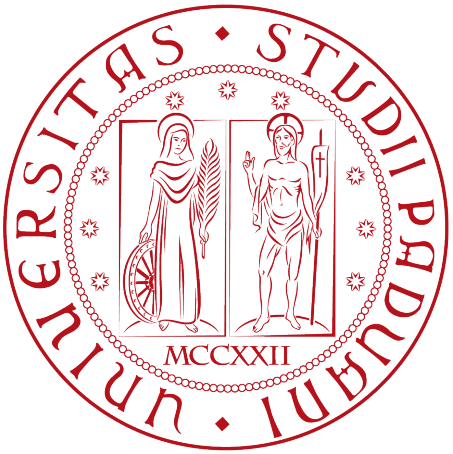
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Java Content Repository per la persistenza di prodotti commerciali

Tesi di laurea triennale

Relatore

Prof. Tullio Vardanega

Laureando

Jordan Gottardo

Anno Accademico 2016-2017

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dal laureando Jordan Gottardo presso l'azienda IBC S.r.l. di Peraga (PD).

Gli obiettivi principali da raggiungere erano due. In primo luogo, veniva richiesto uno studio degli *standard* JSR 170 e JSR 283, che descrivono le API per l'utilizzo di Java Content Repository (JCR). Era richiesta la produzione di documentazione ed esempi di codice sorgente che sfruttassero tali API.

Il secondo obiettivo riguardava l'implementazione di un prototipo, sottoforma di *web app*, che permettesse la memorizzazione di prodotti commerciali aventi attributi variabili. Era richiesta inoltre l'implementazione di funzionalità di ricerca per effettuare selezioni mirate di prodotti in più passi.

La libreria da utilizzare per la persistenza delle informazioni era Apache Jackrabbit, mentre il *framework* per la realizzazione dell'interfaccia grafica era di libera scelta.

Il presente documento è organizzato in quattro capitoli:

* **L'azienda:** in questo capitolo presento l'azienda che ha ospitato lo *stage*, IBC S.r.l., fornendo descrizioni del contesto aziendale e del modo di lavorare. Descrivo inoltre i prodotti e i servizi che essa offre sul mercato.

* **L'offerta di *stage*:** all'interno di questo capitolo descrivo il progetto di *stage* offerto, soffermandomi sulle motivazioni aziendali e personali che hanno portato a questa scelta. Elencherò inoltre gli obiettivi da raggiungere.

* **Svolgimento del progetto:** in questo capitolo presento le attività svolte durante lo *stage* per il raggiungimento degli obiettivi prefissati.

* **Analisi retrospettiva:** all'interno di questo capitolo fornisco un'analisi retrospettiva sugli obiettivi dello *stage*. Fornisco inoltre una descrizione di alcune mancanze nell'insegnamento accademico che dovrebbero essere aggiunte al piano didattico per un efficace approdo nel mondo del lavoro.

Nel documento utilizzerò le seguenti notazioni tipografiche:

* *Italico*: termine in lingua straniera.

* `Monospace`: nome di *file*, classe o codice sorgente.

* Termine in azzurro: termine a glossario, solo la per la prima occorrenza di ogni capitolo. Cliccando sul termine è possibile leggere la spiegazione.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Tullio Vardanega, relatore della mia tesi, per l'aiuto fornitomi durante la stesura del documento.

Ringrazio IBC, in particolare Denis Corà e Stefano Gesuato, per il supporto durante il periodo di stage.

Desidero infine ringraziare la mia famiglia per il sostegno che mi ha fornito in questi anni e Giulia, Giovanni P. e Giovanni D. per il lavoro svolto durante la realizzazione del progetto di ingegneria del software.

Padova, Settembre 2017 Jordan Gottardo

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Tools and applications

In this chapter I will describe the tools and softwares I have utilized to carry out the simulations.

1.1 Network Simulator 3

Network Simulator 3 (ns-3)is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. ns-3 is free software, licensed under the GNU GPLv2 license, and is publicly available for research, development, and use.

ns-3 development began in 2006 by a team lead by Tom Henderson, George Riley, Sally Floyd and Sumit Roy. Its first version was released on June 30, 2008.

ns-3 is the successor of ns-2, released in 1989. The fact that the former was built from scratch makes it impossible to have backward compatibility. In fact, ns-2 used oTCL scripting language to describe network topologies and C++ to write the core of the simulation. This choice was due to avoid the very time consuming C++ code recompilation, exploiting the interpreted language oTCL. ns-2 mixed the "fast to run, slow to change" C++ with the "slow to run, fast to change" oTCL “aaaa”

Capitolo 2

L'offerta di *stage*

2.1 *Stage* in IBC: motivazioni aziendali

IBC è un'azienda che in passato ha già offerto rapporti di *stage*, anche se questo è il primo anno che essa partecipa a StageIt. Per non essere una perdita di tempo e risorse aziendali, gli *stage* in IBC devono avere obiettivi e motivazioni che portano vantaggi anche a quest'ultima, oltre che allo stagista.

* Il primo obiettivo che l'azienda tenta di raggiungere è lo studio di nuove tecnologie per andare ad espandere (e in alcuni casi sostituire) gli strumenti utilizzati. I dipendenti di IBC infatti sono quasi sempre impegnati in progetti con scadenze stringenti, il che rende molto difficile dedicare risorse alla scoperta e all'apprendimento di nuove tecnologie. Questi compiti sarebbero solitamente svolti dal reparto ricerca e sviluppo, assente in IBC. Le attività svolte durante lo *stage* coprono quindi in parte questa mancanza, permettendo all'azienda di ottenere informazioni su nuovi strumenti e *proof of concept* di prodotti di futura realizzazione.

* Il secondo obiettivo riguarda la prospettiva di assunzione di nuovo personale. L'azienda infatti tratta lo *stage* come periodo di prova pre-assunzione, in modo da poter verificare le capacità dello stagista e fargli apprendere i meccanismi aziendali. IBC, al momento dell'offerta, era alla ricerca di programmatori Java e ha presentato una proposta proprio in quell'ambito. Assumere il tirocinante nello stesso ambito alla fine dello *stage* fa risparmiare all'azienda tempo e risorse rispetto a un ulteriore addestramento in un'altra area.

* La terza e ultima motivazione è il vantaggio che IBC può ottenere da una mente giovane e creativa, come ad esempio quella di uno studente che sta per concludere una laurea triennale in Informatica. Al contrario del personale abituato da anni a portare a termine gli stessi compiti nella stessa maniera, uno studente è più propenso ad inventare soluzioni originali e talvolta non convenzionali. Non sempre è detto che tali soluzioni siano adottabili e manutenibili, quindi è necessaria una revisione da parte di personale esperto prima che l'azienda adotti le proposte dello stagista.

2.2 Il progetto

2.2.1 Dominio applicativo

Il progetto di *stage* riguardava la memorizzazione di informazioni di prodotti commerciali. Il continuo rapporto con clienti in ambito *retail* da parte di IBC porta alla necessità di avere una persistenza delle informazioni dei prodotti che essi offrono sul mercato.

Figura 2.1: Informazioni di un prodotto (goo.gl/pxeYU1).

Questi dati sono utilizzati in molti ambiti, come ad esempio:

* La stampa dell'etichetta da apporre sulla confezione di un prodotto.
* La categorizzazione di tipologie di prodotto sugli scaffali di un supermercato o sul sito di un *e-commerce*.
* L'identificazione e la ricerca di prodotti con particolari caratteristiche.
* La gestione di magazzino.
* L'identificazione di allergeni o particolari agenti chimici.
* La raccolta di dati statistici e la generazione di reportistica.

Dati i molti utilizzi, la memorizzazione delle informazioni è una necessità che negli anni ha avuto varie soluzioni e implementazioni.

Attualmente IBC utilizza un *database* relazionale per la persistenza dei dati. Questo porta al problema principale che il progetto offerto deve risolvere. Data la natura stessa del modello relazionale, è necessario definire una struttura prima di poter memorizzare un qualsiasi elemento. Gli attributi dei prodotti però sono variabili e non è raro trovare due prodotti appartenenti alla stessa categoria con qualche attributo non in comune.

Un altro fattore da considerare è il fattore umano: dato che le informazioni dei prodotti vengono spesso fornite a IBC da personale non tecnico, capita a volte che gli attributi non rispettino i limiti imposti dalla struttura relazionale, portando all'impiego di risorse per riprogettare la struttura o tradurre gli attributi in un modello valido.

Tenendo conto dei due punti appena esposti, l'attuale soluzione adottata dall'azienda prevede la definizione di una struttura che contempli tutti i possibili attributi di ogni

categoria di prodotto. Le particolari istanze di ogni prodotto che non riportano un qualche attributo avranno il valore di quest'ultimo impostato a `null`.

Questa soluzione è però poco logica ed è imposta dal modello relazionale. Un *Content Repository* fornisce un'alternativa senza lo svantaggio appena esposto.

IBC era quindi alla ricerca di una soluzione flessibile, che permettesse l'aggiunta di prodotti aventi proprietà variabili, utilizzando il modello JCR. Il *tutor* ha affermato che l'obiettivo del prototipo da realizzare era verificare se fosse possibile implementare una soluzione utilizzando la libreria Jackrabbit. Anche in base ai risultati da me ottenuti, l'azienda deciderà in futuro se intraprendere un progetto su più ampia scala per la realizzazione di un *software* utilizzando questa tecnologia.

2.2.2 Introduzione a JCR

Date le mie scarse conoscenze del dominio del progetto, ho impiegato i primi giorni lavorativi per effettuare uno studio preliminare. Ho consultato principalmente risorse presenti *online*, tra cui:

* *Paper* "JCR or RDBMS: why, when, how?", per comprendere le differenze tra Relational DataBase Management System (d'ora in poi RDBMS) e Java Content Repository (d'ora in poi JCR).
* Articolo "What is Java Content Repository" (`https://goo.gl/8HWDRZ`), per avere una descrizione di base di JCR.
* Standard JSR 170 e JSR 283.

I risultati delle mie ricerche sono contenuti all'interno di due documenti che ho prodotto per IBC:

* **Confronto tra RDBMS e JCR:** documento che racconta la storia di JCR e analizza le differenze tra il modello relazionale e il modello JCR.
* **Struttura JCR e funzionamento Jackrabbit:** documento che, a partire dagli *standard* precedentemente citati, fornisce una spiegazione delle principali API per l'accesso a JCR. Inoltre, date le funzionalità aggiuntive fornite da Jackrabbit, nella seconda parte del documento descrivo la struttura di Jackrabbit e il suo funzionamento nel dettaglio.

