

RELAZIONE PROGETTO

# **“Mini-Market Billing System”**

Mangini Alessandro

Pesenti Alessandro

Pesenti Luca

Ingegneria del Software

a.a. 2023/2024



# Indice

Software Engineering Management .....	5
Software Life Cycle .....	6
Configuration Management .....	7
People Management and Team Organization .....	10
Software Quality .....	11
Requirements Engineering .....	13
Modeling .....	18
Software Architecture .....	23
Software Design .....	25
Software Testing .....	28
Software Maintenance .....	30



## Software Engineering Management

Per quanto concerne il piano di progetto, si rimanda al relativo file disponibile contestualmente al repository GitHub: <https://github.com/mangini-a/cash-register/blob/main/project-plan.md>.

L'unica variazione da segnalare rispetto alla versione originaria dello stesso è legata alla decisione, dettata dal graduale processo di espansione delle funzionalità offerte dall'applicativo, di proporre casi d'uso aggiuntivi (tra cui la possibilità di effettuare operazioni di natura gestionale da parte del/dei titolare/i del negozio di riferimento).

## Software Life Cycle

Si è scelto di adottare una forma di sviluppo evolutivo nota come **RAD** (Rapid Application Development), la quale risulta particolarmente adatta in presenza di applicativi primariamente guidati dai requisiti dell'interfaccia utente su cui lavorano team di dimensione limitata.

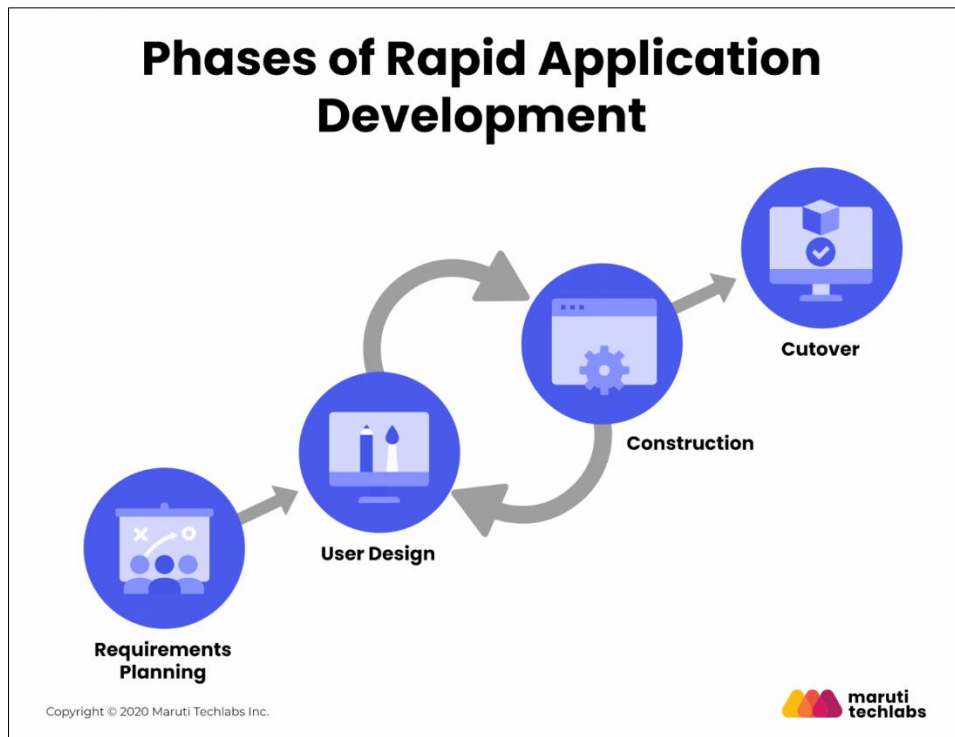


Figura 1: Stadi caratterizzanti il modello di processo adottato

Tale tipologia di processo di sviluppo è stata ritenuta appropriata anche per via della sua capacità di facilitare compiti per cui è necessaria una certa inventiva, oltre che la costante evoluzione dei prototipi: dimostrazioni regolari di esecuzione dell'applicazione possono aiutare a convalidare il design ed il set di funzionalità proposto, consentendo agli utenti di fornire input che possono essere immediatamente tenuti in considerazione oppure incorporati nelle iterazioni successive.

Va inoltre segnalato che testare continuamente i prototipi e raccogliere il feedback degli utenti dà modo di identificare ed affrontare le potenziali problematiche nelle fasi embrionali del ciclo di vita del software, riducendo sensibilmente il rischio di fallimento del progetto.

Tornando alla stretta relazione tra la costruzione della UI ed il modello adottato, quest'ultimo si può rivelare particolarmente utile nello sviluppo di applicativi desktop, nel contesto dei quali gli elementi dell'interfaccia utente possono essere rapidamente integrati mediante framework quali JavaFX e **Swing**: è proprio quest'ultima la casistica alla quale ci si trova di fronte.

# Configuration Management

Il progetto ha tratto grande vantaggio dai frequenti confronti diretti tra i membri del gruppo, che si sono riuniti regolarmente in presenza: questi incontri sono stati l'occasione per esaminare a fondo ciascuna fase dello sviluppo, chiarire i requisiti e discutere quale modello di processo risultasse più adatto alla gestione ed all'implementazione del codice.

Grazie a quest'ultimi è stato possibile definire un workflow ottimizzato, in grado di sfruttare in toto le funzionalità di GitHub per il controllo delle versioni, la condivisione della documentazione ed il mantenimento di una base di codice stabile (durante o alla fine di questi incontri, una sola persona si è occupata di eseguire la maggior parte dei commit).

Il repository, ospitato su GitHub, è stato organizzato in modo da garantire un'ordinata gestione di tutta la documentazione e del codice sorgente; è stato articolato nelle seguenti cartelle principali:

- `./docs/`: include tutta la documentazione di progetto, compreso il piano di lavoro e i diagrammi utili al coordinamento ed alla pianificazione.
- `./code/`: contiene il codice sorgente, pronto per essere eseguito ed aggiornato.

Tale assetto strutturato facilita l'accesso centralizzato ai vari contributi ed agevola il monitoraggio delle modifiche, supportando efficacemente il lavoro collaborativo.

```
commit e847426afb1f5e3e677b83b0d7e8df83d352e8eb
Author: lucapesentii <163282885+lucapesentii@users.noreply.github.com>
....skipping...
commit d8407421e38d719357ef266cad7d619e7df8fb4 (HEAD -> main, origin/main, origin/HEAD)
Author: mangini-a <a.mangini@myyahoo.com>
Date: Mon Nov 11 19:30:58 2024 +0100

    Uploaded the package diagram

    It will be useful to describe the application's architecture.

commit d099e074793ad418b985a6c0e27deb96b3ad29ac
Author: lucapesentii <163282885+lucapesentii@users.noreply.github.com>
Date: Mon Nov 11 18:39:43 2024 +0100

    Uploaded state chart diagram

commit bdaccb890b4788ca585f426c6cde6e44e808ec74
Author: lucapesentii <163282885+lucapesentii@users.noreply.github.com>
Date: Mon Nov 11 18:36:54 2024 +0100

    Delete docs/uml-diagrams/state-chart-diagram.pdf

commit e847426afb1f5e3e677b83b0d7e8df83d352e8eb
Author: lucapesentii <163282885+lucapesentii@users.noreply.github.com>
Date: Mon Nov 11 18:35:33 2024 +0100

    Upload

commit d00efe3bb7c08b83e11fb9e77bc014afe1e7bcea
Author: mangini-a <a.mangini@myyahoo.com>
Date: Sun Nov 10 22:25:17 2024 +0100

    Uploaded three of the UML diagrams

commit 54f6fe50840bfb25b56719b1da34993e93b598df
Author: Alessandro Mangini <145766525+mangini-a@users.noreply.github.com>
Date: Sun Nov 10 22:21:54 2024 +0100

    Update README.md

commit 60f65a47f706dd208533ae89d56afdbfe7a8d73f
Author: mangini-a <a.mangini@myyahoo.com>
Date: Sat Nov 9 22:24:26 2024 +0100

    Changed the visibility of some classes, constructors, and methods

    The possibility of doing so was discovered through the use of the UCDetector tool (installed from the Eclipse marketplace).
```

Figura 2: Parte dello storico dei commit, ottenuto mediante il comando `git log`

Nel corso degli incontri periodici GitHub è stato utilizzato per gestire le attività di progetto per mezzo degli *issue*, che hanno trovato due principali impieghi:

- **Segnalazione di problemi (bug)** riscontrati durante lo sviluppo, con l'obiettivo di garantirne una risoluzione rapida e coordinata.
- **Definizione di nuove funzionalità o modifiche**, suggerite al fine di migliorare il progetto o adeguarlo ai requisiti aggiornati.

