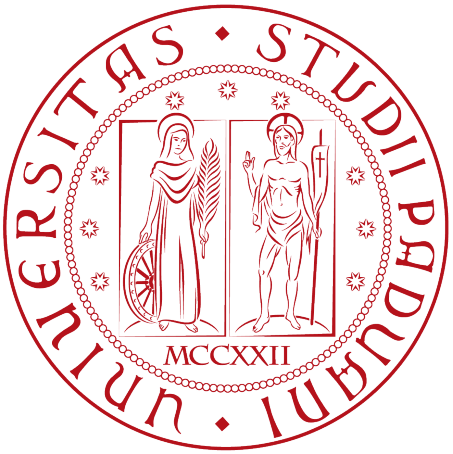
Università degli Studi di Padova

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.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

L'azienda

1.1 Contesto aziendale

IBC è nata nel 1980 come concessionaria NCR. Le sue prime attività per conto di NCR riguardavano la fornitura di attrezzature *hardware* per i punti vendita, come ad esempio POS e *scanner*. Successivamente, essa si è specializzata nella produzione di *software* specifici per il mercato *retail*, offrendo soluzioni personalizzate in base alle esigenze del singolo cliente.

Nel 1995 IBC fonda la sua sede a Peraga di Vigonza, in provincia di Padova. Ha sede tutt'ora nello stesso luogo, con tre filiali ad Alessandria, Trieste e Viterbo.

Figura 1.1: Logo IBC (https://www.ibc.it/).

Figura 1.2: Cronologia IBC (https://www.ibc.it/).

IBC è stata una delle prime *software house* italiane a realizzare progetti riguardanti la *fidelity* e la profilazione dell'utente finale. Correntemente gestisce circa mille punti vendita, avendo installato quattromila casse e quattrocento postazioni *self checkout*.

Attualmente, l'azienda opera principalmente su tre aree:

* **Sviluppo progetti**.
* **Fornitura prodotti *software* e *hardware***.

* **Servizi e assistenza**.

Fornirò spiegazioni ed esempi riguardo queste aree nelle sezioni successive di questo capitolo. L'azienda possiede inoltre due certificazioni:

* **Certificazione di qualità UNI EN ISO 9001:2008:** per la commercializzazione e l'assistenza di misuratori fiscali, strumenti di pesatura, strumenti per l'automazione del punto vendita e POS bancari.
* **Certificazione per la verifica periodica dei misuratori fiscali:** IBC è abilitata alla verificazione periodica dei misuratori fiscali. Inoltre, è anche riconosciuta come laboratorio accreditato presso la CCIAA di Padova per le verifiche metriche degli strumenti di pesatura

Per fornire prodotti e servizi aggiornati, l'azienda collabora con vari *partner* tecnologici, tra cui: NCR, Verifone, Motorola, Datalogic, Bizerba, Lenovo e Maind informatica.

Inoltre, per garantire il servizio di assistenza, l'azienda intrattiene rapporti con: Master Office, InfoMaint, IT-Avantec, BSS, GAB Tamagnini.

I clienti principali di IBC fanno tutti parte della GDO. Tra di essi, troviamo sia clienti nazionali, come ad esempio il Gruppo UniComm, Benetton, Despar, Lando e Rossetto, sia internazionali, come ad esempio Würth Superstore.

1.1.1 Prodotti

I prodotti che IBC fornisce si dividono principalmente in due categorie:

* *Hardware*.
* *Software*.

Hardware

I prodotti *hardware* sono costituiti principalmente da strumenti di cassa e di pagamento. L'azienda non produce direttamente questo tipo di prodotti, ma opera da distributore, installatore e manutentore.

1. **Soluzioni *self*:** prodotti che consentono al cliente di concludere la spesa ed effettuare il pagamento autonomamente. Alcuni esempi di questa tipologia di prodotti sono le casse NCR Fastlane Selfserv Checkout Versione 6 (a sinistra nell'immagine) e i Kiosk NCR Selfserv 85 (a destra).

Figura 1.3: Soluzioni *self* (https://www.ibc.it/).

2. **Soluzioni POS:** si tratta di terminali che permettono al cliente di pagare utilizzando carte di credito, di debito o prepagate.
3. **Periferiche:** *scanner*, stampanti, *monitor* e tutte le altre periferiche per aggiungere funzionalità ai POS e snellire le operazioni di *checkout*.

Figura 1.4: *Scanner* Datalogic Gryphon M4130 (https://goo.gl/MZqts3).

4. **Terminali *mobile*:** palmari che utilizzano sia Android che Windows, in modo da poter fornire compatibilità con la maggior parte dei sistemi dei clienti.
5. **Terminali di pagamento:** strumenti (come ad esempio i PIN Pad) che permettono al cliente di inserire il numero della sua carta durante il pagamento. Garantiscono sicurezza e velocità durante l'esecuzione di questa operazione.

Figura 1.5: Pin PAD Verifone VX675 (https://www.ibc.it/).

6. ***Server*:** prodotti da installare nei punti vendita. Essi riescono a gestire un elevato numero di richieste concorrenti, caratteristica fondamentale soprattutto nelle operazioni di gestione magazzino.
7. **Bilance:** strumenti che garantiscono alta precisione nella pesatura. Sono inoltre programmabili, personalizzabili e integrabili allo *scanner* per ampliare le funzionalità della postazione cassa. Sono anche utilizzate nella maggior parte dei reparti ortofrutta, permettendo al cliente di effettuare le operazioni di pesatura autonomamente.

Figura 1.6: Bilancia Bizerba EC 100 (https://goo.gl/dhTc9F).

Software

Oltre a fornire *hardware*, IBC produce anche *software*. Solitamente i clienti richiedono la realizzazione di soluzioni personalizzate. Per far ciò, l'azienda si è dotata di soluzioni modulari e flessibili.

1. **JStore:** questo *software* è una *suite* completa che offre servizi per gli ambienti *retail*. JStore, una volta installato in sede, permette la gestione e il controllo di tutti i punti vendita. Una delle caratteristiche principali di questo software è la sua modularità, in modo da poter essere ampliato e modificato senza intaccare le altre funzionalità. JStore è realizzato in linguaggio Java, quindi è particolarmente adatto ad ambienti multipiattaforma.

Figura 1.7: Moduli JStore (https://www.ibc.it/).

La *suite* copre quattro aree strategiche principali. Ogni area utilizza vari moduli per fornire le funzionalità necessarie.

(a) ***E-commerce*:** la prima area è dedicata agli acquisti via *web*, sia di tipo classico (ovvero dalla creazione dell'ordine *online* fino alla consegna), sia di tipo *click & collect*. Quest'area fa utilizzo di due moduli:

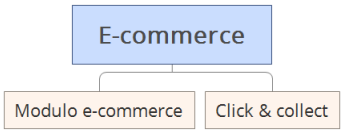


Figura 1.8: Moduli area *e-commerce*

- **Modulo *e-commerce*:** questo modulo sfrutta un'integrazione della piattaforma di *content management* Magento che gestisce la parte di logistica, preparazione dell'ordine e organizzazione della spedizione integrata con il magazzino.
- **Modulo *click & collect*:** si integra con il sistema centrale per la divulgazione anagrafiche e prezzi del cliente, e con gli altri moduli di JStore per la gestione di *fidelity*, *coupon*, promozioni e fatturazione. Il modulo supporta anche *tablet* e PDA. Inoltre, fornisce funzionalità multispesa (l'operatore può preparare contemporaneamente più spese)

e multioperatore (più operatori possono preparare contemporaneamente la stessa spesa).

(b) ***Marketing*:** la seconda area è dedicata alla gestione delle promozioni e dei *coupon*. I moduli di quest'area si occupano di monitorare il flusso di informazioni, generare reportistica e tenere traccia dei *coupon* durante tutti i passaggi di stato.

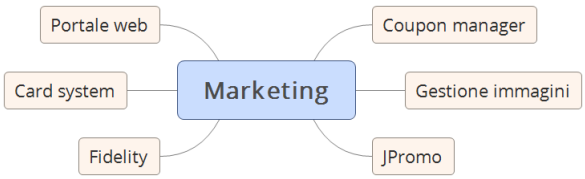


Figura 1.9: Moduli area marketing

- **Modulo *coupon manager*:** permette la gestione e la definizione di caratteristiche di valore, fruizione e validità dei buoni spesa.
- **Modulo gestione immagini:** permette di pubblicare le immagini nei formati richiesti, effettuando il *resize* automatico dell'immagine.
- **Modulo JPromo:** il modulo si occupa della gestione delle promozioni di negozio. Se utilizzato in ambiente centralizzato, permette di generare un pacchetto promozioni di un'intera catena di negozi.
- **Modulo *fidelity*:** gestisce in modo centralizzato le funzionalità relative alle carte fedeltà, come ad esempio accumulo e utilizzo punti e ritiri dei premi.
- **Modulo *card system*:** piattaforma *web* che permette di amministrare le *gift card*.
- **Modulo portale *web*:** modulo che fornisce un canale di comunicazione tra il punto vendita e il cliente fidelizzato, offrendo varie informazioni su promozioni e saldo punti.

(c) **Gestione operativa del punto vendita:** la terza area soddisfa le esigenze di negozio, dalla fatturazione, la tracciabilità e l'inventario fino alla gestione delle comunicazioni con i clienti.

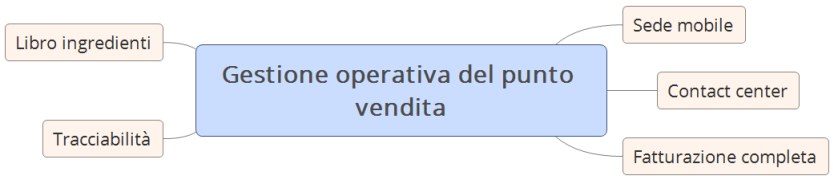


Figura 1.10: Moduli gestione operativa del punto vendita

- **Modulo sede *mobile*:** modulo che gestisce in modo centralizzato l'inventario permanente dei dispositivi mobili.
- **Modulo *contact center*:** piattaforma *web* in grado di gestire la registrazione delle richieste (come ad esempio i reclami dei clienti), le soluzioni proposte e le risposte dei clienti.

