

# Brew Day!

Luca Pincioli - 885969, Marco Ferioli - 879277,  
Francesco Trolli - 889039, Stefano Spinelli - 887511

21 febbraio 2024



# Indice

<b>1 Introduzione</b>	<b>3</b>
1.1 Traccia del progetto . . . . .	3
1.2 Processo di sviluppo . . . . .	4
1.2.1 Suddivisione delle iterazioni . . . . .	4
1.2.2 Applicazione del modello UP . . . . .	5
<b>2 Analisi</b>	<b>6</b>
2.1 Modello dei casi d'uso . . . . .	6
2.1.1 Attore primario: utente . . . . .	7
2.1.2 Gestisci ricette . . . . .	7
2.1.3 Gestisci ingredienti disponibili . . . . .	7
2.1.4 Gestisci birre . . . . .	8
2.1.5 Gestisci attrezzi . . . . .	8
2.1.6 Aggiungi nota . . . . .	8
2.1.7 Crea lista della spesa . . . . .	8
2.2 Modello di dominio . . . . .	12
2.3 Diagrammi di sequenza di sistema (SSD) . . . . .	13
<b>3 Progettazione</b>	<b>16</b>
3.1 Android Clean Architecture . . . . .	16
3.2 Model-View-ViewModel (MVVM) Pattern . . . . .	18
3.3 Repository Pattern . . . . .	19
3.4 Data Source, Room, DAO . . . . .	20
3.5 Diagramma dell'architettura . . . . .	21
3.6 Diagrammi di sequenza (SD) . . . . .	22
3.7 Diagramma delle classi software . . . . .	24
3.8 Diagrammi degli stati ed attività . . . . .	25
<b>4 Pattern applicati e principi seguiti</b>	<b>27</b>
4.1 Design pattern . . . . .	27
4.1.1 Service Locator . . . . .	27
4.1.2 Factory Method . . . . .	28
4.1.3 Observer . . . . .	29
4.1.4 Singleton . . . . .	30
4.1.5 Strategy . . . . .	31
4.2 Design Principles . . . . .	32
4.3 Analisi con Understand . . . . .	32

# 1 Introduzione

## 1.1 Traccia del progetto

“Brew Day!” è un’applicazione che consente ai birrai domestici di mantenere un database organizzato delle loro ricette di birra. L’applicazione consente agli utenti di creare, memorizzare e modificare le ricette, e in seguito di cancellarle, se l’utente lo desidera. L’applicazione è destinata esclusivamente ai produttori di birra “all grain” e quindi tutte le ricette sono destinate a questo tipo di birra (le birre “da estratto” non sono supportate).

Ogni birraio domestico ha un’attrezzatura specifica, le cui caratteristiche portano a un particolare ”batch size”, il numero massimo di litri che possono essere prodotti in una singola sessione.

Le ricette comprendono, oltre all’acqua, anche altri ingredienti:

- Malto
- Luppolo
- Lieviti
- Zuccheri
- Additivi

Sebbene i birrai preferiscano creare ricette riferite a valori concreti, come chilogrammi di uno specifico malto o grammi di uno specifico luppolo, l’applicazione deve memorizzare queste ricette in qualche misura ”assoluta”, che consenta una conversione diretta della ricetta quando l’attrezzatura, e di conseguenza la dimensione del lotto, viene aggiornata. Ad esempio, è possibile esprimere le quantità di malto come percentuale del totale e il luppolo come grammi per litro di mash.

Oltre alle ricette vere e proprie, l’applicazione deve mantenere istanze di ricette, cioè particolari birre basate su una ricetta; queste istanze possono essere accompagnate da note che si riferiscono a questioni che possono influenzare la birra risultante e che i birrai desiderano tenere registrate. Un tipo particolare di note è quello delle note di degustazione, che consente ai birrai di tenere traccia delle opinioni su una birra di un particolare prodotto.

Oltre a queste caratteristiche tradizionali di Brew Day, l’applicazione mantiene un elenco degli ingredienti disponibili. Questo permette ai birrai di essere avvisati degli ingredienti mancanti per la prossima produzione. Un’istanza di ricetta, cioè una particolare birra, dovrebbe consentire agli utenti di aggiornare l’elenco degli ingredienti

disponibili, sottraendo gli ingredienti usati da quelli disponibili. In relazione a queste informazioni, Brew Day! deve supportare una funzione utile per i birrai: La funzione "cosa devo fare oggi?" passa in rassegna il database delle ricette e sceglie la ricetta che massimizza l'uso degli ingredienti disponibili, tenendo ovviamente conto della capacità dell'attrezzatura.

