

UNIVERSITÀ DEGLI STUDI DI MODENA E
REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Laurea Triennale in Ingegneria Informatica

Applicazione per la gestione del cibo

Relatore:

Prof: Francesco Guerra

Candidato:

Saverio Napolitano

Anno Accademico 2023/2024

Indice

1	Introduzione	6
2	Specifica dei requisiti	8
2.1	Requisiti funzionali	8
2.2	Requisiti non funzionali	13
3	Descrizione del progetto	14
4	UML	17
4.1	Use case diagram	18
4.2	Activity diagram	19
4.2.1	Activity diagram della dispensa	19
4.2.2	Activity diagram della lista della spesa	20
4.2.3	Activity diagram delle ricette	21
4.3	State diagram	22
4.3.1	State diagram della ricetta	22
4.3.2	State diagram della lista della spesa	23
4.4	Class diagram	24
4.5	Sequence diagram	25
4.5.1	Sequence diagram della dispensa	26
4.5.2	Sequence diagram della lista della spesa	27
4.5.3	Sequence diagram delle ricette	28
5	Design Patterns	30
5.1	Factory	30
5.2	Iterator	31
5.3	Observer	32
5.4	Singleton	32
5.5	Decorator	33
6	ORM	34
6.1	Vantaggi	34
6.2	Svantaggi	36
6.3	Stress tests	38

6.3.1	Inserimento di prodotti nel database	38
6.3.2	Eliminazione di prodotti dal database	39
6.3.3	Modifica di prodotti nel database	40
6.3.4	Join fra tabelle del database	41
7	Conclusioni	42

Elenco delle figure

1.1	Logo dell'applicazione.	6
3.1	Finestra per la gestione di dispensa e lista della spesa.	14
3.2	Finestra per la modifica dei dati del prodotto.	15
3.3	Finestra per la gestione delle ricette.	16
4.1	Diagramma dei casi d'uso.	18
4.2	Diagramma delle attività della dispensa.	19
4.3	Diagramma delle attività della lista della spesa.	20
4.4	Diagramma delle attività delle ricette.	21
4.5	Diagramma degli stati della ricetta.	22
4.6	Diagramma degli stati della lista della spesa.	23
4.7	Diagramma delle classi.	24
4.8	Diagramma di sequenza della dispensa	26
4.9	Diagramma di sequenza della lista della spesa	27
4.10	Diagramma di sequenza delle ricette	28
6.1	Inserimento di prodotti nel database.	38
6.2	Eliminazione di prodotti dal database.	39
6.3	Modifica di prodotti nel database.	40
6.4	Join fra tabelle del database.	41

Elenco delle tabelle

6.1	Inserimento di prodotti nel database.	38
6.2	Eliminazione di prodotti dal database.	39
6.3	Modifica di prodotti nel database.	40
6.4	Join fra tabelle del database.	41

Capitolo 1

Introduzione

Expiration Date è un'applicazione nata per permettere la gestione completa di tutto ciò che riguarda i prodotti alimentari; con essa è infatti possibile gestire gli aspetti relativi:

- alla lista della spesa (prodotti che si desidera acquistare)
- alla dispensa (prodotti già posseduti)
- alle ricette



Figura 1.1: Logo dell'applicazione.

Nel seguito verranno illustrati:

- i requisiti (funzionali e non funzionali) alla base dello sviluppo dell'applicazione
- una panoramica generale dell'applicazione, con particolare attenzione all'interfaccia grafica
- i diagrammi UML (Unified Modeling Language) che descrivono nel dettaglio le funzionalità dell'applicazione e la sua business logic
- i design patterns implementati, le motivazioni e i vantaggi del loro impiego
- la tecnica ORM (Object-Relational Mapping) per la gestione della persistenza dei dati e l'interfacciamento con il database

Questo elaborato estende l'applicazione sviluppata per l'esame di "Programmazione a oggetti" con l'obiettivo di acquisire nuove competenze nell'ambito dell'interoperabilità fra programmazione a oggetti e database relazionali. Tramite l'utilizzo di ORM, infatti, ho avuto la possibilità di confrontarmi con un approccio alla gestione dei dati totalmente diverso da quelli visti in precedenza, acquisendo le capacità di:

- modellare le entità classiche usate dai database (e definite dal modello relazionale) con un nuovo formalismo, orientato ai linguaggi a oggetti
- rappresentare le associazioni e il modo con cui le suddette entità interagiscono tra loro attraverso strutture dati tipiche della programmazione (a oggetti ma non solo)
- integrare concetti tipici della programmazione a oggetti (come l'ereditarietà) in un contesto che nativamente non li prevede
- accedere a dati memorizzati su database con modalità proprie dei linguaggi di programmazione a oggetti
- studiare e valutare pregi e difetti di questa metodologia per capirne l'effettiva utilità in un contesto reale, eventualmente confrontandola con altre tecniche (JDBC)

Inoltre, rispetto alla versione originale, l'applicazione è stata riprogettata e ha adottato diversi design patterns. In questa fase ho avuto l'opportunità di:

- approfondire il concetto di mantenibilità del codice, le best practices per raggiungerla e i suoi vantaggi nel medio e lungo termine
- esplorare soluzioni standard a problemi comuni, in modo da poter comunicare efficacemente con esperti del settore

Capitolo 2

Specifica dei requisiti

Nelle sezioni seguenti verranno illustrati i principali requisiti che l'applicazione si pone come obiettivo di soddisfare.

2.1 Requisiti funzionali

Essi descrivono le funzionalità che il sistema deve offrire, intendendo con questo sia le operazioni che l'applicazione svolge sia le interazioni che la stessa ha con l'utente o con altri sistemi esterni con cui si interfaccia.

RF01	Aggiunta alimenti in dispensa
Input	Dati alimento acquistato - nome alimento, data di scadenza, quantità
Processo	Memorizzazione su lista e database degli alimenti
Output	Visualizzazione alimenti

RF01.1	Generazione notifica scadenza alimento
Input	Aggiunta di un prodotto alla dispensa
Processo	Si crea l'avviso su calendario relativo alla scadenza del prodotto
Output	Aggiunta dell'avviso sul calendario dell'utente

RF02	Rimozione alimenti in dispensa
Input	Click su pulsante "Delete"
Processo	Eliminazione alimento dalla lista e dal database
Output	Visualizzazione lista aggiornata

RF02.1	Rimozione avviso su calendario
Input	Rimozione prodotto dalla dispensa
Processo	Cancellazione avviso su calendario
Output	Rimozione avviso su calendario

RF03	Ordinamento prodotti dispensa per nome
Input	Click sulla colonna "Product"
Processo	Ordinamento prodotti in modo crescente o decrescente in base al nome
Output	Visualizzazione prodotti ordinati

RF04	Ordinamento prodotti in dispensa per data di scadenza
Input	Click sulla colonna "Expiration Date"
Processo	Ordinamento prodotti in modo crescente o decrescente in base alla data di scadenza
Output	Visualizzazione prodotti ordinati

RF05	Modifica nome alimento in dispensa
Input	Doppio click sul nome alimento e inserimento nuovo nome
Processo	Aggiornamento nome alimento in memoria e in database
Output	Visualizzazione alimento con nome aggiornato

RF06	Modifica dati alimento in dispensa
Input	Doppio click su data di scadenza e modifica dati
Processo	Si apre una nuova finestra che consente la modifica dei dati dell'alimento; Aggiornamento dati alimento in memoria e in database
Output	Visualizzazione alimenti aggiornati

RF06.1	Modifica avviso calendario
Input	Modifica data di scadenza e/o nome prodotto
Processo	Aggiornamento data di scadenza e/o nome prodotto sul calendario
Output	Visualizzazione calendario aggiornato

RF07	Aggiunta alimenti in lista della spesa
Input	Nome alimento da acquistare
Processo	Salvataggio su memoria e database della lista della spesa
Output	Visualizzazione lista della spesa

RF08	Rimozione alimenti in lista della spesa
Input	Click su pulsante “Delete”
Processo	Eliminazione alimento dalla lista e dal database della lista della spesa
Output	Visualizzazione aggiornata lista della spesa

RF09	Alimento in lista della spesa comprato
Input	Utente clicca sulla checkbox che segnala l’acquisto di un elemento nella lista della spesa
Processo	Viene azionata la stessa procedura che implementa il requisito RF01
Output	Visualizzazione alimenti e aggiornamento lista della spesa

RF10	Pulizia lista della spesa
Input	Click sul pulsante “Clear” della lista della spesa
Processo	Rimozione dalla lista della spesa, sia in memoria che in database, degli alimenti già acquistati
Output	Visualizzazione lista della spesa aggiornata

RF11	Visualizzazione ricette memorizzate
Input	Click sul pulsante “Recipe” nella dispensa e all’interno della schermata “Recipe” attraverso le frecce di navigazione è possibile passare da una ricetta alla successiva
Processo	Caricamento dal database delle ricette
Output	Visualizzazione ricette

RF12	Aggiunta ricette
Input	Click su pulsante “Add” e inserimento dati ricetta - titolo, durata, numero porzioni, categoria, lista dei tag, lista ingredienti, procedimento ricetta
Processo	Salvataggio su database e in memoria della ricetta
Output	Visualizzazione ricetta

RF13	Modifica dati ricetta
Input	Modifiche su dati ricetta
Processo	Salvataggio in memoria e database dei dati della ricetta aggiornati
Output	Visualizzazione ricetta aggiornata

RF14	Controllo disponibilità ingredienti
Input	Visualizzazione ricetta o modifica della stessa
Processo	Confronto fra i prodotti in dispensa e gli ingredienti necessari per la ricetta
Output	Visualizzazione ingredienti disponibili e non disponibili per la ricetta attuale

RF15	Eliminazione ricetta
Input	Click sul pulsante “Delete” nella schermata delle ricette
Processo	Rimozione da memoria e database della ricetta attuale
Output	Visualizzazione di un’altra ricetta o aggiunta di una nuova ricetta vuota se non ci sono più ricette

RF16	Import ricette da file
Input	File contenente una lista di ricette
Processo	Salvataggio in memoria e database dei dati della ricette sul file importato
Output	Visualizzazione ricette importate

RF17	Export ricette su file
Input	Click su pulsante “Export”
Processo	Creazione di un file su cui sono memorizzate le ricette presenti in database
Output	File con lista delle ricette

2.2 Requisiti non funzionali

Essi descrivono le caratteristiche che l'applicazione deve avere che non sono direttamente collegate alle operazioni che esegue.

RNF01	Tempi di risposta
Descrizione	Il software dovrà garantire tempi di risposta minimi per tutte le operazioni, consentendo agli utenti di interagire con l'applicazione in tempo reale

RNF02	Database (salvataggio dei dati)
Descrizione	L'applicazione dovrà utilizzare un sistema di database affidabile e sicuro per garantire la persistenza dei dati relativi ai prodotti nella dispensa, ai prodotti presenti nella lista della spesa e alle ricette

RNF03	Portabilità
Descrizione	L'applicazione deve essere progettata in modo da essere compatibile sia con i sistemi Android che iOS

RNF04	Integrazione con servizi di terze parti
Descrizione	Il sistema deve essere in grado di interagire con il calendario predefinito dell'utente

Capitolo 3

Descrizione del progetto

Per questa descrizione si farà riferimento alla versione desktop dell'applicazione.