Un *Content Repository* è un modello utilizzato per la memorizzazione di qualsiasi tipo di dato. Gli *standard* JSR 170 e JSR 283 definiscono le API per JCR.

Le differenze tra il modello relazionale e il JCR possono essere suddivise in varie aree, come analizzato nel seguente *paper*: `https://goo.gl/ngzgKt`.

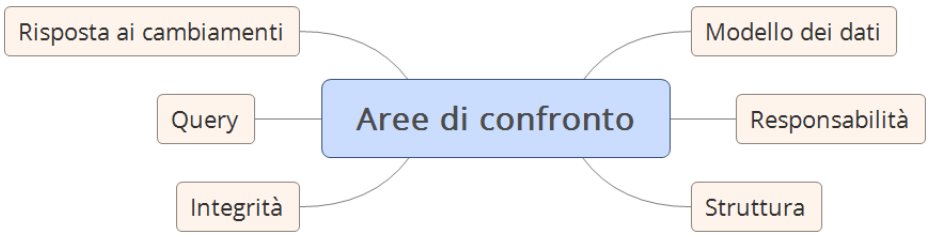


Figura 2.2: Aree di confronto tra RDBMS e JCR.

1. **Modello dei dati**
   Con "modello dei dati" intendiamo il modo con cui i dati vengono organizzati, acceduti e messi in relazione tra di loro.

   RDBMS Il modello relazionale si basa sulla teoria degli insiemi e sulla definizione matematica di relazione, che ricordiamo essere un sottoinsieme del prodotto cartesiano tra n insiemi. Dato che ognuno di questi dev'essere distinguibile dagli altri, ogni insieme è definito come dominio. Ad esempio, facendo riferimento alla tabella sottostante, i domini sono quello dei nomi (N), cognomi (C) ed età (E).

Nome (N)	Cognome (C)	Età (E)
Mario	Rossi	30
Giovanna	Bianchi	25
Enrico	Neri	40

Figura 2.3: Tabella che rappresenta una persona (https://goo.gl/ngzgKt).

   La definizione di relazione non implica la possibilità di creare associazioni tra le relazioni. Per fare questo, è necessario utilizzare l'algebra relazionale.

   JCR Il modello JCR si basa principalmente su una struttura ad albero, unendo le caratteristiche dei modelli gerarchici a quelle dei modelli a rete. Il risultato è una struttura ad albero che permette la connessione dei nodi orizzontalmente.

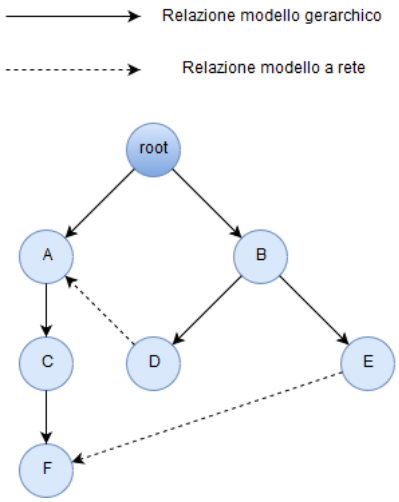


Figura 2.4: Unione del modello a rete con il modello gerarchico (https://goo.gl/ngzgKt).

2. **Responsabilità**
   Quando si tratta di *database* e persistenza dei dati, generalmente possono essere identificati tre ruoli principali:

   * Il ***database administrator* (DBA)**, che mantiene il *database* in uno stato utilizzabile eseguendo attività di installazione, configurazione, *backup* e *data recovery*.

* L'***application programmer***, che scrive *software* che accede al *database*.
* L'**utente**, che utilizza il *software* per leggere, scrivere e modificare i dati nel *database*.

I due modelli si differenziano anche sotto il punto di vista dei ruoli. Più precisamente, cambiano le responsabilità e i campi di interesse di ogni ruolo.

I campi di interesse presi in esame sono:

* **Contenuto**: tutti i dati inclusi nel *database*.
* **Struttura**: il modo con cui i dati sono suddivisi.
* **Integrità**: lo stato di completezza dei dati.
* **Coerenza**: la relazione ordinata, logica e consistente delle parti.

RDBMS Nel modello relazionale generalmente è il DBA ad avere il controllo sulla struttura. L'*application programmer* solitamente ha un qualche tipo di influenza sulle decisioni prese in questo campo, ma la decisione finale spetta al DBA. L'utente non ha alcuna responsabilità per quanto riguarda la struttura e può solo interagire con il *database* tramite le operazioni fornite dal *software*.

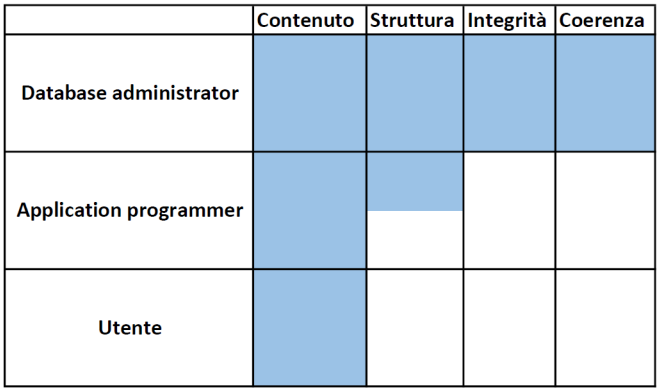


Figura 2.5: Responsabilità dei ruoli in RDBMS (https://goo.gl/ngzgKt).

JCR Nel modello JCR invece la struttura è responsabilità di tutti e tre i ruoli. Infatti, il controllo sulla struttura è più incentrato verso l'*application programmer* e l'utente, riducendo di fatto le responsabilità del DBA in questo campo.

Uno dei vantaggi principali di questo approccio è che solitamente il ruolo di *application programmer* è più vicino all'utente finale rispetto al DBA, quindi una collaborazione tra questi due ruoli per la definizione della struttura è solitamente più efficace.

È anche possibile costruire *software* che permettono al solo utente finale di definire la struttura, aggiungendo attributi ai dati a tempo di esecuzione, sottostando ai vincoli definiti dal DBA e dall'*application programmer*.

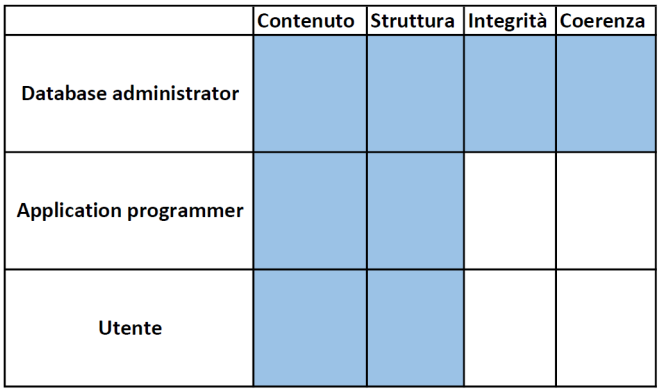


Figura 2.6: Responsabilità dei ruoli in JCR (https://goo.gl/ngzgKt).

3. **Struttura**
   Con "struttura" intendiamo il modo con cui i dati sono suddivisi e a quali costrizioni essi sono sottoposti.

   Le differenze in termini di struttura rendono i due modelli diametralmente opposti, con vantaggi e svantaggi in entrambi gli approcci.

   RDBMS Nel modello RDBMS, i dati sono guidati dalla struttura. Un dato, per essere istanziato, ha bisogno che la struttura sia completamente definita. Questo modello si basa sull'assunzione che dati e struttura siano sempre completamente separati e indipendenti, ma nella realtà quest'assunzione non è sempre valida. Come esposto precedentemente, ci sono casi d'uso in cui la struttura del dato cambia nel tempo, portando all'aggiunta di nuovi campi o ad un'intera riprogettazione nei casi più sfortunati.

   JCR In JCR non è richiesta la definizione di alcuna struttura per istanziare i dati. Nodi, attributi e valori possono essere creati senza nessun prerequisito. Infatti, la struttura emerge con l'inserimento dei dati. Con il modello JCR non è più necessario definire tutti i possibili attributi al momento della creazione di un tipo di dato, garantendo una maggiore flessibilità ed estendibilità.

4. **Integrità**
   L'integrità di un *database* indica l'impossibilità di distruzioni e alterazioni dei dati, siano esse accidentali o intenzionali. Questa caratteristica è implementata in diversi modi a seconda del modello.

   RDBMS Il modello relazionale adotta una strategia simile ad una *white list*, ovvero i dati possono essere salvati solo se è definita una struttura. È quindi quest'ultima che garantisce buona parte dell'integrità, ad esempio attraverso i vincoli di dominio.

   JCR In opposizione al modello RDBMS, JCR si basa su un approccio a *black list*. Un nodo generico del *Content Repository* può avere qualunque nodo figlio e qualsiasi proprietà, senza vincoli su tipi e valori.

Eventuali vincoli possono essere imposti assegnando ai nodi dei tipi. Un tipo di nodo descrive vincoli sul tipo dei nodi figli o sui valori delle proprietà che il nodo stesso può avere. Assegnando un tipo anche ai nodi figli e continuando a procedere in questo modo è possibile imporre sempre più limiti alla struttura.

5. ***Query***
   Anche il tipo e la potenza delle *query* differenzia i due modelli.

   RDBMS Data la definizione di relazione, il modello RDBMS si basa sull'algebra relazionale per la definizioni delle operazioni di base. Il vantaggio di questo modello è che sia l'*input* che l'*output* delle operazioni sono relazioni. È quindi possibile concatenare espressioni complesse senza troppe difficoltà. Inoltre, la maggior parte dei linguaggi di *query* fornisce anche la possibilità di effettuare cambiamenti sequenziali al risultato di una *query*.

   JCR Nel JCR è necessario utilizzare un modello di *query* astratto per effettuare operazioni. Questo modello astratto serve a mappare il modello JCR con le nozioni di relazione, domini, tuple e attributi tipiche del modello relazionale.

   Uno dei principali svantaggi è che, con l'implementazione JCR di *default*, non è possibile effettuare cambiamenti sequenziali con una *query*.

   Nel complesso, JCR offre un supporto più limitato rispetto al modello relazionale per quanto riguarda le *query*, ma ha vantaggi prestazionali nell'esecuzione di ricerche *full-text*.