- **Modulo fatturazione completa:** permette la rilevazione di fatture e scontrini emessi nei punti vendita. Inoltre, supporta l'invio automatico ad intervalli regolari e personalizzabili degli scontrini dal punto vendita alla sede.
- **Modulo tracciabilità:** modulo che gestisce la tracciabilità dei lotti carne e ittici nei punti vendita.
- **Modulo libro ingredienti:** il modulo permette di memorizzare e riconoscere gli allergeni presenti all'interno dei prodotti. Questa funzionalità consente la pubblicazione del libro degli ingredienti secondo le normative europee.

(d) **Gestione strategica e di monitoraggio:** la quarta e ultima area comprende tutti i moduli che riguardano l'osservazione e il controllo dei sistemi e delle informazioni. Lo scopo di quest'area è definire le strategie di gestione, come la produttività del lavoro dei cassieri e i dati del venduto.

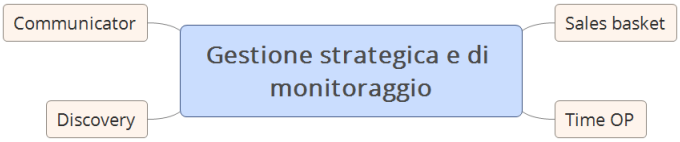


Figura 1.11: Moduli gestione strategica e di monitoraggio

- **Modulo *sales basket*:** strumento che permette di analizzare le informazioni sul venduto a partire dagli scontrini. Supporta l'attivazione di *alert* in base a eventi, con la possibilità di inviare messaggi via SMS o *e-mail*.
- **Modulo *time OP*:** modulo che permette l'analisi delle casse, fornendo dati sulla produttività del lavoro dei cassieri e consentendo di effettuare comparazioni tra i punti vendita della catena.
- **Modulo *discovery*:** si occupa di monitorare e recuperare le informazioni tecniche e di stato dei sistemi, inviando segnalazioni di errori quando necessario.
- **Modulo *communicator*:** modulo per la gestione e il monitoraggio della divulgazione di informazioni in JStore.

2. **i_STORE:** *software* di *back office* che permette di gestire dalla sede le principali esigenze dei punti vendita. È particolarmente adatto alle aziende del settore distributivo, dato il *focus* sulla movimentazione delle merci. Uno dei vantaggi principali di i_STORE è il funzionamento in modo indipendente rispetto ai modelli di casse e bilance installate nel punto vendita.

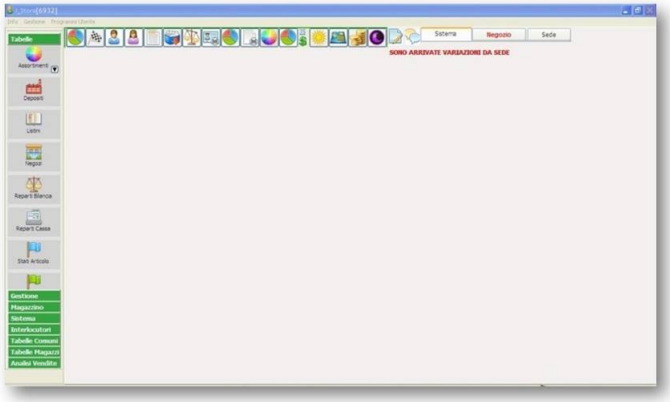


Figura 1.12: *Screenshot* di i_STORE (https://www.ibc.it/).

3. **ARS:** *software* realizzato da NCR e installato da IBC. Questo *software* è installato su casse tradizionali e *self-checkout*, indipendentemente dall'*hardware* e dal sistema operativo. Gestisce l'applicazione delle logiche promozionali durante la vendita e il pagamento.
4. **UPB:** *software* NCR che permette alle casse di offrire vari servizi, come il pagamento di utenze, tasse di abilitazione di carte prepagate e la possibilità di effettuare ricariche telefoniche. Con questo *software* è possibile portare a termine, durante il pagamento in cassa, qualsiasi attività che tipicamente viene svolta dalla tabaccheria o dalle poste.
5. **WinEPTS**: è un *software* NCR per i pagamenti elettronici che consente al *retailer* di rendersi completamente autonomo dalle banche, diminuendo (e in alcuni casi azzerando) le commissioni sui pagamenti effettuati tramite bancomat.
6. **Customer Point:** soluzione IBC installata su Kiosk per permettere al cliente di avere informazioni su prodotti e servizi. Il *software* è anche integrabile su dispositivi *touch*.

Figura 1.13: Un Kiosk (https://goo.gl/PhTxdc).

7. **MobileStore:** *app mobile* per la raccolta remota dei dati del punto vendita. Supporta la lettura dei codici a barre e la memorizzazione di informazioni, effettuandone anche una prima elaborazione direttamente sul terminale.

8. **Libro guida ordini:** *app mobile* disponibile su *tablet* che permette il riordino degli articoli in maniera digitale, sostituendo il tradizionale libro cartaceo ed eliminandone i costi di gestione.

9. **Assistente di negozio:** *app mobile* mirata a dare supporto al cliente nella vendita *no food*. Il *software* permette di dare informazioni e caratteristiche tecniche dei prodotti accedendo direttamente all'anagrafica di sede.

10. **Fidelity:** *app mobile* per la gestione delle carte fedeltà rivolta al cliente. L'installazione dell'applicazione è semplificata in quanto è possibile installarla rilevando direttamente il QR *code*. Attraverso l'*app* il cliente può compiere varie operazioni, come ad esempio prenotare la spesa, gestire i *coupon* e visualizzare gli scontrini.

11. **Yourself:** *app* installabile sui lettori portatili in grado di scannerizzare i codici a barre dei prodotti, in modo da velocizzare il pagamento in cassa. Collegandosi a JStore, l'applicazione permette al cliente di evitare la fila alle casse, offrendogli la possibilità di imbustare direttamente la spesa.

1.1.2 Servizi

Nel mercato odierno, i prodotti non sono l'unica caratteristica che permette ad un'azienda di aver successo. Data la collaborazione, in molti casi pluriennale, tra IBC e i suoi clienti, l'azienda fornisce vari servizi.

1. **Assistenza:** IBC fornisce assistenza *software* e *hardware* attraverso servizi di *call center*, *help desk* e interventi *on-site*. Una delle caratteristiche di forza dell'assistenza IBC è la gestione delle segnalazioni in tempi brevi e garantiti. Per garantire ciò, l'azienda rimane aperta quasi tutto l'anno, compresi i *weekend*.

Figura 1.14: Percorso assistenza IBC.

   Una richiesta di assistenza a IBC attraversa vari stadi:

   (a) ***Call center*:** riceve e registra le richieste di assistenza, indicandone l'urgenza e assegnando un codice identificativo. Successivamente, inoltra la chiamata ai tecnici di *help desk*.

   (b) ***Help desk*:** analizza il problema tecnico dichiarato e fornisce una soluzione per via telefonica. Nel caso la soluzione non sia efficace o l'*help desk* non abbia strumenti o competenze sufficienti, quest'ultimo assegna la risoluzione della segnalazione al reparto IBC più adatto.

   (c) **Assistenza *hardware*:** qualora il problema segnalato sia di natura *hardware*, il personale incaricato provvede ad effettuare le operazioni di manutenzione necessarie e ripristina le normali condizioni di funzionamento presso la sede del cliente.

2. **Laboratorio metrologico:** come descritto in 1.1, IBC è anche laboratorio accreditato presso la CCIAA di Padova per le verifiche metriche degli strumenti di pesatura. I servizi offerti riguardano la verifica periodica prevista per legge di

bilance e strumenti per la pesatura, sia automatici che non. La verifica, oltre che alla scadenza, è obbligatoria anche dopo un'attività di manutenzione che ha rimosso i sigilli dallo strumento.

In aggiunta ai servizi previsti per legge da un laboratorio metrologico, IBC offre in aggiunta servizi ulteriori, come ad esempio il trasporto delle masse necessarie per effettuare le prove, la conservazione dei dati presso l'archivio aziendale e la gestione automatica della periodicità delle scadenze. Inoltre, per legare il servizio di assistenza al servizio di laboratorio, nel caso di bilance che non risultino idonee alla verifica periodica, IBC offre gratuitamente la seconda uscita dell'ispettore metrico per l'intervento di riparazione.

1.2 Organizzazione aziendale

1.2.1 Organizzazione e reparti

IBC, internamente, è divisa in due macro aree:

* **Amministrativa:** area che comprende le funzioni aziendali di amministrazione, risorse umane, organizzazione e finanza.
* **Prodotti e assistenza**: area che comprende lo sviluppo di prodotti, principalmente *software*, e l'assistenza post vendita.

Essendo stato collocato all'interno del team Java 3 per lo svolgimento dello *stage*, ho potuto comprendere meglio il funzionamento dell'area prodotti e assistenza. Per questo motivo mi concentrerò maggiormente sull'analisi di quest'area. In aggiunta a ciò, ho avuto rapporti molto limitati con l'area amministrativa, per cui non sono in grado di fare un'analisi approfondita del funzionamento interno di quest'ultima.

L'area prodotti e assistenza è suddivisa in vari reparti:

* **Reparto Analisi:** reparto che si occupa di comprendere le necessità dei clienti e formulare un'analisi comprensibile dal personale tecnico aziendale. Gli analisti sono il primo passo verso la formulazione di una soluzione *software*. Questo reparto opera sia presso il cliente che presso la sede IBC.
* **Reparto Sviluppo:** reparto che si occupa della realizzazione effettiva del prodotto finale, utilizzando varie tecnologie e linguaggi di programmazione. Il reparto sviluppo è composto da:
  - Tre *team* Java, che si occupano della realizzazione di *web app* e dello sviluppo di JStore.
  - Un *team device*, le cui mansioni sono la realizzazione e la manutenzione delle applicazioni *mobile*.
  - Un *team* che si occupa della realizzazione e manutenzione del *software* delle casse.
* **Reparto *Customer Care* e *Help Desk***: reparto che si occupa dell'assistenza post vendita, sia su prodotti *hardware* che *software*. Ho potuto notare che molte segnalazioni di natura *software* vengono fatte risolvere al reparto sviluppo, spesso provocando ritardi in altre attività.