Il software presenta due schermate principali:

- una per la gestione della dispensa e della lista della spesa
- una per la gestione delle ricette

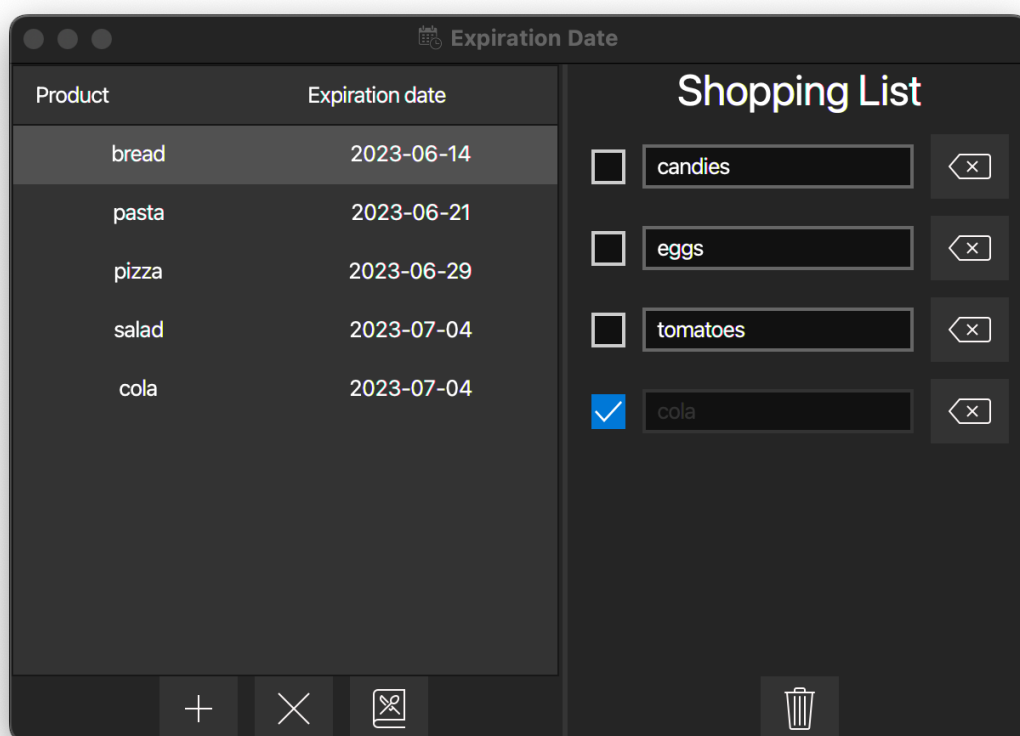
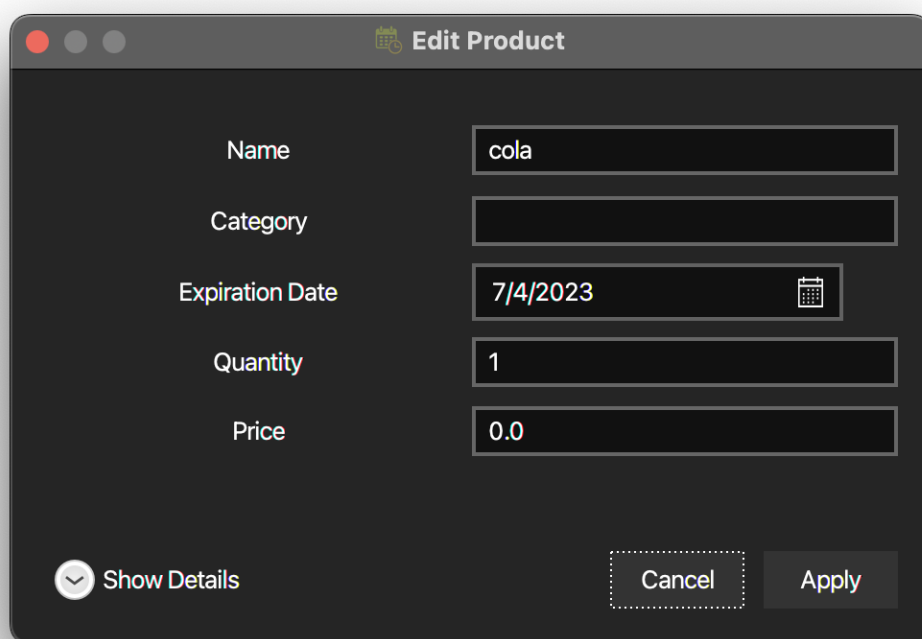


Figura 3.1: Finestra per la gestione di dispensa e lista della spesa.

La schermata per la gestione della dispensa e della lista della spesa si divide in due parti:

- la parte di sinistra permette di aggiungere prodotti alla dispensa, eliminare prodotti dalla dispensa, modificare i dati di un prodotto o passare alla schermata delle ricette
 - nel caso si desideri modificare i dati di un prodotto comparirà una schermata apposita per la modifica



Name	cola
Category	
Expiration Date	7/4/2023
Quantity	1
Price	0.0

Show Details Cancel Apply

Figura 3.2: Finestra per la modifica dei dati del prodotto.

- la parte di destra permette di aggiungere prodotti alla lista della spesa, eliminare prodotti dalla lista della spesa, contrassegnare un prodotto della lista della spesa come "acquistato" (e aggiungerlo in automatico alla dispensa) e rimuovere tutti i prodotti contrassegnati come acquistati

La schermata per la gestione delle ricette permette di navigare tra le ricette con le apposite frecce, aggiungere una ricetta, rimuovere una ricetta, modificare i dati di una ricetta, importare ed esportare liste di ricette.

The screenshot shows a macOS-style window titled "Recipe". The menu bar contains "File" and "Edit". The main content area is titled "Frozen Pizza". Below the title, there are several input fields and dropdown menus: "Duration: 30.0" with a "minutes" dropdown, "Portions: 4", "Category: first course" with a dropdown, and "Tag" with three dropdowns containing "low calories", "vegan", and an empty one. Below these is an "Ingredients" section with a green checkmark and the text "Ingredients you have: Done". Underneath, there is an input field with "pizza", a "Quantity: 1.0" field, a "Unit: kg" dropdown, a button with a minus sign and "x", and a green checkmark. A "+" button is centered below the ingredients section. At the bottom, there is a "Steps" section with a text area containing the placeholder "Add recipe steps...". Navigation arrows (left and right) are located on the left and right sides of the ingredients section.

Figura 3.3: Finestra per la gestione delle ricette.

Capitolo 4

UML

I diagrammi UML permettono di descrivere graficamente l'applicazione sotto vari punti di vista, garantendo la comprensione del funzionamento del software, delle esigenze che esso soddisfa e delle relazioni che intercorrono fra i suoi componenti.

Esistono diverse tipologie di diagrammi UML, quelle riportate nel presente documento sono:

- diagramma dei casi d'uso
- diagramma delle attività
- diagramma degli stati
- diagramma delle classi
- diagramma di sequenza

Per ciascuno di essi verrà offerta una breve panoramica che includerà:

- significato del diagramma
- contesto in cui viene utilizzato
- funzionalità specifica dell'applicazione che viene descritta
- eventuali precisazioni/omissioni all'interno del diagramma e motivazione delle stesse

4.1 Use case diagram

Lo use case diagram permette di identificare gli attori che interagiscono col sistema e le attività che essi possono svolgere. Lo scopo è descrivere le principali funzionalità del software. Nel caso in esame:

- gli attori sono l'utente e l'applicazione stessa
- i casi d'uso sono
 - gestione delle ricette e loro importazione/esportazione
 - gestione della lista della spesa
 - gestione della dispensa
- ogni caso d'uso prevede la gestione degli errori che possono verificarsi, nonché l'esecuzione di alcune sub-routine necessarie per il corretto funzionamento dell'applicazione

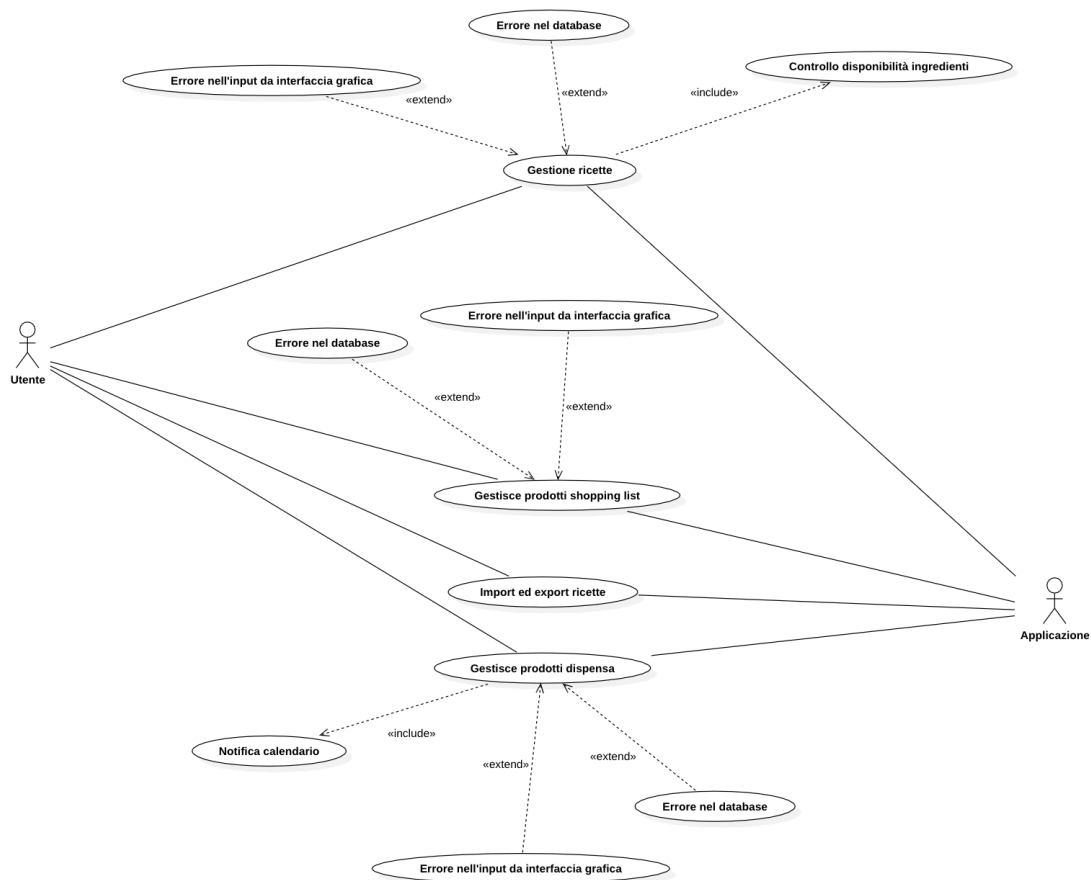


Figura 4.1: Diagramma dei casi d'uso.