6. **Risposta ai cambiamenti**
   Nonostante un'analisi dei requisiti svolta in maniera impeccabile, è possibile che nuovi requisiti emergano dopo che l'architettura di un sistema è già stata definita. Un modello di sviluppo non strettamente sequenziale, come ad esempio quello incrementale, permette solitamente di soddisfare i nuovi requisiti senza dover riprogettare interamente il sistema. Tuttavia, un impatto a livello di architettura è spesso inevitabile e comporta dei costi. Per diminuire questi costi, è preferibile adottare un modello dei dati che riesca ad accettare i cambiamenti in maniera trasparente.

   RDBMS Nel modello relazionale, quasi tutti i cambiamenti architetturali richiedono un cambiamento a livello di logica dei dati. Questo modello non è quindi molto adatto a casi in cui sono necessari molti cambiamenti.

   JCR Dato che un'architettura basata sul modello JCR è molto lasca, è possibile aggiungere nuovi campi dati (e quindi soddisfare i requisiti che lo richiedono) senza modificare il livello di logica dei dati. Con JCR si ha quindi un disaccoppiamento molto forte tra dati e logica dell'applicazione. Data questa caratteristica, l'aggiunta di eventuali campi dati impatterà solo il livello di logica dell'applicazione e di interfaccia. Alcuni *framework* si occupano di un ulteriore disaccoppiamento tra logica e interfaccia operando in maniera simile. Questa combinazione genera un sistema che risponde ai cambiamenti in modo estremamente dinamico e con costi contenuti.

2.2.3 Obiettivi

Dopo vari incontri con il *tutor* aziendale, abbiamo definito gli obiettivi da raggiungere, suddividendoli in obbligatori, desiderabili e facoltativi.

A grandi linee, nelle trecentoventi ore previste dallo *stage* l'azienda si aspettava:

* Uno studio e la produzione di documentazione sulle differenze tra *database* relazionale e *Content Repository*.
* Uno studio e la produzione di documentazione sugli *standard* JSR 170 e JSR 283, rispettivamente Java Content Repository 1.0 e 2.0.
* La produzione di esempi di codice sorgente riguardanti l'utilizzo della libreria Jackrabbit.
* La realizzazione di un prototipo che permettesse operazioni di aggiunta, visualizzazione, modifica e rimozione di prodotti commerciali e dei loro attributi

Con il *tutor* abbiamo discusso anche dell'eventuale possibilità dello studio e dell'implementazione di soluzioni distribuite, ma dato il tempo limitato a disposizione e la corposità delle librerie da apprendere abbiamo deciso di non inserire questa richiesta negli obiettivi.

A seguire includo una lista dettagliata degli obiettivi suddivisi per importanza.

Obiettivi obbligatori
Studio e documentazione sulle differenze tra *database* relazionale e *Content Repository*
Studio e documentazione sulla storia di *Content Repository*
Studio di JSR 170 e JSR283: Content Repository for Java (JCR), con produzione di codice e documentazione
Studio e documentazione della struttura di JCR
Studio e documentazione della definizione di nodo
Studio e documentazione riguardo aggiunta, rimozione e modifica di proprietà di un nodo
Studio e documentazione riguardo l'aggiunta e la rimozione di tipologie di nodo
Studio e documentazione riguardo la referenziazione di elementi
Studio e documentazione riguardo l'esecuzione di *query* utilizzando XPath e JCR-SQL2
Studio e documentazione riguardo l'indicizzazione
Progettazione di un prototipo di applicazione che gestisca le informazioni di prodotti commerciali
Realizzazione di un prototipo di applicazione che gestisca le informazioni di prodotti commerciali
Obiettivi desiderabili
Realizzazione della GUI del prototipo
Obiettivi facoltativi
Studio e documentazione riguardo i *workspace* multipli

Tabella 2.2: Obiettivi del progetto

2.2.4 Vincoli

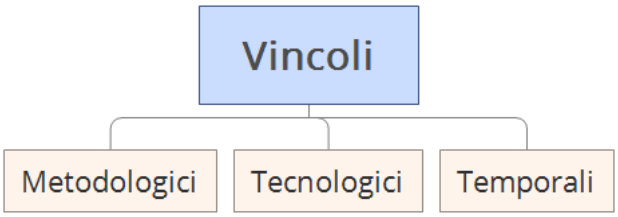


Figura 2.7: Vincoli del progetto

Metodologici

La prima tipologia di vincoli a cui il progetto era sottoposto erano i vincoli metodologici.

Con il *tutor* aziendale abbiamo stabilito che il lavoro doveva essere svolto presso la sede aziendale, per avere miglior approccio e comunicazione con il *tutor* stesso e gli altri colleghi. L'azienda ha posto questo vincolo anche per cercare raggiungere l'obiettivo di prospettiva di assunzione descritto nella sezione 2.1.

Un altro vincolo stabilito riguardava l'interazione con il *tutor* e la richiesta di informazioni tecniche ai colleghi. Dati i frequenti impegni del *tutor*, nel caso di necessità di informazioni tecniche avrei dovuto chiedere ai colleghi d'ufficio facenti parte del *team* Java 3, senza però abusare di tale possibilità. Uno degli obiettivi che l'azienda ha cercato di raggiungere con questo vincolo è quello di migliorare le mie capacità di *problem solving* e di lavoro in autonomia, insegnandomi a riconoscere i problemi risolvibili da me e quelli che invece necessitano di personale più esperto. I rapporti con il *tutor* si sarebbero dovuti limitare a richieste riguardo i requisiti e a revisioni periodiche per valutare i risultati raggiunti.

Tecnologici

Gli unici vincoli tecnologici imposti dall'azienda riguardavano l'implementazione di esempi di codice e di un prototipo basato sulla libreria Jackrabbit, utilizzando quindi il linguaggio Java.

Per quanto riguarda il versionamento, IBC ha predisposto un *repository* SVN su cui avrei dovuto effettuare i *commit* di codice e documentazione.

Non abbiamo fissato vincoli stretti riguardo la gestione della configurazione, anche se il *tutor* mi ha fortemente consigliato di utilizzare Maven, data l'esperienza positiva che l'azienda ha avuto con tale strumento.

La scelta di eventuali *framework* per l'implementazione dell'interfaccia grafica del prototipo era libera, a patto che fosse possibile l'interfacciamento con il JCR offerto da Jackrabbit. Questo mi ha portato a dover effettuare una scelta:

* La prima opzione che ho considerato è stato l'utilizzo del linguaggio PHP, da me già conosciuto. Ho presto realizzato che per percorrere questa strada avrei dovuto utilizzare la libreria Jackalope, che fornisce un'implementazione di JCR accessibile tramite API PHP. Nonostante la presenza di Jackalope-Jackrabbit, un'implementazione basata sul JCR fornito da Jackrabbit, ho trovato questa soluzione troppo complicata e scarsamente documentata.

* La seconda opzione che ho considerato è stato JavaServer Faces (JSF), un *framework* Java basato sul *design pattern* MVC per lo sviluppo di applicazioni *web*. Dopo aver letto varie opinioni *online*, ho scartato questa scelta in quanto risulta essere molto complicata. Uno dei motivi principali di questa complessità è, come citato da ThoughtWorks nell'articolo raggiungibile al link `https://goo.gl/dfxpaC`, "pensiamo che [JSF] sia imperfetto in quanto tenta di astrarre troppo l'HTML, il CSS e l'HTTP".

* La terza opzione, quella da me scelta, è stato Apache Wicket, anch'esso un *framework* Java che utilizza MVC con la caratteristica aggiuntiva di essere basato su componenti. Questa caratteristica, unita al fatto che Wicket permetteva l'interfacciamento senza alcuna complicazione al JCR e che era utilizzato anche da IBC, mi hanno portato a preferirlo alle altre tecnologie.

Alla luce di questa e altre decisioni, includo una tabella che elenca le tecnologie utilizzate durante il progetto.

Documentazione	**Config. e versionamento**	**Sviluppo**
LibreOffice	Maven	Eclipse
	SVN	Jackrabbit
	Tomcat	Wicket
		Wicket-Bootstrap
		JUnit

Tabella 2.4: Principali tecnologie utilizzate nel progetto.

Temporali

Per quanto riguarda i vincoli temporali, gli orari di lavoro erano gli stessi del personale IBC, ovvero dal Lunedì al Venerdì con orario dalle 8:30 alle 12:30 e dalle 14:00 alle 18:00. L'azienda non ha richiesto moduli o procedure particolari per l'assenza da lavoro o la variazione di orario per motivi universitari, tranne la comunicazione a voce al *tutor* o ad un collega.

2.2.5 Pianificazione del lavoro

La pianificazione del lavoro ha dovuto tener conto dei vincoli temporali esposti nella sezione precedente.

Ho pianificato lo svolgimento delle attività in otto settimane lavorative da quaranta ore ciascuna, come mostrato nel Gantt sottostante.

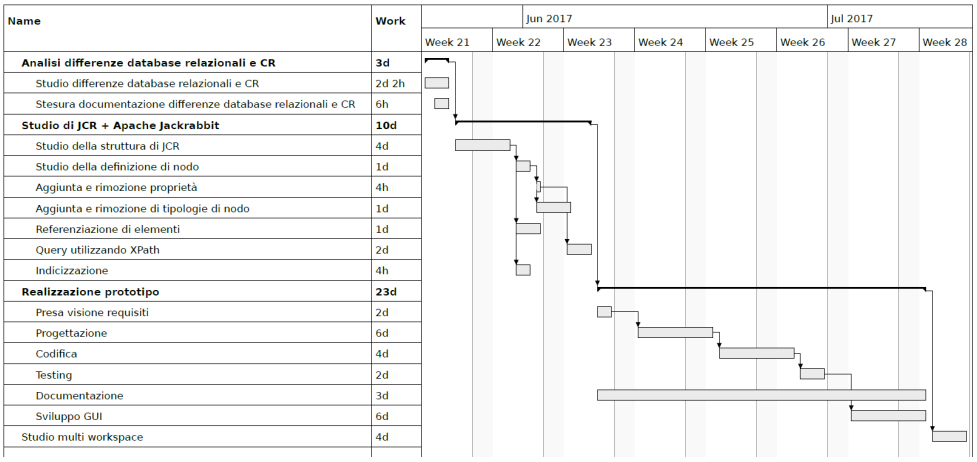


Figura 2.8: Pianificazione temporale.