## 1.2 Processo di sviluppo

È stato utilizzato il processo di sviluppo UP (Unified Process); si tratta di un processo iterativo appartenente ai modelli di sviluppo agili, dove per "agilità" si intende la capacità del modello di essere flessibile ed adattabile.

### 1.2.1 Suddivisione delle iterazioni

Per rispettare i tempi di scadenza per il progetto, il lavoro sviloto è stato diviso in due iterazioni:

1. **Prima iterazione.** È stata effettuata una prima fase di analisi, andando a produrre una visione di insieme del progetto attraverso il diagramma dei casi d'uso e il modello di dominio. Per quanto riguarda la progettazione, in questa fase è stata inizialmente progettata l'architettura dell'applicazione secondo le linee guida della "Clean Architecture", lo schema architettonico consigliato da Google per lo sviluppo di applicazioni Android. Si è passati quindi ad un'analisi più approfondita dei requisiti critici, i quali costituiscono una solida base per l'intera applicazione: dopo averli descritti sotto forma di casi d'uso, sono stati prodotti i relativi diagrammi di sequenza di sistema, al fine di descrivere il comportamento atteso dall'applicazione per ciascun caso d'uso considerato. Per quanto riguarda la progettazione, sulla base dell'analisi effettuata sono stati redatti il diagramma delle classi (in forma parziale) e i diagrammi di sequenza che modellano le operazioni più complesse.
2. **Seconda iterazione.** L'analisi e la progettazione si sono concentrate sui requisiti funzionali non approfonditi nell'iterazione precedente: essi sono stati descritti sotto forma di casi d'uso e documentati attraverso i relativi diagrammi di sequenza di sistema. Successivamente, il diagramma delle classi di progetto è stato aggiornato e sono stati redatti il diagramma degli stati e il diagramma delle attività, i quali hanno permesso di modellare adeguatamente alcuni aspetti critici della realtà presa in esame.

### 1.2.2 Applicazione del modello UP

Lungo le due iterazioni, il processo di analisi e produzione del software è stato suddiviso in quattro fasi, così come suggerito dal modello UP:

1. **Ideazione.** Sono stati definiti gli obiettivi del progetto, analizzando la traccia fornita. In particolare, sono state formulate le prime stime approssimative sui requisiti.
2. **Elaborazione.** I requisiti sono stati definiti in maniera più dettagliata e analizzati mediante i vari elaborati UML necessari, al fine di avere una corretta visione d'insieme sul progetto. Il nucleo del sistema è stato progettato ed implementato, così da garantire subito una base solida all'applicativo.
3. **Costruzione.** Si tratta della fase che ha richiesto il maggior sforzo, poiché l'obiettivo da raggiungere è stato quello di implementare e testare il sistema in modo da soddisfare i requisiti richiesti.
4. **Transizione.** Il lavoro è stato finalizzato svolgendo le ultime prove e preparando il sistema per poterlo consegnare.

Per la prima volta abbiamo provato l'esperienza di lavorare come un "team" e abbiamo cercato di seguire correttamente il processo di sviluppo software scelto, rispettando gli obiettivi che ci siamo prefissati inizialmente e mettendo in pratica gli insegnamenti che abbiamo acquisito durante il corso.

Durante lo sviluppo abbiamo organizzato stand-up meeting giornalieri con lo scopo di fare il punto della situazione: in questo modo è risultato semplice rilevare rapidamente eventuali problematiche e potervi porre rimedio. Inoltre, i meeting giornalieri hanno anche permesso al team di mantenere il corretto focus sugli obiettivi da raggiungere.

Al fine di velocizzare lo sviluppo, abbiamo anche sfruttato la pratica del pair programming: dividendo il team in due coppie è stato possibile lavorare in parallelo sulle parti più complesse del sistema, risparmiando molto tempo e risorse.

## 2 Analisi

Per poter raccogliere e organizzare i requisiti sono stati compilati gli elaborati UML presentati di seguito. In particolare, per quanto riguarda i casi d'uso, abbiamo deciso di descrivere in un formato dettagliato quelli che abbiamo ritenuto più complessi. Tutti gli altri sono stati rappresentati in un formato più conciso, dove viene descritto esclusivamente lo scenario di successo.