4.2 Activity diagram

L'activity diagram consente di specificare come il sistema realizzerà le funzionalità mostrate nello use case. Lo scopo è connettere tra loro azioni di alto livello per rappresentare un processo che viene svolto nel sistema. Per una migliore comprensione del funzionamento, nei diagrammi seguenti si è scelto di non modellare i casi di errore del database e dell'interfaccia grafica.

4.2.1 Activity diagram della dispensa

In questo diagramma viene mostrato il processo relativo all'interazione fra l'utente e l'applicazione per la gestione della dispensa. In particolare, le azioni dell'utente sul prodotto:

- sono propagate sia in memoria che sul database per garantire la consistenza dei dati
- hanno un impatto anche sulla generazione, modifica ed eliminazione delle notifiche sul calendario, per garantire che gli avvisi sulla scadenza dei prodotti siano coerenti con i dati dei prodotti stessi

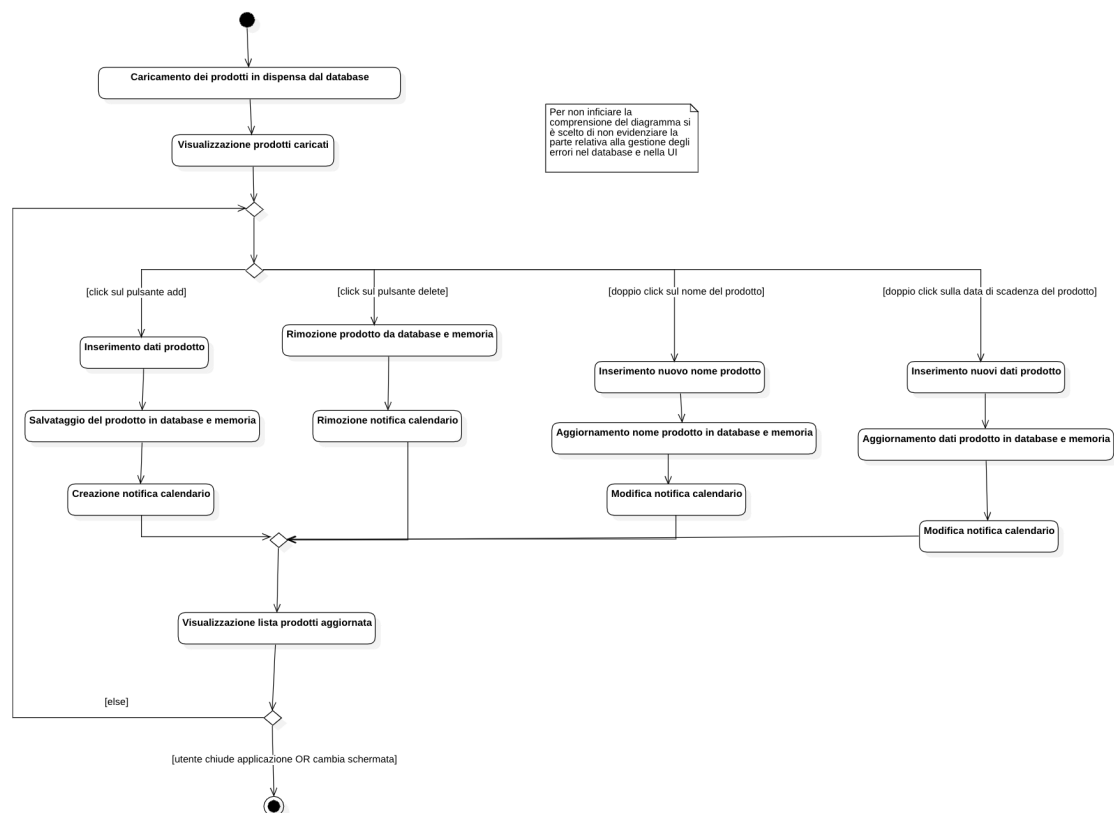


Figura 4.2: Diagramma delle attività della dispensa.

4.2.2 Activity diagram della lista della spesa

In questo diagramma viene mostrato il processo relativo all'interazione fra l'utente e l'applicazione per la gestione della lista della spesa. Si noti come:

- si faccia riferimento all'activity diagram precedente per evitare ripetizioni ed avere maggiore chiarezza in fase di modellazione e presentazione
- rispetto al diagramma delle attività della dispensa, in questo caso si modella anche lo stato del prodotto, dal momento che esso può variare in base all'azione effettuata dall'utente
- la struttura logica del diagramma sia molto simile a quella mostrata in precedenza, in quanto le azioni possibili sono quasi del tutto speculari

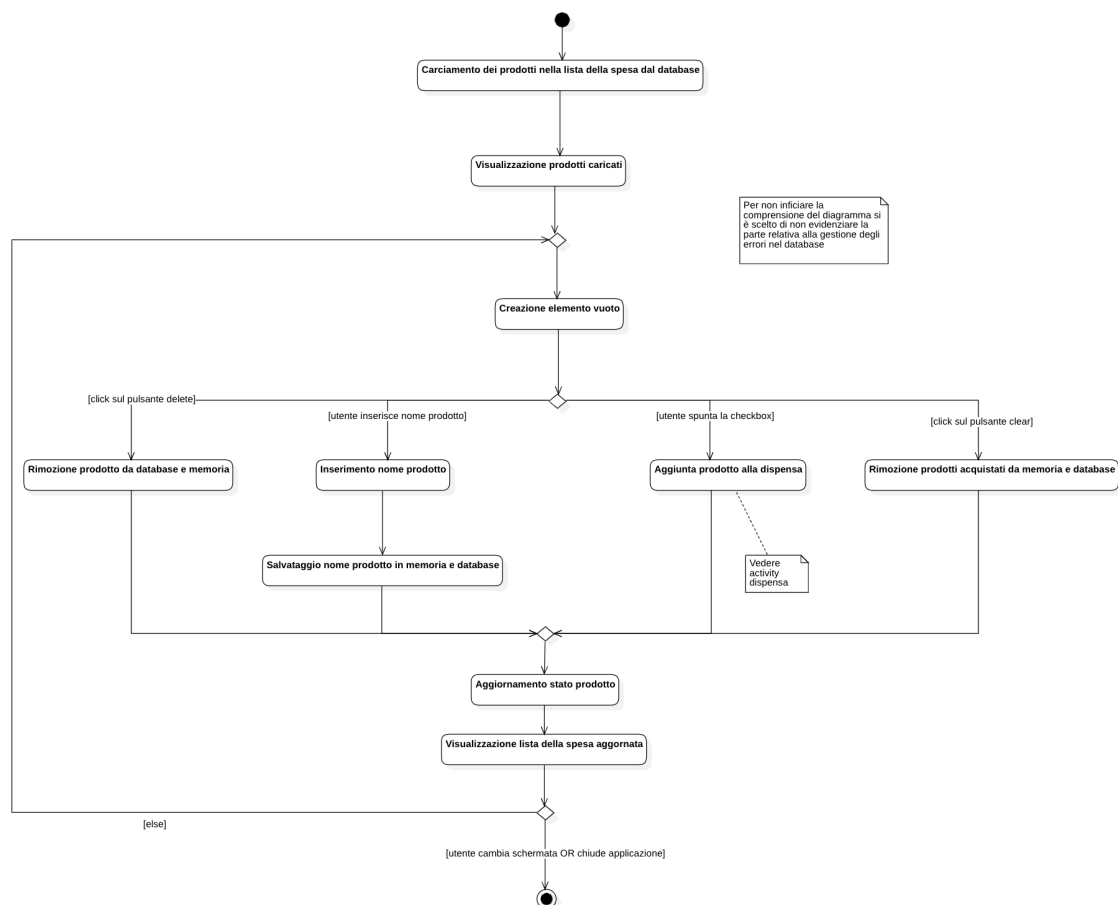


Figura 4.3: Diagramma delle attività della lista della spesa.

4.2.3 Activity diagram delle ricette

In questo diagramma viene mostrato il processo relativo all'interazione fra l'utente e l'applicazione per la gestione delle ricette, inclusa la possibilità di importare ed esportare ricette. Elementi distintivi rispetto ai due casi precedenti sono:

- la distinzione fra le due possibili casistiche (nessuna ricetta disponibile oppure almeno una ricetta salvata nel database) in modo da poter mostrare all'utente il risultato più adatto alle sue necessità, come una ricetta vuota in cui inserire i dati nel caso non siano presenti ricette (caso che si verifica, per esempio, al primo avvio dell'applicazione)
 - si noti come nel caso di ricetta presente nel database, venga eseguita anche la sub-routine relativa al controllo della disponibilità degli ingredienti in modo da fornire all'utente la consapevolezza circa l'effettiva realizzabilità del piatto
- la già citata possibilità di importazione ed esportazione, utile per favorire la condivisione reciproca di ricette personali o di nicchia permettendone una rapida espansione

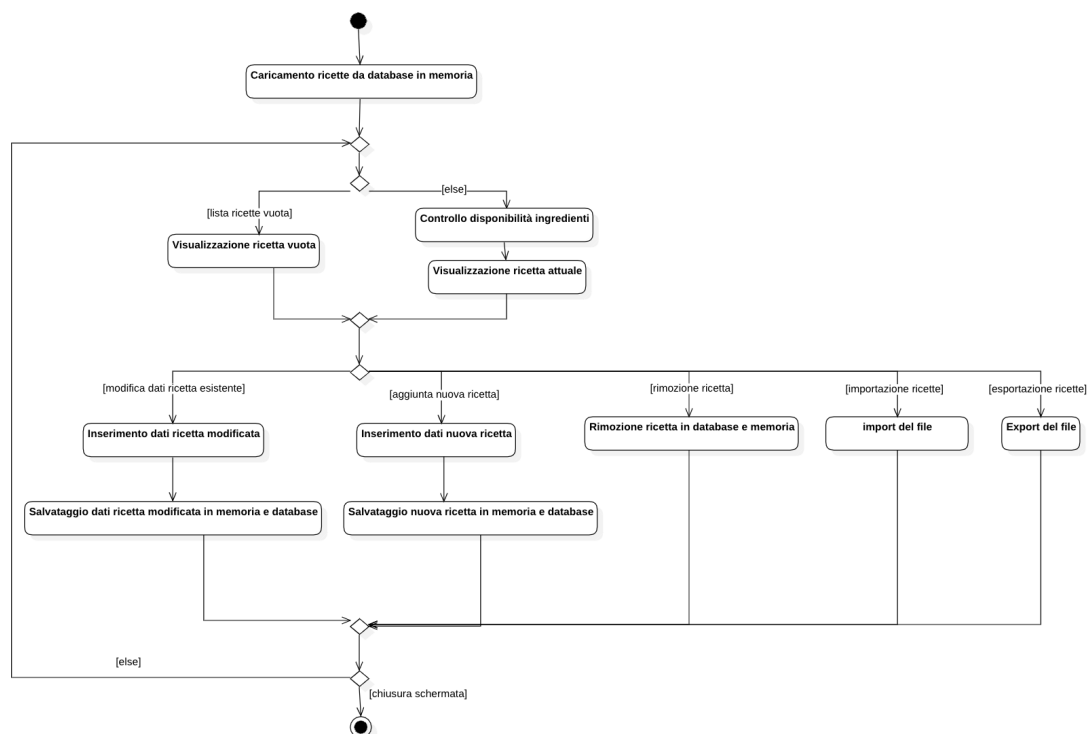


Figura 4.4: Diagramma delle attività delle ricette.