Proprio grazie alla gestione centralizzata degli *issue* è stato possibile monitorare agevolmente l'avanzamento delle attività e la distribuzione dei compiti tra i membri della squadra.

L'impiego dei *branch* è stato fondamentale per impostare un lavoro in parallelo, permettendo ad ogni membro del gruppo di sviluppare e testare le proprie modifiche in autonomia: i *branch* separati hanno evitato interferenze tra i vari contributi e preservato la stabilità del ramo di sviluppo principale (*main*), il quale è stato integrato, volta per volta, esclusivamente con modifiche verificate; una volta completato e rivisto il lavoro, ciascun *branch* è stato unito al *main* tramite una *pull request* approvata, assicurando così un elevato livello di controllo della qualità ed una stabilità ottimale.

Per la gestione delle modifiche principali è stata seguita una procedura ben definita, la quale ha ottimizzato il flusso di lavoro e garantito una gestione rigorosa delle versioni, i cui passaggi principali sono stati quelli elencati di seguito:

1. **Apertura di un issue:** ogni modifica viene introdotta aprendo un *issue* dettagliato, che descrive chiaramente il problema da risolvere o la funzionalità da implementare.
2. **Creazione di un branch dedicato:** viene creato un *branch* specifico al fine di lavorare sulla modifica, evitando così di generare conflitti con il *main*.
3. **Creazione di una pull request:** terminato il lavoro sul *branch*, viene generata una *pull request* per sottoporre le modifiche ad una revisione collettiva da parte del team.
4. **Approvazione o richiesta di ulteriori modifiche:** la squadra esamina la *pull request* e decide se approvarla oppure richiederne ulteriori modifiche.
5. **Merge:** una volta approvata la *pull request*, le modifiche vengono integrate nel *main*.
6. **Chiusura del branch:** in seguito all'operazione di *merge*, il *branch* viene chiuso con l'intento di mantenere il repository in ordine.



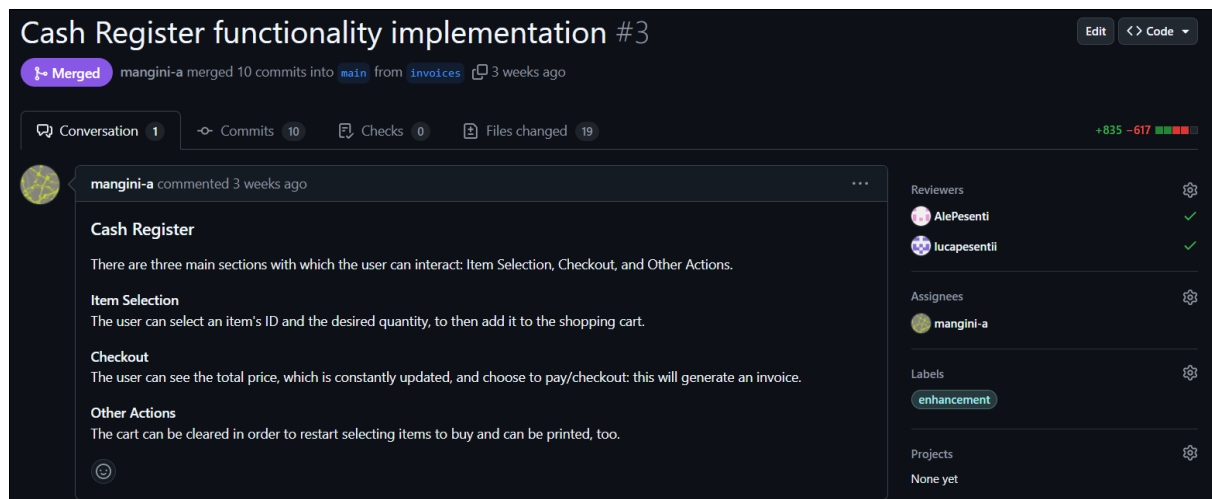


Figura 3: Esempio dimostrativo della strategia adottata

Tale flusso di lavoro ha consentito di mantenere il progetto ben strutturato nel corso della sua evoluzione ed ha permesso ad ogni membro del team di contribuire efficacemente alla realizzazione dello stesso, garantendo stabilità, qualità ed una visione chiara dei progressi ottenuti.

## People Management and Team Organization

Il gruppo è stato organizzato seguendo il modello **SWAT** (Skilled Workers with Advanced Tools), il cui approccio prevede l'impiego di professionisti competenti supportati da strumenti avanzati per massimizzare l'efficienza con cui si porta avanti il lavoro.

Non sono stati assegnati ruoli specifici ai membri del gruppo: in questo modo, ciascun componente ha potuto partecipare a tutte le fasi dello sviluppo del progetto; tale scelta organizzativa ha portato vantaggi significativi, non solo in termini di crescita delle competenze specifiche legate alle tecnologie impiegate nel progetto, ma anche da un punto di vista tecnico.

La distribuzione equilibrata delle attività ha infatti consentito a tutti, specialmente durante la fase di scrittura del codice, di approfondire la propria conoscenza in ambiti chiave nella sfera dello sviluppo software, quali il pattern architetturale **MVC** e l'interazione con un database relazionale per mezzo di **Hibernate ORM**, in grado di mappare dinamicamente le tabelle della base di dati a classi Java appartenenti al layer di persistenza dell'applicativo.

Rifacendosi alle indicazioni del modello SWAT, il team ha adottato canali di comunicazione informali per facilitare il flusso di lavoro; seguono le modalità adottate:

- **Comunicazione sincrona:** si è tenuta per mezzo di incontri di persona in biblioteca e riunioni virtuali su Google Meet.
- **Comunicazione asincrona:** si è espletata tramite l'apertura ed i commenti contestuali agli *issue* su GitHub, oltre che attraverso scambi di messaggi su chat parallele.

Sebbene non siano stati definiti incarichi specifici, i componenti del gruppo si sono concentrati maggiormente sulle aree operative in cui disponevano di competenze approfondite e di cui ritenevano di avere una buona padronanza: ciò ha permesso al team di procedere con maggiore efficienza, valorizzando le abilità individuali all'interno del contesto collaborativo:

	Progettazione	Documentazione	Sviluppo	Testing
Mangini A.	x	x	x	x
Pesenti A.	x	x		
Pesenti L.		x	x	

## Software Quality

Nel corso dello sviluppo del progetto sono stati tenuti in considerazione i fattori di qualità definiti da McCall al termine degli anni '70 del secolo scorso:

**Correttezza:** il software realizzato rispetta i requisiti prioritari stabiliti durante la fase di ingegneria dei requisiti; la metodologia MoSCoW ha guidato l'assegnazione delle priorità, assicurando che le funzionalità critiche fossero implementate (i requisiti considerati meno urgenti sono stati pianificati per future versioni del prototipo).

**Affidabilità:** la stabilità del programma è stata garantita tramite test rigorosi, che hanno confermato la precisione e la puntualità delle funzionalità chiave; il sistema si è dimostrato in grado di eseguire le funzioni richieste con un livello di affidabilità soddisfacente.

**Efficienza:** il codice è stato ottimizzato per limitare quanto più possibile il consumo delle risorse di sistema, sia in termini di memoria che di CPU; tale attenzione nei riguardi dell'efficienza assicura che l'applicazione mantenga alte prestazioni anche in ambienti che dispongono di risorse limitate.

**Integrità:** sono state implementate misure di controllo dell'accesso per proteggere il sistema da utilizzi non autorizzati; la sicurezza dei dati è a sua volta assicurata da autorizzazioni specifiche, così da impedire ad utenti esterni di accedere alle funzionalità o ai dati sensibili.

**Usabilità:** l'interfaccia grafica sviluppata è intuitiva e facile da usare, con comandi autoesplicativi e una disposizione logica delle funzionalità, raggruppate in finestre tematiche; tali scelte progettuali facilitano l'interazione dell'utente con il sistema e ne riducono il tempo di apprendimento.

**Manutenibilità:** grazie all'uso di design pattern il software presenta una struttura chiara e modulare che ne facilita la comprensione e la gestione; la documentazione dettagliata segue lo standard definito da Javadoc e descrive le funzionalità dei metodi chiave, rendendo più semplice la manutenzione e l'aggiornamento futuro del codice sorgente.