Durante lo svolgimento iniziale delle attività di analisi e studio ho seguito il piano, ma durante la realizzazione del prototipo non ho rispettato la netta sequenzialità tra realizzazione del prototipo e sviluppo della GUI. Il motivo di questa decisione è dovuto al fatto che ho deciso di raggiungere l'obiettivo desiderabile stabilito dal piano di lavoro. Ho avuto quindi la necessità di iniziare ad apprendere il *framework* scelto per l'interfaccia al più presto, portandomi a svolgere le attività di realizzazione della logica del prototipo e della parte grafica in parallelo.

Tenendo conto di questo punto, il reale svolgimento delle attività è stato il seguente.

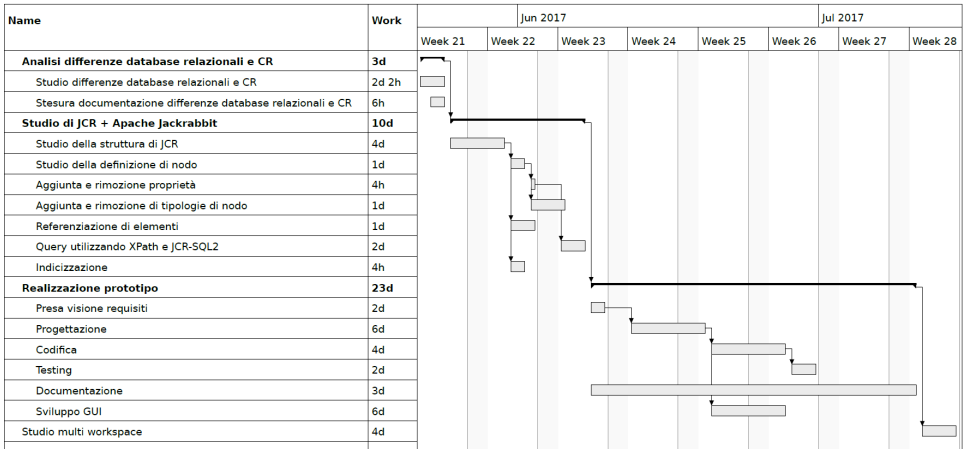


Figura 2.9: Svolgimento attività.

2.3 *Stage* in IBC: motivazioni personali

Durante la partecipazione a StageIt ho sostenuto colloqui con undici aziende. La mia scelta è ricaduta su IBC per una serie di motivi suddivisibili in tre tipologie: economici e logistici, professionali, personali.

Economici e logistici

* L'azienda, al contrario di molte altre, offriva un rimborso spese. Personalmente lo considero come un modo di riconoscere del valore nel lavoro svolto dallo stagista. Inoltre, la gratificazione ricevuta da questo riconoscimento è un buon punto di partenza per un rapporto che potrebbe continuare dopo la fine dello *stage*.

* Il posizionamento del luogo di lavoro, situato a dieci minuti da casa e vicino a Padova, era ideale per permettermi di raggiungere in breve tempo la sede dell'università. Infatti, data la necessità di terminare il progetto didattico di Ingegneria del *software*, ho dovuto presenziare ad alcuni incontri con i miei compagni di progetto dopo l'orario di lavoro. Un'azienda situata più lontano non mi avrebbe permesso tale flessibilità.

Professionali

* IBC è un'azienda che non si occupa solamente di consulenza, ma produce anche *software* proprio. Svolgere lo *stage* presso IBC mi ha permesso di essere immerso in un ambiente che unisce entrambe le realtà.

* Data la diffusione del linguaggio Java in ambito aziendale, ho valutato positivamente un'esperienza in una realtà che, oltre ad usare tale linguaggio, produce applicazioni che si basano su Java EE.

Personali

* Con questo *stage* ho voluto valutare se l'impiego presso un'azienda che produce *software* fosse adatto a me. Inoltre, dato che questa sarebbe stata la mia prima esperienza lavorativa, mi sono posto come obiettivo quello di rapportarmi con il personale esperto per avere consigli ed informazioni su come gestire un lavoro in campo informatico.

Capitolo 3

Svolgimento del progetto

3.1 Modello di sviluppo

Il modello di sviluppo che ho utilizzanto durante lo svolgimento del progetto è il modello iterativo. Questo modello prevede un'esecuzione ciclica di:

* Analisi.
* Progettazione.
* Produzione di prototipi.
* Test.
* Raffinamento dei prototipi.

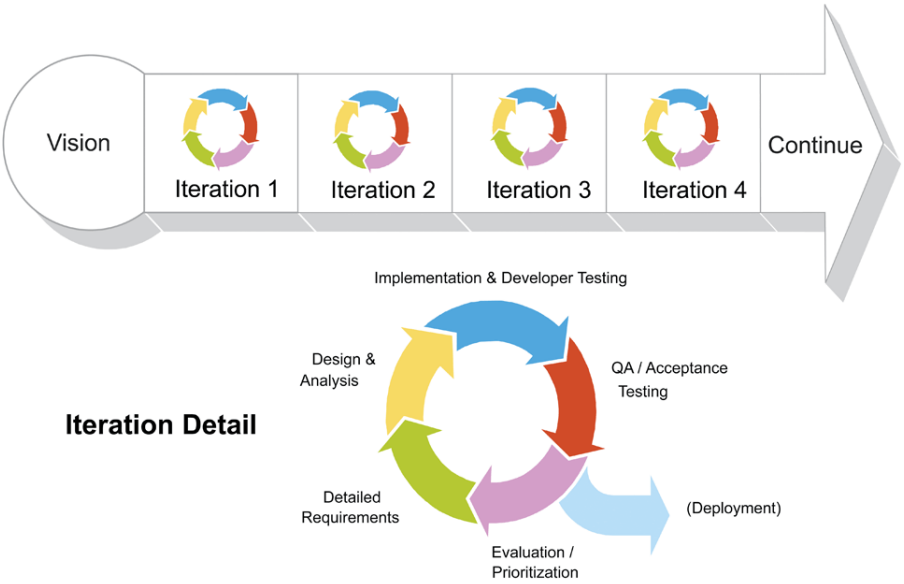


Figura 3.1: Modello iterativo (https://goo.gl/YcTb7w).

Il principale rischio del procedere per iterazioni piuttosto che per *incrementi* è quello di non convergere mai ad una soluzione. Ho quindi adottato delle soluzioni per ridurre questo rischio:

* La prima soluzione è stata fissare con il *tutor* un insieme minimo di requisiti obbligatori da implementare nel prodotto, in modo da avere una soluzione accettabile in un periodo relativamente poco avanzato del progetto.
* La seconda soluzione è stato l'impiego di prototipi da presentare durante gli incontri con il tutor per dare una migliore idea del prodotto in corso di realizzazione.

Il procedere per prototipi, oltre ad essere utilizzato ampiamente in IBC, è un metodo che si è rivelato efficace anche nel mio caso. Infatti un prototipo, rispetto alla sola documentazione:

* Fornisce una migliore visione d'insieme del prodotto, garantendo maggiore comprensibilità anche da parte del personale non tecnico.
* Richiede relativamente poco tempo per essere prodotto e modificato, permettendo maggiore flessibilità nel caso di modifiche. Inoltre, diminuisce di molto il tempo che intercorre tra il momento della decisione della modifica e la presentazione del prototipo successivo che la implementa.
* Permette di comprendere meglio anche il *design* grafico e l'esperienza di utilizzo volute dal committente nei periodi iniziali del progetto.

Nei primi periodi i prototipi da me prodotti hanno avuto forma cartacea, per poi evolversi in pagine *web* appena ho preso dimestichezza con Wicket.

3.2 Analisi dei requisiti

Dopo aver completato lo studio di fattibilità e scelto tutte le tecnologie necessarie, ho proceduto a effettuare l'analisi dei requisiti.

La metodologia di conduzione dell'analisi che ho utilizzato si è basata principalmente su interviste al *tutor* in modo da identificare innanzitutto i casi d'uso e successivamente i requisiti.

Periodicamente ho effettuato incontri con il *tutor* per verificare la corretta comprensione dei requisiti. In questo modo, nonostante io non abbia prodotto documentazione formale sull'analisi, ho ridotto il rischio di incomprensioni difficili da correggere se mantenute durante tutta la durata del processo di sviluppo.

Inoltre, per ridurre ulteriormente il rischio di implementare funzionalità non volute o in modo errato, ho prodotto quasi fin da subito dei prototipi da presentare durante gli incontri, come esposto nella sezione precedente.

3.2.1 Scopo del prodotto

L'applicazione prodotta doveva essere una *web app* che permettesse all'utente di inserire, visualizzare, modificare e rimuovere prodotti commerciali di varie tipologie. L'applicazione doveva considerare categorie e sottocategorie di prodotti. Ad esempio, la pasta è una sottocategorie dei prodotti alimentari.

Per ogni prodotto, l'utente doveva essere in grado di inserire le informazioni, obbligatorie od opzionali, imposte dalla categoria di prodotto. Oltre a queste informazioni,

l'utente doveva avere la possibilità di inserire un qualsiasi numero di proprietà aggiuntive non previste dalla tipologia. Ogni prodotto poteva avere associata una o più immagini che lo rappresentassero.

Opzionalmente, l'applicazione doveva fornire la possibilità di definire da interfaccia anche nuove tipologie e sottotipologie di prodotto.

3.2.2 Attori

Il primo passo che ho svolto è stato l'identificazione degli attori. Dato che il *tutor* non ha richiesto l'implementazione di gerarchie di utenti o dell'autenticazione, ho identificato solamente un attore, che d'ora in poi chiamerò "Utente".

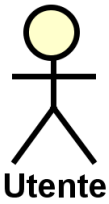


Figura 3.2: Attore utente.

3.2.3 Casi d'uso

Una volta identificato l'attore, ho proceduto a identificare le azioni che egli intendeva svolgere utilizzando il prodotto. A scopo esemplificativo, in questa sezione rappresento alcuni casi d'uso che ho identificato..