4.3 State diagram

Lo state diagram permette di modellare gli stati in cui si trova un oggetto e gli eventi che causano le transizioni fra gli stati.

4.3.1 State diagram della ricetta

In questo diagramma vengono mostrati gli stati in cui si può trovare una ricetta in base alla presenza, e alla disponibilità, degli ingredienti. L'obiettivo è fornire all'utente una panoramica immediata su:

- ricette che può preparare con gli ingredienti attualmente a disposizione
- ricette per cui manca un numero limitato di ingredienti per poter essere preparate
- ricette per cui la cui maggioranza degli ingredienti non è disponibile

e mantenere questa informazione aggiornata in modo tale da poter aiutare l'utilizzatore dell'applicazione a decidere cosa fare in base ai vincoli (sia di ingredienti che di tempo) che deve rispettare.

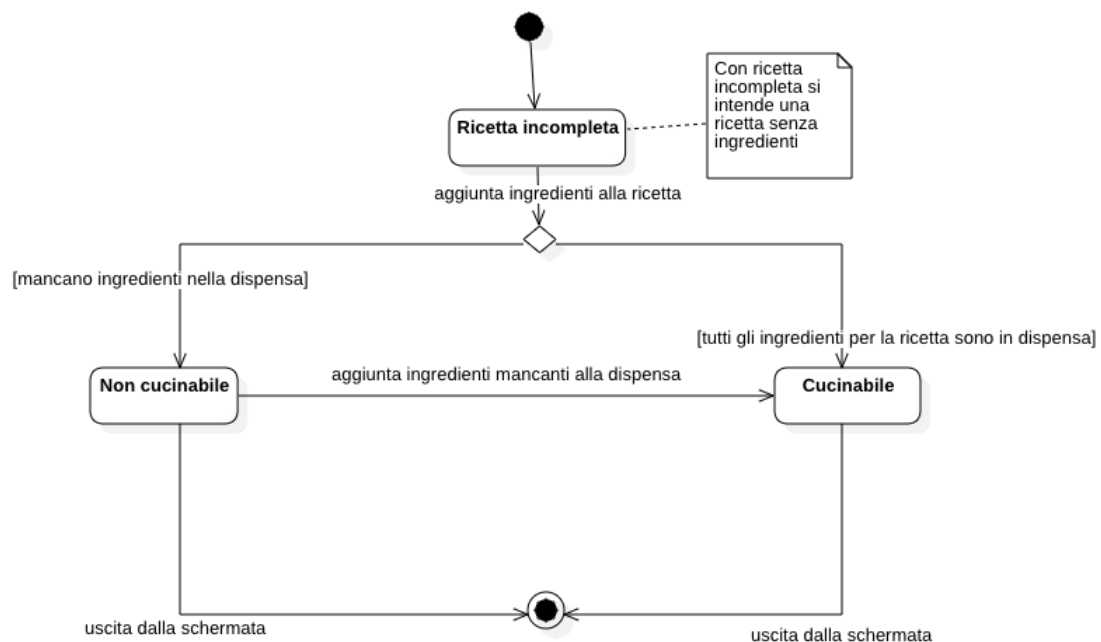


Figura 4.5: Diagramma degli stati della ricetta.

4.3.2 State diagram della lista della spesa

In questo diagramma vengono mostrati gli stati in cui si può trovare un prodotto presente nella lista della spesa. Ciò permette all'utente di:

- mantenere ordinata la lista della spesa, in particolare
 - eliminando prodotti già acquistati, riducendo l'impatto visivo della lista e mettendo in evidenza i prodotti ancora da comprare
 - eliminando prodotti che si erano inseriti "per sbaglio" o per cui si è cambiata idea riguardo all'acquisto, garantendo flessibilità e adattabilità alle esigenze (dinamiche) dell'utente

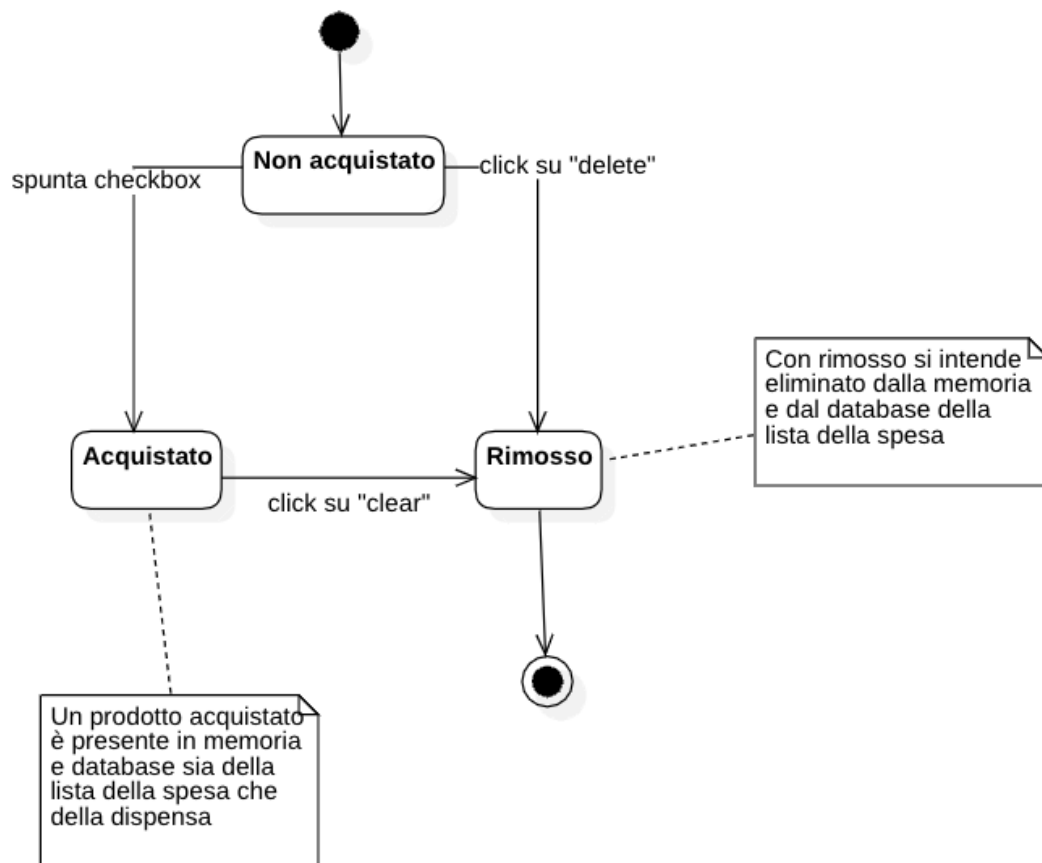


Figura 4.6: Diagramma degli stati della lista della spesa.

4.4 Class diagram

Il class diagram permette di descrivere i tipi di oggetti presenti nel sistema e le relazioni statiche che intercorrono tra di essi. Esso descrive inoltre gli attributi e i metodi di ogni classe, oltre ai vincoli che si applicano nel collegamento tra gli oggetti.

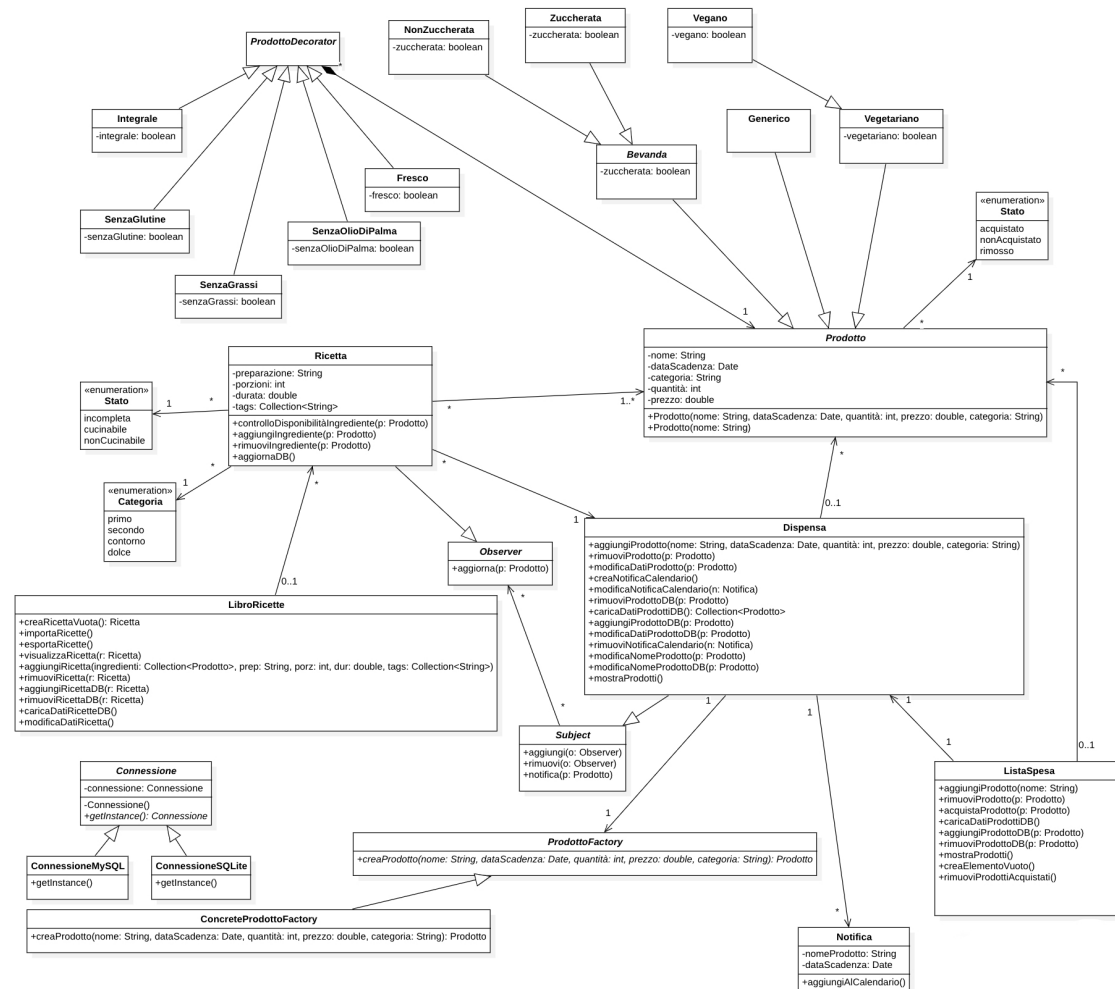


Figura 4.7: Diagramma delle classi.