Nell'elenco mancano reparti come Ricerca e Sviluppo e reparti che si occupano di progettazione. Questa mancanza è dovuta al fatto che IBC non ha dei reparti dedicati per questi scopi, ma si affida a singoli dipendenti (o in ogni caso gruppi molto ristretti) che non costituiscono reparti a sé stanti. Ho potuto notare questa propensione anche in relazione al mio *stage*: per l'azienda, le attività da me svolte rientrano nella funzione ricerca e sviluppo. Fornirò maggiori informazioni su questo punto, insieme ad altri obiettivi aziendali legati agli *stage*, nel capitolo 2.

La tendenza nel far prendere decisioni importanti ad un numero ristretto di persone si sposa con la struttura aziendale che ho potuto rilevare. Anche se giuridicamente IBC si presenta come una S.r.l., nella pratica il suo funzionamento è quello di un'impresa a conduzione familiare, data la consanguineità dei ruoli di più alto livello.

Tra i vantaggi di questo approccio ho potuto notare la facilità nel prendere decisioni anche importanti in tempi relativamente brevi. Se le stesse decisioni avessero dovuto attraversare vari organi aziendali prima di essere prese, sicuramente sarebbe passato molto più tempo e alcune avrebbero riportato una perdita di efficacia. Gli svantaggi principali sono invece:

* Troppe libertà e responsabilità lasciate al personale tecnico, specialmente ai programmatori. Data la mancanza di un reparto che si occupa di progettazione di dettaglio, il programmatore ha troppa libertà decisionale su come implementare la soluzione che gli viene assegnata. Alcune volte ho potuto notare come decisioni prese da un programmatore abbiano causato incomprensioni e ritardi nelle attività di altri *team* di sviluppo.

* Ritardi e impossibilità nel prendere decisioni di natura architetturale quando anche soltanto uno dei (pochi) dipendenti che si occupa di progettazione è assente.

1.2.2 Processi

Il modello di sviluppo che IBC ha adottato si rifà ai principi del modello Agile, consultabili al seguente indirizzo:

http://agilemanifesto.org/iso/it/principles.html

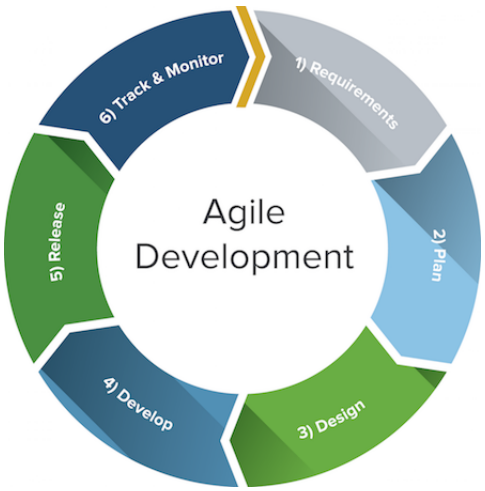


Figura 1.15: Ciclo di sviluppo Agile (https://goo.gl/ESua3X).

I principi che ho percepito come i più seguiti sono:

* "Committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto". Ho constatato che il personale tecnico di alcuni clienti di IBC ha contatti quotidiani con i programmatori dell'azienda, fino ad arrivare ad influenzare il modo con cui le funzionalità sono implementate.

* "Una conversazione faccia a faccia è il modo più efficiente e più efficace per comunicare con il *team* ed all'interno del *team*". Nonostante l'utilizzo di *software* di *ticketing* e la presenza di procedure ben definite per il contatto tra membri di *team* diversi, il personale predilige un rapporto faccia a faccia la maggior parte delle volte. In molti casi ho potuto rilevare che problemi di incomprensioni tra sviluppatori sono stati risolti in modo molto efficace semplicemente parlando di persona.

Tuttavia, IBC non segue i principi alla lettera, ma adatta le sue reazioni a seconda del caso. Ad esempio, l'azienda non accetta cambiamenti sostanziali nei requisiti anche a stadi avanzati dello sviluppo. Nel caso in cui ciò avvenisse, il cliente dovrebbe pagare una somma di denaro per finanziare le ore aggiuntive necessarie a sviluppare (o modificare) il *software* in modo che copra le nuove richieste. Personalmente sono d'accordo con questa linea di pensiero.

Un altro punto di distacco tra il modello adottato da IBC e Agile sono le riunioni. Contrariamente alla prassi adottata dal modello Agile, ovvero riunioni giornaliere (*Daily standup meeting*), IBC tiene riunioni settimanali.

Tenendo conto di questi (e altri) punti, ho potuto percepire che l'azienda non adotta ciecamente il modello Agile, ma ne sfrutta solamente i punti di forza, tralasciando completamente le "cerimonie" che la metodologia è solita portare con sé.

1.2.3 Progetti

Il lavoro all'interno dell'azienda è organizzato a progetti. Ogni *team* si trova a lavorare contemporaneamente a più progetti, richiedendo un cambio di contesto a volte molto rapido.

Per quanto riguarda il *team* Java con cui ho lavorato a più stretto contatto, ho potuto notare che, oltre ai progetti, il *team* doveva gestire anche l'infrastruttura e alcuni moduli di JStore. Nonostante la differenziazione dei *task* assegnati e i molti ambiti gestiti, i membri del gruppo hanno quasi sempre gestito il lavoro in modo organizzato. Questo denota una buona organizzazione all'interno del *team*. Talvolta, più *team* hanno dovuto lavorare su moduli differenti dello stesso progetto e anche in quei casi la comunicazione si è dimostrata efficace.

1.3 Tecnologie a supporto dei processi

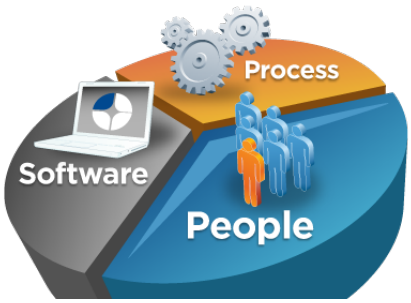


Figura 1.16: Persone, processi e tecnologie (https://goo.gl/KX59QR).

Con l'aumentare delle dimensioni di un'azienda, quest'ultima ha sempre più bisogno di tecnologie che supportino i processi. Infatti, nonostante le persone siano una parte fondamentale per il raggiungimento degli obiettivi aziendali, le tecnologie aiutano sia a rendere il raggiungimento di tali obiettivi ripetibile sia a contenere i costi.

1.3.1 Gestione di progetto

SysAid

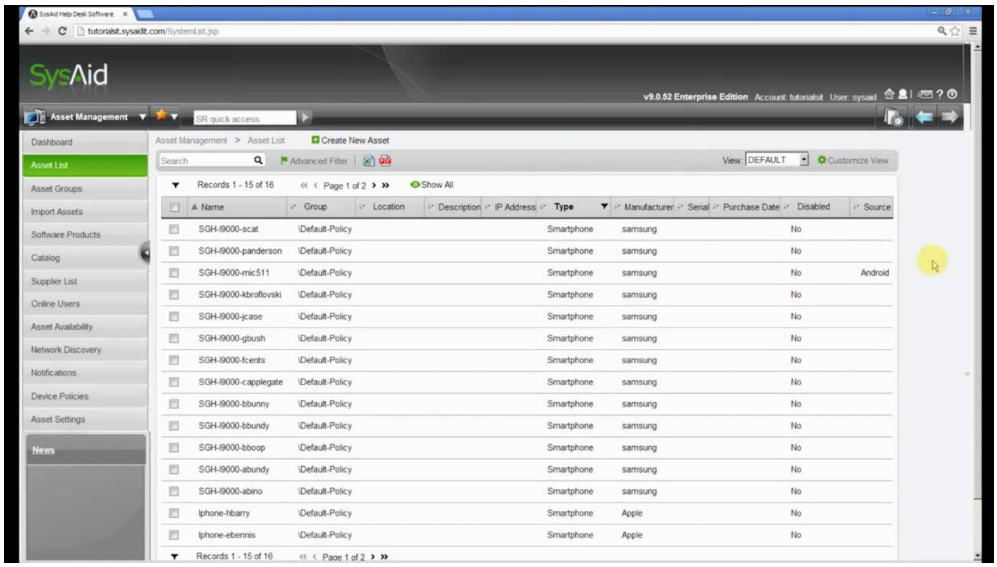


Figura 1.17: *Screenshot* di SysAid (https://goo.gl/cACkfo).

Il principale strumento adottato da IBC per la gestione di progetto è SysAid. SysAid è un *software* di *help desk* completo, utilizzato in quasi ogni reparto IBC. Questo *software* offre varie funzionalità ed è integrabile anche su dispositivi *mobile*. Ecco un elenco delle sue principali caratteristiche:

* **Gestione *ticket*:** SysAid permette di inserire *ticket* e di chiuderli una volta risolti. Questa funzionalità permette al reparto sviluppo di IBC di ricevere *ticket* direttamente dal reparto *customer care* e *help desk* e di risolverli in autonomia, qualora non ci siano incomprensioni o problemi più gravi. Il *software* fornisce anche funzionalità di notifica e di regolazione delle priorità.
* **Gestione *asset*:** funzionalità che permette di rilevare automaticamente e gestire i dispositivi collegati alla rete aziendale.
* ***Knowledge base*:** permette di memorizzare documentazione e guide che consentono la risoluzione dei problemi più frequenti.
* **Gestione *dashboard*:** SysAid offre un rapporto in tempo reale dello stato dei *ticket* e permette di avere *report* di vario genere.

1.3.2 Gestione della configurazione e versionamento

SVN

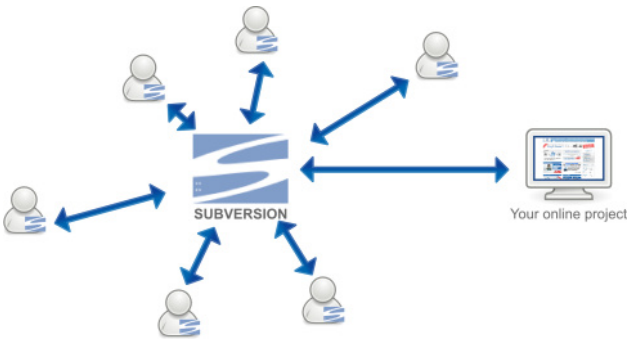


Figura 1.18: Gestione centralizzata di SVN (https://goo.gl/67AyyR).

