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

Durante la scrittura della relazione ho provato ad essere il più preciso possibile sia nel racconto del processo di sviluppo del progetto a cui ho partecipato sia nelle conseguenti riflessioni.

Ho provato a includere il massimo possibile di dettagli e contenuti, sotto la condizione che fossero coerenti con l'argomento scelto per la tesi. Il progetto del tirocinio, però, è stato molto più ampio di quanto qua riesco a riportare sotto un unico filo logico. Mi viene allora la necessità di dire che tanto è stato lasciato fuori, tante nuove conclusioni e apprendistati che si sono verificati lungo tutti i mesi di progettazione e sviluppo.

Tra questi argomenti stanno le riflessioni sull'organizzazione di un team autogestito, senza nessuna tipo di gerarchia netta, le conseguenti difficoltà e come siamo stati in grado di superarle, e sulla (per noi) nuova esperienza di *pair programming*. Sul filo più pratico, lascio fuori tante delle curiose sfide tecniche e decisioni che abbiamo preso, qualche particolarmente interessante soluzione per l'integrazione dei dati o ottimizzazione algoritmica.

Ho provato a fornire definizioni e spiegazioni dei concetti utilizzati sempre che giudicavo necessario. Però, come in qualsiasi ambito, in quello informatico ce ne sono concetti generalmente diffusi, per cui mi sono limitato spesso a descrizioni limitate e non esaustive. Ulteriori informazioni possono sempre essere facilmente trovate in rete.

Note di stile

L'italiano non è la mia lingua madre, e perciò prevedo che molte frasi non vi risuoneranno in modo idiomatico. Ho cercato di utilizzare l'italiano il più spesso possibile, ma ho comunque mantenuto molti termini in inglese. Prevalentemente sono termini tecnici, e, in occasione del primo utilizzo, cerco di fornire una piccola descrizione e/o traduzione come nota a piè di pagina.

Ci sono però un paio di citazioni che mi aiutano a illustrare determinati ragionamenti, che sono state ottenute a partire da letteratura in inglese. Ho inserito allora una mia traduzione degli opportuni brani, e indicato a piè di pagina la versione originale in inglese che ho utilizzato. In questo modo rimane al lettore la possibilità di capire meglio in originale o disambiguare la traduzione.

Ho provato a tenere il flusso di lettura leggero, senza troppo carico teorico o di definizioni (anche per compensare la difficoltà di esprimere aspetti complessi in un'altra lingua), spiegando esaustivamente i concetti ma evitando di ripetermi troppo. Si tratta di un racconto, una relazione su un progetto sviluppato, sugli obbiettivi raggiunti, gli ostacoli trovati e le lezioni acquisite.

Per quanto riguarda lo stile, vorrei notare che l'uso alternato tra prima persona plurale (noi) e prima persona singolare (io) è intenzionale, per il fatto che ho partecipato a un progetto come membro di un team. Perciò uso il plurale per la descrizione degli aspetti collettivi (nostri obbiettivi, compiti, ecc), ma passo al singolare per descrivere gli aspetti su cui mi sono concentrato particolarmente di più, i miei ragionamenti, riflessioni o opinioni.

Per chiarezza ho evidenziato i termini in inglese con l'uso del corsivo, ad eccezione di quando sono all'interno di paragrafi di citazione o quando si trattano di nomi propri.

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.

TODO - CAMBIARE: Su questa relazione descrivo il processo di sviluppo della suddetta applicazione da zero, con particolare rilievo alla ricerca della buona progettazione e all'utilizzo di una modalità operativa 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

La prima sfida per lo sviluppo di un progetto è quella di capire i requisiti e i problemi coinvolti. A seconda della tipologia di progetto, ciò può richiedere tempo e dedicazione considerevole. Molto spesso però il capire avviene soltanto durante lo sviluppo stesso: sono casi in cui la comprensione del problema avviene, contraddittoriamente, con la sua risoluzione.

Il nostro 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, con decine di migliaia di righe su ognuno, il che è ragionevole, se consideriamo che

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

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).

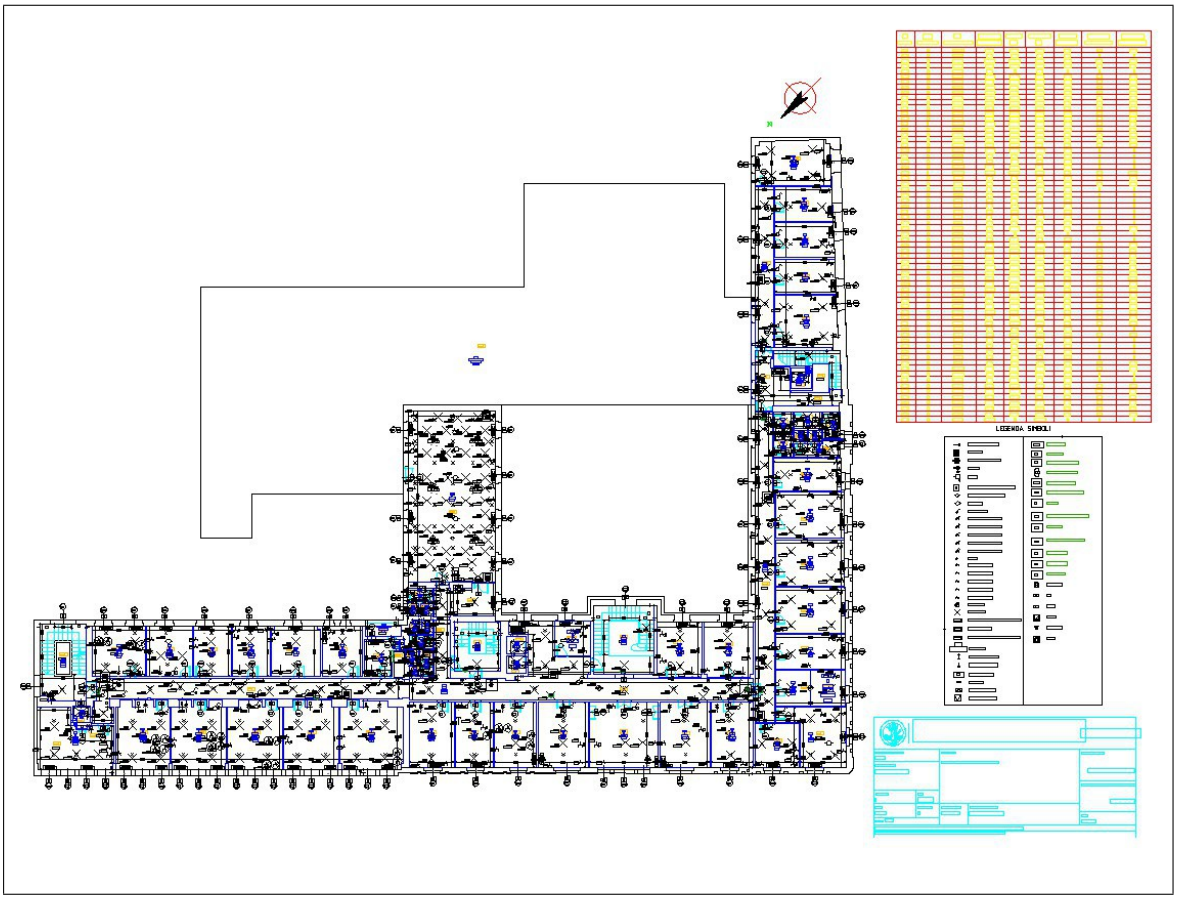

Figura 1: Esempio del contenuto di un file DWG di media complessità (Dipartimento di Informatica, 1° piano). La quantità di informazioni presente rende difficile la comprensione del contenuto informativo del file manualmente.