### 2.1 Modello dei casi d'uso

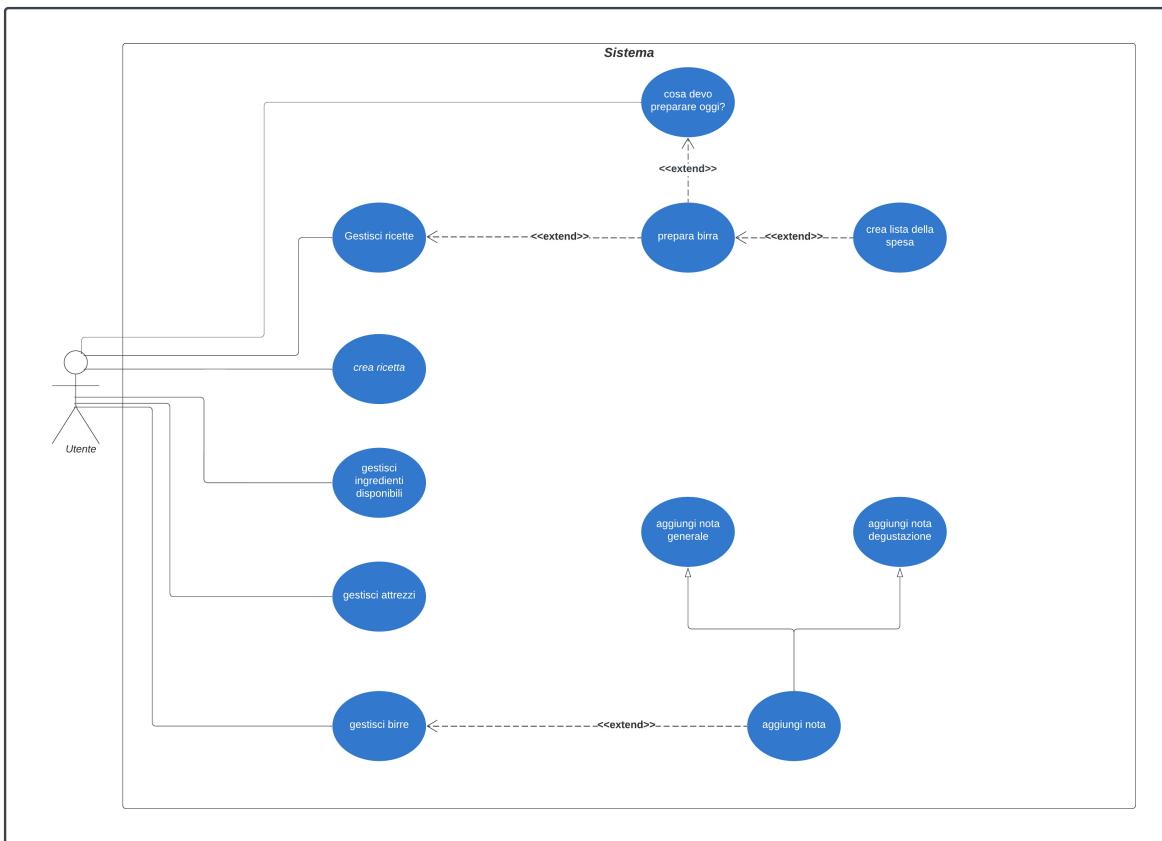


Figura 1: Diagramma dei casi d'uso

### **2.1.1 Attore primario: utente**

L’utente che utilizza l’applicativo è l’attore primario a cui sono collegati i casi d’uso definiti.

### **2.1.2 Gestisci ricette**

Il caso d’uso ”gestisci ricetta” indica che l’utente può visualizzare, modificare e cancellare una tra le ricette che ha registrato nel sistema. Lo scenario principale di successo è il seguente:

1. L’utente sceglie una ricetta tra quelle disponibili.
2. Il sistema permette all’utente di visualizzare i dettagli della ricetta che ha scelto.  
Se l’utente decide di modificare la ricetta:
  - (a) L’utente inserisce i nuovi parametri che caratterizzano la ricetta.
  - (b) La ricetta viene aggiornata e memorizzata nel sistema.

Se invece l’utente sceglie di cancellare la ricetta, quest’ultima viene eliminata dal sistema.