UC1 - Visualizzazione lista prodotti

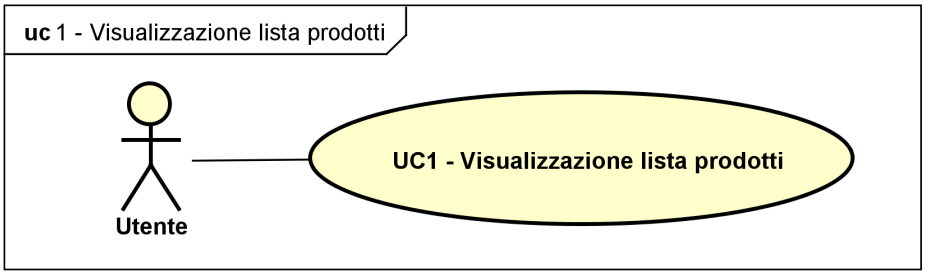


Figura 3.3: UC1 - Visualizzazione lista prodotti.

UC1 - Visualizzazione lista prodotti	
Attori	Utente.
Descrizione	L'utente visualizza la lista dei prodotti presenti all'interno dell'applicazione.
Pre-condizione	L'utente ha aperto l'applicazione.

UC1 - Visualizzazione lista prodotti	
Post-condizione	L'utente ha visualizzato la lista dei prodotti presenti all'interno dell'applicazione.
Scenario principale	1. UC1 - Visualizzazione lista prodotti.

UC2 - Esecuzione ricerca

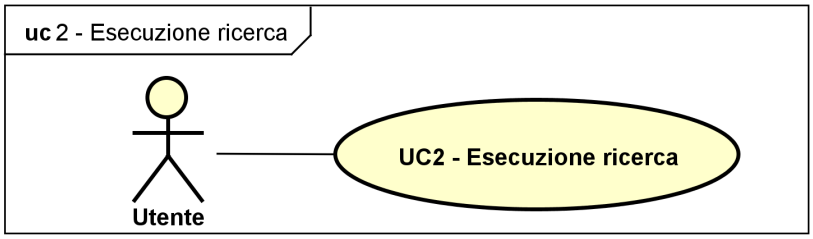


Figura 3.4: UC2 - Esecuzione ricerca.

UC2 - Esecuzione ricerca	
Attori	Utente.
Descrizione	L'utente esegue una ricerca.
Pre-condizione	L'utente ha aperto la schermata di ricerca.
Post-condizione	L'utente ha eseguito la ricerca e ne visualizza i risultati.
Scenario principale	1. UC2 - Esecuzione ricerca.

3.2.4 Requisiti

Successivamente, dopo aver identificato i casi d'uso, ho provveduto a trasformarli in requisiti elementari che descrivessero le caratteristiche che il prodotto avrebbe dovuto avere.

Rendere elementari i requisiti scendendo ad un basso livello di dettaglio è importante per aumentarne la comprensibilità e garantirne l'atomicità, ovvero fare in modo che essi si riferiscano ad una singola e precisa necessità senza alcuna ambiguità.

Nella tabella di seguito fornisco un elenco di alto livello delle principali funzionalità, ricavate dall'analisi dei requisiti, che l'applicazione doveva offrire. Ad ogni funzionalità è associata una certa importanza (obbligatoria, desiderabile o facoltativa).

Funzionalità	**Importanza**
Gestione prodotti	Obbligatoria
Gestione immagine prodotto	Desiderabile
Gestione categorie prodotti	Facoltativa
Gestione catalogo immagini prodotti	Facoltativa

Tabella 3.3: Principali funzionalità ricavate dai requisiti.

3.3 Progettazione

Per quanto riguarda la progettazione, ho seguito un approccio "*meet-in-the-middle*", ovvero ho utilizzato sia l'approccio *top-down* che *bottom-up*.

Il motivo di questa scelta è dovuto a esperienze passate vissute durante lo svolgimento del progetto di Ingegneria del *software*. Infatti, io e il mio *team* avevamo progettato in modo puramente *top-down* senza avere una conoscenza profonda delle tecnologie utilizzate. La conseguenza è stata una progettazione che impiegava *design pattern* non adatti alle tecnologie, la quale ha causato ritardi dovuti alla riprogettazione necessaria a risolvere gli errori.

Per non ripetere lo stesso errore, sono arrivato anche a progettare e codificare alcune componenti di basso livello prima di fissare completamente l'intera architettura. Questa scelta ha avuto un duplice vantaggio:

* In primo luogo, mi ha permesso di convergere ad una soluzione architetturale funzionante e che si adattava bene alle tecnologie utilizzate.
* In secondo luogo, la codifica delle componenti grafiche mi ha permesso di avere dei prototipi in periodi relativamente poco avanzati del progetto, con i vantaggi descritti nelle sezioni precedenti.

L'architettura del prodotto si basa sul *design pattern Model-View-Controller*, alla cui base c'è il principio di separazione delle responsabilità. Questa scelta si adatta anche al *framework* Wicket, che effettua già una prima separazione della *View* fornendo pagine HTML statiche da popolare tramite codice Java.

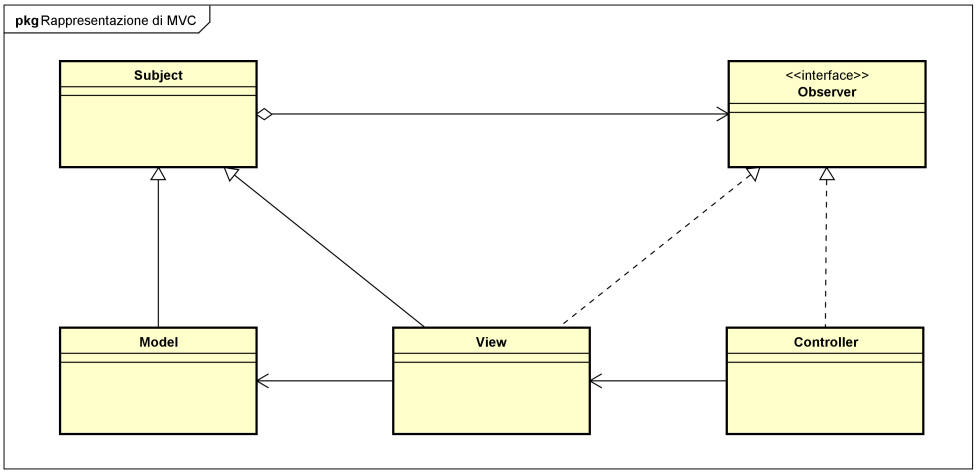


Figura 3.5: Rappresentazione dell'architettura MVC con *pattern* Observer. (https://goo.gl/NYKkpQ).

3.3.1 DAO

L'accesso ai dati contenuti nel JCR richiede l'utilizzo di API di relativamente basso livello per la lettura e la scrittura. Per separare ulteriormente la logica dell'applicazione da queste API, ho utilizzato il *design pattern* Data Access Object (DAO).

Questo *pattern* consiste nel racchiudere tutte le operazioni che effettuano l'accesso ad un *database* (in questo caso, il JCR) in apposite classi. Gli oggetti di queste

classi nascondono l'esistenza di *query* fornendo dei metodi che restituiscono oggetti di *business* utilizzabili direttamente dall'applicazione.

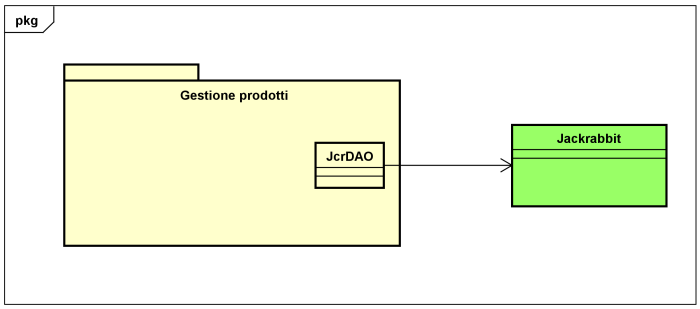


Figura 3.6: Ruolo del DAO nell'architettura.

3.3.2 Struttura JCR

Ho progettato la struttura del contenuto del JCR sotto forma di albero, così come previsto dal modello JCR.

Ogni elemento memorizzato è definito come nodo dell'albero. Ogni nodo può avere delle proprietà e ogni proprietà a sua volta ha un valore. I dati concreti sono memorizzati solamente come valore di una proprietà, le quali diventano quindi le foglie dell'albero.

Includo un esempio minimale di struttura di JCR. Nella rappresentazione, i nodi hanno una forma circolare mentre le proprietà assumono una forma rettangolare.

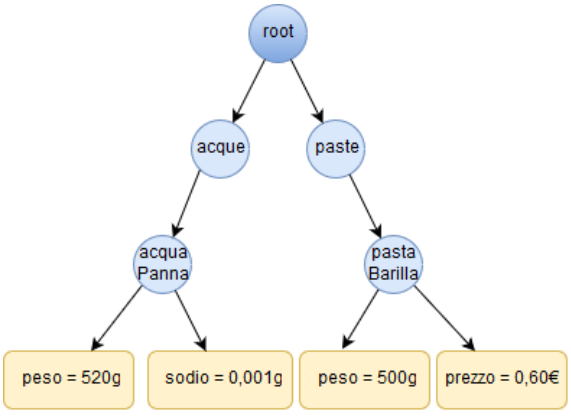


Figura 3.7: Esempio struttura JCR.

3.3.3 *View*

Durante la progettazione della *View* ho sfruttato i componenti `Panel` di Wicket per massimizzare il riuso di codice.

Un `Panel` infatti può essere utilizzato in diversi punti dell'applicazione senza la necessità di dover ricopiare il *markup* ogni volta. Fornisco un piccolo esempio di progettazione di due pagine diverse che utilizzano lo stesso `Panel`.

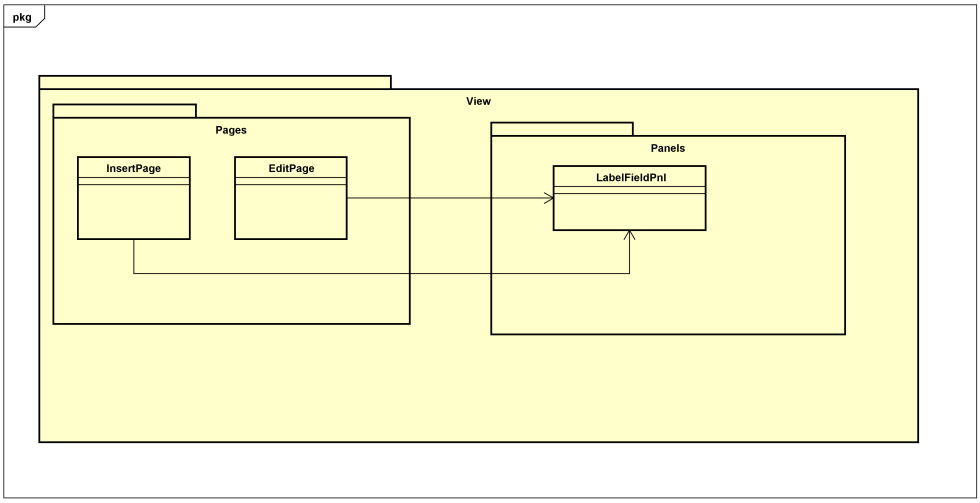


Figura 3.8: Esempio di utilizzo di `Panel` di Wicket.

