# DOCUMENTAZIONE

## 1.Ingegneria dei requisiti:

### 1.1 **Obbiettivo**

Si vuole realizzare un sistema per la gestione degli ordini di una pasticceria. Gli utenti dovranno avere necessariamente un account per accedere al sistema e questo proprio perché l'utente avrà la possibilità di cambiare la sua classificazione da cliente standard a cliente premium ottenendo ulteriori sconti. La registrazione consisterà nell'inserimento di dati come username e password, una volta effettuata dovranno solo usufruire del sistema per poter effettuare e modificare gli ordini.

### 1.2 **Target**

Questa applicazione è sviluppata per poter essere utilizzata da tutti. Infatti, l'interfaccia di utilizzo è studiata per essere semplice ed intuitiva. I diversi servizi vengono visualizzati nella pagina principale. Nessuna conoscenza o requisito specifico viene richiesta.

### 1.3 **Servizi offerti all'utente**

L'utente registrato potrà effettuare:

- Ordine standard: qui l'utente potrà visualizzare le diverse tipologie di torte proposte dalla pasticceria e aggiungerli al carrello.
- Ordine personalizzato: in questa sezione gli utenti modificano il prodotto inserendo il numero di persone e l'occasione per cui la torta dovrà essere servita, il numero di piani, e la data di ritiro dell'ordine.
- Sconto: l'utente premium ha diritto ad uno sconto giornaliero del 30% che applicherà all'ordine.

Ciò che differenzia il cliente standard da quello premium è il pagamento di un abbonamento annuale che gli permetterà di usufruire dello sconto che verrà applicato sul totale dell'ordine.

## 1.4 **Requisiti funzionali e non funzionali**

I requisiti sono stati definiti secondo l'acronimo MoSCoW:

1.MUST HAVE: sono requisiti assolutamente necessari.
2.SHOULD HAVE: sono requisiti importanti, ma non assolutamente necessari.
3.COULD HAVE: sono requisiti che vengono implementati solo se il tempo lo consente, solitamente implementati solo per portare potare valore aggiuntivo.
4.WON'T HAVE: requisiti che sono stati discussi ma che alla fine sono stati posticipati o esclusi dal progetto.

FUNZIONALI:

1)Registrazione dell'utente, visualizzare il catalogo, effettuare la prenotazione, pagamento abbonamento.

2) Visualizzare il carrello e modificare prodotto, applicare lo sconto.

3) Visualizzare i dati Utente, cancellare carrello e visualizzare gli ingredienti dei prodotti.

4) Modificare e visualizzare i dati dell'abbonamento, riepilogo dell'ordine standard.

NON FUNZIONALI:

1) Prestazione(applicazione veloce e reattiva), Usabilità (interfaccia utente semplice e intuitiva),

2) Riutilizzabilità(gestione del catalogo e dei prodotti)

3) Sicurezza (dati utenti protetti),

## 1.5 **Vincoli**

Vincoli legati all'uso delle funzionalità:

- L'utente deve effettuare una registrazione prima di effettuare il login
- L'utente deve registrarsi come utente standard prima di diventare premium
- Senza registrazione l'utente non può effettuare alcun ordine

- Il pagamento verrà effettuato solamente dall'applicazione

# 2.Ciclo di vita del software:

## 2.1 **Metodo Rad**

Il metodo di sviluppo scelto è il metodo Rad, la motivazione di questa scelta è legata agli impegni dei componenti del Team.

Questo metodo infatti prevede di fissare una serie di requisiti che dovranno poi essere soddisfatti entro un intervallo di tempo fissato (time box). Se non si possono soddisfare tutti i requisiti entro il periodo di tempo stimato allora alcune funzionalità possono essere sacrificate.

Per questo motivo si ha bisogno di stabilire una priorità dei requisiti, effettuata durante l'analisi, tramite un processo chiamato Triage che utilizza l'acronimo MoSCoW.

Le fasi del ciclo di vita del rad sono:

1. *Pianificazione Requisiti:* l'obbiettivo è comprendere ciò che il cliente si aspetta dal sistema e quindi raccogliere i requisiti che dovrà avere tramite interviste o documentazioni.  (importante la collaborazione con gli utenti finali)
2. *Progettazione dell'applicazione:* Dai requisiti raccolti al punto precedente si sviluppa un prototipo iniziale che verrà utilizzato come base per le iterazioni successive.
3. *Costruzione:* qui viene scritto il codice effettivo e implementate le funzionalità. È una fase iterativa, ci possono essere cicli di sviluppo rapidi per apportare le modifiche.
4. *Taglio:* qui il software viene distribuito e reso disponibile agli utenti finali, offrendo anche attività di formazione e supporto.