**Testabilità:** nonostante la presenza di un database integrato (disk-based), il software è stato reso facilmente testabile mediante metodi specifici (contrassegnati da apposite annotazioni della JPA) in grado di ripristinare il DB di testing (in-memory) al termine di ciascuno dei test di unità effettuati, garantendo flessibilità durante la verifica funzionale e senza alterare i dati reali.

**Flessibilità:** l'architettura ben strutturata dell'applicazione permette di apportare modifiche al codice in maniera semplice e veloce, facilitando l'adattamento del software a nuove esigenze o requisiti in evoluzione.

**Portabilità:** il linguaggio Java offre di per sé un'elevata portabilità grazie all'utilizzo della JVM ed al bytecode, così come il fatto di ricorrere a Maven per la gestione delle dipendenze; ciò consente di eseguire l'applicazione su diverse piattaforme, facilitando il trasferimento tra vari ambienti software e hardware.

**Riutilizzabilità:** la progettazione basata sul noto pattern MVC (Model-View-Controller) favorisce il riutilizzo dei componenti; grazie alla modularità, le componenti sviluppate possono essere riutilizzate in applicazioni analoghe, ad esempio per creare una web app client/server evitando di doversi trovare a riprogrammare la logica di business o la persistenza dei dati.

**Interoperabilità:** il software è stato progettato per integrarsi facilmente con altre applicazioni; l'utilizzo di standard e tecnologie ampiamente adottati incentiva l'interazione con altri sistemi, facilitando lo scambio di dati e l'integrazione con quest'ultimi.

# Requirements Engineering

Si riporta, a seguire, il risultato della fase di analisi dei requisiti, ovvero la specifica dei requisiti: tale documento costituisce la base a partire da cui si può procedere con la fase di progettazione, oltre che un punto di ancoraggio rispetto al quale possono essere giustificati i passaggi successivi.

Si è cercato di proporre un documento il più possibile conforme a quanto stabilito dallo standard IEEE 830 (1998):

---

## **Introduzione**

- *Obiettivo*
- *Scopo*
- *Definizioni, acronimi ed abbreviazioni*
- *Riferimenti*
- *Panoramica*

## **Descrizione generale**

- *Prospettiva del prodotto*
- *Funzioni del prodotto*
- *Caratteristiche dell'utente*
- *Vincoli*
- *Presupposti e dipendenze*

## **Requisiti specifici**

---

## *Introduzione*

### **Obiettivo**

L'obiettivo del sistema è sviluppare un'applicazione desktop che permetta a un piccolo esercizio commerciale (ad esempio, un negozio di alimentari o un minimarket) di gestire le operazioni principali legate alla vendita e alla gestione del negozio.

L'applicativo sarà realizzato utilizzando **Java Swing** per l'interfaccia utente e **Hibernate ORM** per la gestione della persistenza dei dati, seguendo il **pattern architetturale MVC (Model-View-Controller)**.

Tale approccio garantisce una chiara separazione delle responsabilità tra la gestione dei dati, la logica applicativa e la visualizzazione delle informazioni, migliorando la manutenibilità e l'estendibilità del sistema, il quale si occuperà delle seguenti operazioni:

- Emissione degli scontrini
- Gestione dell'inventario (ossia, degli articoli a magazzino)
- Amministrazione del personale del negozio
- Visualizzazione dello storico delle transazioni e dei profitti giornalieri

### **Scopo**

Lo scopo di questo documento è fornire una specifica dettagliata dei requisiti per il *Mini-Market Billing System*: sarà infatti utilizzato come base per la progettazione e lo sviluppo del sistema, oltre che come punto di riferimento per valutare la correttezza e la completezza del prodotto finale.

### **Definizioni, acronimi ed abbreviazioni**

- **Hibernate ORM**: Object-Relational Mapping framework per Java, utilizzato per la gestione della persistenza dei dati;
- **Java Swing**: Libreria Java per la creazione di interfacce grafiche;
- **MVC**: Model-View-Controller, pattern architetturale che separa il layer di persistenza dei dati (Model), la logica di controllo (Controller) e la visualizzazione (View);
- **Entity**: Oggetto mappato su una tabella del database relazionale in uso;
- **Controller**: Componente software che gestisce la logica dell'applicazione e si occupa della comunicazione tra l'interfaccia utente (al di sopra di esso) e le entità persistenti (al di sotto di esso).

## Riferimenti

- IEEE 830-1998: Standard per la descrizione dei requisiti software
- Documentazione ufficiale di Hibernate 6 (version 6.6.3.Final):  
[https://docs.jboss.org/hibernate/orm/6.6/introduction/html\\_single/Hibernate\\_Introduction.html](https://docs.jboss.org/hibernate/orm/6.6/introduction/html_single/Hibernate_Introduction.html)
- Documentazione ufficiale della piattaforma Java (Standard Edition) 17:  
<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

## Panoramica

Il documento di specifica descrive dettagliatamente i requisiti funzionali e non funzionali del sistema *Mini-Market Billing System*; si forniranno dettagli relativi all'ambiente operativo, alle funzionalità principali ed agli utenti previsti, unitamente alle ipotesi ed ai vincoli che influenzano lo sviluppo.

### *Descrizione Generale*

## Prospettiva del prodotto

*Mini-Market Billing System* è un'applicazione desktop destinata all'uso interno da parte di un negozio di alimentari di medio-piccole dimensioni, una bottega o un minimarket: si tratta di un sistema indipendente, che non richiede l'integrazione di app di terze parti.

Esso consentirà al personale di gestire le operazioni quotidiane di vendita e la contestuale registrazione di scontrini, oltre ad offrire la possibilità di mantenere aggiornato l'inventario e monitorare le prestazioni (in termini economici) del negozio.

## Funzioni del prodotto

- **Emissione di scontrini:** Registrazione degli acquisti, calcolo dell'ammontare della transazione e generazione (opzionale) di un documento riepilogativo;
- **Gestione degli articoli a magazzino:** Inserimento, modifica, eliminazione e visualizzazione degli articoli disponibili;
- **Gestione dello staff:** Inserimento ed aggiornamento delle informazioni relative al personale (in particolare, le credenziali di accesso al sistema), inclusa la definizione dei ruoli;
- **Visualizzazione dello storico delle transazioni:** Accesso ai dati storici delle vendite e calcolo dei profitti giornalieri, graficati a schermo nella forma di un istogramma.

## Caratteristiche dell'utente

Il sistema può essere utilizzato da due tipologie di utenti:

- **Operatore di cassa:** Ha la sola facoltà di emettere scontrini, avendo accesso unicamente al registratore di cassa.
- **Manager:** Ha accesso a tutte le funzionalità previste, ovvero al registratore di cassa, alle schermate deputate alla gestione del personale e dell'inventario ed a quella destinata alla visualizzazione dello storico delle transazioni e dell'istogramma corrispondente.

## Vincoli

Il sistema è tenuto a rispettare i seguenti vincoli:

- Architettura basata sul noto pattern **MVC**, al fine di garantire una netta separazione delle responsabilità dei componenti software;
- Utilizzo della libreria **Java Swing** per la realizzazione dell'interfaccia utente (UI) e del framework **Hibernate ORM** per la gestione della persistenza dei dati;
- Utilizzo di un DBMS relazionale per la gestione della base di dati integrata (salvata su disco, all'interno del workspace che include il progetto).

## Presupposti e dipendenze

- Il laptop o PC in dotazione al minimarket disporrà di una connessione stabile al database locale, in grado di scongiurare la necessità di preoccuparsi della stessa;
- Gli utenti, agevolati dalla curva di apprendimento particolarmente semplificata, avranno una conoscenza di base dell'utilizzo del programma, a cui accedere direttamente dal desktop;
- Il sistema sarà installato su una macchina con il Java Runtime Environment preinstallato.