Con l'aiuto anche della divisione di Sistemi Informativi, rappresentata in speciale da Vincenzo Pupillo, abbiamo ottenuto in formato testuale CSV le informazioni utilizzate dal sistema Easyrooms, 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 dei codici identificativi presenti anche su quelle. Dalle piante ottenevamo anche la localizzazione di aree di interesse come biblioteche, sale studio, bagni, spazi di ristorazione, eccetera.

In sostanza, il lavoro di integrazione consisteva nel far confluire in prima istanza i dati testuali dei dipartimenti di Sistemi Informativi e Edilizia. Si trattava allora di relazionare informazioni sui palazzi e sulle aule didattiche. Il risultato di questa operazione, dovevamo correlare con le informazioni presenti nelle piantine edili. Lo scopo finale è quello di avere un ulteriore set di dati disponibile, con la pretesa di essere più preciso e di averne più informazioni delle singole sorgenti di partenza.

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.

Da questo segue una nuova sfida: prima ancora di procedere all'integrazione, dovrebbe esserci un efficace meccanismo di validazione dei dati di partenza. Questo è diventato un ulteriore valore dell'applicazione, cioè la capacità di fornire un'analisi della correttezza sintattica dei file in input, permettendo un valido *feedback* per ulteriori miglioramenti e correzioni manuali.

Nessuna assunzione

La molle di dati era significativa, in speciale per i file DXF: 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.

[2]Più dettagli nel capitolo 3

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. Possiamo vedere le *list comprehentions* come un'applicazione più leggibile del paradigma map-filter-reduce.
- 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, generazione di immagini, ecc).
- 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 un linguaggio 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 valide, e ciò si ripeteva anche sugli altri dati. L'uso di un DBMS relazionale avrebbe portato a degli *schema* con considerevoli campi *null* e denormalizzato, due aspetti considerati *anti-patterns* nella progettazione di *database* relazionali. 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

Una mia particolare preoccupazione in fase iniziale del progetto, era quella di garantire la costante e ricorrente ricerca per la buona progettazione, prima, durante e dopo l'implementazione delle funzionalità. A parte i vantaggi ovvi per il prodotto finale di un approccio del genere, avevo lo scopo personale di sanare un mio *gap* tecnico: sentivo che le mie conoscenze di (buona) progettazione del Software erano molto spesso di carattere teorico, e mi mancava allora l'assimilazione pratica dei principali concetti. Nella mia opinione l'esistenza di questo *gap* è abbastanza comune tra gli altri studenti del mio stesso anno. Infatti, il perfezionamento di questo tipo di abilità può soltanto avvenire al confronto dello studio teorico con l'applicazione pratica, e in questo ho trovato un forte contributo nello sviluppo di questo lavoro.

In precedenza, nella mia carriera professionista e accademica, poche sono state le opportunità in cui ho potuto pensare al design del software complessivamente. Nella maggior parte dei casi, applichiamo precetti di design a livello *micro*: singolo algoritmo, singola procedura, al massimo singola classe/modulo. Molto spesso neanche quello potevo fare, dato che la preoccupazione principale era quella di sintetizzare un algoritmo funzionante e performante, oppure di esplorare una determinata tecnica o risolvere un problema pratico in un determinato orizzonte temporale. Pensare invece a un sistema più complesso, con più componenti e elementi, organizzarli a vicenda, gestire le loro dipendenze, garantire un isolamento e disaccoppiamento salutare, non va fatto sotto stretti vincoli temporali, e allora non ha mai fatto parte della mia esperienza personale.

La maggior parte della mia carriera professionista ha avuto luogo in ambito di sviluppo web, e dovevo sempre partire da un qualche *framework*[1] o sistema preesistente, che mi dava già una risoluzione del *design* a livello macro, di solito con l'utilizzo di pattern strutturali come MVC / MVP. A me restava più che altro rispettare le convenzioni fornite dall'ambiente di sviluppo e assicurarmi che a livello *micro* (metodi, classi) non causare troppi danni.

Su questo progetto non avevamo una struttura di partenza, e non avevamo qualcuno all'interno del *team* che ci mostrasse le soluzioni migliori. In altre parole, la mancanza di esperienza nella costruzione del (buon) software da zero rappresentava una significativa sfida didattica e pratica su questo progetto.

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. Dovevamo allora scoprire in che modo la libreria dxfgrabber ci poteva aiutare. Naturalmente, abbiamo aperto la REPL[2] di Python, e abbiamo incominciato ad esperimentare con la libreria e i dati.

[1] La maggior parte della mia carriera come sviluppatore ha avuto luogo utilizzando *Ruby on Rails*, un *framework* noto per non solo fornire una struttura MVC, ma per essere anche fortemente *opinionated*, cioè di avere le *best practices* definite e incentivate in modo molto chiaro.

[2] REPL - *Read Eval Print Loop*. Si tratta di un ambiente di programmazione interattivo che ci permette di inserire comandi del rispettivo linguaggio di programmazione ed avere il risultato

La libreria dxfgrabber gestiva il compito di leggere i file da disco, interpretare le sue informazioni e restituirci 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* [3] con cui il file veniva organizzato. La 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'applicazione sistematica 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, suite di test troppo lenti 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 continuavano ad aumentare a seconda che esploravamo più file.

stampato in sequenza

[3] 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.

Per evitarle dovevamo essere in grado di fare qualche assunzione, ma non riuscivamo a trovare proprietà che venissero rispettate nel 100% dei casi. Abbiamo capito un aspetto imprescindibile per un lavoro così complicato, su dati così eterogenei: non dovevamo 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, e infatti era esattamente quello che avevamo già fatto alla prima apertura della REPL per l'esplorazione della libreria utilizzata: volevamo capire meglio in che modo lo strumento ci potrebbe aiutare nella gestione 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 caratterizzato dall'incertezza:

> Quando si sta in produzione, sviluppando su standard predefiniti, si vogliono i requisiti. Quando si sta in un ambiente di incertezza, si vogliono ipotesi.[4]
>
> *(Traduzione mia, originale in inglese[4])*

[4] "When you're in production, building to known standard, you want requirements. When you're in an environment of uncertainty, you want hypotheses."

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, con l'interprete python bpython[5], 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*).

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

- 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.

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 a volte per aspetti di completezza (può non gestire ad esempio casi limite), a volte per qualità del codice o prestazioni (viene scritto il più veloce possibile in modo da non bloccare il *workflow*, ma non necessariamente si tratta di un'implementazione efficace o chiara).

Non abbiamo scelto esplicitamente nessuna di queste metodologie di lavoro, piuttosto ci siamo fatti guidare dalla natura dei problemi e dei dati da trattare.

3.3 *Esempio pratico dell'uso della sperimentazione e REPL*

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. Si noti che con la REPL partivamo alla esplorazione, mirando a una comprensione maggiore del problema e dei dati, e finivamo per risolvere il problema. In questo modo non era più possibile scrivere i test *a priori*, invalidando uno dei punti cardine dell'uso sistematico del TDD.

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*[6].

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[7].

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, ma un po' più complicato in termini algoritmici.