Per non complicare ulteriormente il diagramma si è scelto di non rappresentare la parte relativa all'interfaccia grafica. I design pattern applicati verranno spiegati nella sezione corrispondente.

4.5 Sequence diagram

Il sequence diagram serve per descrivere singoli scenari dell'applicazioni. Lo scopo è mostrare come il sistema svolge le proprie funzioni. Per una migliore comprensione del funzionamento, nei diagrammi seguenti si è scelto di non modellare i casi di errore del database e dell'interfaccia grafica.

Gli elementi principali di un sequence diagram, usati in tutti gli esempi successivi, comprendono:

- lifeline: rappresenta l'esistenza di un'entità in un dato istante della sequenza (utile quando un'entità è creata o eliminata durante una sequenza)
- entità: attore/parte che interagisce con altri attori/parti; sono dotate di un nome ed eventualmente di una classe
- messaggi: segnali che un partecipante (un'entità) invia ad un altro partecipante oppure provenienti da agenti esterni; possono essere specificati parametri e tipo del valore di ritorno
- frammenti: operatori usati per modellare varie strutture di controllo, permettono di rappresentare molteplici flussi di esecuzione in maniera compatta e precisa
- Esempi di frammenti utilizzati nei diagrammi seguenti
 - alt: rappresenta un *if* con più condizioni (o *guardie*)
 - opt: rappresenta un *if* con una sola condizione
 - loop: usato per modellare i cicli
- Esempi di altri frammenti, usati per modellare l'ordine di esecuzione (seq, strict, par, critical) o l'occorrenza dei messaggi (ignore, consider, assert, neg)
 - break: simboleggia la conclusione anticipata di un ciclo rispetto alla sua canonica esecuzione, spesso in seguito alla rilevazione di un'eccezione
 - seq: rappresenta l'ordinamento di default (ordinamento debole)
 - strict: rappresenta un ordinamento forte
 - par: permette di separare l'ordine degli operandi rendendoli indipendenti l'uno dall'altro (e potenzialmente eseguibili contemporaneamente)
 - critical: definisce un'area atomica nell'interazione, garantendo che essa non venga interrotta da eventi inaspettati
 - ignore: indica che il messaggio ricevuto è irrilevante
 - consider: indica che il messaggio ricevuto è di particolare importanza per l'interazione che si sta analizzando
 - assert: usato per definire situazioni che devono avvenire
 - neg: permette di modellare un'interazione invalida, come ad esempio i messaggi di errore

4.5.1 Sequence diagram della dispensa

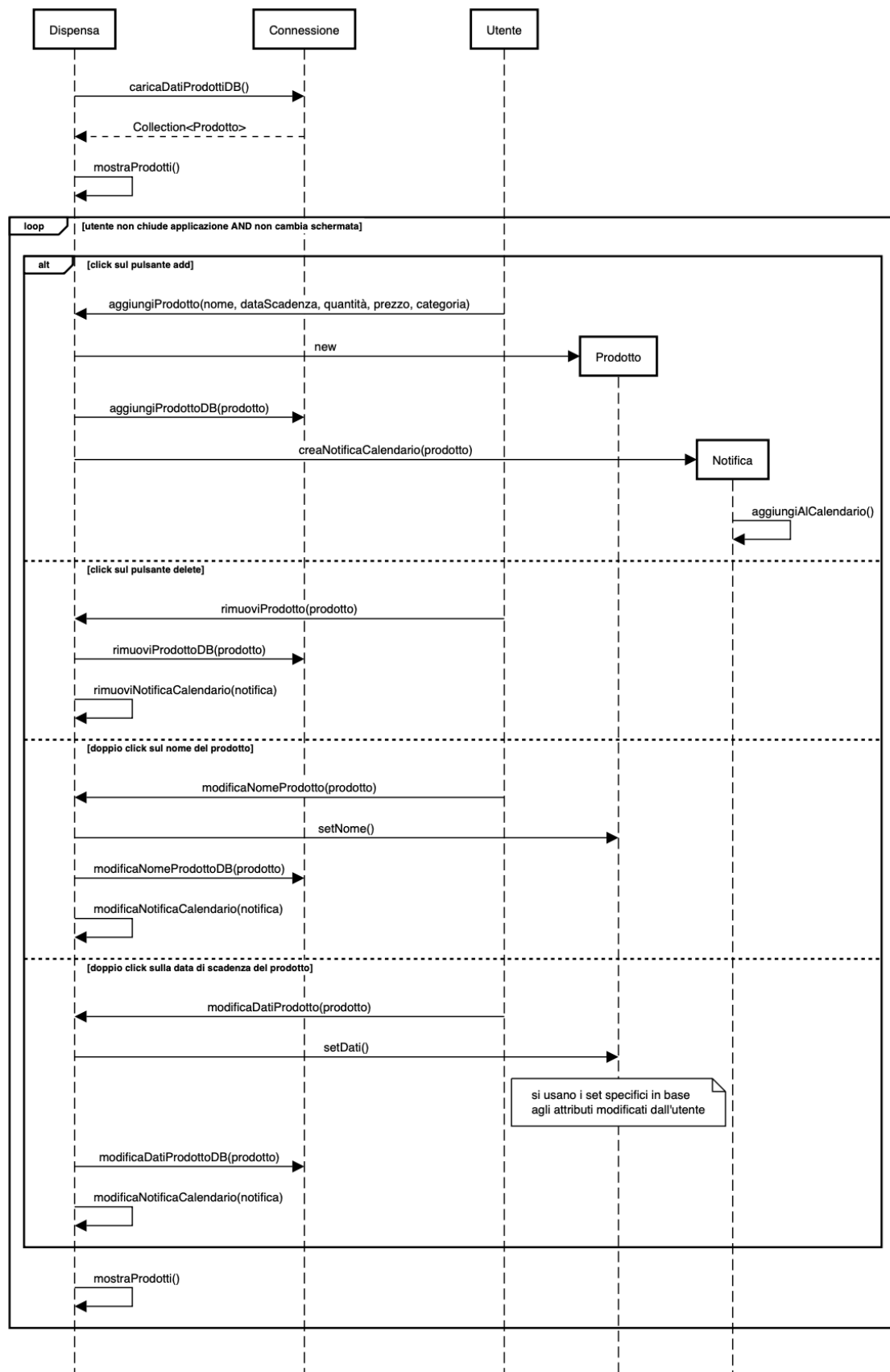


Figura 4.8: Diagramma di sequenza della dispensa

4.5.2 Sequence diagram della lista della spesa

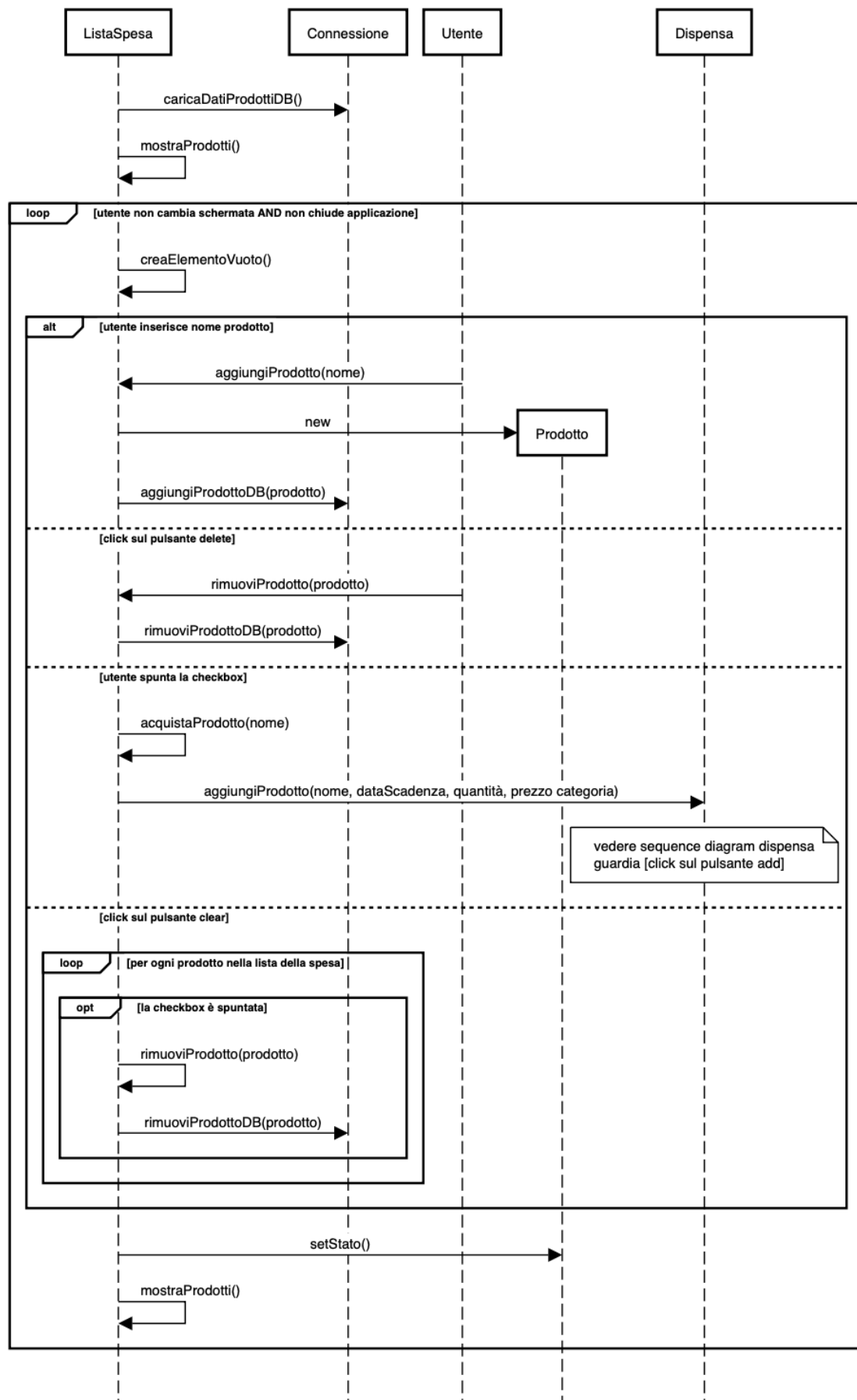
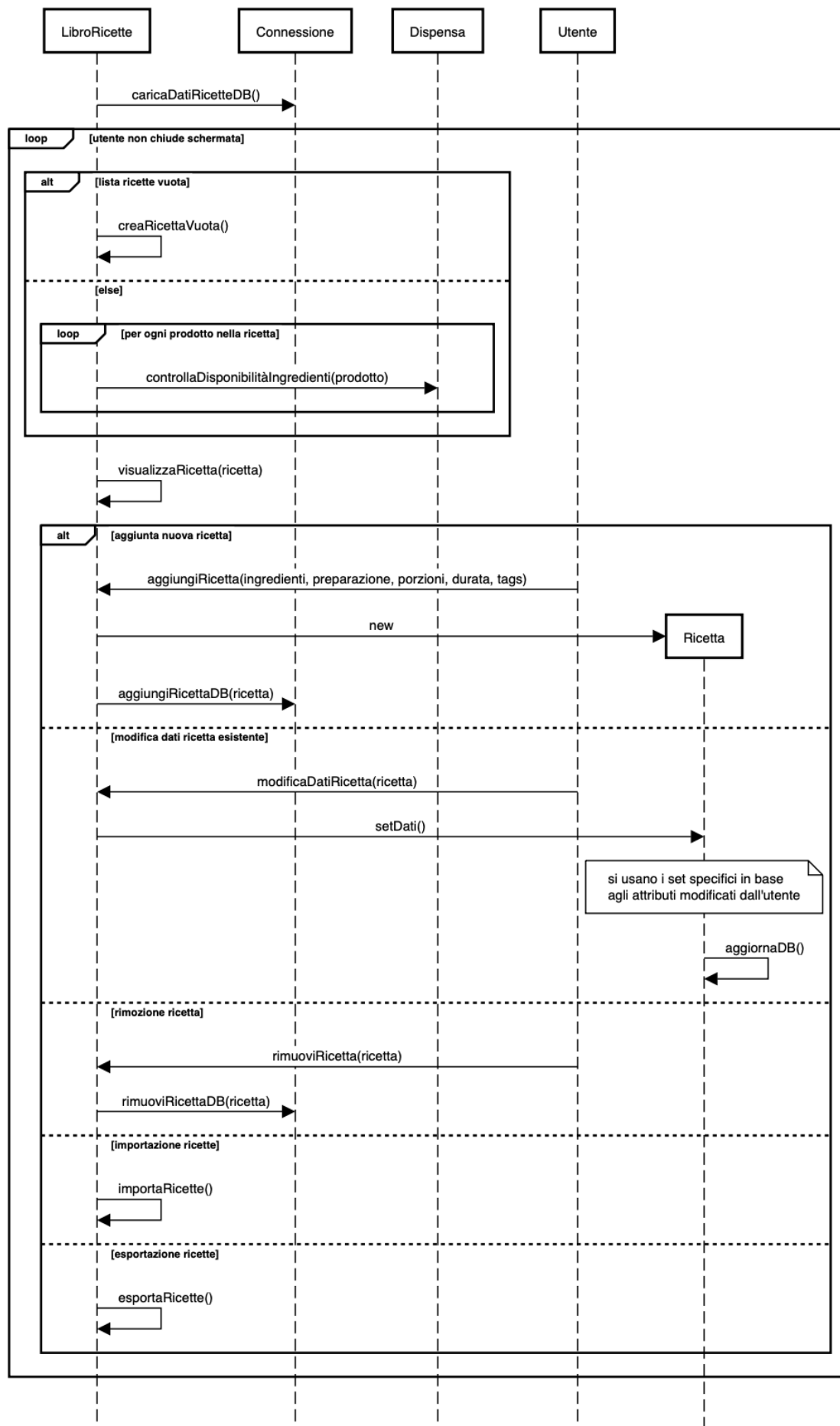