## *Requisiti Specifici*

### Requisiti Funzionali

- **Emissione Scontrino**
  - Il sistema deve consentire al cassiere di selezionare articoli dal magazzino, specificando la quantità desiderata dal cliente al momento dell'acquisto;
  - Calcolo automatico del totale e applicazione di eventuali sconti;
  - Possibilità di eliminare gli articoli prima della conferma finale dello scontrino;



- Registrazione della transazione nel database, associata al cassiere che ha effettuato la vendita.
- **Gestione Articoli a Magazzino**
  - Inserimento di nuovi articoli con dettagli quali nome, quantità, prezzo unitario e categoria di riferimento;
  - Modifica delle informazioni relative agli articoli esistenti;
  - Visualizzazione dell'elenco dei prodotti a disposizione.
- **Gestione dello Staff**
  - Aggiunta di nuovi membri dello staff con nome, cognome, password e ruolo;
  - Aggiornamento delle informazioni del personale, incluse le credenziali di accesso al programma e la modifica del ruolo da Cassiere a Manager (promozione);
  - Rimozione di membri dello staff che non sono più operativi;
  - Visualizzazione dell'elenco del personale con dettaglio dei ruoli.
- **Visualizzazione Storico Transazioni**
  - Accesso all'elenco delle transazioni registrate;
  - Visualizzazione a schermo di un istogramma in grado di chiarire immediatamente l'ammontare giornaliero dei profitti.

## **Requisiti Non Funzionali**

- **Usabilità:** L'interfaccia utente deve essere semplice e intuitiva, riducendo al minimo i click necessari per completare un'operazione;
- **Affidabilità:** Il sistema deve garantire la consistenza dei dati anche in caso di interruzioni inaspettate;
- **Performance:** L'attuazione delle operazioni quotidiane (inserimento di articoli a magazzino, modifica del prezzo unitario di quest'ultimi) deve essere rapida, con tempi di risposta inferiori a 2 secondi per le operazioni principali;
- **Sicurezza:** Le informazioni relative agli utenti ed alle transazioni devono essere protette tramite autenticazione e gestione dei permessi.

Tale descrizione rappresenta una linea guida per la progettazione e lo sviluppo del sistema, assicurando che tutte le funzionalità richieste siano chiaramente identificate e documentate.

Una fase essenziale lungo lo sviluppo del progetto è stata quella della modellazione: attraverso la creazione di diagrammi UML (sia di natura statica che dal taglio comportamentale) siamo stati in grado di definire e strutturare aspetti del sistema molto delicati in modo chiaro e organizzato.

Mentre *Class Diagram* e *Package Diagram* verranno presentati nel prossimo capitolo (dedicato all'architettura software dell'applicativo), si dettagliano qui i seguenti diagrammi dinamici:

- *Activity Diagram*
- *Sequence Diagram*
- *State Chart Diagram*
- *Use Case Diagram*

### Activity Diagram

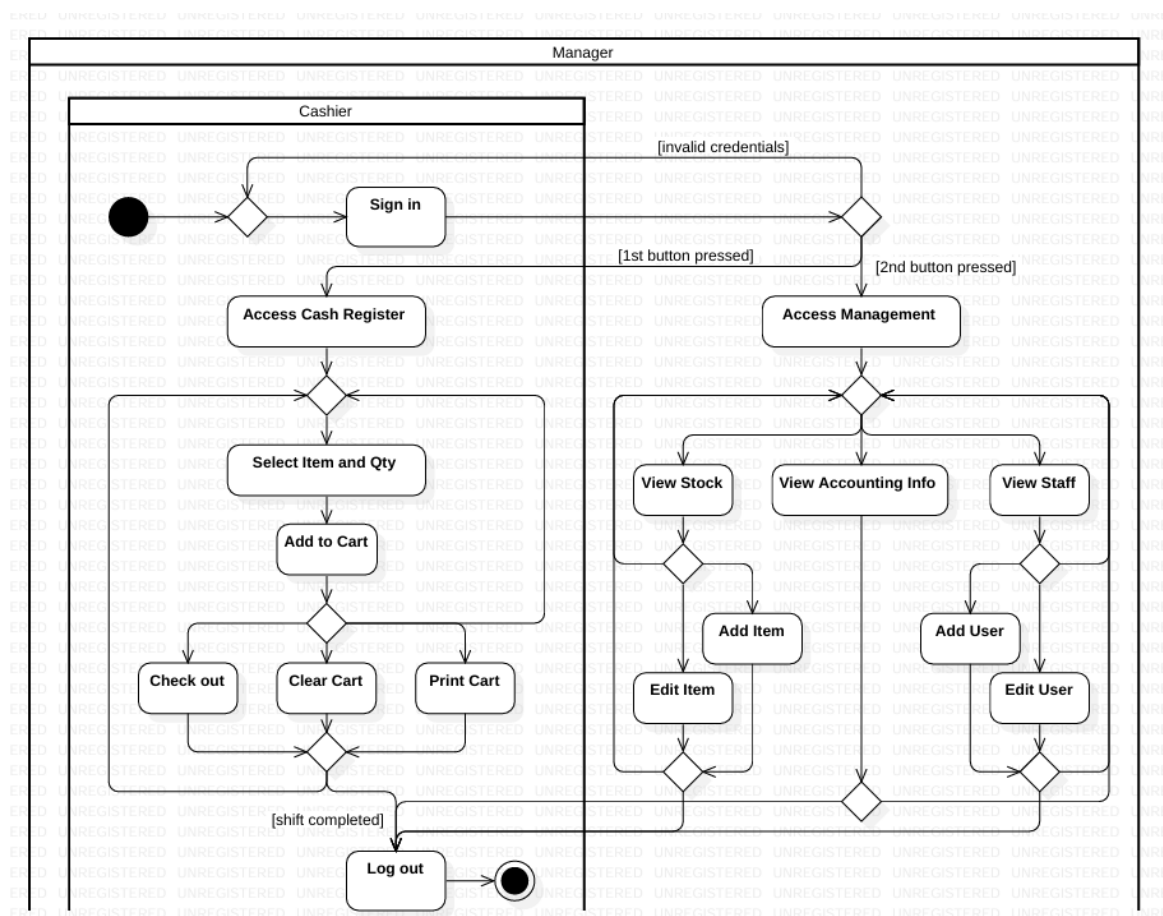


Figura 4: Diagramma di attività

In tale diagramma dinamico viene mostrato il flusso di operazioni eseguibili da chi utilizza il programma: vengono infatti rappresentate tutte le possibili sequenze/combinazioni di azioni, decisioni, eventi e risultati, evidenziando il percorso che queste delineano all'interno del sistema.