Un altro esempio di utilizzo dei `Panel` è in combinazione con i componenti `Repeater`, i quali permettono di creare un componente grafico per ogni elemento presente all'interno di una collezione.

3.4 Codifica

Ho realizzato il prodotto utilizzando i seguenti linguaggi:

* **Java 8:** per l'utilizzo delle API JCR e la realizzazione della *View* con Wicket.
* **HTML5:** per la creazione del *markup* delle componenti Wicket.
* **CSS:** per assegnare lo stile alle pagine *web*.
* **XML:** per la configurazione di Maven.
* **Compact Namespace and Node Type Definition (CND):** per la definizione dei tipi di nodo e della struttura di JCR.

Nonostante Jackrabbit offrisse API aggiuntive rispetto al JCR, ho utilizzato principalmente quelle specificate dallo *standard* JSR 283, contenute all'interno di `javax.jcr.*`. Questa scelta assicura:

* Maggiore compatibilità con altre implementazioni di JCR nel caso di sostituzione della libreria Jackrabbit con un'altra libreria.
* Minori problemi in caso di aggiornamento della libreria Jackrabbit, in quanto le API definite dallo *standard* possono cambiare solamente a causa di un aggiornamento dello *standard* stesso. Le API di Jackrabbit invece possono cambiare molto più facilmente e senza preavviso.

Durante la codifica in Java ho utilizzato alcune caratteristiche del linguaggio che non avevo appreso durante gli studi universitari, tra cui:

- **Java Annotations:** una forma di metadato da associare ad un qualche elemento del codice in modo da fornire maggiore leggibilità da parte di altri sviluppatori e sopratuttto da parte del compilatore. Alcune delle *annotation* che ho utilizzato maggiormente sono state `@Override` per la ridefinizione dei metodi e `@Test` per la specifica dei *test case*.
- ***Reflection*:** meccanismo tramite il quale è possibile ottenere informazioni sul codice e modificare metodi, classi e attributi a *runtime*. La *reflection* viene utilizzata sopratutto da Wicket.

Per quanto riguarda la rappresentazione degli oggetti di *business*, ho utilizzato dei JavaBean. Questi ultimi sono degli oggetti di classi sottoposte a vari vincoli, tra cui avere solo il costruttore di *default* e permettere l'accesso ai propri campi dati attraverso metodi `get` e `set`. Quest'ultimo vincolo, unito alla *reflection*, permette a Wicket di legare i campi dati dei *bean* ai valori dei componenti grafici, in modo da permettere inserimenti e visualizzazioni dei dati.

Per meglio documentare il codice ho prodotto, attraverso l'utilizzo di Javadoc, una pagina *web* che fornisce descrizioni di classi e metodi.

3.5 Verifica e validazione

Durante tutto lo svolgimento del progetto ho eseguito attività di verifica. Le revisioni periodiche con il *tutor* e i colleghi durante le quali ho dimostrato prototipi, documentazione ed esempi di codice sorgente sono state utili a verificare che il progetto procedesse secondo quanto pianificato. Inoltre, le verifiche sono state utili per la risoluzione dei problemi che ho incontrato durante la codifica.

3.5.1 *Test*

Un altro tipo di verifica che ho impiegato è quella automatica tramite *test* utilizzando JUnit4. Nonostante io abbia eseguito i *test* di sistema solamente nel periodo finale del progetto, ho effettuato verifiche più mirate tramite *test* di unità nel corso del progetto, sopratutto per quanto riguarda le componenti della *View*. L'utilizzo di `WicketTester` ha permesso *test* di unità di componenti Wicket senza troppo dispendio di tempo.

Per evitare di dover ridefinire i meccanismi di funzionamento del JCR durante l'esecuzione dei *test* di integrazione, ho utilizzato la classe `JackrabbitRepositoryStub` fornita da Jackrabbit. Questa classe fornisce uno *stub* di un JCR; l'ho utilizzata per testare l'interazione delle altre componenti con il *repository*.

I *test* di integrazione e di sistema che interagiscono con la *View* sono guidati da un oggetto della classe `WicketTester`, che permette la simulazione di eventi come l'inserimento di caratteri da tastiera o la pressione di un bottone.

Una delle difficoltà principali che ho incontrato durante l'esecuzione dei *test* di sistema è stata l'interazione tra eventi Ajax e l'invio di *form*. `WicketTester` infatti fornisce un oggetto di tipo `FormTester`, che permette l'impostazione di valori degli oggetti contenuti in una data *form* e il successivo invio della stessa. Data la mia inesperienza con gli eventi Ajax, ho impiegato un po' di tempo durante il periodo finale del progetto per capirne il funzionamento e completare i *test* in maniera corretta. Anche grazie all'aiuto della *mailing list* di Wicket, sono riuscito a risolvere i problemi incontrati e ad implementare la maggior parte dei *test* previsti.

Grazie al *plug-in* EclEmma per Eclipse ho potuto calcolare i gradi di copertura del codice da parte dei *test*. Complessivamente, *test* di unità, integrazione e sistema hanno dato i seguenti risultati:

Tipo di *coverage*	**Percentuale**
Statement coverage	85%
Branch coverage	80%

Tabella 3.4: Resoconto copertura del codice.

Durante la pianificazione e l'implementazione dei *test* ho preferito testare più profondamente le funzionalità di maggior importanza e a maggior rischio di errore, dati anche i vincoli temporali dello *stage*. Per questo motivo le percentuali di copertura non raggiungono il 100%, ma si attestano in ogni caso su risultati accettabili.

Test prestazionali

Uno degli obiettivi del prototipo era l'implementazione di ricerche *full-text* tra gli elementi inseriti. Il *tutor* si è dimostrato interessato alla valutazione prestazionale delle due tipologie di ricerca offerte da JCR:

* La ricerca "classica", ovvero eseguita tramite operatore `LIKE`.
* La ricerca *full-text* eseguita tramite operatore `CONTAINS`.

Ho eseguito *test* prestazionali su un insieme di 20.000 prodotti di tipo pasta strutturati nel seguente modo:

Nome attributo	**Tipo attributo JCR**
id	`String`
prezzo	`Decimal`
minutiCottura	`Long`
volume	`Decimal`
marca	`String`

Tabella 3.5: Descrizione struttura prodotto utilizzato nei *test* prestazionali.

Eseguendo per dieci volte *query* equivalenti, ovvero che ritornavano lo stesso insieme di risultati, ho ottenuto i seguenti risultati:

Operatore	**Tempo di esecuzione (ms)**										**Media**
	Numero *test*										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
CONTAINS	396	476	528	292	805	687	436	557	418	510	510,50
LIKE	503	636	697	432	531	747	807	873	772	651	665,90

Tabella 3.6: Resoconto *test* prestazionali.

Come dimostrato dai *test*, l'operatore `CONTAINS` restituisce i risultati impiegando mediamente un tempo minore del 23% rispetto all'operatore `LIKE`. Il motivo di questa differenza è dovuta al fatto che Jackrabbit sfrutta il motore di ricerca testuale fornito

da Apache Lucene per l'esecuzione delle ricerche *full-text* con l'operatore `CONTAINS`. Questo motore è conosciuto per la sua velocità in questo tipo di ricerche.

Ho prodotto un documento destinato a IBC che illustra il procedimento utilizzato per effettuare i *test* prestazionali e i risultati ottenuti.

Validazione

Ho eseguito attività di validazione durante gli ultimi giorni dello *stage* mostrando il prodotto al *tutor* e dimostrandogli le funzionalità implementate. Grazie ai *test* di sistema che avevo precedentemente eseguito ho potuto portare a termine la validazione in sicurezza, avendo la garanzia che il sistema rispondesse correttamente ai casi d'uso previsti. Complessivamente, la validazione ha avuto esito positivo.

3.6 Prodotto finale

Il prodotto finale assume la forma di una *web app single-page*. Per istruire l'utilizzatore finale all'utilizzo del prodotto ho redatto un manuale utente comprensivo di immagini.

Procedo a illustrare alcune funzionalità offerte dal prodotto.

3.6.1 *Homepage*

L'*homepage* presenta una tabella con struttura ad albero tramite la quale è possibile visualizzare le informazioni dei prodotti. Cliccando sul pulsante "+" è possibile espandere la visualizzazione delle categorie, fino ad arrivare alla visualizzazione dei prodotti desiderati.

Figura 3.9: *Homepage* con nessun prodotto visualizzato.

Figura 3.10: *Homepage* con alcuni prodotti visualizzati.

3.6.2 Finestra di inserimento

La finestra di inserimento permette di aggiungere nuovi prodotti a partire dalle categorie già esistenti. Selezionando la tipologia di prodotto, l'applicazione come prima cosa richiede la compilazione dei campi dati predefiniti. Quelli obbligatori sono segnalati da un asterisco di colore rosso.

Una volta compilati i dati obbligatori, l'utente ha la possibilità di aggiungere proprietà aggiuntive tramite l'utilizzo del pulsante "Aggiungi riga proprietà". Una proprietà consiste di tre elementi:

* Un nome.
* Un tipo, a scelta fra `String`, `Long`, `Decimal`, `Double`.
* Un valore, in accordo con il tipo scelto.

Infine, l'utente ha la possibilità di aggiungere una sola immagine da abbinare al prodotto.

Figura 3.11: Finestra di inserimento prodotto.

3.6.3 Finestra di visualizzazione dettaglio e modifica

Cliccando sul pulsante "Modifica" affianco al nome di un prodotto verrà aperta la finestra di visualizzazione dettaglio e modifica.

All'interno di questa finestra l'utente ha la possibilità di:

* Visualizzare e modificare le informazioni di dettaglio del prodotto.
* Rimuovere eventuali proprietà aggiuntive precedentemente inserite. Le proprietà predefinite non possono essere rimosse per non invalidare la struttura imposta dalla categoria del prodotto.
* Aggiungere ulteriori proprietà.
* Visualizzare e modificare l'immagine che rappresenta il prodotto.