Il sistema di versionamento utilizzato da IBC è Subversion (d'ora in poi SVN). Questo sistema offre un *repository* centralizzato su cui gli sviluppatori effettuano dei *commit* per pubblicare i cambiamenti da loro prodotti. Alcune caratteristiche di SVN sono:

* L'atomicità dei *commit*: qualora un *commit* dovesse essere interrotto, il *repository* non verrebbe lasciato in uno stato di inconsistenza.
* Un'efficiente gestione dei *file* binari.
* Il *branching* è un'operazione che richiede un tempo indipendente dalla dimensione dei dati.
* La licenza è *open source*.

La centralizzazione di SVN, la caratteristica principale che garantisce la sua semplicità rispetto a soluzioni distribuite come Git, è anche il suo principale svantaggio. Infatti, in caso di impossibilità di accesso al *repository*, è impossibile effettuare *commit* e la gran parte delle funzionalità di versionamento è inutilizzabile. IBC sopperisce a questo rischio fornendo una connessione *internet* affidabile all'interno dei propri stabili.

Maven

Apache Maven è un *software* utilizzato per la gestione della configurazione e delle dipendenze tra un progetto e librerie esterne. In IBC viene utilizzato per gestire i progetti Java, anche se è possibile configurarlo per altri linguaggi. Alla base di Maven c'è il POM (Project Object Model), ovvero un *file* XML che descrive le *directory* di progetto, le dipendenze e definisce come deve avvenire il processo di *build*. Il *download* delle dipendenze è gestito in modo automatico, tipicamente appoggiandosi ad un *repository* centralizzato.

1.3.3 Sviluppo

Wicket

Apache Wicket è un *framework web* lato *server* che utilizza Java per lo sviluppo di *web app*. Il *framework* fornisce un insieme di componenti grafiche pronte all'uso, che permettono un'alta produttività a discapito della personalizzazione. Wicket risulta essere adatto allo sviluppo di *web app* per conto dei clienti di IBC. Infatti, la maggior parte delle funzionalità richieste dai clienti è già implementata e gestita dai componenti di Wicket, minimizzando lo sviluppo di componenti personalizzate.

Gestione di progetto	Config. e versionamento	Sviluppo	IDE	Vari
SysAid	SVN	Java	Eclipse	LibreOffice
	Maven	C++	Android Studio	Skype
	Ant	Wicket		
		Bootstrap		
		Hibernate		

Tabella 1.2: Principali tecnologie utilizzate da IBC.

1.4 Rapporto con l'innovazione

Negli ultimi anni, il mondo del *retail* in Italia sta avanzando in termini tecnologici. Le casse automatiche sono sempre più diffuse all'interno dei punti vendita e, con l'espansione di supermercati e ipermercati, molti clienti sentono la necessità di avere servizi aggiuntivi, come le ricariche telefoniche o i servizi tipici delle tabaccherie.

IBC, per poter fornire soluzioni adatte alle richieste del mercato e dei clienti, necessita di stare al passo dal punto di vista tecnologico. Questo è il motivo per cui intrattiene rapporti con alcuni dei fornitori che offrono tecnologie più avanzate, come ad esempio NCR e Motorola. Queste collaborazioni permettono la fornitura e la manutenzione di *hardware* sempre aggiornato.

Anche lo sviluppo di soluzioni multipiattaforma è uno dei punti di forza dell'azienda. Per poter soddisfare le richieste dei clienti, IBC offre soluzioni sia *desktop* che *mobile* compatibili con tutte le principali piattaforme. L'efficacia nello sviluppo di queste soluzioni multipiattaforma è data dall'utilizzo di linguaggi di programmazione come Java, particolarmente adatto a questo caso d'uso.

Dal punto di vista dei *framework* e delle librerie adottate, l'azienda ha recentemente adottato Apache Wicket che, nonostante sia un *framework* abbastanza vecchio, continua ad essere aggiornato e ad avere un discreto supporto.

L'architettura alla base del prodotto principale di IBC, JStore, è basata su *web service* per garantire modularità ed interoperabilità. A mio avviso, una scelta architetturale di questo genere è matura e rivolta al futuro, in modo da abbandonare le architetture monolitiche del passato.

Una caratteristica negativa dal punto di vista dell'innovazione è la mancanza di implementazione ed esecuzione di *test* automatici. Infatti, IBC non sfrutta alcun tipo di *test* automatico per verificare o validare i *software* prodotti. A mio avviso, una delle conseguenze più gravi di questa mancanza è la regressione, ovvero la possibilità di introdurre errori in *software* precedentemente funzionante senza accorgersene immediatamente.

Ho assistito ad alcuni esempi di regressione durante la mia permanenza presso l'azienda. Ogni occasione ha portato all'impiego di numerose ore persona per risolvere i problemi introdotti. La maggior parte di queste situazione avrebbe potuto essere evitata sfruttando *suite* di *test* opportunamente configurate.

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

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 Java Content Repository (d'ora in poi JCR).
Le differenze tra il modello relazionale (d'ora in poi definito anche come RDBMS, *Relational DataBase Management System*) e il JCR possono essere suddivise in varie aree, come analizzato nel seguente *paper*: `https://goo.gl/ngzgKt`.

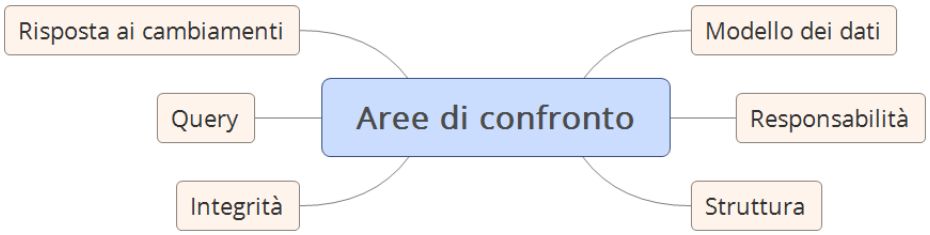


Figura 2.2: Aree di confronto tra RDBMS e JCR.

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.

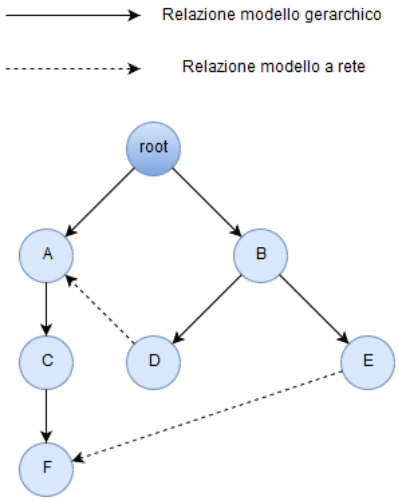


Figura 2.4: Unione del modello a rete con il modello gerarchico (https://goo.gl/ngzgKt).

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*.

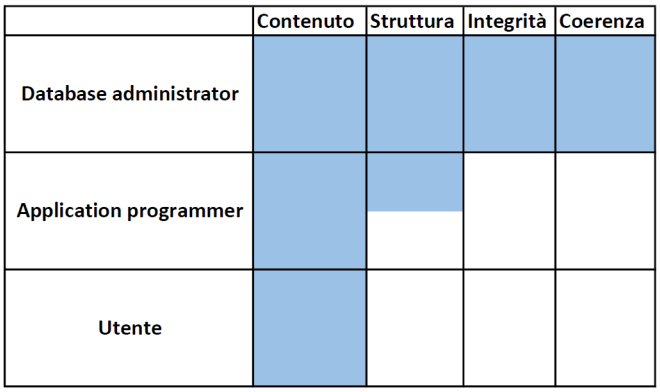


Figura 2.5: Responsabilità dei ruoli in RDBMS (https://goo.gl/ngzgKt).

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*.

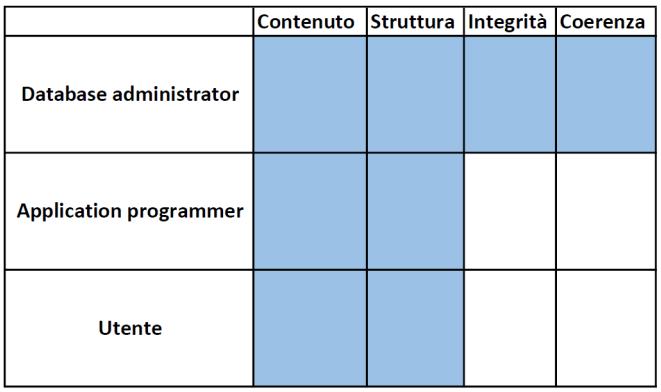


Figura 2.6: Responsabilità dei ruoli in JCR (https://goo.gl/ngzgKt).

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

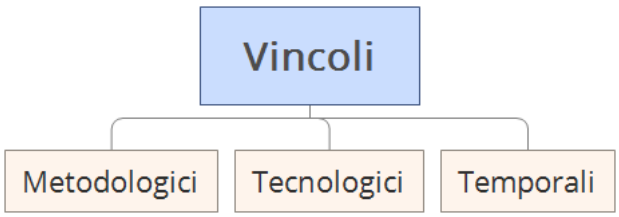


Figura 2.7: Vincoli del progetto

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.

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.