Figura 4.9: Diagramma di sequenza della lista della spesa

4.5.3 Sequence diagram delle ricette



Si noti che, per evitare sovrapposizioni nel diagramma che ne avrebbero resa più difficile la comprensione, il metodo `creaRicettaVuota()` (che si limita a mostrare dei campi di testo vuoti all'utente tramite l'interfaccia grafica) non crea un oggetto di tipo `Ricetta` (come invece avviene nel caso del metodo

`aggiungiRicetta(ingredienti, preparazione, porzioni, durata, tags)` in cui la ricetta creata presenta valori non nulli nei propri attributi).

Capitolo 5

Design Patterns

L'applicazione implementa diversi Design Patterns per garantire una manutenzione più agevole del codice, rendere il software più modulare semplificando l'aggiunta di nuove funzionalità, consentire la parallelizzazione dello sviluppo.

5.1 Factory

Il design pattern Factory ha il compito di occuparsi della creazione dei prodotti. L'utilizzo di questo design permette di evitare di programmare verso le implementazioni concrete, favorendo la programmazione verso le interfacce. Senza di esso, quando in futuro dovranno essere apportare modifiche o estensioni, sarebbe necessario riaprire il codice e cambiarlo di conseguenza (violando quindi l'open-closed principle). Incapsulare la creazione dei prodotti in una apposita classe permette di evitare questo scenario, oltre a facilitare la manutenzione e l'aggiornamento dell'applicazione: le modifiche sono infatti localizzate unicamente nella classe Factory, mentre in mancanza di essa sarebbe necessario cambiare manualmente tutti i punti del codice in cui c'è un "new" di un prodotto. L'utilizzo del design pattern Factory rispetta inoltre il dependency inversion principle.

```
1 // Beverage.java
2 public abstract class Beverage extends Product{
3     boolean sweet;
4     public Beverage(String name, LocalDate expirationDate,
5         int quantity, double price, String category){
6         super(name, expirationDate, quantity, price, category);
7     }
8 }
9 // SweetBeverage.java
10 public class SweetBeverage extends Beverage{
11     public SweetBeverage(String name, LocalDate expirationDate,
12         int quantity, double price, String category) {
13         super(name, expirationDate, quantity, price, category);
14         sweet = true;
15     }
16 }
17
18
```

```
19 // ProductFactory.java
20 public abstract class ProductFactory {
21     public abstract Product createProduct(String name,
22         LocalDate expirationDate, int quantity, double price,
23         String category);
24 }
25 // ConcreteProductFactory.java
26 public class ConcreteProductFactory extends ProductFactory{
27     @Override
28     public Product createProduct(String name, LocalDate expirationDate,
29         int quantity, double price, String category) {
30         if("sweet beverage".equals(category)){
31             return new SweetBeverage(name, expirationDate, quantity,
32                 price, category);
33         } else if("non sweet beverage".equals(category)){
34             return new NonSweetBeverage(name, expirationDate, quantity,
35                 price, category);
36         } else if("generic product".equals(category)){
37             return new GenericProduct(name, expirationDate, quantity,
38                 price, category);
39         } else if("vegetarian product".equals(category)){
40             return new VegetarianProduct(name, expirationDate, quantity,
41                 price, category);
42         } else if("vegan product".equals(category)){
43             return new VeganProduct(name, expirationDate, quantity,
44                 price, category);
45         } else return null;
46     }
47 }
```

Si noti come, nella classe `ConcreteProductFactory`, sia stata utilizzata una tecnica di defensive programming per evitare una `NullPointerException`.

5.2 Iterator

Il design pattern Iterator consente di incapsulare il modo di iterare su una collezione di elementi, nascondendo l'implementazione interna della struttura dati e fornendo un modo unico per navigare attraverso tipi di dato diversi (a patto che implementino l'iterator). Questo design pattern permette di programmare verso le interfacce, disaccoppiando il codice che si occupa di svolgere l'iterazione dalle classi concrete su cui itera. Java ha un proprio iterator e la maggior parte delle collezioni di oggetti implementa la sua interfaccia, pertanto è sufficiente utilizzare i metodi già offerti dalla libreria standard.

```
1 public class Recipe implements Observer{
2     List<Product> ingredients;
3     // other attributes and methods of the class
4     public void checkIngredientAvailability(Product p){
5         for(Iterator<Product> iterator = ingredients.iterator();
6             iterator.hasNext();){
7             Product product = iterator.next();
8             // method implementation
9         }
10    }
11 }
```

5.3 Observer

Il design pattern Observer permette agli oggetti di essere notificati quando qualcosa di loro interesse cambia. Questo design pattern permette di programmare verso le interfacce, svincolandosi dalle implementazioni concrete ed evitando di violare l'open-closed principle tramite l'incapsulamento di ciò che varia e la sua separazione da ciò che non varia. Applicando questo design pattern si ottiene inoltre una riduzione dell'accoppiamento tra le classi che vengono osservate e le classi che osservano. Per esempio, le ricette possono osservare la dispensa in modo da essere notificate quando un prodotto viene aggiunto o rimosso e poter aggiornare la disponibilità degli ingredienti.

```
1 public class Pantry implements Subject {
2     List<Observer> observers = new ArrayList<>();
3     ProductFactory productFactory;
4     // other attributes and methods of the class
5     public void addProduct(String name, LocalDate expirationDate, int
6         quantity, double price, String category){
7         Product p = productFactory.createProduct(name, expirationDate,
8             quantity, price, category);
9         // method implementation
10        notify(p);
11    }
12    public void notify(Product p){
13        for(Iterator<Observer> iterator = observers.iterator(); iterator.
14            hasNext()){
15            Observer observer = iterator.next();
16            observer.update(p);
17        }
18    }
19 }
20
21 public class Recipe implements Observer {
22     // attributes and methods of the class
23     public void update(Product p){
24         checkIngredientAvailability(p);
25     }
26 }
```

5.4 Singleton

Per la connessione al database si utilizza il design Pattern Singleton in modo da garantire l'unicità dell'istanza creata e un punto di accesso globale a questa istanza. Applicando questo design pattern si rispetta il dependency inversion principle, poiché si dipende da una interfaccia per la connessione al database: in questo modo modifiche nell'implementazione concreta del database non impatteranno sulle classi che ne fanno uso. Inoltre rispetta anche l'open-closed principle, in quanto per aggiungere un nuovo database basterà creare una nuova classe che implementi l'interfaccia senza dover modificare il codice relativo alle implementazioni precedenti.