### **2.1.3 Gestisci ingredienti disponibili**

L’utente ha la possibilità di visualizzare la lista di tutti gli ingredienti che ha registrato nel sistema e di interagire con essa. Il caso d’uso definisce il seguente scenario di successo:

1. L’utente visualizza la lista di tutti gli ingredienti che ha registrato nel sistema.
2. L’utente può modificare manualmente la quantità degli ingredienti che ha registrato.

L’operazione di aggiunta di un ingrediente corrisponde al caricamento di un prodotto acquistato dall’utente. La rimozione di una certa quantità di ingrediente è stata modellata per permettere all’utente di mantenere coerenza tra la quantità di ingredienti registrati nell’app ed il suo magazzino. Infatti, l’utente potrebbe voler utilizzare i suoi ingredienti in preparazioni non collegate all’uso dell’applicativo e quindi deve avere la possibilità di ridurre la quantità di ingredienti registrati nell’applicazione.

#### **2.1.4 Gestisci birre**

Il caso di utilizzo "gestisci birre" modella i requisiti funzionali che riguardano le birre. L'utente può:

1. visualizzare la lista delle birre la cui produzione è terminata o in corso.
2. controllare quali ingredienti sono stati utilizzati per produrre una birra.
3. visualizzare la lista degli attrezzi utilizzati da una birra in fase di produzione.
4. terminare la produzione di una birra manualmente.

#### **2.1.5 Gestisci attrezzi**

L'utente possiede degli attrezzi e questi ultimi devono essere registrati nel sistema per poter scegliere ed avviare delle ricette. Lo scenario di successo è definito in questo modo:

1. L'utente visualizza la lista degli attrezzi memorizzati nel sistema.
2. L'utente può scegliere se:
  - (a) Registrare un attrezzo nel sistema.
  - (b) Modificare un attrezzo nel sistema.
  - (c) Rimuovere un attrezzo dal sistema.

#### **2.1.6 Aggiungi nota**

Si tratta di una generalizzazione dei casi d'uso "Aggiungi nota generale" ed "aggiungi nota degustazione". Quando l'utente prepara una birra può registrare una nota generale, mentre la nota di degustazione può essere associata esclusivamente alle birre la cui produzione è terminata.

#### **2.1.7 Crea lista della spesa**

Il caso di utilizzo definisce la possibilità dell'utente di generare una lista della spesa a partire dagli ingredienti che consuma per effettuare la produzione di una certa quantità di litri di una determinata birra.

---

**Caso d'uso: "Cosa prepariamo oggi?"**

---

<b>Portata:</b>	Sistema BrewDay
<b>Livello:</b>	Obiettivo utente
<b>Attore primario:</b>	Utente
<b>Precondizioni:</b>	Nessuna
<b>Garanzia di successo:</b>	All'utente viene proposta la preparazione di una ricetta tra quelle inserite nel sistema, selezionata in base agli attrezzi e agli ingredienti attualmente disponibili
<b>Scenario principale di successo:</b>	<ol style="list-style-type: none"><li>1. L'utente indica se vuole una ricetta che massimizzi i litri prodotti o il consumo degli ingredienti</li><li>2. Il sistema consulta la lista delle ricette disponibili</li><li>3. Il sistema sceglie la ricetta che meglio rispetta il criterio di massimizzazione scelto dall'utente</li><li>4. Il sistema mostra all'utente la ricetta scelta</li></ol>
<b>Estensioni:</b>	<ol style="list-style-type: none"><li>3. (a) <b>Fallimento:</b> il sistema non trova nessuna ricetta registrata</li><li>(b) <b>Fallimento:</b> il sistema non trova ricette compatibili con gli ingredienti attualmente a disposizione</li><li>(c) <b>Fallimento:</b> il sistema non trova ricette compatibili con gli attrezzi attualmente a disposizione</li></ol>
<b>Altre informazioni:</b>	Nessuna

---

---

**Caso d'uso: "*Prepara birra*"**

---

<b>Portata:</b>	Sistema BrewDay
<b>Livello:</b>	Obiettivo utente
<b>Attore primario:</b>	Utente
<b>Precondizioni:</b>	L'utente deve aver registrato almeno una ricetta
<b>Garanzia di successo:</b>	Viene creata un'istanza di ricetta scelta dall'utente
<b>Scenario principale di successo:</b>	<ol style="list-style-type: none"><li>1. L'utente seleziona una ricetta dalla lista di ricette</li><li>2. L'utente seleziona la quantità di birra che vuole produrre</li><li>3. L'utente visualizza gli ingredienti richiesti per produrre la quantità di birra scelta in relazione alla ricetta selezionata</li><li>4. L'utente conferma l'inizio della preparazione della birra scelta</li></ol>
<b>Estensioni:</b>	<ol style="list-style-type: none"><li>3. (a) <b>Fallimento:</b> il sistema non dispone degli attrezzi necessari per dare inizio alla preparazione</li><li>(b) <b>Fallimento:</b> il sistema non dispone degli ingredienti necessari per dare inizio alla preparazione</li></ol>
<b>Altre informazioni:</b>	Nessuna

