborarlo.

Nel caso venisse scartato, un approccio di tipo Test Driven Design (TDD) veniva utilizzato per la sua ricostruzione. In questo modo potevamo centralizzare gli sforzi nella definizione del design e dell'interfaccia della componente, dato che le procedure per la risoluzione del problema erano già state esplorate.

In qualche casi, però, ci sembrava evidente che una nuova versione fatta da zero avrebbe avuto molto in comune con il prototipo in mano. In questi casi abbiamo scelto di rielaborarlo e risistemarlo, in particolare assicurandoci di:

- Ripensare le sue interfacce utilizzando una metodologia per il *refactoring* ispirata al TDD, facendo finta che nessuna implementazione esistesse, scrivendo test contro l'interfaccia ideale, e poi trasformando l'implementazione fino a far passare i test.
- Gestire casi estremi e eccezioni.
- Stabilire una modularità adeguata.
- Rispettare gli standard per la buona progettazione.
- Aggiungere *test coverage*, in speciale test di unità di tipo *white box*, *non regression test* e test di accetazione *end to end*.

Rimane da dire come è stata superata la difficoltà del testing. Una volta che avevamo affrontato la diversità dei dati con un approccio di esplorazione, avevamo dimostrato diverse delle nostre ipotesi, e a quel punto eravamo anche in grado di scrivere dei test che testassero in che modo il nostro codice si relazionava con quelle ipotesi. Non dovevamo più testare la funzionalità di per se (o i requisiti), ma le ipotesi che avevamo definito con il margine definito.

Attraverso la creazione di opportuni moduli e classi che ci permettessero di gestire astrazioni delle informazioni dei file DXF, e non gli elementi veri estratti dalla biblioteca dxfgrabber, siamo stati in grado di definire una quantità di test di qualità significativa, con sufficiente isolamento rispetto ad altre funzionalità / moduli del sistema.

3.4 Non regression test

Questo *workflow* fece si che nessun componente prototipale rimanesse intoccato per molto tempo. In conseguenza dell'aggiunta di nuove funzionalità o di cambiamenti del codice esistente, per ad esempio gestire più casi, le componenti venivano rielaborate, ritrasformate e pulite. In sostanza, il *refactoring* di pezzi preesistenti diventò parte essenziale del processo di sviluppo, e per renderlo possibile abbiamo dato più importanza ai textitnon regression test e a una modularità e incapsulamento che ci permettessero di cambiare il codice con sufficiente confidenza.

Avere un insieme di test su cui ci fidavamo è stato essenziale per rendere possibile e veloce lo sviluppo in queste modalità. Un aspetto curioso da discutere è proprio questo: in che modo aumentare il grado di affidabilità fornito da una *suite* di test?

Un altro contesto in cui l'analisi statistica dei dati e dei file è stata essenziale è stato durante la fase di integrazione di informazioni sulle stanze e piani di ogni edificio. Alla fine della lettura e estrazione, abbiamo scoperto che non esisteva nessun standard identificativo per i piani che venisse rispettato da tutte le sorgenti. Non c'era un modo diretto per associare due piani dello stesso palazzo su sorgenti diverse, e ciò rendeva l'integrazione dell'informazione delle stanze molto difficile.

Con l'uso della REPL, abbiamo stabilito però che sul 99% dei piani era possibile trovare almeno una stanza, utilizzando il codice identificativo, in tutte e tre le sorgenti. Sapendo in quale piano una stanza veniva rimappata in ogni sorgente ci dava una forte indicazione di come quei piani dovrebbero venir relazionati. L'unico problema erano i casi estremi, in cui due stanze di uno stesso piano su una sorgente venivano rimappate su piani diversi in un'altra sorgente, o quando per qualche piano nessuna associazione di stanza con le altre sorgenti era possibile. L'algoritmo finale che abbiamo concepito gestisce tutti i possibili casi, e se ne accorge quando non riesce a trovare un'associazione fra piani di due o più sorgenti.

In questo modo all'utente vengono segnalati i conflitti, sia quelli che abbiamo risolto in modo automatico che quelli la cui risoluzione non è possibile. Con l'uso di questa strategia e una serie di eurisitche per renderla più efficiente, siamo stati in grado di associare il 99% dei piani correttamente. L'1% che avanza è costituito da precisamente tre casi, due dei quali si presenta solo a causa di un errore di battitura dei dati originali (stanze identificate in modo sbagliato), e comunque vengono tutti e

tre segnalati per la revisione dell'utente.

3.5 Refactoring guidato - SOLID

Lorem Ipsum dolor sit

"We cannot solve the problems we have created with the same thinking we used in creating them." - A. Einstein

Bibliografia

[1] Seiden, Josh, Replacing Requirements with Hypotheses, speach, http://www.slideshare.net/jseiden/2012-feb-25-agile-ux-nyc-seiden-requirements-to-hypotheses?from=ss_embed, 2012.