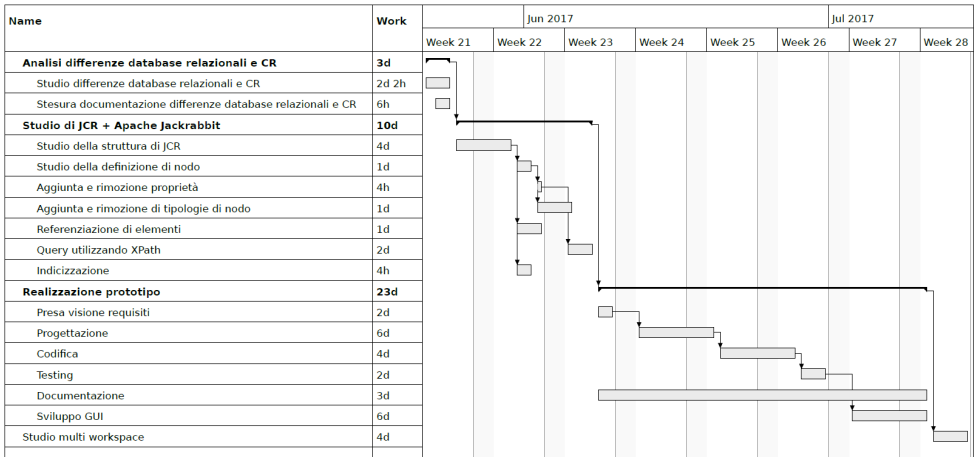


Figura 2.8: Pianificazione temporale.

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.

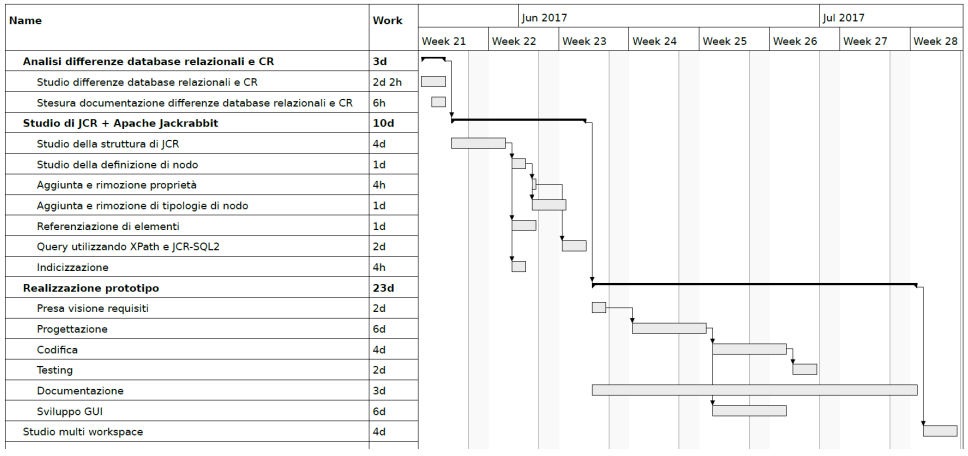


Figura 2.9: Svolgimento attività.

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 Studio del dominio

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 RDBMS e 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.

Per quanto riguarda le motivazioni e le problematiche riguardo la persistenza dei dati dei prodotti commerciali, ho ricavato le informazioni necessarie dalle prime discussioni con il *tutor* e gli altri dipendenti di IBC.

3.2 Studio di fattibilità

Come indicato in 2.2.4, l'unica scelta tecnologica rimanente in quanto non imposta dall'azienda era quella che riguardava la realizzazione della GUI.

* 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.

3.3 Tecnologie utilizzate

3.3.1 Documentazione

* **LibreOffice**
  Suite open source per l'ufficio. Ho utilizzato principalmente Writer per la produzione di testi e Calc per la produzione di fogli di calcolo.

3.3.2 Gestione della configurazione e versionamento

* **Maven**
  Software per la gestione della configurazione, già descritto nella sezione 1.3.2. Ho utilizzato Maven per la gestione delle dipendenze del progetto, utilizzando le risorse presenti presso il sito `https://mvnrepository.com/`.

  Non ho avuto la necessità di produrre configurazioni complicate per il funzionamento del progetto. Le configurazioni di *default* di Maven si sono dimostrate efficaci nella maggior parte dei casi.

* **SVN**
  Sistema di versionamento centralizzato, già descritto nella sezione 1.3.2. Ho utilizzato SVN per effettuare il versionamento sia di documenti che di codice.

  Dato che l'unica persona ad effettuare *commit* ero io, non ho ritenuto necessario utilizzare configurazioni di *branching* complesse. Ho però mantenuto come regola quella di inserire nel *repository* solamente codice funzionante.

* **Tomcat**
  Server web che fornisce una piattaforma per l'esecuzione di applicazioni *web* sviluppate in linguaggio Java.

3.3.3 Sviluppo

* **Eclipse**
  Ambiente di sviluppo integrato utilizzato nel progetto per la scrittura di codice Java, l'avvio del *server* e dei *test*. Eclipse supporta vari *plug-in* per estendere le sue funzionalità.

* **Jackrabbit**
  Libreria che fornisce un'implementazione di JCR. Nel progetto ho utilizzato Jackrabbit per la persistenza dei dati.

* **Wicket**
  Framework per lo sviluppo di *web app* già descritto nella sezione 1.3.3.

* **Wicket-Bootstrap**
  Libreria aggiuntiva di componenti Wicket che utilizzano lo stile predefinito di Bootstrap.

* **JUnit**
  Framework per l'esecuzione di *test* in linguaggio Java.

3.4 Modello di sviluppo

Il modello di sviluppo che ho seguito durante lo svolgimento del progetto è il modello iterativo.

Al contrario del tradizionale modello a cascata, che è strutturato in una sequenza lineare di fasi che vedono l'esecuzione di:

* Analisi dei requisiti.
* Progettazione.
* Sviluppo.
* Accettazione.
* Manutenzione.

una volta soltanto, il modello iterativo procede per iterazioni.

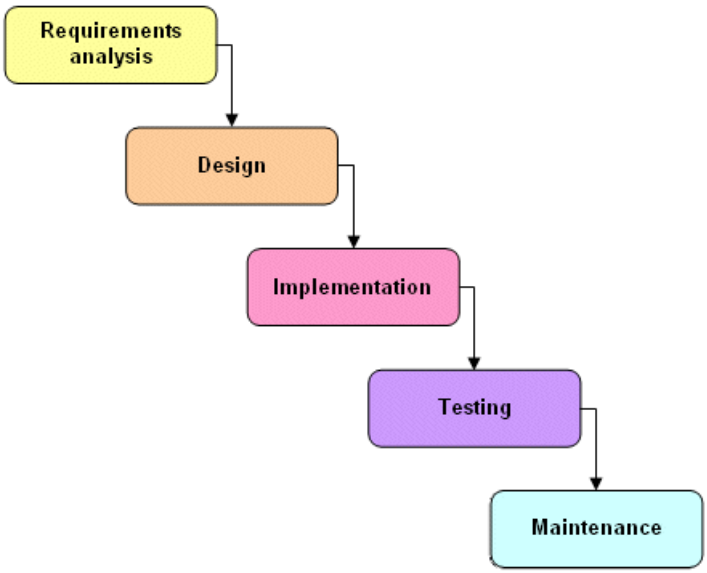


Figura 3.1: Modello a cascata (https://goo.gl/8QTr1T).

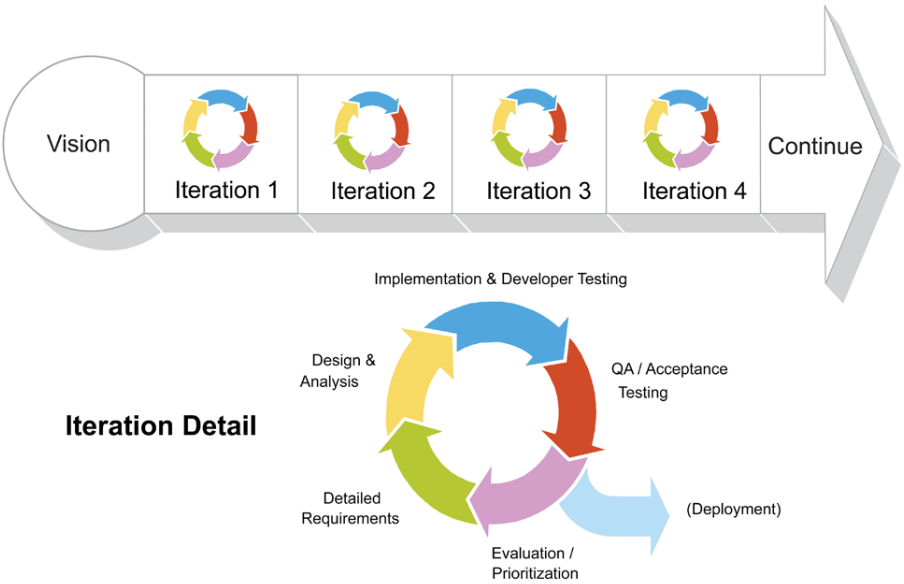


Figura 3.2: Modello iterativo (https://goo.gl/YcTb7w).

Questo modello prevede un'esecuzione ciclica di:

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

Non posso dire di avere utilizzato il modello incrementale in quanto non ho fissato completamente i requisiti e le funzionalità da implementare all'inizio, ma li ho via via ampliati per meglio accomodare le richieste del *tutor*. Inoltre, ho proceduto con l'implementazione dei *test* automatici solamente nell'ultimo periodo del progetto. Un incremento prevede, oltre all'esecuzione di *test* di unità e integrazione, anche la validazione dell'intero sistema prima di passare allo sviluppo dell'incremento successivo.

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 per dare una migliore idea del prodotto in corso di realizzazione, come descritto nella sezione successiva.

3.5 Documentazione e prototipazione

Durante lo *stage* ho prodotto vari documenti. Nonostante io abbia redatto anche documentazione supplementare rispetto ai documenti richiesti, il livello di dettaglio e formalità non è paragonabile a quello tipico dell'ingegneria del *software*. Il motivo di questo fatto è dovuto alla corposità delle librerie da apprendere, sopratutto Wicket, e ad alcuni ritardi avuti durante l'esecuzione dell'attività di *test*.

In ogni caso, ho completato tutta la documentazione richiesta dagli obiettivi definiti nel piano di lavoro. I documenti che ho prodotto sono:

* **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* JSR 170 e JSR 283, 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.
* **Resoconto *test* prestazionali:** documento che fornisce il resoconto dei *test* prestazionali sulla velocità di esecuzione delle *query* in JCR, soprattutto per quanto riguarda le ricerche *full-text*.
* **Manuale utente:** documento che fornisce una guida all'uso del prototipo.
* **Documentazione del codice**: pagina *web* prodotta attraverso l'utilizzo di Javadoc che fornisce descrizioni di classi e metodi.