# 3.Organizzazione del Team:

## 3.1 **Struttura del Team**

Il team è composto da:

- Tea Benzoni (matr. 1065725)
- Antonio Marmo (matr. 1060572)
- Andrea Vaerini (matr. 1067398)

Il team ha una struttura molto semplice dove i membri si suddividono i vari compiti coordinandosi ed effettuando controlli reciproci.

## 3.2 **Suddivisione del lavoro**

Il lavoro è stato diviso proporzionalmente tra i membri in base alle proprie conoscenze. Il primo passo consiste nella realizzazione dei diagrammi UML usati per delineare le linee guida per stesura del codice. Dopo di che si passa alla effettiva stesura del codice il quale verrà controllato tramite opportuni test ed infine il completamento della documentazione.

## 3.3 **Comunicazione del Team**

Durante lo sviluppo del progetto le comunicazioni avvenivano tramite chat di gruppo e riunioni per aggiornare il team del lavoro svolto e programmare il lavoro mancante.

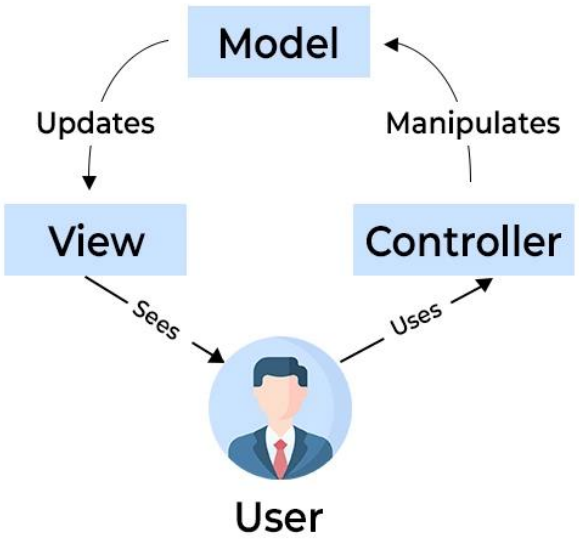
## 3.4 **Configuration Management**

La gestione del progetto è avvenuta tramite GitHub, una piattaforma di sviluppo collaborativo di software. Il team ha tenuto traccia delle modifiche del codice e dei file situati all’interno della repository tramite la creazione di branch per astrarre momentaneamente il lavoro personale dal branch principale ,dopo di che effettuare richieste di pull cosi da integrare nuove funzionalità o correzioni di bug.

# 4. Architettura del Software:

La struttura del software si basa sullo stile architetturale Model-View-Controller. L'obbiettivo è di suddividere l'applicazione in tre componenti principali (Model, View, Controller) ognuna con delle responsabilità specifiche.

- *Model:* Rappresenta la parte di dati e della logica dell'applicazione è quindi responsabile della gestione e dell'aggiornamento dei dati, e in tal caso notifica anche le altre componenti degli eventuali cambiamenti
- *View:* Rappresenta la parte di interfaccia utente dell'applicazione, si occupa di visualizzare i dati provenienti dal model e di interagire con l'utente.
- *Controller:* Esso funge da intermediario tra il Model e la View. Riceve in ingresso i comandi dell'utente tramite la View e reagisce di conseguenza modificando il Model o View.



# 5. Qualità del software:

La qualità può essere definita come il grado con cui un sistema soddisfa le esigenze o aspettative di un utente.

Il sistema è realizzato seguendo la norma ISO 9126 per la quale sono state definite un insieme di caratteristiche di prodotto:

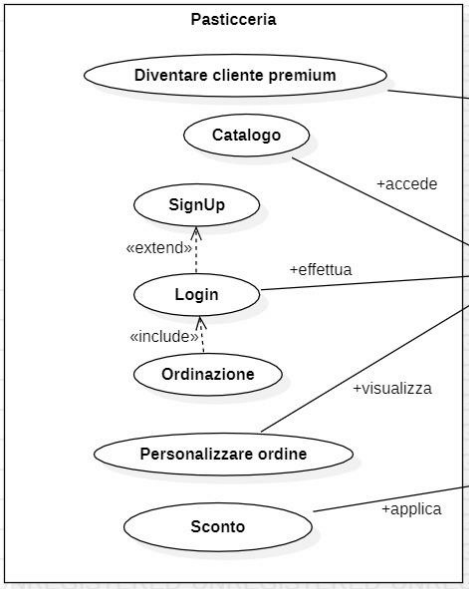
- Funzionalità: Il sistema deve fornire le funzioni che soddisfino le esigenze dell'utente.
- Affidabilità: Il sistema deve essere in grado di mantenere un determinato livello di prestazioni.
- Usabilità: Facilità di comprensione dell'uso del sistema.
- Efficienza: Il sistema deve fornire le prestazioni adeguate.
- Manutenibilità: Il software deve essere facilmente modificato per il suo miglioramento.