5.5 Decorator

Il design pattern Decorator permette di aggiungere funzionalità (metodi o attributi) agli oggetti dinamicamente. Esso rispetta il principio di programmazione che afferma di favorire la composizione rispetto all'ereditarietà. In questo modo si hanno classi meno accoppiate tra loro, la manutenzione del codice è più semplice, il codice è più flessibile (estensioni future non violeranno l'open-closed principle). Nello specifico, il design pattern è applicato ai prodotti: in questo modo si potranno gestire agevolmente aggiunte e modifiche future, siano esse relative agli attributi dei prodotti o ad operazioni riservate solo a particolari tipologie di prodotti.

```
1 // Product.java
2 public abstract class Product {
3     String name;
4     LocalDate expirationDate;
5     String category;
6     int quantity;
7     double price;
8     public Product(String name, LocalDate expirationDate,
9         int quantity, double price, String category) {
10         this.name = name;
11         this.expirationDate = expirationDate;
12         this.quantity = quantity;
13         this.price = price;
14         this.category = category;
15     }
16     // other methods' implementation
17 }
18 // GenericProduct.java
19 public class GenericProduct extends Food{
20     public GenericProduct(String name, LocalDate expirationDate,
21         int quantity, double price, String category) {
22         super(name, expirationDate, quantity, price, category);
23     }
24 }
25 // ProductDecorator.java
26 public abstract class ProductDecorator {
27     Product product;
28     public ProductDecorator(Product product) {
29         this.product = product;
30     }
31 }
32 // FreshProductDecorator.java
33 public class FreshProductDecorator extends ProductDecorator{
34     boolean fresh;
35     public FreshProductDecorator(Product product) {
36         super(product);
37         fresh = true;
38     }
39 }
40 // GlutenFreeProductDecorator.java
41 public class GlutenFreeProductDecorator extends ProductDecorator{
42     boolean glutenFree;
43     public GlutenFreeProductDecorator(Product product) {
44         super(product);
45         glutenFree = true;
46     }
47 }
```

Capitolo 6

ORM

Per la parte relativa al database si è scelto di utilizzare la tecnica ORM (Object-Relational Mapping) per facilitare l'integrazione di database relazionali con software aderenti al paradigma della programmazione orientata agli oggetti. L'ORM si pone infatti come layer intermedio fra un servizio (in questo caso l'applicazione) e il database utilizzato dal servizio (in questo caso MySQL). Di seguito verranno analizzati i vantaggi e gli svantaggi di tale tecnica, in che contesti è conveniente utilizzarla e perché si è scelto di implementarla nell'applicazione oggetto di questa tesi; infine saranno mostrati degli stress tests volti a confrontare le performance di ORM (in questo caso specifico si è utilizzato come applicativo Hibernate) e JDBC. Nel seguito, per preservare la generalità della trattazione, si farà riferimento ad "ORM" e "Non-ORM" per indicare rispettivamente Hibernate e JDBC.

6.1 Vantaggi

Usare la tecnica ORM risulta vantaggiosa in termini di:

- produttività
- progettazione del codice
- testing
- versatilità
- gestione della cache
- sicurezza
- ecosistema
- gestione delle query

Di seguito si analizza ognuno dei punti precedenti più nel dettaglio.

Produttività: senza l'ORM, l'ingegnere del software che progetta l'applicazione deve occuparsi anche della scrittura del corrispondente codice SQL; in particolare, a seconda del database utilizzato, il linguaggio SQL specifico può essere diverso e deve essere conosciuto dal programmatore. La scrittura di statements SQL può richiedere molto tempo, delegandola all'ORM si permette all'ingegnere del software di occuparsi solo della parte di codice relativa al linguaggio a oggetti, riducendo il tempo necessario allo sviluppo dell'intera applicazione e, di conseguenza, il suo time-to-market. L'ORM è quindi particolarmente indicato per i progetti che hanno vincoli di tempo cruciali per il successo del prodotto, come per esempio nel caso di prodotti che devono essere lanciati per la prima volta sul mercato. Permette inoltre di gestire automaticamente operazioni comuni come CRUD (Create, Read, Update, Delete), con conseguente risparmio di tempo e impegno.

Progettazione del codice: nel momento in cui si implementa correttamente l'ORM, esso induce anche l'utilizzo di design patterns che fanno uso di best practices per la progettazione dell'applicazione. Si ottiene quindi un codice meglio strutturato e più facilmente comprensibile, ciò semplifica anche la sua manutenzione, che è l'aspetto chiave per il successo (anche in termini economici) del software e il suo aggiornamento.

Testing: dal momento che l'ORM si occupa di generare il codice SQL necessario, una volta che è stato testato il codice per l'accesso ai dati, non è necessario testarlo nuovamente a meno che non venga cambiata la logica con cui i dati sono acceduti. Molti ORM permettono inoltre di gestire i test in maniera tale da garantire che non lascino dati residui nel database.

Versatilità: la generazione del codice ad opera dell'ORM permette di cambiare facilmente database senza la necessità di modificare il codice; nel caso specifico dell'applicazione in esame, è possibile passare facilmente da MySQL (utilizzato per la versione desktop) a SQLite per una versione mobile. Sarà infatti l'ORM che si occuperà della generazione del codice nell'SQL proprio del nuovo database selezionato.

Gestione della cache: essendo le entità salvate in memoria è necessario meno tempo per il loro caricamento sul database e si riduce il numero di query effettuate.

In generale, come linea guida si può affermare che l'utilizzo di ORM sia particolarmente indicato nel caso in cui gli oggetti e le modalità di accesso ad essi non siano particolarmente complessi. Per query semplici, come per esempio la restituzione di oggetti dal database, l'impiego di ORM permette di risparmiare molto tempo. Sebbene vi siano diversi ORM (fra cui Hibernate) che permettono di accedere alla connessione al database molto facilmente, se vi è necessità di numeroso codice SQL specifico dell'applicazione il rischio è quello di non sfruttare appieno l'ORM (il cui obiettivo è proprio minimizzare la scrittura di codice SQL).

Sicurezza: se non c'è una appropriata validazione dei valori in input (come possono essere, ad esempio, quelli dei cookies) prima di passarli a delle query SQL eseguite dal database, ci si espone al rischio di un attacco tramite SQL injection, una del-

le più semplici e potenzialmente una delle più pericolose minacce per la sicurezza di un'applicazione. Questa eventualità può essere scongiurata tramite l'utilizzo di un ORM (a patto che non vi sia del puro SQL in altre parti dell'applicazione) dal momento che spesso fa uso di query parametrizzate, tuttavia non è strettamente necessario ed uno sviluppatore esperto può facilmente risolvere questo problema senza il bisogno di ricorrere a un ORM.

Ecosistema: ORM popolari hanno spesso ampie communities, offrono plugins, estensioni e supporto che possono accelerare lo sviluppo dell'applicazione.

Gestione delle query: molti ORM offrono costruttori di query o linguaggi specifici per un particolare dominio, che semplificano la creazione di query complesse rendendole più leggibili e mantenibili.

6.2 Svantaggi

Fra gli svantaggi da considerare se si utilizza l'ORM vi sono:

- performance overhead
- controllo limitato
- curva di apprendimento
- problematiche di astrazione
- problemi di scalabilità
- problemi di compatibilità
- complessità del mapping
- problemi di sicurezza

Di seguito si analizza ognuno dei punti precedenti più nel dettaglio.

Performance overhead: l'ORM genera spesso query SQL complesse che possono essere meno efficienti rispetto a query SQL scritte direttamente, questo si nota soprattutto con datasets molto grandi o operazioni complesse.

Controllo limitato: adattare le query SQL in funzione delle performance è più difficile con l'ORM, dal momento che esso astrae l'SQL sottostante. Inoltre l'ORM fornisce un approccio comune a tutti i database, limitando l'uso di funzionalità avanzate specifiche di un certo DBMS ed eventuali ottimizzazioni.

Curva di apprendimento: imparare come un ORM mappa l'SQL può essere complesso, comprendere le implicazioni sulle performance e le best practices richiede molto tempo. Identificare problemi durante il debugging può essere più difficile poiché gli sviluppatori devono avere cognizione sia delle astrazioni dell'ORM che delle query SQL sottostanti che esso genera.

Problematiche di astrazione: le differenze concettuali tra la programmazione orientata agli oggetti e i database relazionali può portare a modelli inefficienti, inoltre il comportamento dell'ORM potrebbe differire da quello atteso, specialmente quando si trova a gestire query personalizzate o relazioni complesse fra le entità.

Problemi di scalabilità: l'ORM può incontrare difficoltà legate alla scalabilità con datasets molto grandi o applicazioni fortemente concorrenti, dove l'ottimizzazione di query e database è cruciale.

Problemi di compatibilità: mantenere la libreria ORM aggiornata con la versione del database può essere difficile, gli aggiornamenti possono causare l'insorgere di incompatibilità. Inoltre i cambiamenti nella stessa libreria ORM possono introdurre modifiche che richiedono un refactoring significativo del codice.

Complessità del mapping: gestire concetti complessi legati alla programmazione orientata agli oggetti come ereditarietà e polimorfismo può essere inefficiente in un contesto come quello dei database relazionali che non li prevedono, inoltre modificare lo schema del database in parallelo con gli oggetti che vengono memorizzati può essere complesso e condurre facilmente ad errori; è pertanto necessario pianificare delle accorte strategie di migrazione.

Problemi di sicurezza: una configurazione errata o un utilizzo improprio dell'ORM può comunque introdurre vulnerabilità, inoltre le query generate automaticamente potrebbero non essere state accuratamente testate per determinate funzioni di sicurezza rispetto a query SQL scritte a mano.

6.3 Stress tests

Di seguito vengono proposti (prima in forma tabellare e poi in forma grafica) una serie di test volti a verificare la differenza di prestazioni fra l'applicazione che non fa uso di ORM e la versione dell'applicazione che lo utilizza.

6.3.1 Inserimento di prodotti nel database

Si consideri come task l'inserimento di prodotti nel database. Ogni prodotto inserito corrisponde ad una riga aggiunta nella tabella.

Tabella 6.1: Inserimento di prodotti nel database.

Numero di prodotti	Tempo (JDBC)	Tempo (ORM)
1K	~ 0.6 secondi	~ 1.5 secondi
10K	~ 2 secondi	~ 4 secondi
100K	~ 14 secondi	~ 25 secondi
1M	~ 134 secondi	~ 231 secondi

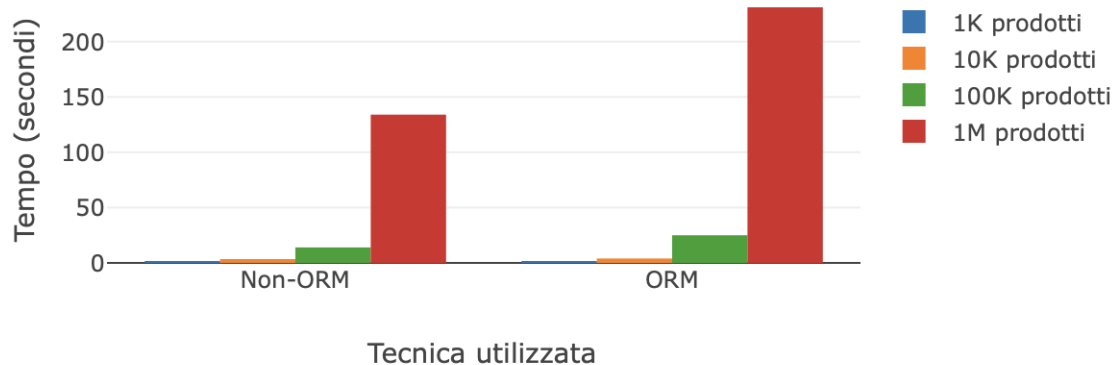


Figura 6.1: Inserimento di prodotti nel database.

In caso di quantità di dati relativamente ridotte, l'overhead introdotto dall'ORM, seppur presente, risulta impercettibile per l'utente. Quando ci si trova a dover gestire quantità di dati elevate, emergono i problemi di scalabilità descritti precedentemente che si concretizzano nell'impossibilità pratica di usare l'applicazione.