---

---

**Caso d'uso: "Crea ricetta"**

---

<b>Portata:</b>	Sistema BrewDay
<b>Livello:</b>	Obiettivo utente
<b>Attore primario:</b>	Utente
<b>Precondizioni:</b>	Nessuna
<b>Garanzia di successo:</b>	Viene creata una ricetta con i dati inseriti dall'utente
<b>Scenario principale di successo:</b>	<ol style="list-style-type: none"><li>1. L'utente inserisce il nome che vuole dare alla ricetta e il numero di litri indicativi</li><li>2. L'utente inserisce gli ingredienti necessari per produrre tale ricetta</li><li>3. L'utente conferma i dati inseriti per la ricetta</li></ol>
<b>Estensioni:</b>	<ol style="list-style-type: none"><li>3. <b>Fallimento:</b> l'utente non ha selezionato abbastanza ingredienti</li></ol>
<b>Altre informazioni:</b>	L'utente deve aver selezionato almeno 4 ingredienti

---

## 2.2 Modello di dominio

Il modello di dominio è stato sviluppato per comprendere il contesto in cui deve operare il sistema, rappresentando le entità chiave con le relative relazioni. Le entità sono state rappresentate come classi concettuali, infatti non sono state definite funzionalità, ma solo gli attributi che le caratterizzano.

Inoltre, questa modellazione ci ha permesso di avere una visione chiara e condivisa sul sistema, oltre che ad approfondire il contesto in cui deve operare l'applicativo.