Oltre alla documentazione, ho prodotto anche dei prototipi da presentare durante gli incontri con il *tutor*. Questo modo di procedere, oltre ad essere utilizzato ampiamente in IBC, 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.

3.6 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.

3.6.1 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.

3.6.2 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".

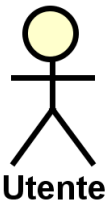


Figura 3.3: Attore utente.

3.6.3 Casi d'uso

Una volta identificato l'attore, ho proceduto a identificare le azioni che egli intendeva svolgere utilizzando il prodotto. In questa sezione rappresento i principali casi d'uso che ho identificato, scendendo nel dettaglio di alcuni dei più importanti.

Panoramica generale

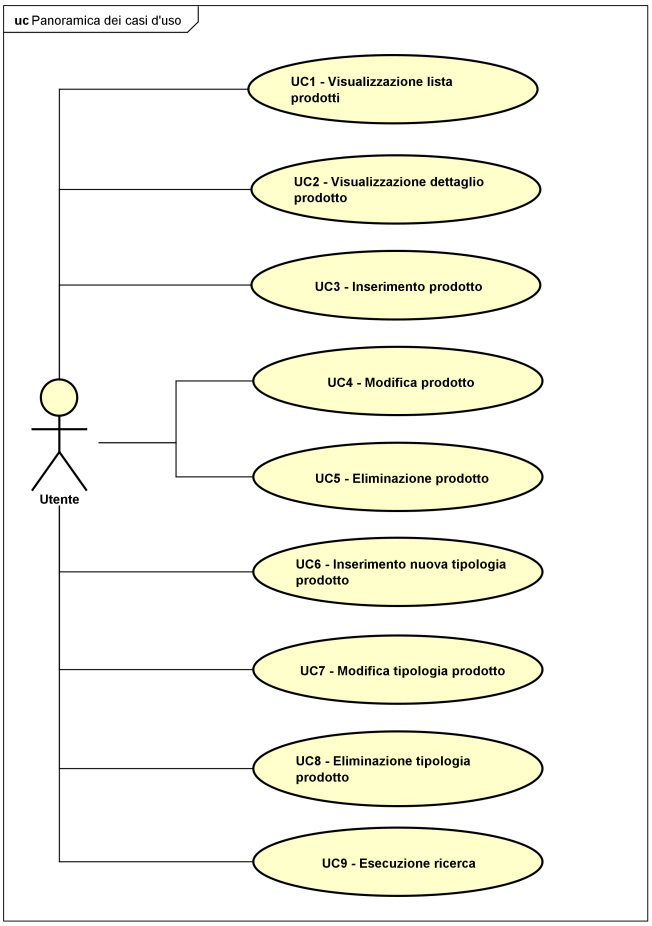


Figura 3.4: Panoramica generale dei casi d'uso.

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

UC2 - Visualizzazione dettaglio prodotto

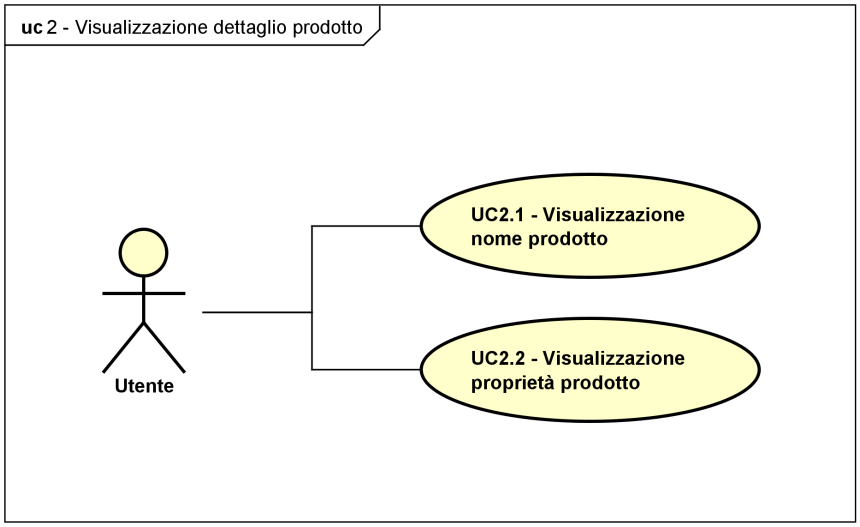


Figura 3.5: UC2 - Visualizzazione dettaglio prodotto.

UC2 - Visualizzazione dettaglio prodotto	
Attori	Utente.
Descrizione	L'utente visualizza le informazioni di dettaglio di un prodotto.
Pre-condizione	L'utente ha aperto l'applicazione.
Post-condizione	L'utente ha visualizzato informazioni di dettaglio di un prodotto.
Scenario principale	1. UC2.1 - Visualizzazione nome prodotto. 2. UC2.2 - Visualizzazione proprietà prodotto.

UC2.1 - Visualizzazione nome prodotto

UC2.1 - Visualizzazione nome prodotto	
Attori	Utente.
Descrizione	L'utente visualizza il nome di un prodotto.
Pre-condizione	L'utente ha aperto l'applicazione.
Post-condizione	L'utente ha visualizzato il nome di un prodotto.
Scenario principale	1. UC2.1 - Visualizzazione nome prodotto.

UC2.2 - Visualizzazione proprietà prodotto

UC2.2 - Visualizzazione proprietà prodotto	
Attori	Utente.
Descrizione	L'utente visualizza le proprietà di un prodotto.
Pre-condizione	L'utente ha aperto l'applicazione.
Post-condizione	L'utente ha visualizzato le proprietà di un prodotto.
Scenario principale	1. UC2.2 - Visualizzazione proprietà prodotto.

UC3 - Inserimento prodotto

UC3 - Inserimento prodotto	
Attori	Utente.
Descrizione	L'utente inserisce un nuovo prodotto.
Pre-condizione	L'utente ha aperto l'applicazione.
Post-condizione	L'utente ha inserito un nuovo prodotto.
Scenario principale	1. UC3 - Inserimento prodotto.

UC4 - Modifica prodotto

UC4 - Modifica prodotto	
Attori	Utente.
Descrizione	L'utente modifica le informazioni di un prodotto.

UC4 - Modifica prodotto	
Pre-condizione	L'utente ha aperto l'applicazione e sta visualizzando le informazioni di un prodotto.
Post-condizione	L'utente ha modificato le informazioni di un prodotto.
Scenario principale	1. UC4 - Modifica prodotto.

UC5 - Eliminazione prodotto

UC5 - Eliminazione prodotto	
Attori	Utente.
Descrizione	L'utente elimina un prodotto.
Pre-condizione	L'utente ha aperto l'applicazione e sta visualizzando le informazioni di dettaglio del prodotto.
Post-condizione	L'utente ha eliminato un prodotto.
Scenario principale	1. UC5 - Eliminazione prodotto.

UC6 - Inserimento nuova tipologia di prodotto

UC6 - Inserimento nuova tipologia di prodotto	
Attori	Utente.
Descrizione	L'utente inserisce una nuova tipologia di prodotto.
Pre-condizione	L'utente ha aperto l'applicazione.
Post-condizione	L'utente ha inserito una nuova tipologia di prodotto.
Scenario principale	1. UC6 - Inserimento nuova tipologia di prodotto.

UC7 - Modifica tipologia prodotto

UC7 - Modifica tipologia prodotto	
Attori	Utente.
Descrizione	L'utente modifica una tipologia di prodotto.
Pre-condizione	L'utente ha aperto l'applicazione. È presente almeno una tipologia di prodotto all'interno dell'applicazione.
Post-condizione	L'utente ha modificato una tipologia di prodotto.

UC7 - Modifica tipologia prodotto	
Scenario principale	1. UC7 - Modifica tipologia prodotto.

UC8 - Eliminazione tipologia prodotto

UC8 - Eliminazione tipologia prodotto	
Attori	Utente.
Descrizione	L'utente elimina una tipologia di prodotto.
Pre-condizione	L'utente ha aperto l'applicazione. È presente almeno una tipologia di prodotto all'interno dell'applicazione.
Post-condizione	L'utente ha eliminato una tipologia di prodotto.
Scenario principale	1. UC8 - Eliminazione tipologia prodotto.

UC9 - Esecuzione ricerca

UC9 - 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. UC9 - Esecuzione ricerca.

3.6.4 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 i principali requisiti da me ricavati. Mantengo la visione ad un alto livello di dettaglio per motivi di brevità.

Ogni requisito è identificato con un codice univoco, secondo la seguente notazione:

R[Importanza][Tipologia][Codice]

L'**importanza** indica quanto gli *stakeholder* valutano quel requisito. Può assumere i seguenti valori:

* **O:** indica un requisito obbligatorio.

* **D:** indica un requisito desiderabile.
* **F:** indica un requisito facoltativo.

La **tipologia** indica la categoria di appartenenza del requisito. Può assumere uno tra i seguenti valori:

* **F:** indica un requisito funzionale;
* **V:** indica un requisito di vincolo.

Il **codice** è un valore numerico progressivo.

Requisito	Tipologia	Descrizione
ROF1	Funzionale	L'utente può visualizzare la lista dei prodotti.
ROF2	Funzionale	L'utente può visualizzare le informazioni di dettaglio di un prodotto.
ROF2.1	Funzionale	L'utente può visualizzare il nome di un prodotto.
ROF2.2	Funzionale	L'utente può visualizzare le proprietà di un prodotto.
ROF3	Funzionale	L'utente può inserire un nuovo prodotto.
ROF4	Funzionale	L'utente può modificare un prodotto precedentemente inserito.
ROF5	Funzionale	L'utente può eliminare un prodotto precedentemente inserito.
RFF6	Funzionale	L'utente può inserire una nuova tipologia di prodotto.
RFF7	Funzionale	L'utente può modificare una tipologia di prodotto precedentemente inserita.
RFF8	Funzionale	L'utente può eliminare una tipologia di prodotto precedentemente inserita.
ROF9	Funzionale	L'utente può eseguire una ricerca.
RDF10	Funzionale	L'utente può associare un'immagine al prodotto.
RFF11	Funzionale	L'utente può associare un catalogo di immagini al prodotto.
ROVF12	Vincolo	Il prodotto deve funzionare su *browser* Firefox 53.0, sistema operativo Linux Mint 18.1 Cinnamon 64bit.