6.3.2 Eliminazione di prodotti dal database

Si consideri come task l'eliminazione di prodotti dal database. Ogni prodotto eliminato corrisponde ad una riga rimossa dalla tabella.

Tabella 6.2: Eliminazione di prodotti dal database.

Numero di prodotti	Tempo (Non-ORM)	Tempo (ORM)
1K	~ 0.6 secondi	~ 1.8 secondi
10K	~ 2 secondi	~ 5 secondi
100K	~ 14 secondi	~ 32.7 secondi
1M	~ 140 secondi	~ 312 secondi

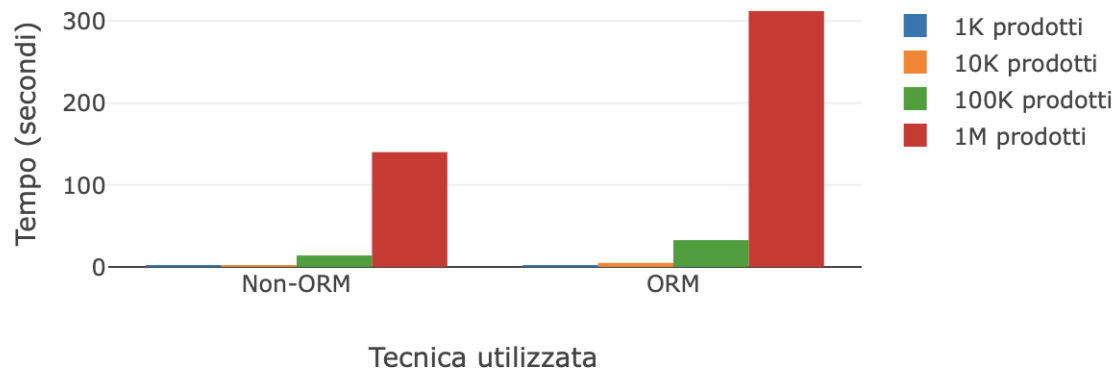


Figura 6.2: Eliminazione di prodotti dal database.

Le prestazioni dell'applicazione che non fa uso di ORM sono simili a quelle ottenute con l'inserimento di prodotti nel database, mentre si osserva un discreto peggioramento dell'ORM. Ciò è dovuto al fatto che, mentre l'operazione di inserimento attraverso ORM consiste nell'aggiungere l'oggetto che si desidera inserire al contesto di persistenza e successivamente nel database, l'eliminazione richiede la riassociazione dell'oggetto che si desidera rimuovere al contesto di persistenza dell'applicazione, la rimozione effettiva dello stesso dal contesto di persistenza, la generazione di una query per la rimozione e la rimozione dell'oggetto dal database.

6.3.3 Modifica di prodotti nel database

Si consideri come task la modifica di prodotti nel database. Ogni prodotto modificato corrisponde ad una riga della tabella aggiornata.

Tabella 6.3: Modifica di prodotti nel database.

Numero di prodotti	Tempo (Non-ORM)	Tempo (ORM)
1K	~ 0.6 secondi	~ 2 secondi
10K	~ 2.3 secondi	~ 6 secondi
100K	~ 16.3 secondi	~ 38 secondi
1M	~ 162 secondi	~ 366 secondi

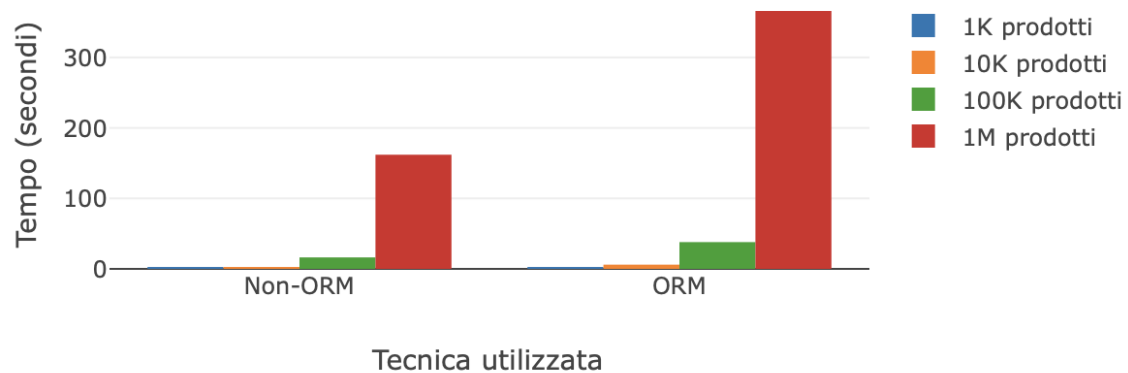


Figura 6.3: Modifica di prodotti nel database.

Entrambe le versioni dell'applicazione subiscono una degradazione delle prestazioni, dovuta principalmente alla maggiore complessità dell'operazione da eseguire.

6.3.4 Join fra tabelle del database

Si consideri come task il join fra tabelle del database. In particolare, per recuperare i dati relativi alle ricette, sarà necessario un join con la tabella dei prodotti. Per testare i limiti sia pratici che teorici, si è scelto di considerare il caso in cui tutte le ricette contengono tutti i prodotti, ed il numero di prodotti è uguale a quello delle ricette. Questo comporta la necessità di ridurre l'ordine di grandezza del numero di elementi testati, in quanto nella tabella che rappresenta l'associazione fra ricetta e prodotto c'è un numero di righe che cresce quadraticamente rispetto al numero di ricette/prodotti (che sono quelli a cui si fa riferimento nella tabella).

Tabella 6.4: Join fra tabelle del database.

Numero di prodotti	Tempo (Non-ORM)	Tempo (ORM)
1K	~ 1.2 secondi	~ 10 secondi
2K	~ 4.2 secondi	~ 40 secondi
2.5K	~ 7.2 secondi	~ 65 secondi
3K	~ 9.5 secondi	~ 110 secondi

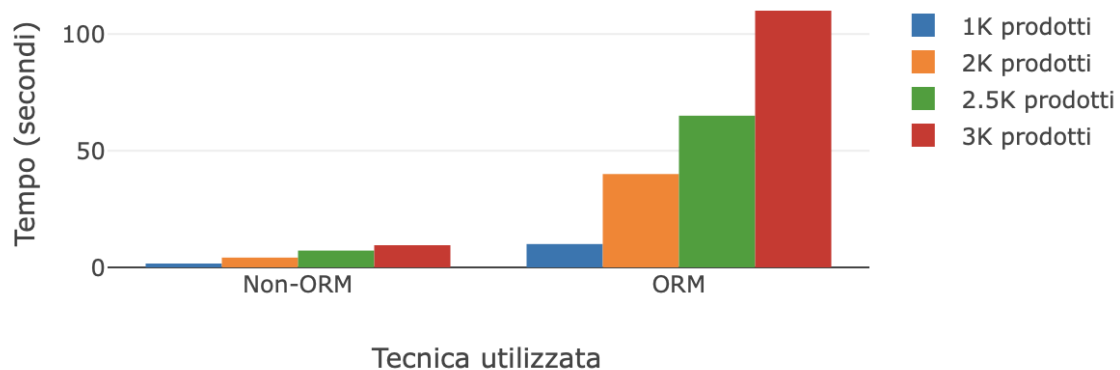


Figura 6.4: Join fra tabelle del database.

Questo test mette in evidenza i problemi di scaabilità che si riscontrano con l'utilizzo di tecniche ORM su datasets di ingenti dimensioni, tuttavia vale la pena notare come gli scenari considerati per questo test abbiano un valore principalmente teorico, dal momento che nell'utilizzo comune è improbabile che un utente registri nel database 1000 ricette ognuna con 1000 ingredienti, perciò per le finalità dichiarate dell'applicazione e il caso di utilizzo di un utente medio non emergeranno, salvo casi eccezionali, queste criticità.

Capitolo 7

Conclusioni

Appare evidente come l'overhead in caso di grandi quantità di dati appaia non trascurabile e risulti nell'impossibilità pratica di utilizzare l'ORM. Tuttavia, nel caso di quantità di dati relativamente ridotte, seppur presente esso è impercettibile per l'utente. Nel caso in esame, essendo gli oggetti inseriti nel database i prodotti presenti nella dispensa di un utente, ed essendo le operazioni svolte su di essi molto semplici, è improbabile che si raggiungano numeri tali da far sperimentare all'utilizzatore dell'applicazione un evidente calo di prestazioni, mentre per il programmatore l'utilizzo di ORM semplifica notevolmente lo sviluppo, la manutenzione e il testing del codice. In particolare, l'adozione dell'ORM permette di semplificare concettualmente la visione dell'applicazione e di ragionare in termini di un unico insieme, dal momento che le tabelle del database vengono mappate, e quindi possono essere gestite, come classi Java, delegando al tool ORM l'onere di far coesistere i due ambiti e risolvere eventuali differenze o criticità, mentre in precedenza ogni funzionalità e/o cambiamento doveva essere ragionato (in fase di progettazione) e codificato (in fase di implementazione) sia in termini di puro SQL (con logica rivolta a database relazionali) che in termini di programmazione a oggetti. Il risparmio garantito dall'ORM si articola non solo in termini di tempo (con tutti i benefici, anche economici, che ne conseguono), ma soprattutto in termini di aree di interesse per lo sviluppatore, che può così concentrarsi sulla logica applicativa di alto livello senza doversi preoccupare di risolvere dettagli puramente implementativi e di basso livello. L'astrazione fornita dall'ORM permette di cambiare facilmente gli elementi concreti sottostanti (ovvero lo specifico database che si utilizza) minimizzando le modifiche necessarie per garantire la compatibilità e il funzionamento con il resto dell'applicazione: questo permette di avere un software flessibile, dinamico, pronto ad adattarsi alle esigenze che si presenteranno nel corso del tempo. Perché se un software viene usato (ed è l'obiettivo, nonché la speranza, di chi lo progetta), esso dovrà inevitabilmente cambiare, e prepararsi in anticipo per questi cambiamenti può fare la differenza fra il successo e il fallimento di un prodotto nel lungo periodo.

Bibliografia

- [1] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Pearson Addison Wesley, 1997.
- [2] Russ Miles and Kim Hamilton. *Learning UML 2.0 A Pragmatic Introduction to UML*. O'Reilly Media, 2006.
- [3] Harsh Singh. Why orms aren't always a great idea. <https://dev.to/harshhhdev/why-orms-arent-always-a-great-idea-41kg>, 2022.
- [4] Mithun Sasidharan. Should i or should i not use orm ? <https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce>, 2016.
- [5] Agridence. To orm or not to orm. <https://www.linkedin.com/pulse/orm-agridence>, 2023.
- [6] Bill Armelin. To orm or not to orm: Do we keep it or replace it? <https://akfpartners.com/growth-blog/to-orm-or-not-to-orm>, 2022.