In seguito la REPL ci ha permesso di 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 89%. 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 relativa 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 soltanto le etichette di tipo "cavedio" e infatti

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

[7]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 sarebbe naturalmente più bassa

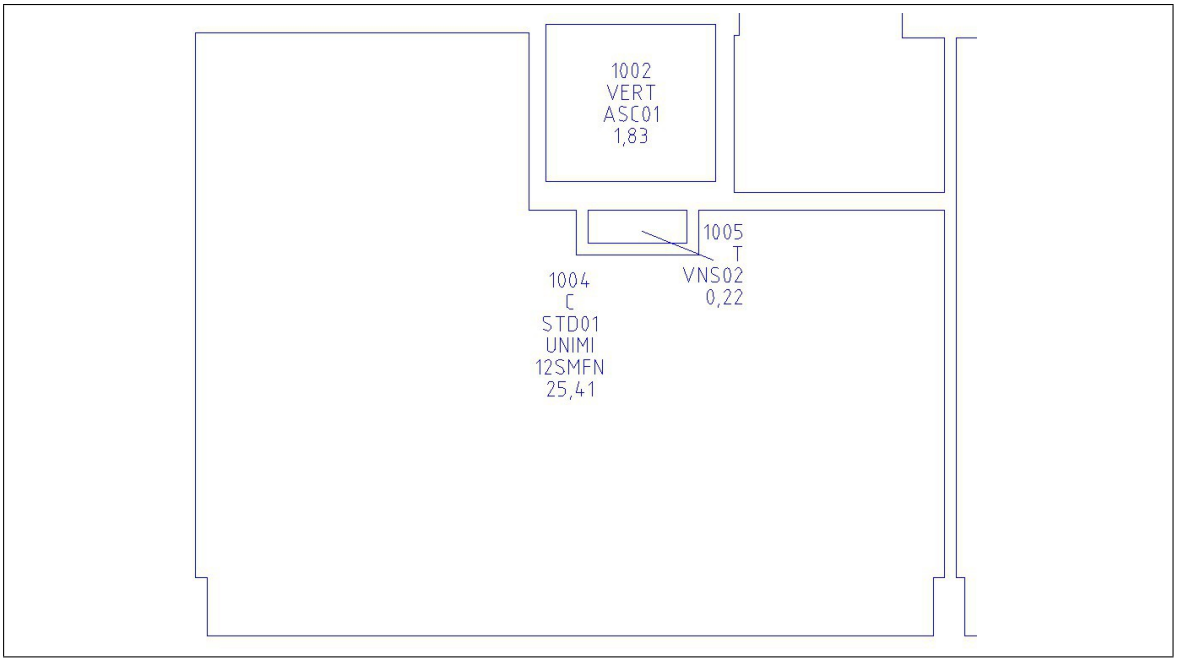

Figura 2: Esempio di stanza con più etichette al suo interno. Le etichette corrette sarebbero quelle indicate dal numero 1004. Quelle di numero 1005 sono riferite al piccolo rettangolo indicato (in questo particolare caso, ma non sempre) dalla linea uscente, e si tratta di un "Cavedio Porta Impianti",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

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. Creando ipotesi, dimostrando che coprissero i casi prevalenti e aggiungendo in seguito euristiche per l'inclusione di più casi specifici, siamo riusciti ad affrontare la maggior parte delle difficoltà.

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 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 di questa metodologia 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.

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. La difficoltà era quella di dire quale piano di una determinata sorgente dovrebbe coincidere con quale piano di un altra sorgente.

Con l'aiuto della REPL abbiamo scoperto che molto spesso determinate sorgenti avevano informazioni su piani che altre sorgenti non conoscevano, o che addirittura Easyroom colassava nel piano terra spesso stanze del piano sotterraneo e del piano terra. Queste caratteristiche creavano degli ostacoli per l'associazione dei piani fra le diverse sorgenti.

La REPL ci ha permesso anche di stabilire che su un impressionante 99% dei piani era possibile trovare almeno una stanza, utilizzando il suo codice identificativo, che venisse rintracciata in tutte e tre sorgenti (piantine, informazioni testuali dell'edilizia e informazioni testuali di Easyroom), e allora la presenza di quella stanza ci dava un'informazione ortogonale ai piani delle diverse sorgenti. Sapendo in quale piano una stanza veniva rimappata per sorgente ci dava una forte indicazione di come quei piani dovrebbero venir relazionati. L'unico problema che rimaneva 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 euristiche 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.

3.4 La naturale sovrapposizione con un approccio prototipale

Molti componenti dell'applicazione finale sono nati a partire da questi esperimenti con la REPL, avvicinando la nostra modalità operativa a un approccio di 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.

Col il nostro obbiettivo per la ricerca continua della buona progettazione, una volta capito meglio il *wicked problem* in questione, lo sviluppo doveva proseguire in una di due possibili strade: scartare il prototipo per rifarlo da capo, o rielaborarlo. Mantenere la versione originale senza ripulirla e testarla non era una scelta possibile, specialmente a lungo termine.

Dati i tempi stretti, nella maggior parte dei casi abbiamo optato per la rielaborazione del prototipo esistente e non per il suo scarto, poiché era evidente che una nuova versione fatta da zero avrebbe avuto molto in comune con il prototipo in mano.

Infatti questa è la maggior differenza del nostro approccio a quelli prototipali più comuni, in cui si cerca di trattare i prototipi in modo usa-e-getta. Nella rielaborazione, ci siamo assicurati di:

- Ripensare le interfacce utilizzando una metodologia per il *refactoring* ispirata al TDD: ignoravamo quella già esistente e partivamo dalla stesura di test contro l'interfaccia che considerassimo ideale. Da questi test proseguivamo all'applicazione di successive trasformazioni all'interfaccia già implementata fino ad avere test passanti.
- Rilevare casi estremi ed eccezioni, testarli e poi gestirli.
- Stabilire una modularità adeguata, limitando le dipendenze.
- Aggiungere *test coverage*, in speciale test di unità di tipo *white box*, *non regression test* e test di accettazione *end to end*.

Rimane da dire come è stata superata la difficoltà del *testing*, che ci aveva fatto rifiutare l'approccio iniziale del TDD. 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 confrontassero il modo in cui 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 e che ci fornivano il margine desiderato. L'esplorazione ci ha fornito la comprensione dei problemi e anche la capacità di testarli.

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.

3.5 *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, l'estensione di pezzi preesistenti diventò il

punto di partenza essenziale del processo di sviluppo, e per renderlo possibile abbiamo dato più importanza ai *non 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, ma non era sufficiente per metterci al riparo della perdita del design a lungo termine. Questo processo di rielaborazione di codice per aggiunta di nuove funzionalità ha finito per portare il nostro *codebase* a un punto da cui non riuscivamo più a procedere. Si trattava del noto fenomeno del *Software Decay*, che purtroppo non avevamo prima anticipato.

Capitolo 4

Un Primo Contatto con il *Software Decay*

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

A un certo punto dello sviluppo mi sono accorto che incominciavamo a fallire. Gli algoritmi funzionavano, il programma eseguiva i suoi compiti, ma il codice diventava intricato e difficile da mantenere.

Non bastava esserci messi l'obbiettivo della ricerca costante della qualità e chiarezza del design. Poco a poco, il nostro sistema era diventato complesso, le modifiche costavano di più e non potevo evitare di pensare che ci fosse qualcosa di sbagliato mentre guardavo il codice, l'organizzazione dei diversi moduli, o addirittura la composizione interna di qualche algoritmi.