Requisito	Tipologia	Descrizione
ROVF13	Vincolo	Il prodotto deve funzionare su *browser* Chromium 59.0.3071.109, sistema operativo Linux Mint 18.1 Cinnamon 64bit.

Tabella 3.12: Requisiti.

3.7 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.

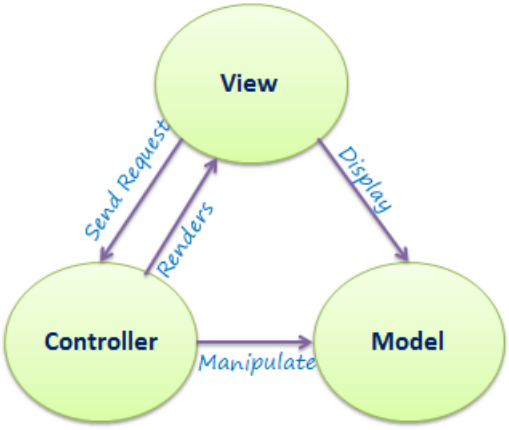


Figura 3.6: Architettura MVC (https://goo.gl/NYKkpQ).

3.7.1 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.

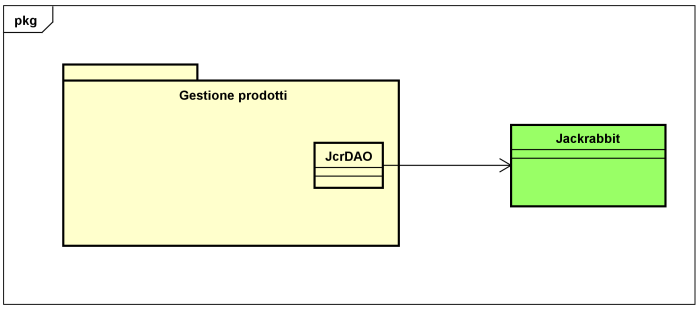


Figura 3.7: Ruolo del DAO nell'architettura.

3.7.2 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.

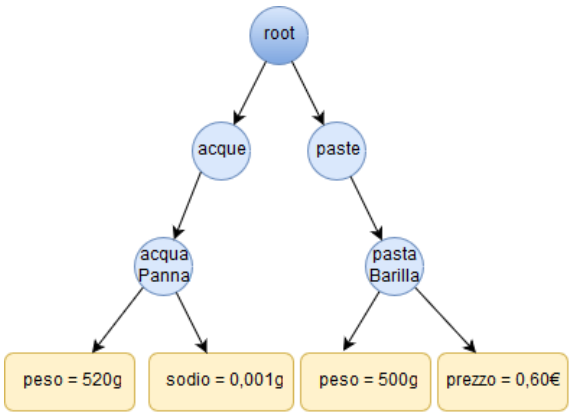


Figura 3.8: Esempio struttura JCR.

3.7.3 *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`.

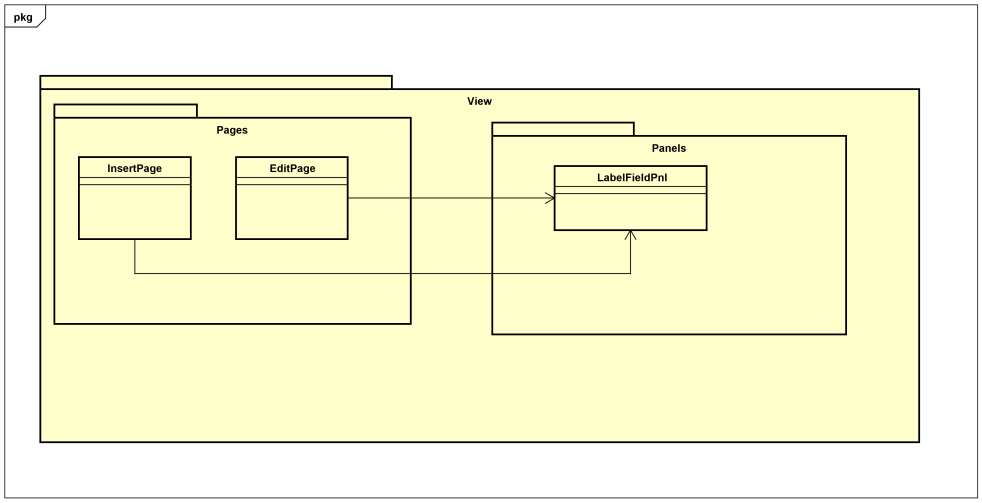


Figura 3.9: Esempio di utilizzo di `Panel` di Wicket.

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.

3.8 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.

3.9 *Test*

Per quanto riguarda l'esecuzione dei *test*, ho utilizzato JUnit4. 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* 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. Fornisco un elenco dei *test* di sistema pianificati ed eseguiti, utilizzando le seguenti abbreviazioni:

* **NI:** non implementato.
* **I:** implementato.
* **NS:** non superato.
* **S:** superato.

Test	Descrizione	Stato	Esito
TS1	Viene verificato che il sistema permetta la visualizzazione dei prodotti.	I	S
TS2	Viene verificato che il sistema permetta l'inserimento di un prodotto.	I	S
TS3	Viene verificato che il sistema permetta la modifica di un prodotto.	I	S
TS4	Viene verificato che il sistema permetta l'eliminazione di un prodotto.	I	S
TS5	Viene verificato che il sistema permetta l'esecuzione di una ricerca.	I	S
TS6	Viene verificato che il sistema permetta la visualizzazione dei risultati di una ricerca	I	S
TS7	Viene verificato che il sistema permetta l'associazione di un'immagine ad un prodotto.	I	S
TS8	Viene verificato che il sistema permetta l'associazione di un catalogo di immagini ad un prodotto.	NI	NS
TS9	Viene verificato che il sistema permetta l'inserimento di una nuova tipologia di prodotto.	NI	NS

Tabella 3.13: Test di sistema.

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 3.14: 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.

3.9.1 *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 3.15: 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 3.16: 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.

3.10 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.

Un altro tipo di verifica che ho impiegato è quella automatica tramite *test*. 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.

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.

3.11 Prodotto finale

Il prodotto finale assume la forma di una *web app single-page*. Procedo a illustrare alcune schermate descrivendo qualche funzionalità offerta.

3.11.1 *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 3.10: *Homepage* con nessun prodotto visualizzato.

Figura 3.11: *Homepage* con alcuni prodotti visualizzati.

3.11.2 Schermata di inserimento

La schermata 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 3.12: Schermata di inserimento prodotto.

3.11.3 Schermata di visualizzazione dettaglio e modifica

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

All'interno di questa schermata 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.

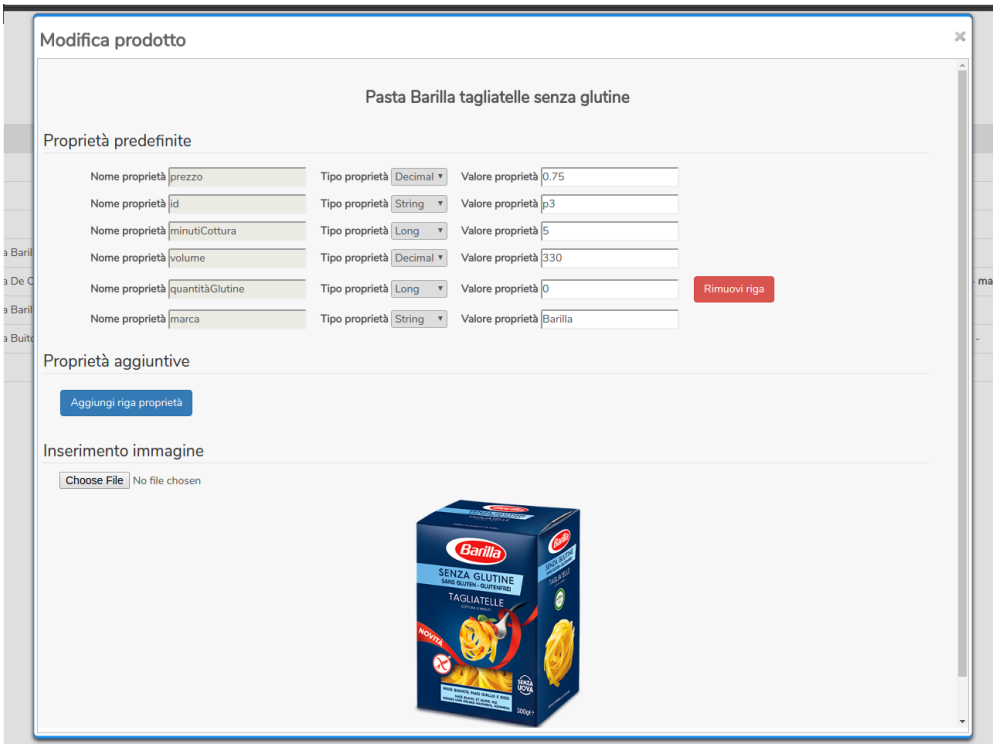


Figura 3.13: Schermata di visualizzazione dettaglio e modifica prodotto.

3.11.4 Schermata di ricerca

Dalla schermata 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 schermata. L'utente è libero di modificare e rimuovere i filtri precedentemente utilizzati, oppure di inserirne di nuovi per raffinare la ricerca.