# 6. Modellazione:

## 6.1 **UML**

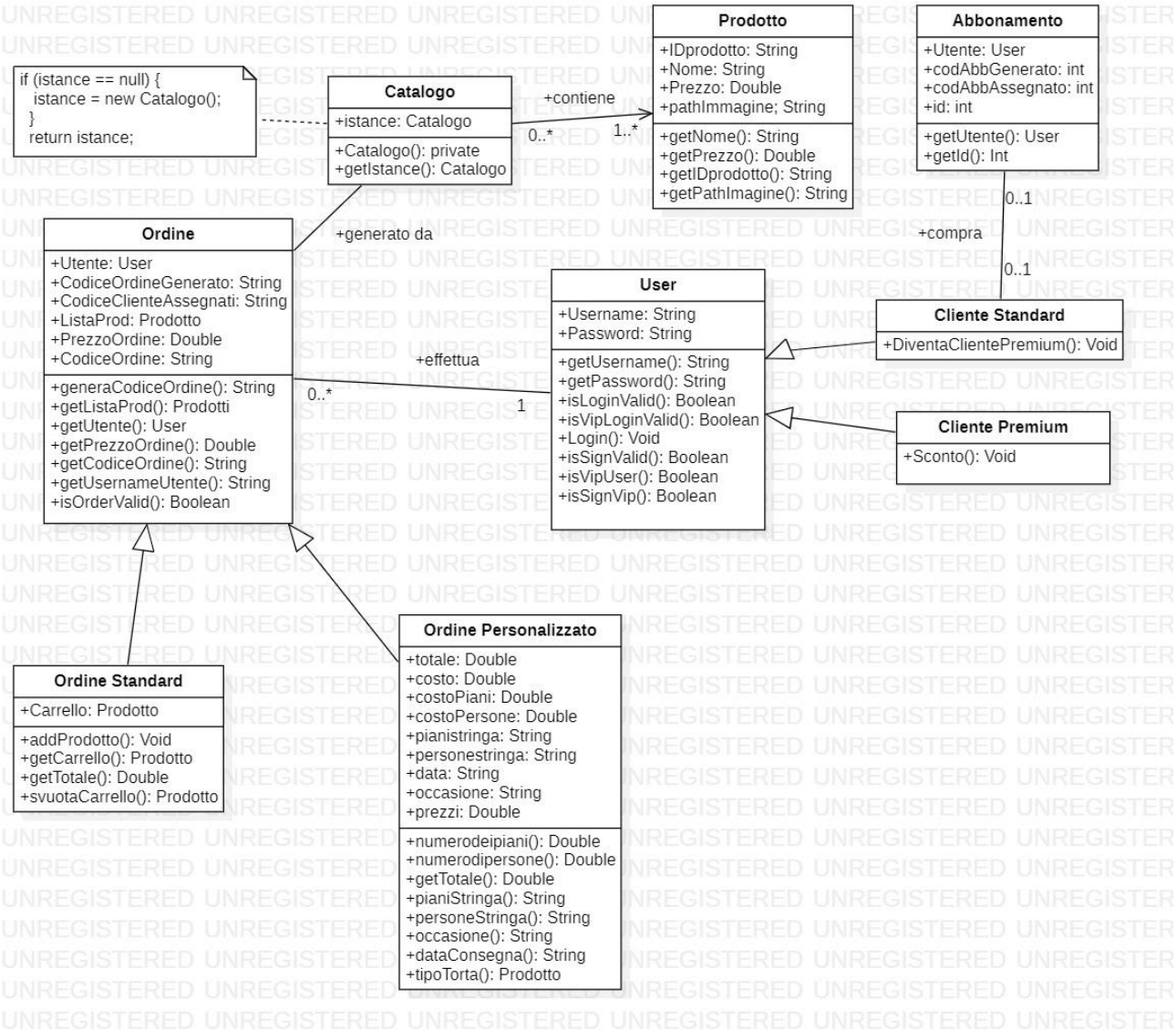
La realizzazione dei diagrammi UML è avvenuta tramite il programma StarUML e resi disponibili nella cartella UML della repository di GitHub. I diagrammi al suo interno sono:

### 6.1.1 **Diagramma dei Casi d'Uso:**



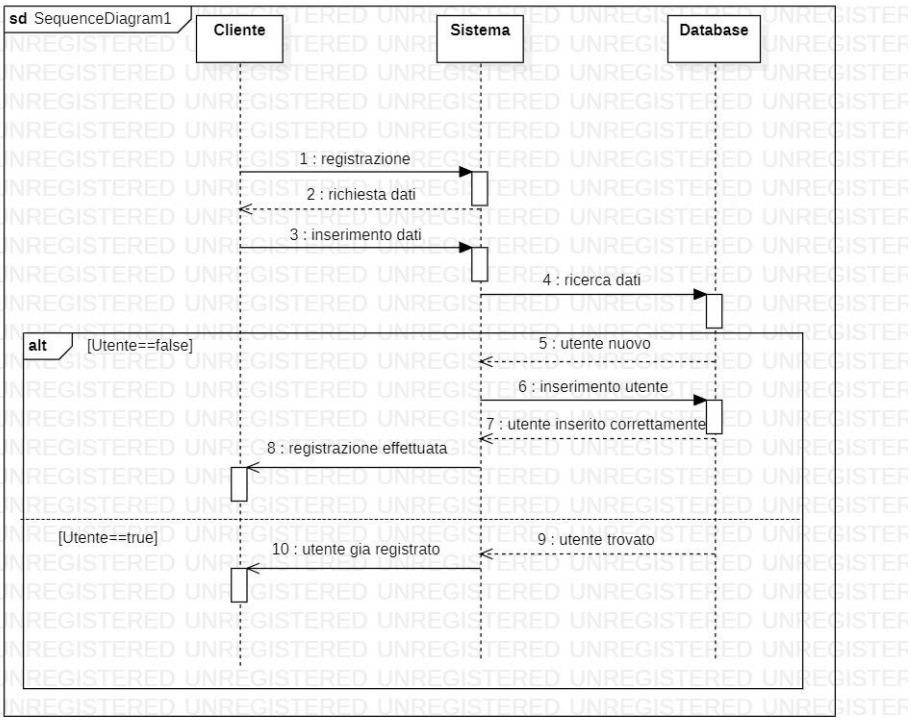
Attraverso i casi d’uso sono state riportate tutte le funzionalità svolte dal sistema delle quali gli attori, che sono i clienti della pasticceria, possono usufruire.

### 6.1.2 **Diagramma delle classi**



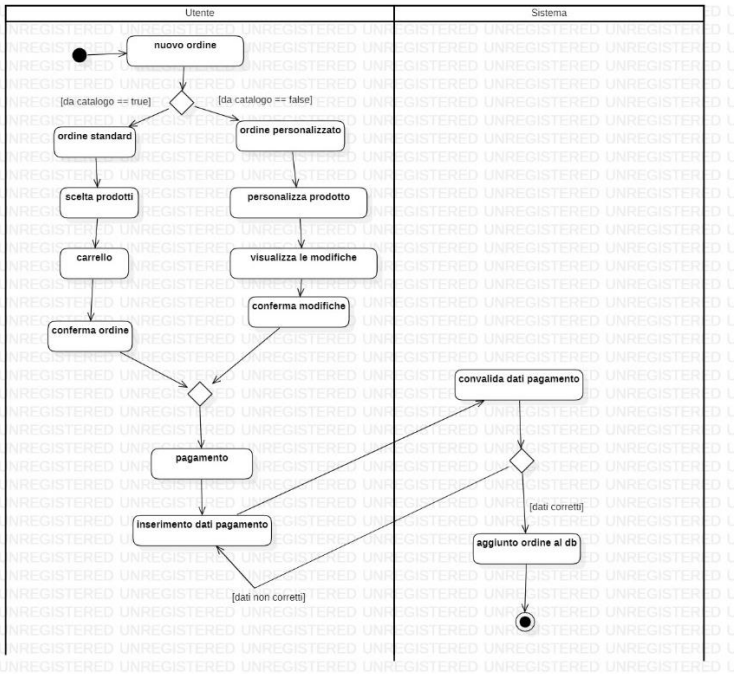

Queste classi modellano la struttura del nostro sistema e, allocate all’interno del package model, vengono utilizzate soprattutto per interagire con il database.