A quanto pare, il nostro progetto, al crescere in dimensione e complessità diventava vittima del fenomeno noto come *software decay*. La nostra intesa "buona progettazione" fatta dall'inizio veniva poco a poco storpia.

L'approccio di sviluppo iniziale basato sulla REPL si era rivelato eccezionale per alcuni aspetti, in speciale nella fase iniziale del progetto dove la necessità di capire era predominante rispetto alle preoccupazioni sulla qualità del codice. Una volta che avevamo superato le limitazioni iniziale dei problemi e dati di partenza, il sistema è

cresciuto velocemente con l'aggiunta delle funzionalità di integrazione dei dati, ulteriori validazioni, interazioni con il database e funzionalità di segnalazione. Cresceva in complessità.

Sentivo ovunque i *code smells*[1], cioè sintomi sul programma che indicavano possibilmente la presenza di problemi più profondi. Tra la percezione dei *code smells* e la capacità di trovare le loro radici esiste però una grossa differenza, e l'esperienza e pratica è ciò che molto spesso ci rende in grado di stabilire il diagnostico completo.

Mi chiedevo che cosa ci fosse di sbagliato, leggendo il codice e studiando il funzionamento del sistema ripetitivamente, senza riuscire però a nominare i problemi. Mi mancava. Mi mancava l'esperienza, il know-how pratico di qualcuno che ne avesse già imparato a fare questo stesso mestiere: guardare al proprio *codebase* e diagnosticare la "puzza".

Incominciava a dominarmi la paura di modificare il codice, di aggiungere o togliere funzionalità. Per compensare mi è venuta l'idea di aggiungere più test al sistema, per assicurarmi che coprissero la maggior parte di casi possibili. Con la complessità acquisita, non riuscivamo più a cogliere cosa doveva fare ogni pezzo del sistema e come si relazionavano. Una cosa che ci ha aiutato è stata l'elaborazione di qualche *activity diagram* per almeno renderci esplicito ad altissimo livello i principali flussi di esecuzione del programma.

Ho resistito alla tentazione di aggiungere test per compensare questi difetti, prima perché mi sembrava sbagliato compensare problemi di *design* (che addirittura non riuscivo ancora a capire) con l'aggiunta di test, ma in secondo luogo perché la maggior parte di questi test verrebbe tautologica, serviva a poco o copriva aspetti già coperti da altri. Sentivo ancora che incominciavamo a fallire, una follia.

4.1 Tre Programmatori in Cerca di Esperienza

(Nota dell'autore: ogni riferimento a scene o persone o drammaturghi italiani in questo sottotitolo non è puramente casuale)

[1] letteralmente "puzza del codice"

Non potevo evitare di pensare a quanto mancasse al team un membro più esperto, qualcuno che avesse già vissuto quella difficoltà e avesse imparato da qualcun altro o addirittura a partire da uno sforzo proprio! Mi sono accorto però che se qualcuno poteva riuscirci da solo, potevo provare anch'io, e chi sa un giorno sarei stato io ad aiutare qualcuno in una situazione simile? E allo stesso tempo la sensazione di aver bisogno di un maestro da farci vedere come procedere era costante.

In mancanza di questa figura all'interno del team, mi chiedevo in che altro modo avrei potuto acquisire quell'esperienza. Il primo tentativo che considero sia stato un successo parziale è stato quello di cercare i concetti teorici della "buona progettazione" e in particolare sono finito a rileggermi ancora una volta i principali materiali disponibili sui principi SOLID[2].

Dei principi SOLID, l'SRP (*Single Responsibility Principle*) è stato quello più utile, in quanto averlo visitato e utilizzato la sua luce per rileggere ancora una volta il nostro *codebase*, ha comportato modifiche significative di tipo strutturale, sia a livello di singolo algoritmo che a livello di modulo / classe.

Per capire le responsabilità di ogni classe/metodo, ho utilizzato un suggerimento di Sandi Metz nel libro "Practical object-oriented design in Ruby: an agile primer":

> Un altro modo per concentrarsi su che cosa una classe sta veramente facendo è provare a descriverla in una sola sentenza. Ricordarsi che una classe deve fare soltanto la più piccola cosa utile possibile. Questa cosa dev'essere semplice da descrivere. Se la descrizione più semplice che si riesce a ottenere usa la congiunzione "e" la classe ha probabilmente più di una responsabilità. Se utilizza la congiunzione "o", allora la classe possiede più di una responsabilità e non sono nemmeno molto correlate. [3]
>
> *(Traduzione mia, originale in inglese[3])*

[2] SOLID è un acronimo per i cinque principi basici della programmazione orientata ad oggetti proposti da Robert C. Martin. I principi sono *Single responsibility, Open-closed, Liskov substitution, Interface segregation e Dependency inversion.*

[3] "Another way to hone in on what a class is actually doing is to attempt to describe it in one sentence. Remember that a class should do the smallest possible useful thing. That thing ought to be simple to describe. If the simplest description you can devise uses the word "and," the class likely has more than one responsibility. If it uses the word "or," then the class has more than one responsibility and they aren't even very related."

Si noti che secondo Sandi Metz, la presenza di una unica di queste congiunzioni nella "frase di responsabilità" è già sufficiente per concludere che quella classe ha più di una responsabilità. Pronto, adesso avevo un principio guida per aiutarmi, a partire dall'analisi sistematica del codice, a trovare l'origine dei *code smells*.

4.2 Esempio pratico di *refactoring* guidato dall'SRP

Un esempio pratico in cui l'SRP ha contribuito in due granularità diverse è la classe chiamata DataUpdater, inizialmente responsabile per l'aggiornamento delle informazioni sui palazzi e sulle stanze provenienti dalle fonti testuali (file CSV). La sua interfaccia era costituita da due principali metodi, update_buildings e update_rooms (aggiorna palazzi e stanze rispettivamente).

Entrambi questi metodi erano parzialmente definiti, nel senso che, cercando già a priori un design di qualità, gli avevamo concepiti come *template methods*[4], delegando parte del compito/lavoro a degli *hook methods* sovrascritti dalle sottoclassi.

In conseguenza DataUpdater era superclasse di due classi più specializzate: EdiliziaDataUpdater e EasyRoomDataUpdater, che, attraverso il polimorfismo, sovrascrivevano questi *hook methods* opportuni per aggiungere il *know-how* specifico a seconda della provenienza dei dati. Come esempio sta la validazione degli identificativi dei palazzi: dati provenienti dal dipartimento di Edilizia presentavano due tipologie di formato diverse, uno nuovo e *standard* e un altro *legacy*, cioè utilizzato in passato ma che dovevamo mantenere per questioni di compatibilità, mentre quelli usati da EasyRoom avevano un solo formato.

In sostanza, il *class diagram* era come segue[5]:

[TODO *Class Diagram* iniziale]

[4] Pattern comportamentale che permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono. In questo modo si può ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune. I punti di "aggancio" perché le sottoclassi possano ridefinire questi comportamenti vengono detti *hook methods*.

[5] Per semplicità ho omesso metodi e attributi privati

Alla stesura della "frase di responsabilità" per DataUpdater, ho ottenuto il seguente: "Gestire la validazione e aggiornamento dei dati delle stanze e/o palazzi in database e far partire la procedura di integrazione con le altre sorgenti". Dalla lunghezza della frase e della classe stessa e dalla quantità di congiunzioni, ho concluso che si trattava di una classe da venir spezzata. Si trattava di una classe considerevolmente lunga, e sicuramente ne gestiva almeno due responsabilità distinte.