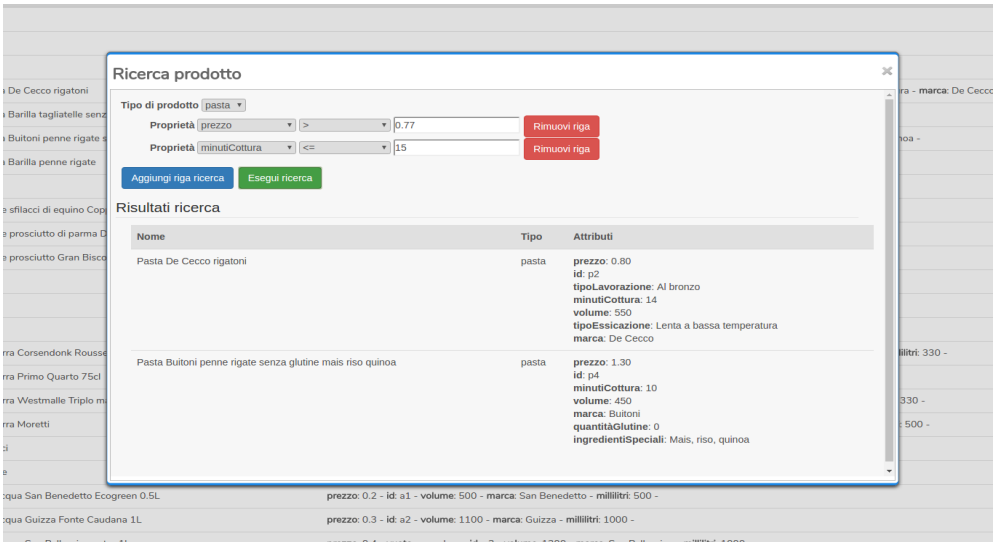


Figura 3.14: Schermata di ricerca, filtro di selezione multiplo.

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.

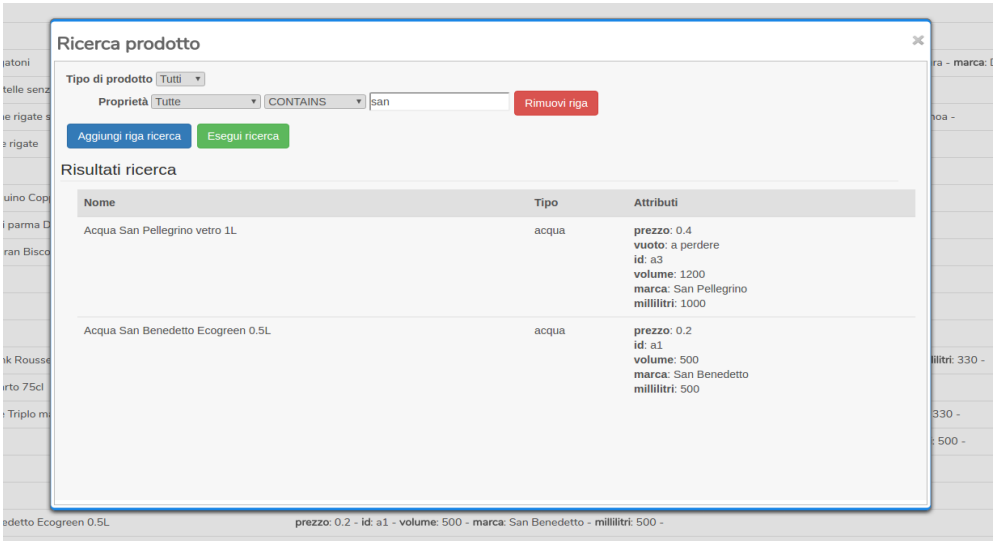


Figura 3.15: Schermata di ricerca, filtro di selezione *full-text*.

Glossario

Back office (ing. dietro ufficio, nel significato di retro-ufficio). Termine che indica la parte di azienda che comprende le attività di gestione operativa, amministrativa e tutte le attività che non riguardano direttamente il cliente. 6, 51

Branch coverage (ing. copertura dei rami). Indica il grado di cammini logici all'interno di un programma coperti durante l'esecuzione dei *test*. 44, 51

Click & collect (ing. prenota e ritira). Metodo di vendita al dettaglio che consiste nella prenotazione, solitamente via *web*, del prodotto da parte del cliente e nel successivo ritiro quando viene segnalata la disponibilità della merce ordinata. La differenza con l'*e-commerce* classico è che la spedizione (in questo caso il ritiro) viene effettuata direttamente dal cliente, senza l'ausilio di corrieri. 4, 51

Fidelity È un insieme di pratiche attuate da un'organizzazione commerciale per favorire la fidelizzazione della clientela attraverso premi, agevolazioni e altri incentivi all'acquisto come la classica raccolta punti. 1, 51

Framework È un'architettura logica di supporto (spesso un'implementazione logica di un particolare *design pattern*) su cui un *software* può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore. iii, 23, 51

Full-text (ing. testo intero). Indica un tipo di ricerca testuale all'interno di un documento o di un *database* in cui il motore di ricerca esamina tutte le parole memorizzate e tenta di trovare un riscontro secondo determinate parole fornite dall'utente. 23, 33, 51

GDO Sigla di Grande Distribuzione Organizzata. Si riferisce al moderno sistema di vendita al dettaglio attraverso una rete di supermercati e ipermercati e di altre catene di intermediari di varia natura. Rappresenta l'evoluzione del supermercato singolo, che a sua volta costituisce lo sviluppo del negozio tradizionale. 2, 51

GUI (ing. *Graphical User Interface*, interfaccia grafica utente). Indica l'interfaccia con cui l'utente interagisce con un *software* attraverso il controllo di oggetti grafici convenzionali. 24, 29, 51

Ingegneria del *software* Corso della Laurea Triennale in Informatica di Padova che richiede lo sviluppo di un impegnativo progetto didattico di gruppo secondo canoni rigorosi di gestione del rapporto cliente-fornitore. 27, 40, 51

Jackrabbit Apache Jackrabbit è una libreria Java open source che fornisce un'implementazione di un Java Content Repository, così come definito dagli standard JSR 170 e JSR 283. iii, 19, 23, 29, 52

Java EE Java Platform, Enterprise Edition. È una specifica impiegata nello sviluppo di applicazioni *web* in linguaggio Java. Inizialmente, la specifica puntava verso la creazione di applicazioni con architetture *multi-tier*, ma grazie alle recenti evoluzioni permette anche di creare applicazioni basate su microservizi. 27, 52

JSR 170 Java Request Specification rilasciato il 17 giugno 2005. Descrive le API per l'utilizzo di Java Content Repository. È conosciuto anche come "JCR v1.0 Specifications". iii, 19, 29, 52

JSR 283 Java Request Specification rilasciato il 25 settembre 2009. Rispetto a JSR 170, aggiunge (e in alcuni casi rimpiazza) alcune API e funzionalità. È conosciuto anche come "JCR v2.0 Specifications". iii, 19, 29, 42, 52

NCR Sigla di National Cash Register. È un'azienda fondata nel 1884 che attualmente opera in gran parte del mondo con soluzioni *retail* e *financial*. Ha sede principale a Dayton (Ohio), U.S.A.; la sede italiana è situata a Milano. Produce principalmente ATM e registratori di cassa. 1, 52

Open source (ing. sorgente aperta). È un termine che indica un *software* di cui i detentori dei diritti rendono pubblico il codice sorgente. Così facendo, altri programmatori possono studiare il codice e apportarvi liberamente modifiche ed estensioni. 13, 30, 52

PDA Sigla di *Personal Digital Assistant*. Indica un computer palmare, ovvero un computer di dimensioni talmente contenute da poter essere portato sul palmo di una mano. Lo schermo del PDA è tattile, in modo da permettere l'interazione con le dita o con un apposito pennino. 4, 52

POS (ing. POS, *Point of Sale*). È il dispositivo elettronico che permette di effettuare pagamenti mediante moneta elettronica, ovvero tramite carte di credito, di debito e prepagate. 1, 52

Proof of concept (ing. prova del concetto). Termine che indica un prototipo o un'incompleta realizzazione di un progetto, in modo da poterne dimostrare la sua fattibilità. 17, 52

Retail (ing. vendita al dettaglio). È una locuzione utilizzata in ambito commerciale per indicare la vendita di prodotti al consumatore finale. È l'ultimo anello della catena di distribuzione, che inizia dal produttore e può passare per un certo numero di grossisti. 1, 18, 52

Stakeholder (ing. portatore di interessi). In economia, indica un soggetto che esercita influenza nei confronti di un'attività economica, come ad esempio un progetto. 38, 52

Statement coverage (ing. copertura delle dichiarazioni). Indica il grado di codice sorgente che viene eseguito durante l'attuazione dei *test*. 44, 52

Stub (ing. abbozzo). Indica una porzione di codice utilizzata in sostituzione di altre funzionalità *software*. È utilizzato sopratutto durante l'esecuzione dei *test* per simulare il comportamento di codice su cui non si sta eseguendo il *test*. 43, 53

Web app (ing. applicazione web). È un'applicazione fruibile tramite *web browser*. iii, 30, 53

Web service Tipo di architettura *software* che si basa sulla comunicazione tra sistemi distribuiti. La comunicazione solitamente avviene solitamente utilizzando linguaggi come XML e JSON, con messaggi trasportati da protocolli *web* (da cui il nome), come HTTP. 15, 53

Bibliografia

Siti web consultati

Articolo "ThoughtWorks Technology Radar Gennaio 2014". URL: https://goo.gl/dfxpaC.

Articolo "What is Java Content Repository". URL: https://goo.gl/8HWDRZ.

Manifesto Agile. URL: http://agilemanifesto.org/iso/it/.

Paper "JCR or RDBMS: why, when, how?" URL: https://goo.gl/ngzgKt.

Sito Apache Jackrabbit. URL: http://jackrabbit.apache.org/jcr/index.html.

Sito Apache Maven. URL: https://maven.apache.org/.

Sito Apache Subversion. URL: https://subversion.apache.org/.

Sito Apache Tomcat. URL: http://tomcat.apache.org/.

Sito IBC. URL: http://www.ibc.it/.

Sito SysAid. URL: https://www.sysaid.com/ita/.

Standard JSR 170. URL: https://jcp.org/en/jsr/detail?id=170.

Standard JSR 283. URL: https://jcp.org/en/jsr/detail?id=283.