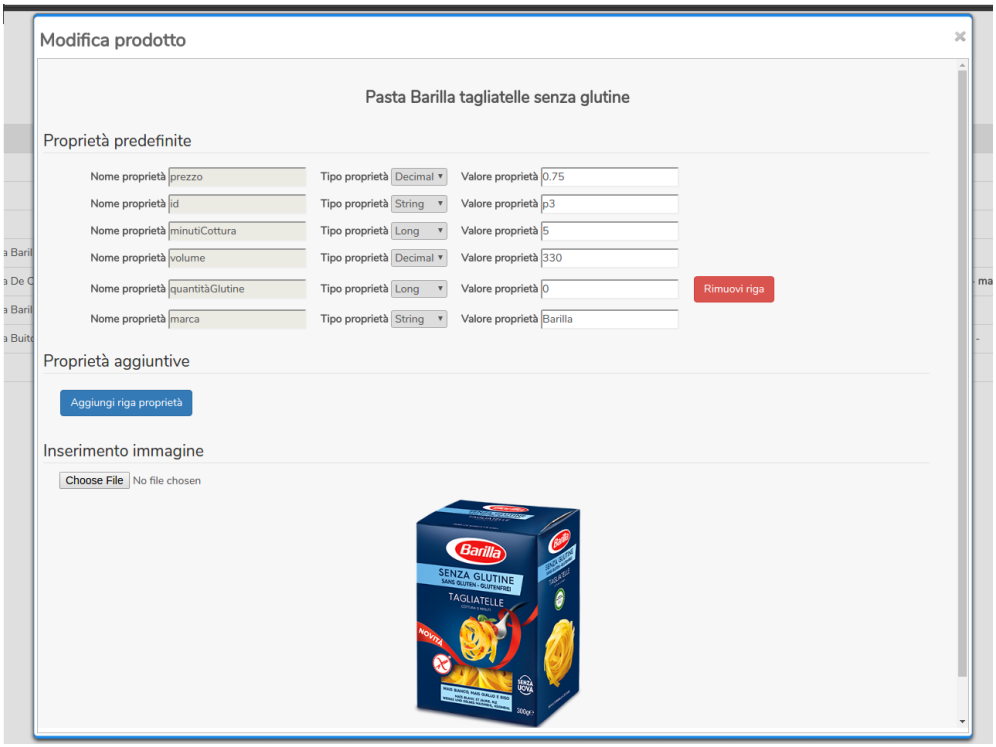


Figura 3.12: Finestra di visualizzazione dettaglio e modifica prodotto.

3.6.4 Finestra di ricerca

Dalla finestra di ricerca l'utente può effettuare ricerche tra i prodotti, raffinando i criteri di selezione fino ad arrivare all'articolo (o all'insieme di articoli) di interesse.

Per effettuare una ricerca è necessario seguire i seguenti passi:

* Come prima cosa, l'applicazione richiede che venga selezionata una tipologia di prodotto su cui fare la ricerca. Le tipologie suggerite sono tutte quelle presenti all'interno del JCR in quel dato momento. Se la tipologia non viene specificata, la ricerca comprenderà i prodotti di tutte le categorie.
* Successivamente, l'utente potrà aggiungere uno o più filtri di ricerca, specificando per ognuno:
  - **Nome della proprietà da ricercare**. L'applicazione suggerisce tutte le proprietà definite dalla struttura della tipologia di prodotto selezionato, più eventuali proprietà aggiuntive.

  - **Operatore da applicare**, come ad esempio >, >=, <, <=, =.
  - **Valore da ricercare**.

* Una volta eseguita la ricerca, i risultati verranno visualizzati nella stessa finestra. L'utente è libero di modificare e rimuovere i filtri precedentemente utilizzati, oppure di inserirne di nuovi per raffinare la ricerca.

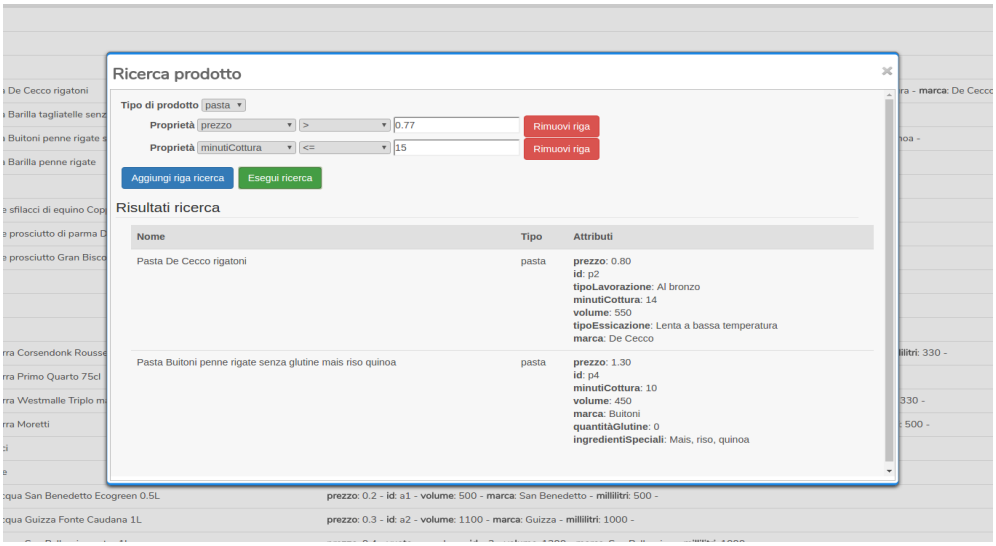


Figura 3.13: Finestra di ricerca, filtro di selezione multiplo.

Per accedere alla funzionalità di ricerca *full-text*, è sufficiente selezionare la voce "Tutte" nel *box* che indica la proprietà da ricercare. Così facendo, l'unico operatore disponibile sarà `CONTAINS`, il quale cercherà il testo immesso dall'utente all'interno di tutte le proprietà di tipo `String` di tutti i nodi della tipologia specificata.

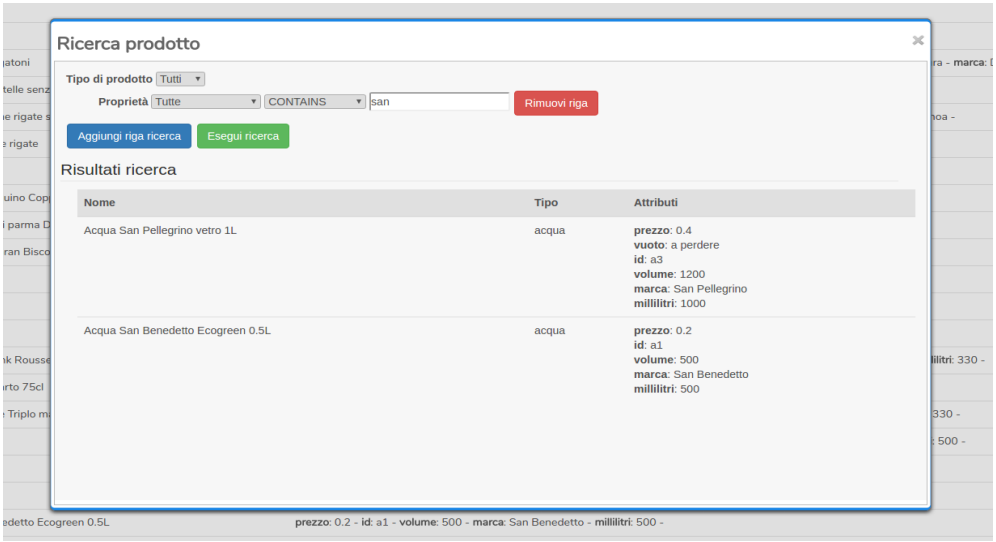


Figura 3.14: Finestra di ricerca, filtro di selezione *full-text*.

Capitolo 4

Analisi retrospettiva

4.1 Raggiungimento degli obiettivi

Nel complesso, lo *stage* ha avuto un esito positivo. Nelle sezioni seguenti presento un resoconto degli obiettivi prefissati e raggiunti, utilizzando la seguente notazione

* ✓**:** obiettivo pienamente soddisfatto.
* ✓**:** obiettivo parzialmente soddisfatto.
* ✗**:** obiettivo non soddisfatto.

4.1.1 Progettuali

Includo una tabella riassuntiva che descrive il grado di raggiungimento degli obiettivi di progetto definiti nella sezione 2.2.3.

Obiettivo	Soddisfacimento
Obiettivi obbligatori	
Studio e documentazione sulle differenze tra *database* relazionale e *Content Repository*	✓
Studio e documentazione sulla storia di *Content Repository*	✓
Studio di JSR 170 e JSR283: Content Repository for Java (JCR), con produzione di codice e documentazione	✓
Studio e documentazione della struttura di JCR	✓
Studio e documentazione della definizione di nodo	✓
Studio e documentazione riguardo aggiunta, rimozione e modifica di proprietà di un nodo	✓
Studio e documentazione riguardo l'aggiunta e la rimozione di tipologie di nodo	✓
Studio e documentazione riguardo la referenziazione di elementi	✓
Studio e documentazione riguardo l'esecuzione di *query* utilizzando XPath e JCR-SQL2	✓
Studio e documentazione riguardo l'indicizzazione	✓
Progettazione di un prototipo di applicazione che gestisca le informazioni di prodotti commerciali	✓

Realizzazione di un prototipo di applicazione che gestisca le informazioni di prodotti commerciali	✓
Obiettivi desiderabili	
Realizzazione della GUI del prototipo	✓
Obiettivi facoltativi	
Studio e documentazione riguardo i *workspace* multipli	✓

Tabella 4.1: Resoconto soddisfacimento obiettivi del progetto.

Com'è possibile evincere dalla tabella ho completato tutti gli obiettivi obbligatori e desiderabili previsti dal piano di lavoro. Ho completato parzialmente l'obiettivo facoltativo riguardante i *workspace* multipli in quanto ne ho effettuato uno studio solamente superficiale e non ho incluso tale funzionalità nel prototipo.

A seguire includo una tabella di soddisfacimento delle funzionalità descritte nella sezione 3.2.4.