Approfittando l'ereditarietà multipla fornita da Python, la separazione più netta era stata quella di dividere DataUpdater in due classi, BuildingDataUpdater e RoomDataUpdater, facendo si che EdiliziaDataUpdater e EasyRoomDataUpdater ereditassero da entrambe. Un altra possibilità sarebbe quella di favorire la composizione invece di usare l'ereditarietà[6]. Ho deciso per l'ereditarietà per due motivi: non avevamo richieste di cambiamenti a *runtime*, e non esistevano metodi o attributi in comune utilizzati da BuildingDataUpdater e RoomDataUpdater, il che garantiva che non ci sarebbero nessun dei noti problemi di *method resolution order* che nascono in contesti di ereditarietà multipla. Da questo cambiamento siamo passati al seguente *class diagram*:

[TODO *Class diagram* dopo separazione]

Dopo aver sistemato i bug introdotti da questo cambiamento e rilevati correttamente dalla nostra suite di test, a quel punto avevo due "frasi di responsabilità" in mano:

- RoomDataUpdater: "Gestire la validazione e aggiornamento dei dati delle stanze in database e far partire la procedura di integrazione con le altre sorgenti"
- BuildingDataUpdater: "Gestire la validazione e aggiornamento dei dati dei palazzi in database e far partire la procedura di integrazione con le altre sorgenti"

Le frasi continuavano grandi, ma adesso la responsabilità era stata divisa esattamente a metà. Una classe gestiva una tipologia di dati, un altra classe l'altra. Ma non ero soddisfatto.

[6] *Favor composition over inheritance* - un altro principio di design del software.

Avevo messo su il mio "cappello del refactoring"[7], e non volevo toglierlo (mi stava bene :D). Ho incominciato con RoomDataUpdater, e ho provato ulteriori suddivisioni, cercando dei pattern da applicare e sono riuscito a fare un paio di modifiche. Alla fine di questa serie di modifiche avevo raggiunto qualcosa che mi sembrava più "modulare". Metto la parola modulare tra virgolette perché quello che avevo fatto in realtà era stato frammentare quel flusso di esecuzione in una quantità elevata di piccole classi che incapsulavano poca o nessuna unità logica di esecuzione. Il flusso dell'algoritmo era stato spezzato in modo che la sua comprensione era praticamente impossibile.

La mia fortuna è che si trattava di fine giornata quando avevo dichiarato "compiuta" quell'impresa. Ero stanco e dovevo ancora risistemare un paio di test di unità che fallivano in conseguenza di quel refactoring perché testavano l'esecuzione precedente in modo vincolato all'implementazione, e allora ho deciso di lasciare la conferma (*commit*) di quelle modifiche verso il nostro sistema di *versioning* per il giorno dopo. La mattina successiva, ansioso per il primo *commit* del giorno, al guardare ciò che avevo fatto ho sentito: puzzava, e un altra volta non capivo il motivo. Mi sono messo a rivisitare altri concetti di Progettazione Software, cercando qualcosa che mi aiutasse a capire perché, dopo aver applicato così sistematicamente l'SRP avevo un ulteriore impressione che ci fosse qualcosa di sbagliato.

4.3 La Redenzione Attraverso il Concetto di *Cohesion*

Un po' per caso sono arrivato a rileggere informazioni sul concetto di *Cohesion*[8]. Nel tentativo di applicare il principio della *singola* responsabilità in modo praticamente fanatico, avevo violato la coesione logica e anche temporale dell'algoritmo di

[7] Allusione alla metafora di "Two Hats" di Kent Beck, raccontata da Martin Fowler in [2], in cui afferma l'importanza di essere sempre cosciente di quale cappello stiamo utilizzando, ovvero di avere chiara la separazione tra il momento di implementare nuove funzionalità e quello di fare textitrefactoring.

[8] La coesione è una misura di quanto strettamente correlate siano le varie funzionalità messe a disposizione da un singolo modulo (o classe). Si intende come una buona qualità in software quando un modulo o classe è considerato coeso. Da non confondere con il termine *coupling*, che sta ad indicare un accoppiamento non necessario fra moduli/classi diversi e che genera più dipendenze e difficoltà per il cambiamento, e di solito è considerato come un aspetto negativo.

partenza. Il motivo per cui il metodo update_rooms incapsulava quella sequenza di passi era funzionale, logico. Per inserire una stanza la dovevo prima pulire (formattando meglio le stringhe, ad esempio) e validare (non dovevo inserire stanze che non godessero, ad esempio, di un identificativo del palazzo a cui appartenevano). In seguito avrei preparato il formato verso il database, inviato i dati e, una volta che i dati originali di questa sorgente finissero memorizzati **correttamente**, avrei dovuto fare una richiesta a un altra classe che avrebbe eseguito l'integrazione con altre sorgenti dati già presenti.

Il compito funzionale delle mie classi sotto analisi era di gestire il flusso degli aggiornamenti, la catena di passi necessari. Erano classi in cui gli elementi godevano di coesione di tipo:

- Funzionale, perché miravano a risolvere uno stesso problema.
- Sequenziale perché ogni *step* dipendeva del successo e usava l'output di quello precedente.
- Di comunicazione perché lavoravano sugli stessi dati.
- Procedurale e temporale perché dovevano essere eseguiti in un preciso ordine.

In altre parole, era giusto che tutto ciò fosse localizzato, in modo coeso, all'interno della stessa classe. Ancora una volta mi è rimasto evidente quanto sia importante analizzare ogni caso in modalità ad hoc, cercando di ottenere un equilibrio instabile tra concetti diversi che, applicati ingenuamente arrivano pure a contraddirsi. Non ci sono regole d'oro o operazioni che portino necessariamente a un buon *design*, per questo, infatti, parliamo di principi e concetti invece di "regole", o *code smells* invece di errori.

Ho concluso allora che il criterio suggerito da Sandi Metz per capire se ci sono più responsabilità in una stessa classe era un po' estremo, e che più spesso conviene analizzare caso a caso. La strategia di sintetizzare le responsabilità in una unica frase è comunque utile nel renderle evidenti.

Avevo allora giustificato l'unione di quelle componenti all'interno della stessa classe per il principio della coesione. Ma non ero ancora soddisfatto. C'era qualcosa che, ancora, non andava bene!

4.4 L'SRP con granularità diversa

La definizione del Single *Responsibilty Principle* più accettata parla esplicitamente di classe. Più precisamente è definita nei seguenti termini: "Una classe deve avere al massimo una ragione per cambiare"[1] *(Traduzione mia, originale in inglese*[9]*).*

Avendo capito meglio e anche nella pratica cosa effettivamente quest'affermazione significa, ho provato ad estendere la sua applicazione non solo con granularità di classe, ma anche a livello di singolo metodo/algoritmo.

Una volta che il file CSV con le informazioni testuali dei palazzi veniva letto, alla classe BuildingDataUpdater veniva delegato il lavoro di eseguire veramente l'aggiornamento, con la chiamata del metodo statico update_buildings, che riceveva appunto la lista di informazioni da processare e finalmente inserire nel database. Come detto in precedenza, l'algoritmo sintetizzato prevedeva tutto il flusso per l'aggiornamento: validazione, pulizia e richiesta dell'integrazione dei dati con le altre sorgenti già presenti in database (ad esempio le piante architettoniche). Tale algoritmo funzionava bene e godeva di una buona quantità di test di unità e di accettazione.