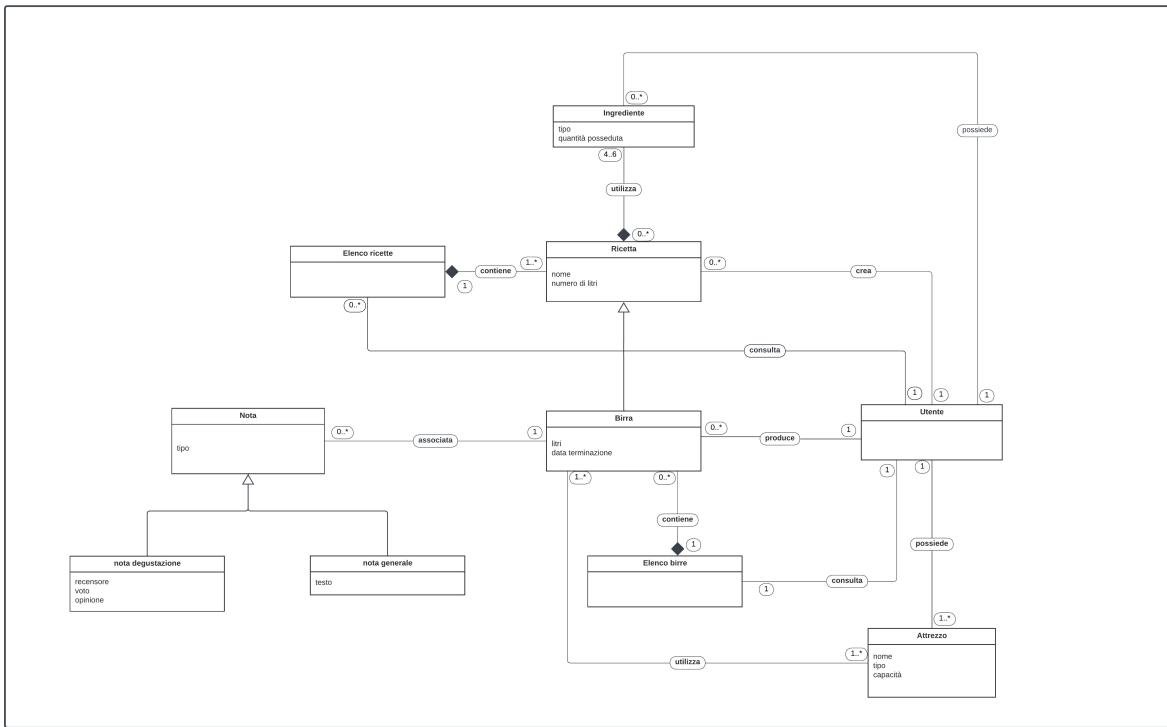


Figure 2: Diagramma del modello di dominio

## 2.3 Diagrammi di sequenza di sistema (SSD)

Il team ha redatto gli SSD al fine di rappresentare le interazioni tra l'attore utente ed il sistema "a scatola nera", ponendo l'attenzione sulle operazioni che possono essere innescate dall'utente e sulle risposte fornite dal sistema, senza analizzare il modo in cui quest'ultimo si occupa di gestirle. In particolare, sono stati rappresentati gli SSD relativi ai casi di utilizzo descritti in forma dettagliata. Questa operazione ha permesso di avere una visione d'insieme migliore sui casi di utilizzo più complessi ed ha inoltre permesso di comprendere in maniera più efficace quali sono le interazioni più rilevanti tra l'utente ed il sistema.