Funzionalità	**Importanza**	**Soddisfacimento**
Gestione prodotti	Obbligatoria	✓
Gestione immagine prodotto	Desiderabile	✓
Gestione categorie prodotti	Facoltativa	✓
Gestione catalogo immagini prodotti	Facoltativa	✗

Tabella 4.2: Soddisfacimento requisiti.

Non ho implementato i requisiti opzionali che riguardavano le operazioni sulle tipologie di prodotto tramite interfaccia grafica per motivi di tempo. Tuttavia, ho fornito in ogni caso la possibilità di eseguire tali operazioni attraverso un file di configurazione scritto in linguaggio CND. In accordo con il *tutor*, abbiamo ritenuto questa scelta accettabile dati i vincoli temporali, seppur non ottimale.

4.1.2 Aziendali

Procedo ad effettuare un'analisi sul raggiungimento degli obiettivi aziendali definiti nella sezione 2.1. Il grado di raggiungimento di tali obiettivi è solamente descritto dal mio punto di vista e in base alla mia percezione dello *stage*. Di seguito non saranno quindi presenti opinioni del *tutor* aziendale o di IBC.

* **Studio di nuove tecnologie** ✓
  L'azienda ha raggiunto pienamente questo obiettivo in quanto ho fornito documentazione, esempi di codice sorgente e un prototipo funzionante che illustrano il funzionamento di JCR e l'interazione con il *framework* per la creazione di *web app* Wicket.

* **Assunzione di personale** ✓
  L'azienda ha raggiunto parzialmente questo obiettivo in quanto è riuscita a farmi apprendere in buona misura i meccanismi aziendali, il modo di lavorare e l'interfacciamento tra reparti. Tuttavia, data la mia intenzione di intraprendere l'istruzione universitaria magistrale, l'azienda non può vedere completato il suo obiettivo di assunzione immediata a fine *stage*. In ogni caso, l'esperienza positiva di *stage* e i risultati ottenuti non pregiudicano un'eventuale assunzione alla fine dei miei studi.

* **Soluzioni originali ✗**
  L'azienda non ha raggiunto questo obiettivo in quanto le soluzioni da me implementate non sono né originali né innovative. Durante lo svolgimento del progetto non mi sono distanziato troppo da quanto dettato dagli *standard* e da quanto suggerito dalle *best pratice* aziendali e di settore. Personalmente reputo che prima di implementare una soluzione innovativa sia necessario conoscere profondamente le soluzioni classiche, obiettivo non raggiungibile secondo le mie capacità in un periodo di tempo di soli due mesi.

4.1.3 Personali

Complessivamente, considero raggiunti gli obiettivi personali definiti nella sezione 2.3. Segue un'analisi più approfondita.

* **Economici e logistici ✓**
  Ho raggiunto gli obiettivi economici e logistici in quanto l'azienda ha mantenuto la promessa di rimborso spese e la vicinanza alla sede universitaria mi ha permesso di completare con successo il progetto di Ingegneria del *software*.

* **Professionali ✓**
  Ho raggiunto solamente in parte gli obiettivi professionali. Ho potuto collaborare con un'azienda che combina consulenza a produzione di *software* proprio, permettendomi di apprendere informazioni sul suo modo di lavorare. Tuttavia, non ho appreso tutte le conoscenze in ambito Java EE che speravo di apprendere, in quanto tale specifica è molto ampia e difficilmente applicabile in un progetto di breve durata.

* **Personali ✓**
  Ho raggiunto pienamente gli obiettivi personali in quanto mi sono rapportato con personale esperto e ho potuto avere consigli in tale ambito anche dopo la fine dello *stage*.

4.2 Bilancio formativo personale

Nel complesso, il bilancio formativo personale dello *stage* è positivo. Nelle sezioni seguenti fornisco una descrizione delle conoscenze, abilità e competenze che ho acquisito.

4.2.1 Conoscenze

Dagli studi effettuati durante lo svolgimento del progetto ho acquisito conoscenze nelle seguenti tecnologie da me non conosciute e nei seguenti campi di interesse:

* Le necessità e le problematiche riguardanti la persistenza dei dati.
* JCR e la libreria Jackrabbit.
* Apache Wicket.
* SVN, anche se solamente nel caso d'uso più semplice, con solo uno sviluppatore che contribuisce al *repository*.
* Maven.

* Tomcat.
* JUnit.
* Eclipse.
* Superficialmente, la specifica Java EE.

Inoltre, per quanto riguarda le tecnologie già conosciute, ho approfondito l'utilizzo di:

* HTML5 e CSS.
* LibreOffice, sopratutto Writer e Calc.
* Sistema operativo Linux Mint.

4.2.2 Abilità

Ho acquisito e approfondito varie abilità, tra cui:

* **Creazione di configurazioni *software*** utilizzando Maven, in modo da gestire le dipendenze e il processo di *build* di un progetto.
* **Implementazione di *test*** per verificare la corretta risposta di un sistema a dei casi d'uso utilizzando JUnit. Considero questa abilità molto importante per poter effettuare la validazione con il committente in sicurezza.
* ***Debug*** del codice a *runtime* sfruttando le funzionalità offerte dall'IDE Eclipse.
* **Interazione con un'azienda** che sfrutta la metodologia Agile.

4.2.3 Competenze

Infine, ho anche acquisito molte competenze. A seguire ne presento un elenco.

* **Progettazione, realizzazione e *test*** di *web app* basate sul *framework* Wicket.
* **Conduzione di interviste** con un soggetto esterno (nel mio caso, il *tutor* aziendale) per comprendere i requisiti di un prodotto *software*. Nonostante avessi già una base di questa competenza grazie al progetto di Ingegneria del *software*, l'interazione con il *tutor* aziendale è stata la prima esperienza in cui ho condotto tali interviste completamente da solo. Questo mi ha quindi permesso di affinare la competenza.
* **Capacità nell'inserirmi in un contesto lavorativo** nuovo e di rapportarmi con colleghi e superiori, rispettando regole aziendali e modello di sviluppo. L'inserimento in un'azienda strutturata mi ha permesso lo sviluppo di questa competenza in misura maggiore rispetto ad un altro tipo di azienda.
* **Impiego della prototipazione** per fissare un'architettura *software*, in modo da evitare costose riprogettazioni e cambiamenti architetturali.
* **Implementazione e comprensione di *test* prestazionali**. Capire il modo con cui sono implementati i *test* di questo tipo è importante per comprendere con efficacia i confronti tra diverse librerie e soluzioni proposte *online*.

4.3 Mancanze nell'insegnamento accademico

Complessivamente, uno *stage* dovrebbe fornire allo stagista un bagaglio di conoscenze nuove che normalmente non possono essere insegnate in ambito scolastico. Tuttavia, l'università dovrebbe dare le nozioni di base per poter svolgere con profitto lo *stage*, senza la necessità di apprendere in ambiente lavorativo determinati argomenti. A seguire includo una lista delle conoscenze che, secondo la mia esperienza, dovrebbero essere inserite nell'insegnamento accademico.

* **Utilizzo di IDE e sistemi di versionamento del codice**. L'attuale assetto didattico non prevede alcuna introduzione all'utilizzo di IDE, uno strumento fondamentale per un programmatore. La prima occasione dove lo studente si trova ad utilizzare seriamente un IDE è durante il progetto di Ingegneria del *software*, collocato alla fine del percorso di studi. Se il gruppo non approfondisce l'utilizzo dell'IDE, lo studente rischia di trovarsi spiazzato durante i primi giorni dello *stage*. Stesso discorso vale per il versionamento del codice, fondamentale quando si lavora in gruppo.

  Soluzione proposta: attualmente le lezioni di laboratorio, specialmente nei primi due anni, sono poco guidate. Sfruttare alcune di queste lezioni, soprattutto durante i corsi di Programmazione e di Programmazione ad oggetti, per insegnare allo studente un utilizzo maturo di IDE e versionamento del codice potrebbe essere una buona soluzione.

* **Inadeguatezza nell'insegnamento di tecnologie *web***. Lo studente che inizia lo *stage* ha una conoscenza troppo basilare dell'ambito *web*, in quanto l'omonimo corso del terzo anno tratta moltissime tecnologie in modo molto superficiale. Inoltre, il *focus* sull'accessibilità concorre a ridurre ulteriormente il tempo a disposizione di altri argomenti. Alcune delle nozioni fondamentali che secondo me dovrebbero essere fornite sono il funzionamento di AJAX e dei servizi REST.

  Soluzione proposta: una soluzione accettabile potrebbe essere trasferire parte delle tecnologie insegnate dal corso di Tecnologie *web* al corso di Basi di dati, in modo da poter approfondire i temi che ho appena citato. Personalmente ripristinerei l'assetto didattico in vigore fino a tre anni fa, che prevedeva l'inclusione di argomenti come HTML e PHP nel corso di Basi. Tuttavia, per renderla una soluzione davvero efficace, sarebbe necessario eliminare la ridondanza presente a quei tempi, durante i quali il corso di Tecnologie *web* riprendeva gli stessi argomenti (in ambito *web*) trattati nel corso di Basi.

* **Educazione alle tecnologie**. Il corso di studi vede gli studenti impegnati nei primi due anni a studiare tecnologie classiche e ormai ben affermate. Durante il progetto di Ingegneria del *software* (e in alcuni *stage*) viene richiesta la scelta dello *stack* tecnologico, e nella maggior parte dei casi le alternative comprendono tecnologie innovative. Ritengo che gli strumenti a disposizione dello studente siano del tutto insufficienti per effettuare una scelta matura e consapevole. Se lo *stack* tecnologico non è fissato, lo studente si trova a dover effettuare una scelta che, molto spesso, viene fatta in completa casualità o seguendo i consigli, a volte non oggettivi, presenti *online*. Dato che non è possibile illustrare tutte le tecnologie esistenti, sarebbe utile dare come base un insieme di criteri che permettano allo studente di prendere decisioni più consapevoli in ambito tecnologico.

Soluzione proposta: i seminari tenuti durante il corso di Ingegneria del *software* sono un buon punto di partenza, ma il *focus* di ogni seminario è una singola tecnologia, difficile da collocare nello *stack* tecnologico per studenti inesperti. Inoltre, essi vengono tenuti durante uno dei periodi più impegnativi del percorso accademico, quindi la partecipazione non è sempre garantita.

Una buona soluzione potrebbe essere tenere dei seminari nel corso del secondo anno che diano criteri di base sulla scelta dello *stack* tecnologico e informazioni sullo stato dell'arte raggiunto in vari campi dell'informatica.

Bibliografia