Il codice era funzionante ma era lungo e difficile da capire, e lo stesso capitava anche per il metodo update_rooms su RoomDataUpdater. Dovevano gestire più aspetti ad ogni singolo *step*: validare (e eventualmente segnalare all'utente inconsistenze), formattare i dati, memorizzarli, richiedere l'integrazione.

Analizzando la struttura di questi metodi e chiedendomi delle loro responsabilità, ho concluso che la causa del *code smell* non era effettivamente a livello di classe ma del metodo stesso. Mancavano basiche applicazioni dei principi di *Object Orientation*, estrazione di codici duplicati in metodi dedicati e una convenzione di *naming* delle variabili che ne favorisse la comprensione del codice.

Avevo deciso allora di metterli a posto, ma non volevo rischiare di ripetere l'errore precedente di frammentarli troppo. Per questo motivo, e anche per il fatto che si trattavano di implementazioni molto fragili, caratterizzate dalla costruzione di molti stati intermedi temporanei e da una logica molto procedurale, ho deciso di eseguire questa operazione di *refactoring* a piccoli passi, apoggiandomi sui test che avevamo già scritto per garantire che le modifiche riportate non danneggiassero il comportamento originale del metodo.

[9] "A class should have only one reason to change"

4.5 Conclusioni sull'esempio pratico di refactoring

Volevo evitare di innescare grossi contenuti di codice, ma sembra opportuno inserire la versione originale e la versione finale del metodo update_buildings per illustrare la differenza. Si noti che per questo discorso non è necessario capire il funzionamento o la logica della versione originale, ma più che altro compararne la leggibilità (e dimensione!) rispetto alla versione ripulita. Per brevità ho accorciato le stringhe di stampa, dato che sono spesso messaggi lunghi e pieni di dettagli all'utente finale, e per chiarezza ho aggiunti commenti (non presenti nell'originale) riga per riga.

Listing 4.1: Versione iniziale del metodo update_buildings

```
namespace = self.get_namespace() # get_namespace e' definito dalle sottoclassi
batch_date = datetime.now()      # Tutti palazzi aggiornati devono avere
                                 # il campo "updated_at" con lo stesso valore
for b in buildings:
  b_id = b.get("b_id", "")       # variabili temporanee, aggiungono stati intermedi
  l_b_id = b.get("l_b_id", "")

  # Step 1 - validazioni iniziali e segnalazione eventuali problemi
  if not Building.is_valid_bid(b_id):              # Validazione ID di palazzo
    if Building.is_valid_bid(l_b_id):              # Tentattivo di risoluzione errori
      Logger.warning("Legacy Id will be used ...") # Segnalazione: inconsist. risolta
      b["b_id"] = l_b_id                           # (messaggio truncato)
    else:
      Logger.error([String multiline omessa])      # Segnalazione: dati invalidi
      continue                                     # Questo palazzo non va aggiornato

  # Step 2 - Queste quattro righe trovano nel DB il documento da aggiornare o creano
  # un nuovo oggetto di tipo Building da venir inserito. Namespaced_attr si riferisce
  # alla parte del documento relativa alla sorgente attuale (easyroom o edilizia)
  building = self.find_building_to_update(b)
  namespaced_attr = building.get(namespace, {})
  building[namespace] = namespaced_attr
  namespaced_attr.update(b)             # aggiornamento dei dati di questa sorgente

  # Step 3 - primo passo della politica di snapshot (garantire che palazzi non
  # contemplati in questo aggiornamento vengano rimossi se necessario)
  deleted_key = "deleted_" + namespace
  if deleted_key in building:
    del building[deleted_key]

  # Step 4 - richiesta di integrazione delle sorgenti gia' presenti in DB
  # e salvataggio
  with Logger.info("Processing "+str(building)):
    building['merged'] = DataMerger.merge_building(
      building.get('edilizia'), building.get('easyroom'), building.get('dxf')
    )
    building['updated_at'] = batch_date          # Definizione della data dell'ultimo
    namespaced_attr["updated_at"] = batch_date   # Aggiornamento sul documento
    building.save()                              # Invio al Database dei dati aggior.

  # Step 5 - Secondo passo della politica di snapshot, rimuovere dal database
  # palazzi non piu' esistenti
  n_removed, b_removed = Building.remove_untouched_keys(namespace, batch_date)
  b_removed = [ b["b_id"] for b in b_removed ]

  if b_removed:
    Logger.info(n_removed, "previously existing buildings are not present...")

  n_destroyed, b_destroyed  = Building.remove_deleted_buildings()
  b_destroyed        = [ b["b_id"] for b in b_destroyed ]
  if n_destroyed:
    Logger.info(n_destroyed, "buildings were effectively removed...")
```

Non ci vuole molto per capire quanti problemi sussistono nel codice sopra, e per

questo motivo evito di commentarlo e passo subito alla visione della versione finale:

Listing 4.2: Versione finale del metodo update_buildings

```
self.batch_date = datetime.now()

for b_data in buildings:
  if not self._validate_building_data(b_data):      # Step 1 - Righe 5-15 dell'originale
    continue

  building = self.find_building_to_update(b_data)   # Step 2 - Righe 20-23
  self._mark_building_as_updated(building)          # Step 3 - Righe 26-28

  with Logger.info("Processing "+str(building)):
    self._update_a_building(building, b_data)       # Step 4 - Righe 31-37

self._destroy_unmarked_buildings()                  # Step 5 - Righe 41-51
```

Da notare le seguenti proprietà di questo *refactoring*:

- Il metodo update_buildings compie ora il ruolo di scheletro dell'esecuzione. Essendo piccolo, ci sta in una sola pagina di codice e permette con un unico sguardo di cogliere il flusso.
- L'esplicitare dello scheletro aiuta nel trovare eventuali problemi concettuali che dipendano dall'ordine delle operazioni.
- I nomi significativi dei metodi illustrano bene il loro ruolo funzionale all'interno del flusso.
- Ogni *step* coincide con una chiamata di metodo, che viene incapsulato e isolato. Cambiamenti della logica di lavoro, come ad esempio quella di validazione dei dati in input, da ora in poi avvengono in modo isolato, riducendo sostanzialmente il costo delle modifiche.
- Limita la profondità dell'indentazione (significativa in Python) che rendeva la comprensione del metodo più difficile, e i spingeva spesso ad abbreviare variabili con lo scopo di mantenere la lunghezza massima delle righe su 80 caratteri.
- Il rinominare della variabile di iterazione "b" a "b_data" rende più chiaro il suo scopo (sono i dati letti da inviare al database) e la differenzia dalla variabile building (che è un oggetto della classe Building, un ODM[10] Model che rappresenta i documenti in database da aggiornare/inserire.

L'incapsulamento all'interno di metodi piccoli e isolati dal resto della logica applicativa è particolarmente importante per i seguenti aspetti:

- Metodi più piccoli sono, in linea di principio, più facili da capire e allora è più probabile che eventuali errori emergano in fase di lettura di codice, e che l'attività di *debugging* diventi anche più semplice.
- Aumenta la *testability* dell'algoritmo complessivo, in quanto ci fornisce un ulteriore strategia per il *testing*: per verificare la correttezza del metodo principale possiamo procedere in modalità *divide and conquer*, testando i singoli *metodi* (*divide*) che lo compongono, e poi verificando che le chiamate interagiscano nel modo corretto (*conquer*).