### 6.1.3 **Diagramma di Sequenza**



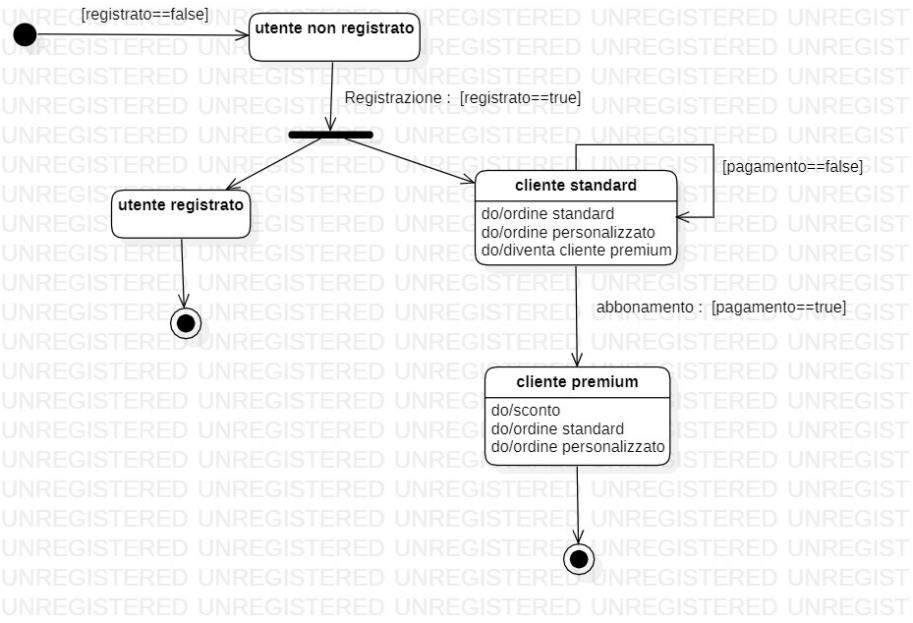

Il diagramma modella lo scenario legato alla registrazione dell'utente all'applicazione.

### 6.1.4 **Diagramma delle Attività**




Viene modellato il processo di ordinazione dei prodotti riportando le opportune modifiche al database e la verifica del pagamento.

### 6.1.3 **Diagramma degli stati**




Rappresentazione degli tutti gli stati che l'utente può assumere all'interno dell'applicazione.

# 7. Design del software:

### 7.1 **Design pattern**

Nello sviluppo del progetto sono stati utilizzati due design pattern: Singleton e Observe.

#### 7.1.1 **Observe**:

È un pattern comportamentale usato per gestire le dipendenze tra gli oggetti in modo efficiente. Prevede che un oggetto (soggetto) mantenga una lista di osservatori che vengono notificati automaticamente quando si verificano dei cambiamenti di stato. Quando gli osservatori ricevono la notifica possono rispondere agli eventi in base alle proprie esigenze.

Nel nostro caso gli oggetti osservati sono i componenti grafici con cui l'utente interagisce per navigare nell'applicazione. Gli osservatori (controllori) si devono registrare e per farlo viene associato un ascoltatore all'oggetto Osservato, dopo di che implementando l'interfaccia ActionListener riescono a gestire gli eventi di azione, che vengono generati a loro volta dalla classe ActionEvent.

### 7.1.2 **Singleton:**

È un pattern creazionale che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale (metodo) a questa istanza. L'istanza viene creata solo quando è richiesta per la prima volta, migliorando le prestazioni poiché viene istanziata solamente se necessaria e quindi non creata inutilmente.

All'interno del progetto è stato utilizzato per istanziare la classe Catalogo. Questa classe ha:

- Variabile statica: chiamata "istance" che conterrà l'unica istanza
- Costruttore privato: per non rendere possibile la creazione di istanze direttamente da altre parti di codice. L'unico modo possibile è di utilizzare il metodo getIstance.
- Metodo getIstance: metodo statico che consente di ottenere l'istanza della classe, una volta chiamato controlla se l'istanza è già stata creat, se non lo è crea un nuovo oggetto e lo restituisce.

## 7.Testing:

Tutti i test sono stati effettuati tramite JUnit e Mockito:

- *JUnit* è un framework di testing per il linguaggio Java che ci ha permesso di scrivere dei test su singole unità di codice, principalmente per i metodi delle classi.
- *Mockito* invece è stato utilizzato per creare degli oggetti (mock) simulati che sostituiscono le componenti reali all'interno dell'applicazione durante i test unitari.

Durante questa fase abbiamo cercato di creare un numero di test adatto per coprire una buona parte del codice (66,4%) e si possono trovare all'interno dei tre package legati ai corrispettivi tre package delle classi.