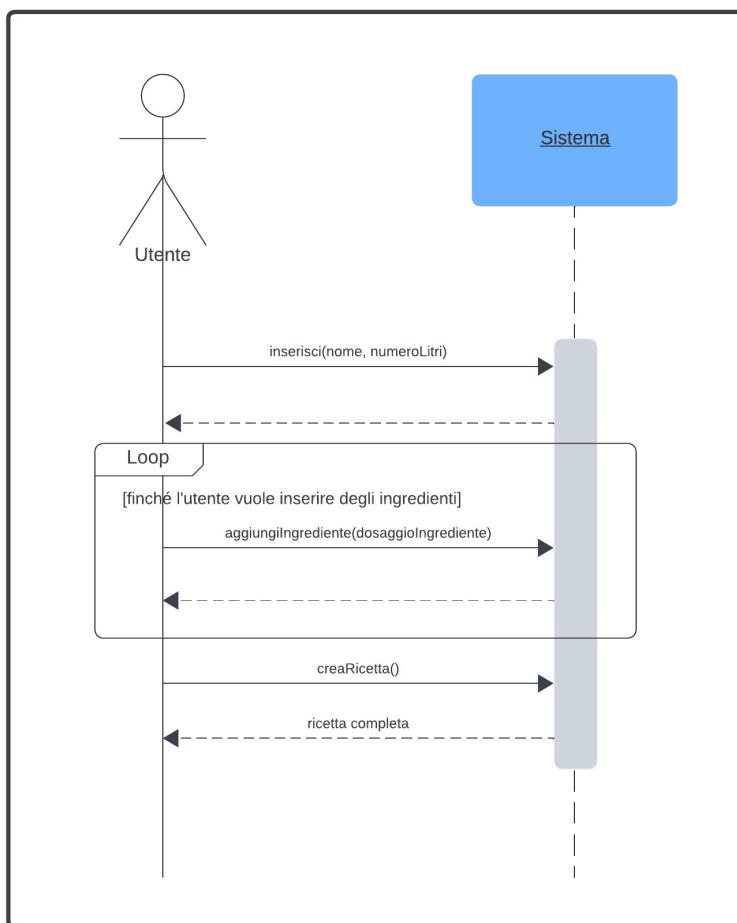


Figure 3: Diagramma SSD - crea ricetta

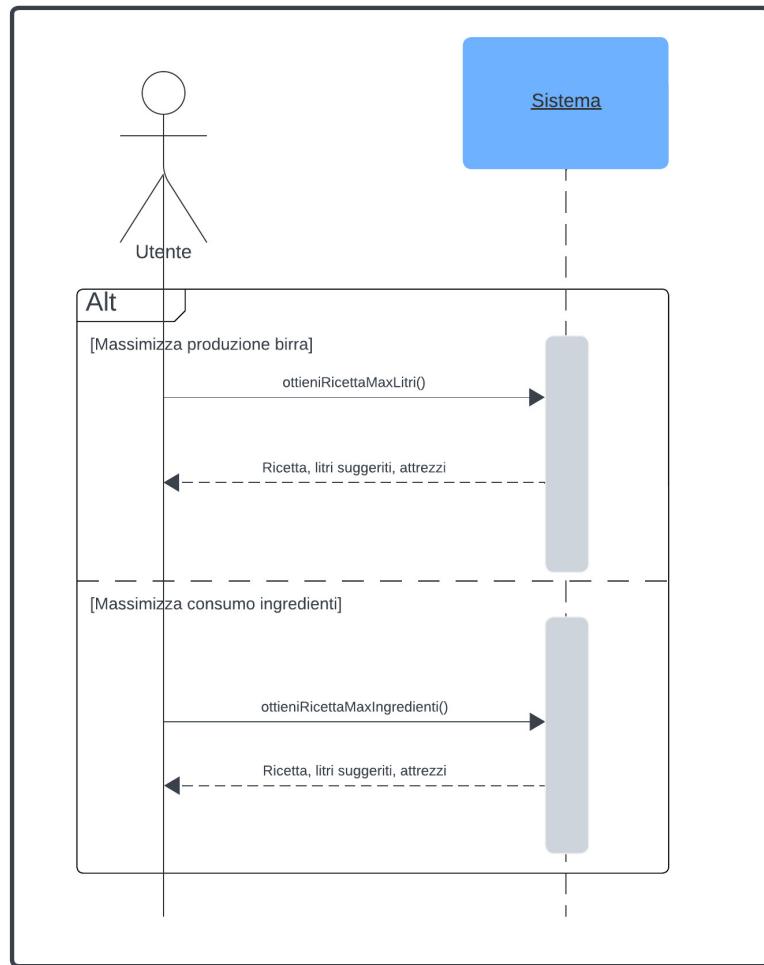


Figure 4: Diagramma SSD - cosa prepariamo oggi?

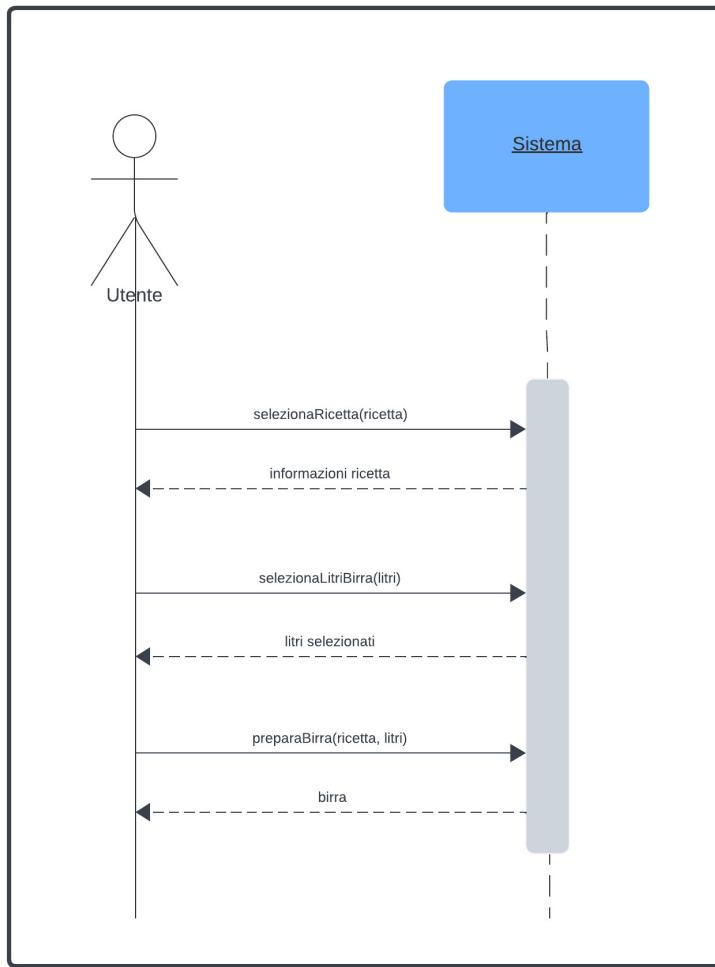


Figure 5: Diagramma SSD - prepara birra

## 3 Progettazione

In questa sezione vengono descritti i diagrammi relativi all'architettura implementata per realizzare l'applicativo, oltre che il tipo di architettura utilizzata in riferimento alla tipologia di applicazione che è stata prodotta.