[10] *Object-Document Mapper*

- Essendo più piccoli, possono lavorare su un sottoinsieme di informazioni e richiedere allora meno *setup* di strutture dati e contesti per l'esecuzione dei test. Di conseguenza su ogni singolo metodo, a parità di tempo e sforzo, vengono testati più casi e aspetti.

Per garantire che ogni aggiornamento di file CSV rappresentasse uno *snapshot*, ovvero che se un giorno un palazzo venisse rimosso dal "catalogo"[11] verrebbe cancellato anche dal nostro database, avevamo pensato a un algoritmo che procedeva in modalità *mark and sweep*: prima marcavamo tutti i palazzi che erano stati contemplati in questo *snapshot*, e, una volta finiti gli aggiornamenti di tutti i palazzi, cancellavamo quelli che non erano stati marcati. Si tratta allora di un algoritmo a due passi, in cui il primo va fatto all'interno del ciclo *for* di aggiornamento, e il secondo alla fine. L'estrazione di questi *step* in metodi dedicati rende anche questa correlazione esplicita dal nome conferito ai metodi: _mark_building_as_updated e _destroy_unmarked_buildings.

Queste modifiche favoriscono ulteriori refactoring per aumentare la leggibilità: i successivi cambiamenti possono essere limitati allo scopo del metodo stesso (ad esempio, se vogliamo dare un nome più significativo a una variabile, dovremo cambiarla soltanto all'interno del metodo - le modifiche sono contenute). Ciò ha reso i singoli metodi significativamente piccoli e semplici, a tal punto che mi permetto di omettere il loro codice sorgente, dato che non contribuiscono per questo discorso.

Ho finito l'attività di *refactoring* di BuildingDataUpdater, applicando altri principi sui singoli metodi, cercando più che altro una buona leggibilità del codice, e ho proceduto a un refactoring molto simile su RoomDataUpdater.

Aver applicato l'SRP più volte su più livelli, e avendo potuto poi contrastarlo con in concetto di *cohesion*, mi ha permesso di cogliere una significativa intuizione pratica da riapplicare in altri contesti, e la dinamica SRP / *Cohesion* mi ha in seguito aiutato a riportare ulteriori *refactoring* di tipo strutturale su altre classi, senza però compiere l'errore commesso nella prima applicazione.

[11] Può sembrare strano che un palazzo venga rimosso, ma ce ne sono casi in cui succede, come ad esempio palazzi che non sono di proprietà dell'università ma vengono rintracciati e allora possono in futuro non più venir utilizzati. Dovevamo inoltre gestire eventuali errori, come ad esempio inserimenti duplicati con codici identificativi sbagliati, casi in cui bisogna rimuovere il duplicato o sbagliato.

Capitolo 5

Refactoring continuo e guidato

Analizzando queste esperienze, ho concluso che l'aver utilizzato uno o più concetti esplicitamente per l'analisi del codice mi ha permesso di cambiare l'approccio, lo sguardo, e mi ha reso capace di eseguire modifiche più significative.

Da ciò segue l'importanza di avere un approccio sistematico al *refactoring*, e che studiare le sue modalità e casi di applicazione è una forte componente dell'esperienza di uno sviluppatore. Inoltre, partendo dal *refactoring* sentivo che le mie capacità di *design* miglioravano. Ad ogni classe rielaborata mi sentivo leggermente più capace di sentire i *code smells* ma anche di capire che cosa li generava.

Essendomi particolarmente interessato all'argomento, ho trovato un libro che mi ha cambiato il modo di vedere il problema già dalla prefazione.

Si trattava del libro "Refactoring: Improving the Design of Existing Code"[2], di Martin Fowler, sviluppatore, autore e *public speaker* per cui provo una sincera ammirazione, specialmente per il suo modo teorico-pratico di affrontare il nostro mestiere.

Già alla *Foreword* scritta da Erich Gamma[1] sentivo di aver trovato un punto di riferimento, e mi sono stupito di vedere che molte delle conclusioni a cui ero arrivato dopo duri processi di tentativi, errori e riflessioni, venivano formalizzate e trattate in modo chiaro ed esplicito.

[1] Erich Gamma (nato nel 1961 a Zurigo) è un ingegnere del software e coautore dell'influente libro di Software Engineering "Design Patterns: Elements of Reusable Object-Oriented Software" e ha dato numerosi contributi teorici e pratici (ad esempio ha creato il framework per il testing JUnit insieme a Kent Beck) sull'argomento.

Ho trovato una definizione più precisa di refactoring, e, più importante, ricorrenti incentivi alla sua **pratica continua**, a prescindere degli ostacoli naturali (management, l'impressione che non porti a niente di nuovo, molto sforzo per nessun valore percepibile dall'esterno, eccetera). Un idea su cui avevo appena riflesso ma che Martin Fowler ha molto bene sintetizzato è quello descritto nel seguente paragrafo:

> Senza il refactoring, il design del programma decade. A misura che le persone cambiano il codice - modifiche per realizzare obiettivi immediati o modifiche fatte senza una comprensione complessiva del design del codice - il codice perde la sua struttura. Diventa più difficile capire il design leggendo il codice. [...] La perdita di struttura del codice ha un effetto cumulativo. Quanto più difficile è vedere il design del codice, più difficile è preservarlo, e più rapidamente esso decade. *Refactoring* regolare aiuta il codice a mantenere la sua forma. [2]
>
> *(Traduzione mia, originale in inglese[2])*

Riflettendo su queste idee, mi sono accorto del (primo) banale errore che avevo commesso, e mi sentivo ingenuo per quello: cercando sempre la "buona progettazione", abbiamo applicato principi e ragionamenti all'inserimento di nuove funzionalità, moduli e classi. Basandoci su estensiva sperimentazione, giudico che avevamo progettato bene ogni modulo / sistema prima della implementazione, e ci eravamo preoccupati di rielaborare ognuno di loro per un po' di tempo.

Però, anche se inizialmente concepiti con particolare cura della chiarezza, questi metodi venivano modificati con piccole correzioni e estensioni di comportamento. In questo naturale ciclo di vita di rielaborazione, trasformazioni, il design iniziale, complessivo, si perdeva, si appannava. Applicando l'attenzione per un buon design alle nuove funzionalità, ci siamo dimenticati di riadeguare quelle già esistenti!

Il secondo errore commesso è stato quello di frammentare eccessivamente la classe BuildingDataUpdater durante il suo refactoring. Mi ero messo l'obbiettivo di "rimuovere di una volta per tutte" i *code smells* su quella classe, e allora mi sono messo a

[2] "Without refactoring, the design of the program will decay. As people change code - changes to realize short-term goals or changes made without a full comprehension of the design of the code - the code loses its structure. It become harder to understand the design by reading the code. [...] Loss of structure of code has a cumulative effect. The harder it is to see the design of the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps code retain its shape."

fare grossi cambiamenti. La verità è che l'unico grosso cambiamento strutturale che era veramente necessario era quello di separare DataUpdater in due classi distinte per l'aggiornamento delle stanze e dei palazzi, e poi eseguire operazioni più piccole di refactoring più localizzato: la maggior parte erano estrazioni di metodi.