## Sequence Diagram

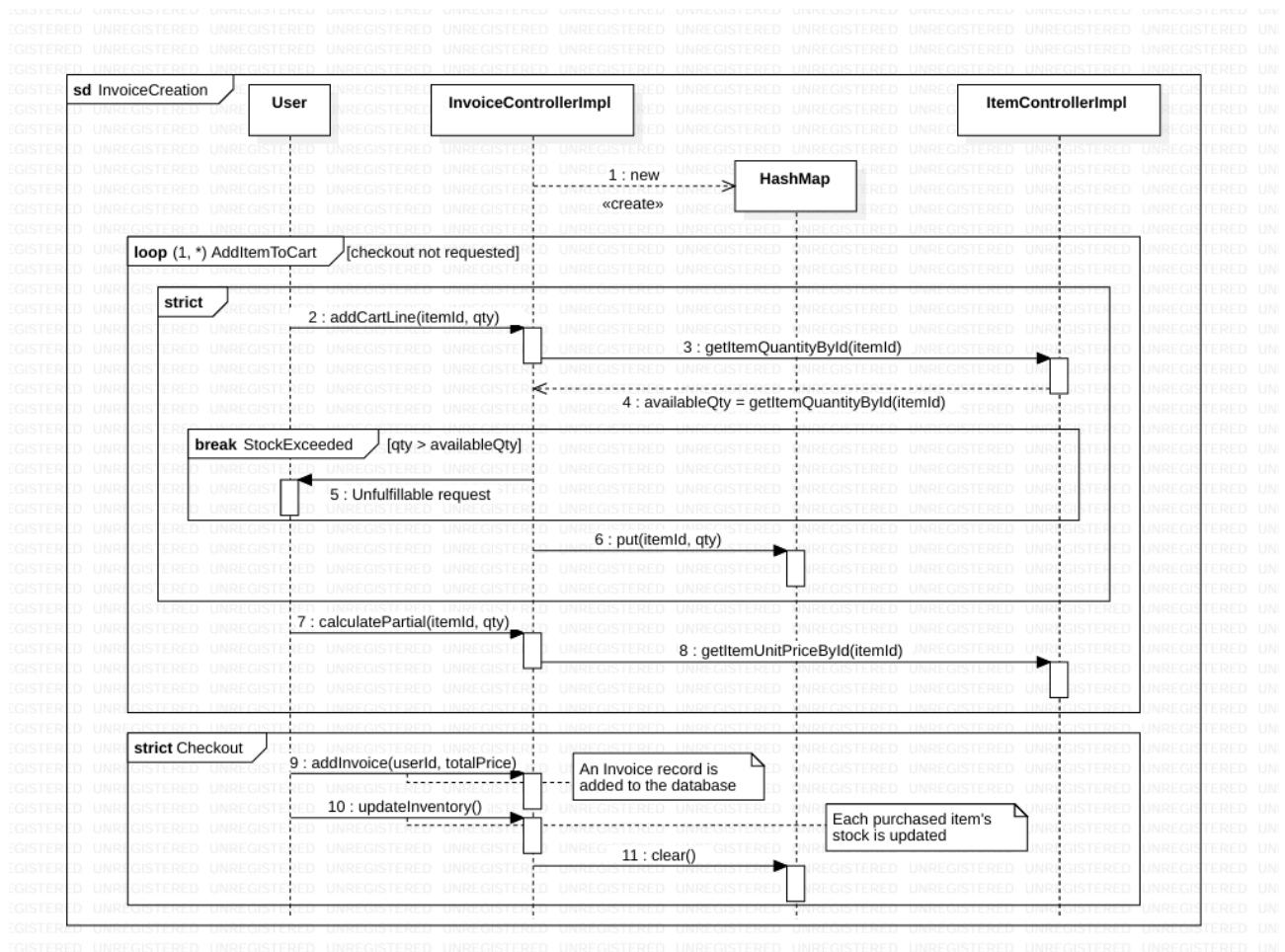


Figura 5: Diagramma di sequenza

Nel diagramma sopra riportato si descrivono le interazioni tra i vari componenti del sistema che concorrono nella creazione di uno scontrino (dal punto di vista dell'utente finale, ci troviamo nella schermata dedicata al registratore di cassa).

Si può notare una serie di frammenti specifici in grado di garantire che le azioni dell'utente ed i processi del sistema siano gestiti in modo prevedibile e corretto, la cui funzione è spiegata di seguito:

### 1. Aggiunta di Articoli al Carrello (*AddItemToCart* loop):

L'utente interagisce con il sistema per aggiungere determinati articoli al carrello della spesa, indicandone sia l'identificativo che la quantità desiderata.

Viene quindi verificato che la quantità richiesta dal cliente del negozio sia disponibile, ricorrendo al metodo *getItemQuantityById()*: se l'esito è positivo, l'articolo viene effettivamente aggiunto al carrello, registrandone ID e quantità in un'apposita struttura dati (HashMap).

Viene calcolato dunque il contributo, in termini di costo, della quantità selezionata del prodotto, andando a processarne il prezzo unitario ottenuto attraverso il metodo *getItemUnitPriceById()*.

## 2. Condizione di errore (*StockExceeded break*):

Se la quantità richiesta dal cliente supera quella disponibile a magazzino, il sistema solleva un errore di richiesta (*Unfulfillable request*), interrompendo l'operazione di aggiunta al carrello della spesa e notificando all'utente la massima quantità che può attualmente selezionare.

## 3. Processo di Checkout (*Checkout strict*):

Al momento del checkout, viene generato un record di fattura nel database richiamando il metodo *addInvoice()*, che va a registrare l>ID dell'utente loggato ed il totale dell'acquisto.

Viene, di conseguenza, aggiornato l'inventario: per ciascun articolo acquistato, le scorte vengono decurtate al fine di riflettere la quantità venduta (metodo *updateInventory()*).

La struttura dati che rappresenta il carrello viene resettata al termine dell'operazione di checkout per mezzo del metodo *clear()*, così da preparare il sistema per il prossimo utilizzo.

```

graph TD
    Start(( )) --> Login
    Login["Login<br/>do/Collect credentials<br/>exit/Authenticate"] -- "Incorrect credentials" --> Login
    Login -- "Login successful" --> Menu
    Menu["Menu<br/>do/View menu items<br/>do/Place order"] -- "Back" --> CashRegister
    Menu -- "[Is MANAGER]" --> Management
    Management["Management<br/>do/Add a new Item<br/>do/Edit item's details"] -- "Back" --> Menu
    CashRegister["Cash Register<br/>do/Add items to cart<br/>do/Checkout<br/>do/Print cart"] -- "Back" --> Menu
    Management --- Junction(( ))
    Staff["Staff<br/>do/Add a new User<br/>do/Edit User's credentials"] --- Junction
    Accounting["Accounting<br/>do/View transaction list<br/>do/View day-to-day profit"] --- Junction
    Junction --- Menu
  
```

Il diagramma include i seguenti stati chiave:

1. **Login:** Stato iniziale, in cui l'utente ha modo di inserire le proprie credenziali al fine di

- accedere alle schermate operative che contraddistinguono il programma: nell'eventualità in cui le credenziali inserite risultino errate, si permane in questo stato; con credenziali corrette si può invece passare allo stato '**Menu**'.
2. **Menu:** Da tale stato si diramano diverse opzioni, a seconda del ruolo dell'utente che si è appena autenticato:
- a. **Manager:** Ha accesso a tutte le funzionalità previste:
    - i. Staff, che permette di gestire il personale, andando ad aggiungere nuove persone, rimuovere dipendenti dal loro incarico, modificarne le credenziali;
    - ii. Stock, che consente di aggiungere nuovi articoli, modificare i dettagli di quelli a magazzino, rimuovere prodotti che si vuole cessare di proporre;
    - iii. Accounting, pensata per visualizzare lo storico delle transazioni effettuate e i profitti giornalieri per mezzo di un istogramma;
    - iv. Cash Register, la funzionalità di base che consente di aggiungere articoli al carrello, procedere al checkout e generare/stampare lo scontrino.
  - b. **Cassiere:** Ha accesso unicamente all'appena descritto registratore di cassa (la cui schermata prende il nome di Cash Register); offre la possibilità, come Manager, di tornare al menu principale tramite l'opzione "Back", che facilita la navigazione.