A rispetto della dimensione dei passi in fase di refactoring, Fowler, in riferimento a un esempio presentato, è abbastanza chiaro:

> La lezione più importante di questo esempio è il ritmo del refactoring: testare, piccola modifica, testare, piccola modifica, testare. È questo ritmo che permette al refactoring di procedere velocemente e con sicurezza[2]
>
> *(Traduzione mia, originale in inglese[3])*

Quando parla di sicurezza, Fowler si riferisce all'importanza dei test, nel garantire che le modifiche apportate non generino nuovi errori su un codice che prima era funzionante. Da questa esperienza che vi ho raccontato aggiungerei che il procedere a piccoli passi ci conferisce anche un ulteriore livello di sicurezza: sulle modifiche nel design. A piccoli passi le trasformazioni avvengono in modo incrementale, e allora esiste una maggior probabilità che ce ne accorgiamo nel momento giusto se le modifiche ci hanno già portato al punto desiderato. Non posso dire che non avrei commesso quell'errore se avessi avanzato a piccoli passi, ma di sicuro ci sarebbe una maggiore probabilità che non dovessi arrivare alla mattina successiva per accorgermi.

Si noti inoltre il rilievo dato da Fowler all'idea di *Refactoring Regolare*. Su questo aspetto ne discute ancora:

> In quasi tutti i casi, mi oppongo all'idea di separare tempo da dedicare al refactoring. Dal mio punto di vista refactoring non è un'attività a cui si dedica un tempo preciso. Refactoring è qualcosa che si fa tutto il tempo in piccole raffiche. Non si decide di fare il refactoring, ma se lo fa perché si vuole fare qualcos'altro, e fare il refactoring ci aiuta a fare quella cosa[2].

[3] "The most important lesson from this example is the rhythm of refactoring: test, small change, test, small change, test, small change. It is that rhythm that allows refactoring to move quickly and safely"

(Traduzione mia, originale in inglese[4])

Assumendo allora che questo approccio di **continuous refactoring** venga portato in avanti sin dall'inizio di un progetto, sono assolutamente d'accordo con tale affermazione. Con questo sforzo continuo e inerente all'attività di implementare nuove funzionalità o estendere quelle esistenti, garantiamo che ad ogni passo il *design* del codice non solo viene mantenuto come migliorato.

Nella mia opinione esiste però un caso in cui separare del tempo preciso per fare il refactoring è necessario. Si tratta del caso specifico che ho esperimentato: quando se ne accorge un po' tardi che l'effetto cumulativo della perdita di struttura del codice ha già causato troppi danni. A quel punto bisogna fermarsi, separare del tempo ed eseguire l'attività di refactoring a piena autonomia.

Vorrei sottolineare che c'è una netta differenza tra le due modalità di refactoring. Nel primo caso, quando si esegue il refactoring come parte del processo di sviluppo stesso, si parla di refactoring per adeguare il codice esistente a delle richieste che dobbiamo implementare di immediato, e allora ne conosciamo già. Saranno appunto queste richieste a guidare i passi del refactoring: in che modo rielaborare il codice già esistente per far si che l'aggiunta/estensione delle richieste funzionalità porti a ugualmente o più elegante design? Su questa tipologia di refactoring, Martin Fowler fornisce un lungo compendio di esempi pratici e di diverse tecniche nel riferito libro[**?**].

L'altro caso, cioè, quando decidiamo che lo stato attuale del codice richiede un impegno dedicato di refactoring, non abbiamo richieste esplicite da guidare le nostre trasformazioni. Le richieste invece tendono ad essere piuttosto generali, come "migliorare la leggibilità del codice", "ridurre le dipendenze", "disaccoppiare moduli", "aumentare la testabilità", eccettera.

In questi casi diventa problematico, specialmente in visione di un *codebase* significativo, decidere da dove partire e scegliere quali modifiche porteranno al maggior beneficio, dato che comunque nei contesti pratici il tempo disponibile a un'attività del genere è anche limitato.

[4] "In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing"

Nella mia particolare esperienza, i criteri utilizzati per guidare la scelta di quali refactoring eseguire e su quali classi/moduli farlo, sono stati i principi SOLID di cui ho prima parlato, con prevalenza dell'SRP. L'SRP è stato il più utile perché molto spesso ho percepito che operare modifiche sul codice in termini di riduzione delle responsabilità di ogni classe rendeva più evidente problemi legati agli altri aspetti, come addirittura il Liskov Substitution Principle. Definire bene le responsabilità di una classe padre e delle sue classi figlio esplicitava la natura della relazione che doveva sussistere fra di loro, e, di conseguenza, quali logiche funzionali violavano l'LSP.

Nel mio caso sento che questa strategia abbia funzionato bene, aiutandomi nella scelta delle modifiche da operare e bilanciando in parte la difficoltà di capire i problemi di design dovuta alla mancanza di un'esperienza più SOLIDa (non per caso in maiuscole).

Capitolo 6

Conclusioni

Sono arrivato alla conclusione

Penso di aver colto aspetti importanti da considerare durante un *refactoring* guidato, e allora di aver sanato parzialmente la sensazione di "mancanza" e immobilità che avevo guardando i codici precedenti. Questo è stato possibile soltanto per il fatto che sono partito da riflessioni proprie, da uno sforzo lungo per capire i difetti di quel design, da tentativi pratici senza, appunto, la presenza di qualcuno che mi guidasse. In seguito a tanti tentativi falliti o con successo soltanto parziale, e allora significativa riflessione pratica, la letteratura e la teoria sull'argomento mi hanno aiutato a portare il ragionamento in avanti, a fare i salti di qualità necessari.

La riflessione sulla letteratura mi ha anche aiutato a capire il motivo per cui inizialmente avevo sbagliato nei tentativi di refactoring, frammentando troppo il flusso di esecuzione del codice in diversi moduli/classi, cioè la necessità di procedere a piccoli passi, riflettendo su ogni cambiamento una volta compiuto e assicurandoci della sua correttezza con l'esecuzione dell'insieme di test dedicati.

Una volta che sono riuscito a compiere questi passi, ho cercato di riapplicare il metodo utilizzando altri concetti. Mi rimane comunque la certezza che mantenere un buon *design* del software è un'abilità che viene acquisita con esperienza, e con uno sforzo continuo. Non basta un ottimo design a priori, perché questo viene fatalmente eroso man mano che riceve modifiche e nuove funzionalità. Non basta neanche la voglia di mantenerlo, ci vuole anche un *know-how* la cui assimilazione non è direttamente proporzionale al tempo o allo sforzo impiegato, ma procede in modi irregolari.

Ciò che abbiamo fatto come team, in seguito a questi episodi, è stato appunto dare dignità di processo all'attività di refactoring, inserendola all'interno del nostro flusso di lavoro. Questo progetto ha servito come applicazione pratica di tanti concetti che avevamo già studiato durante il nostro percorso di studi, e, nel mio caso particolare, penso che ne abbia contribuito significativamente per farmi assorbire in modo più profondo qualche dei principi di Progettazione Software.

In letteratura ho trovato il riscontro che sentivo di aver bisogno per poter sviluppare le mie proprie idee e capacità. Ciò rinforza a me l'idea di aver scelto una professione in cui non si deve mai smettere di studiare. Ma quello lo sapevamo già.

Bibliografia

[1] Martin, Robert C.. Agile Software Development, Principles, Patterns, and Practices, 2003.

[2] Fowler, Martin. Refactoring: Improving the Design of Existing Code, 2002.

[3] Metz, Sandi. Practical object-oriented design in Ruby : an agile primer, 2013.

[4] 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.