## Use Case Diagram

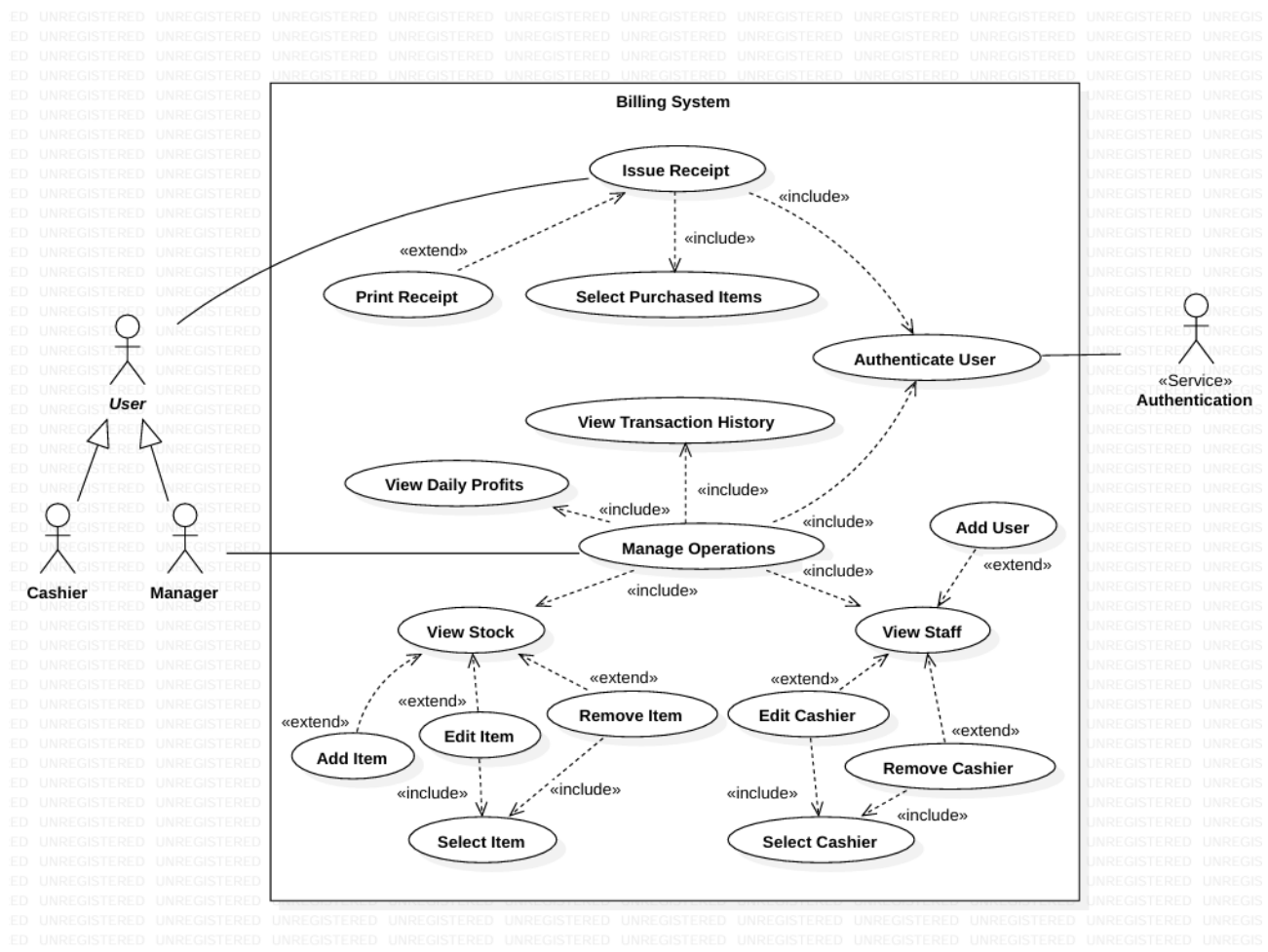


Figura 7: Diagramma dei casi d'uso

In tale diagramma, a sua volta di natura dinamica, sono state rappresentate le funzionalità messe a disposizione degli attori coinvolti; si possono individuare, in particolare, due attori umani ed un attore di servizio:

- **Cashier:** Ha unicamente modo (soltanto dopo essersi autenticato) di utilizzare il registratore di cassa, con la finalità di emettere scontrini ed, eventualmente, di stamparli sotto forma di documenti di testo;
- **Manager:** Oltre ai casi d'uso eseguibili dagli operatori di cassa, ha la possibilità di accedere a funzionalità di natura gestionale, quali la gestione dell'inventario e del personale ed il monitoraggio delle transazioni effettuate in corrispondenza di ciascuna vendita.

Questi due attori possono essere generalizzati da **User**, in quanto entrambi devono effettuare l'accesso al sistema per poter utilizzare il programma e svolgere il loro lavoro; l'ultimo attore è il servizio di autenticazione, che si occupa della verifica e della memorizzazione delle credenziali.

## Software Architecture

L'app sviluppata si basa sul pattern architetturale **MVC** (Model-View-Controller), il quale si presta ad applicazioni interattive con un'interfaccia utente flessibile, la cui separazione dalla logica e dalla persistenza dei dati è auspicabile per via dei continui adattamenti per essa previsti.

Sinteticamente parlando, la componente Model si occupa della persistenza dei dati interagendo con il database, mentre la componente View si prefigge di mostrare l'output all'utente in modo chiaro e comprensibile; la componente Controller, d'altro canto, lavora a metà tra le precedenti, acquisendo l'input fornito dalla View e restituendolo alla stessa (ove necessario, non prima di averlo processato con l'aiuto della prima).

Si noti che ognuna di queste componenti essenziali è stata implementata in modo modulare, sfruttando la capacità di distribuzione funzionale offerta da uno specifico costrutto di Java: i package, sotto-sistemi interni ad un progetto.

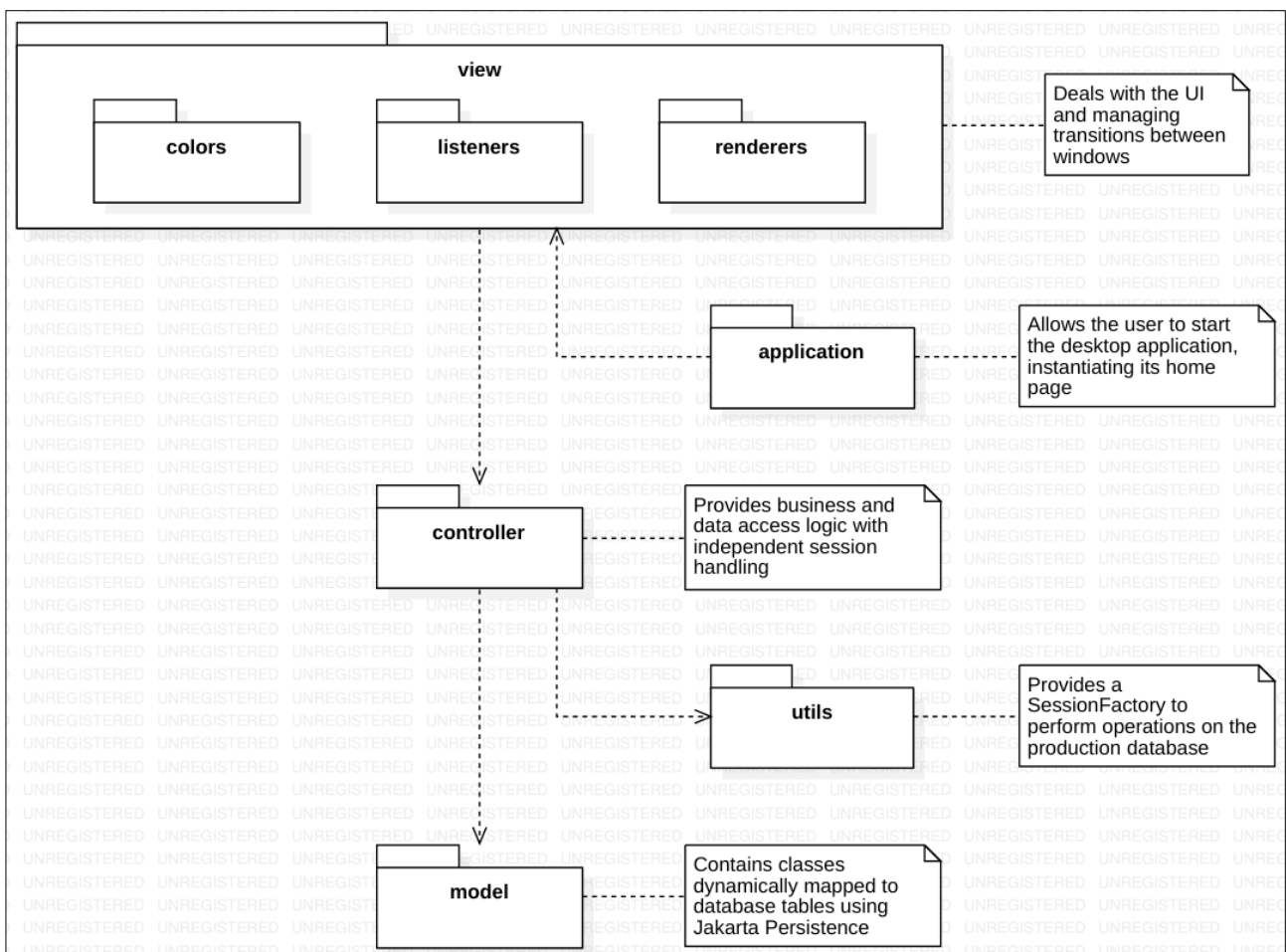


Figura 8: Diagramma dei package, che propone un punto di vista sull'implementazione

Passando ora da una visione del sistema in termini di package ad una descrizione di quest'ultimo nell'ottica degli elementi di progettazione principali e delle loro interazioni, ci si può concentrare sulle componenti costituenti i due package **Model** e **Controller**: nel primo caso si tratta di classi (**Invoice**, **User** ed **Item**) che vengono dinamicamente mappate a tabelle del database ricorrendo alla *Jakarta Persistence API*, con le prime due legate da una relazione *many-to-one*; nel secondo, invece, di interfacce (**InvoiceController**, **UserController** ed **ItemController**) che definiscono una serie di metodi specificati dalle classi che le implementano, le quali costituiscono il raccordo tra i due package in esame.

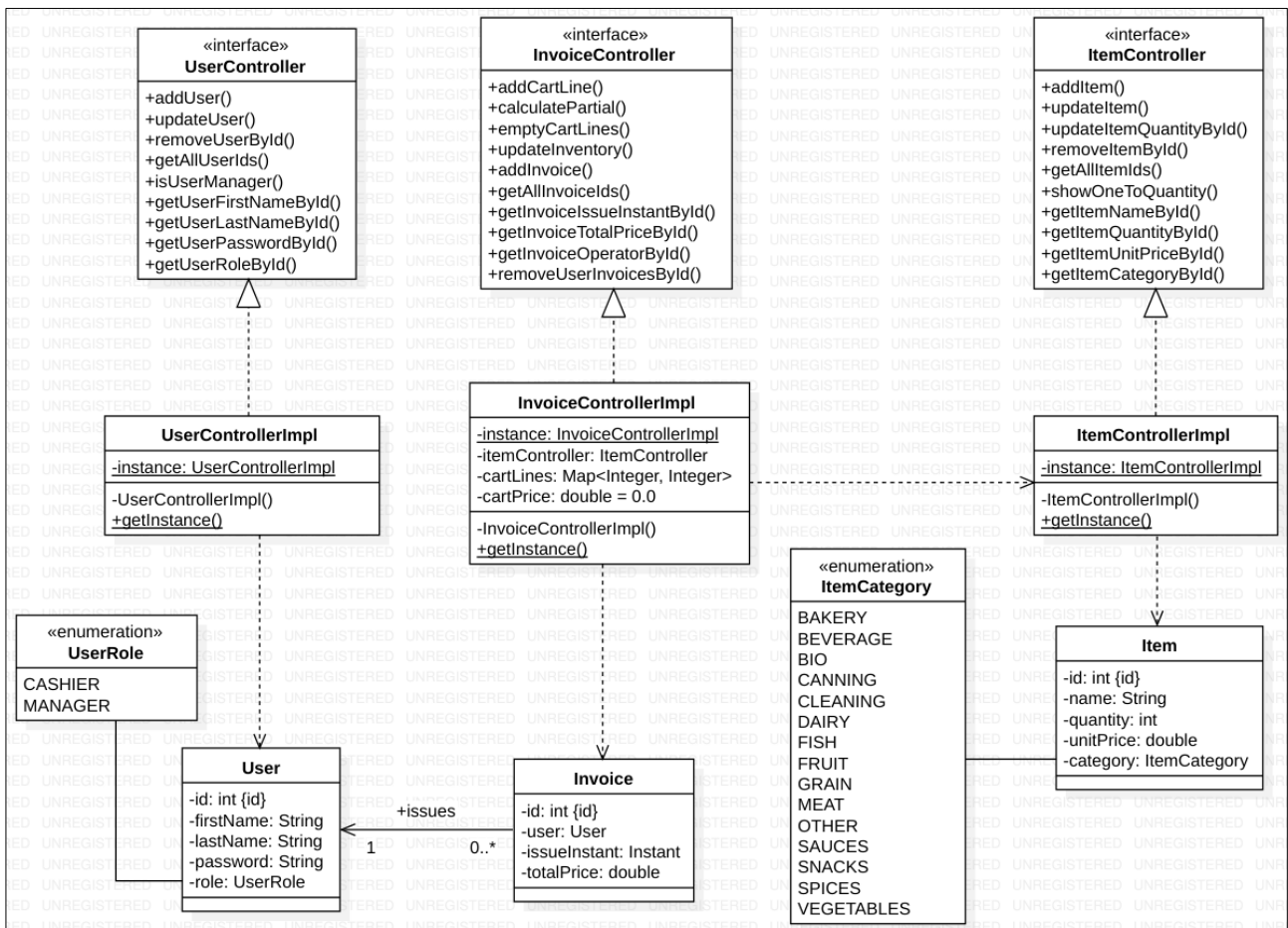


Figura 9: Diagramma delle classi, che offre un punto di vista logico



## Software Design

Tra i principi di progettazione più rilevanti nel contesto di un progetto Java orientato agli oggetti si ritrovano l'astrazione (la quale si ottiene separando la propria struttura applicativa in progetti, package e classi) e l'occultamento delle informazioni (dall'inglese "information hiding"), strettamente legato a quest'ultima, nel senso che, nel momento in cui lo sviluppatore decide di nascondere un determinato elemento, l'utente può fare a meno di conoscerlo.

Con riferimento al package Controller, ad esempio, il grado di astrazione ottenuto è pari a  $\frac{1}{2}$  (senza considerare un'eccezione custom, che ne fa parte), connesso alla decisione di prevedere tre interfacce e lo stesso numero di classi, la cui istanziazione risulta necessaria al fine di effettuare qualsivoglia operazione sul database.

Nell'ottica di massimizzare il grado di occultamento di classi e relativi costruttori, campi e metodi si è rivelato decisamente efficace **UCDetector** (<http://www.ucdetector.org>), un plug-in deputato all'individuazione di codice sorgente il cui modificatore di visibilità può essere rivisto, in quanto privo della necessità di essere pubblico e, dunque, di essere utilizzato dagli altri package.

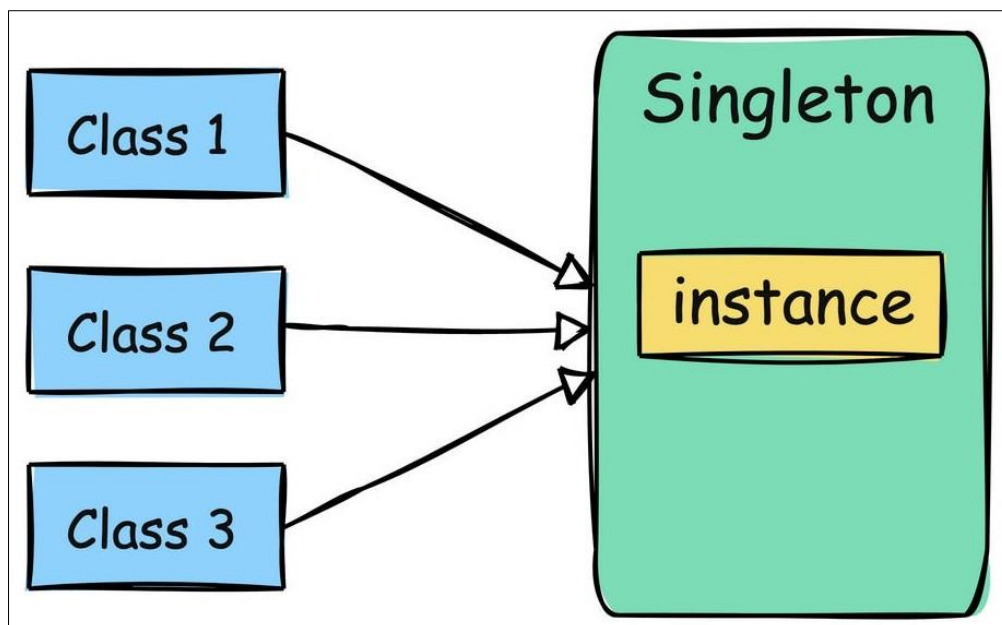


Figura 10: Schema semplificato del design pattern creazionale Singleton

Oltre al pattern architetturale Model-View-Controller, a cui ci si è attenuti al fine di strutturare l'applicazione nel suo complesso, un altro design pattern del cui utilizzo è emersa l'esigenza è il qui raffigurato **Singleton**, il quale consente di fare in modo che una classe abbia una ed una sola istanza, mettendo a disposizione un punto di accesso globale alla stessa.

Nel dettaglio, le classi che hanno sollevato il bisogno di ricorrervi sono quelle appartenenti al package Controller, i cui metodi vengono utilizzati da diverse classi del package View.

Al fine di misurare le metriche indicate nella tabella seguente (valutando così la qualità della progettazione del software) si è utilizzato il tool **JDepend**.

<i>Package</i>	CC	AC	Ca	Ce	Astrazione	Instabilità	Distanza
view.renderers	4	0	1	1	0.00	0.50	0.50
view.listeners	0	1	1	0	1.00	0.00	0.00
view.colors	1	0	2	0	0.00	0.00	1.00
view	16	0	1	10	0.00	0.90	0.09
utils	1	0	2	3	0.00	0.60	0.39
model	5	0	2	1	0.00	0.33	0.66
controller	4	3	2	4	0.42	0.66	0.09
application	1	0	0	6	0.00	1.00	0.00

Nel contesto di cui:

1. **CC (Classi Concrete):** Numero di classi concrete per package;
2. **AC (Classi Astratte):** Numero di classi astratte (o interfacce) per package;
3. **Ca (Accoppiamento Afferente):** Numero di package che dipendono dal package in esame (si tratta di un indicatore di responsabilità);
4. **Ce (Accoppiamento Efferente):** Numero di package da cui il package in esame dipende (si tratta di un indicatore di indipendenza);
5. **Astrazione:** Rapporto tra le classi astratte (o interfacce) ed il totale delle classi appartenenti ad un package (se pari a 0 indica package del tutto concreti, se pari a 1 package completamente astratti);
6. **Instabilità:** Rapporto tra l'accoppiamento efferente ed il totale delle dipendenze di un singolo package, va da 0 (completamente stabile) a 1 (completamente instabile);
7. **Distanza:** Misura quanto il package si distanzia dalla linea ideale di equilibrio tra astrazione e stabilità, dove 0 indica package ottimizzati e 1 package lontani dall'equilibrio.

Il seguente grafo delle dipendenze è invece stato generato utilizzando **Java Dependency Viewer**, un plug-in per l'ambiente di sviluppo integrato che abbiamo utilizzato (Eclipse IDE) progettato per visualizzare le dipendenze tra i package (o le classi) appartenenti ad un progetto Java.

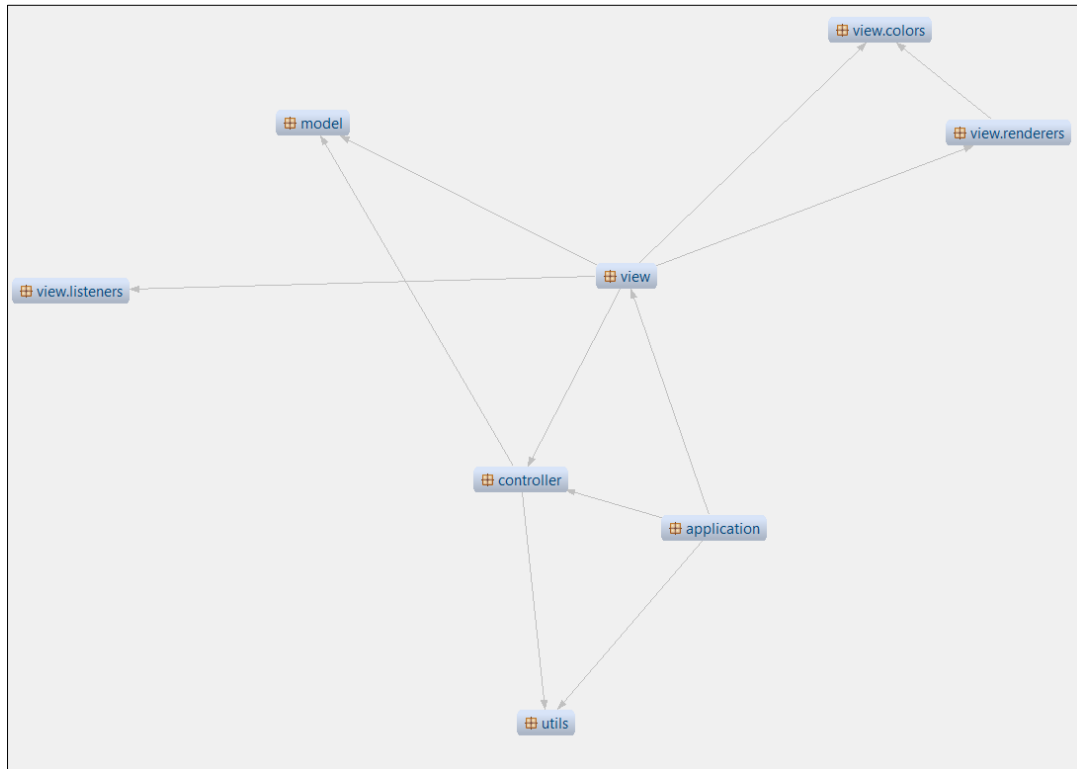


Figura 11: Grafo delle dipendenze tra i package costituenti l'applicazione

## Software Testing

Oltre a simulazioni di utilizzo dell'applicativo si è deciso di effettuare test di unità mirati per mezzo di **Junit 5** all'interno dell'ambiente di sviluppo integrato (Eclipse IDE): ad essere interessate da quest'ultimi sono state sia le classi del package Model che quelle appartenenti al package Controller, i cui metodi sono stati ritenuti i più critici in quanto interagenti con il database.

Al fine di disaccoppiare le operazioni attuate sul database di produzione (disk-based, il cui contenuto viene aggiornato in concomitanza con ciascuna query) si è optato per la configurazione di un ulteriore database esclusivamente adibito alla fase di testing (in-memory, il cui schema viene generato prima dell'esecuzione di ogni test di unità ed eliminato successivamente alla stessa).

Un dettaglio tecnico: entrambe le basi di dati sono state realizzate utilizzando **H2 Database Engine**, un DBMS relazionale particolarmente leggero, con supporto per le transazioni e pienamente conforme alle specifiche Java.

A seguire, con la finalità di mostrare l'approccio intrapreso, si riportano un paio di casi di test scritti al fine di verificare il comportamento di un metodo della classe UserController:

*@Test*

```
public void testIsUserManager_UserIsNotManager() {  
    // Get the ID of the first user in the testing database  
    Integer userId = userController.getAllUserIds().get(0);  
    // Store the boolean value returned by UserController's method  
    boolean result = userController.isUserManager(userId);  
    // Check that the result meets expectations  
    assertFalse(result, "User should be identified as a manager");  
}
```

*@Test*

```
public void testIsUserManager_UserDoesNotExist() {  
    // Assuming 999 is a non-existent user ID  
    boolean result = userController.isUserManager(999);  
    // Check that the result meets expectations  
    assertFalse(result, "Non-existent user should not be identified as a manager");  
}
```




















Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼  MPP_BillingSystem	 26.5 %	2,243	6,236	8,479
▼  src/main/java	 14.7 %	1,071	6,201	7,272
>  view	 0.0 %	0	5,259	5,259
>  controller	 62.8 %	756	447	1,203
>  view.renderers	0.0 %	0	329	329
>  application	0.0 %	0	57	57
>  view.colors	0.0 %	0	55	55
>  utils	0.0 %	0	31	31
>  model	93.2 %	315	23	338
▼  src/test/java	 97.1 %	1,172	35	1,207
>  model	 92.6 %	415	33	448
>  controller	 99.7 %	757	2	759

Figura 12: Misura di copertura globale relativa ai casi di test, realizzata con **EclEmma**

## Software Maintenance

La manutenzione assume un ruolo cruciale nel preservare la qualità e la funzionalità del sistema nel corso del suo intero ciclo di vita: per questo motivo è stato adottato un approccio integrato alla stessa, che ha combinato miglioramenti strutturali del codice con interventi correttivi, portando a mettere in atto un processo di ottimizzazione continuo.

Già in una prima fase si è condotto un approfondito refactoring del codice, concentrandosi sull'eliminazione dei metodi non essenziali, sull'ottimizzazione di quelli esistenti e sull'adeguamento delle visibilità dei metodi, con l'obiettivo di scorporare il più possibile le tre viste e semplificare l'architettura: tale lavoro preparatorio ha reso il software più leggero e facilmente adattabile, ponendo le basi per una gestione semplificata delle problematiche che sarebbero eventualmente emerse durante i test successivi.

A seguito di queste accortezze, una serie di test ha permesso di identificare l'area più critica in assoluto (costituita dai metodi delle classi appartenenti al package Controller, che interagiscono con il database mediante il meccanismo transazionale) e di intervenire tempestivamente per risolverne i malfunzionamenti, migliorando in tal modo la stabilità complessiva del sistema e rendendolo più resiliente e predisposto all'evoluzione.

Parallelamente all'ottimizzazione del codice ed alle correzioni resesi necessarie, non si è affatto trascurata la documentazione di sistema, che è infatti stata predisposta e mantenuta aggiornata con lo scopo di fornire una guida strutturata agli sviluppatori, facilitando la comprensione delle funzionalità e rendendo ogni intervento futuro più efficace e mirato.

Segue il risultato dell'attività di refactoring che ha interessato la struttura complessiva del progetto.

La definizione di sotto-package di View ha permesso di separare nettamente, nel contesto del livello di presentazione, le classi rappresentanti una finestra (**JFrame**) o un pannello (**JPanel**) da quelle prettamente incaricate di gestire la personalizzazione del contenuto, oltre che da un'interfaccia deputata alla gestione dell'aggiornamento dei pannelli in concomitanza con ciascuna transizione tra gli stessi (contenuta nel sotto-package Listeners); un procedimento banale, che però contribuisce a rendere manutenibile non solo il codice sorgente, ma anche l'intero progetto, aiutando ad individuarne i macro-blocchi costitutivi ed i file peculiari, come quelli di configurazione (con estensione **.cfg.xml**).