### 3.1 Android Clean Architecture

Dopo aver effettuato un breve confronto tra i membri del team, è stato deciso di realizzare una mobile app nativa per il sistema operativo **Android** (compatibile da Android 7.0 ad Android 14.0), utilizzando il linguaggio di programmazione **Java**. L'architettura è stata realizzata in riferimento a quanto suggerito dalla documentazione ufficiale Android. Si tratta di un'architettura organizzata su più livelli, ciascuno dei quali detiene la propria responsabilità all'interno del sistema:

- **UI Layer.** Si tratta del livello che gestisce la logica UI, quindi la sua responsabilità è quella di mantere lo stato dell'interfaccia e presentare i dati all'utente.
- **Domain Layer.** È un livello intermedio opzionale, che è stato introdotto per gestire operazioni complesse sui dati e fornire alla UI un punto di accesso ai livelli sottostanti.
- **Data Layer.** Questo livello si occupa di gestire le interazioni con le fonti di dati.

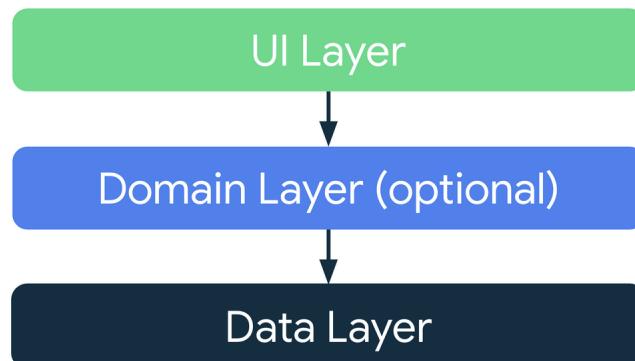


Figure 6: Clean Architecture

Durante la progettazione dell’architettura software dell’applicazione sono stati applicati i seguenti principi fondamentali (facenti parte delle best-practices della programmazione Android):

- **Separation of concerns.** Le responsabilità devono essere correttamente distribuite nei vari livelli dell’architettura, aumentando la coesione e riducendo l’accoppiamento tra le classi: in questo modo è possibile ottenere un’architettura robusta e scalabile.
- **Drive UI from data models.** Devono essere create opportune classi volte a rappresentare il modello dei dati all’interno dell’applicazione: tali classi vengono sfruttate dalla UI pur essendo indipendenti da essa.
- **Single source of truth.** Ad ogni tipologia di dato deve essere assegnata una “singola fonte di verità”, cioè un componente eletto a “proprietario” del dato. Esso deve essere l’unico a gestire letture, aggiunte, modifiche o cancellazioni relative al tipo di dato in questione.
- **Unidirectional data flow.** I dati devono confluire in una sola direzione e gli eventi che li modificano devono confluire in direzione opposta. In particolare, i dati devono confluire a partire dal Data Layer verso lo UI Layer, mentre gli eventi devono confluire a partire dallo UI Layer verso il Data Layer.

## 3.2 Model-View-ViewModel (MVVM) Pattern

L'architettura di un'app Android, che rispetta il concetto di Clean Architecture, può essere realizzata applicando il pattern architettonico **Model-View-ViewModel**: il punto centrale del pattern è la separazione della logica di presentazione dall'interfaccia utente, garantendo così estensibilità e manutenibilità del codice.

Tale pattern suggerisce di definire tre componenti fondamentali:

1. **Model**. È il componente che rappresenta i dati manipolati all'interno del sistema e la logica di business; può includere dati provenienti da qualsiasi fonte. Esso è totalmente indipendente dallo stato dell'interfaccia.
2. **View**. Si tratta del componente che rappresenta l'interfaccia utente e visualizza i dati forniti dai ViewModel; esso si occupa di indicare come i dati devono essere presentati all'utente. Non contiene logica di accesso diretto ai dati, ma si basa sul ViewModel per ottenere e presentare i dati.
3. **ViewModel**. È il componente che funge da intermediario tra la View ed il Model. Esso si occupa di mantenere lo stato di una View e fornisce la logica di interazione con il Model.

In riferimento alla Clean Architecture proposta da Android, il pattern MVVM è applicato come segue: la View è rappresentata dalle classi **Activity** e **Fragment**, le quali si presentano all'utente mediante il loro layout composto da elementi grafici. Le classi ViewModel si occupano invece di mantenere lo stato di Activity e Fragment, effettuando talvolta i dovuti aggiornamenti; forniscono inoltre l'accesso ai livelli sottostanti, che corrispondono al Model.

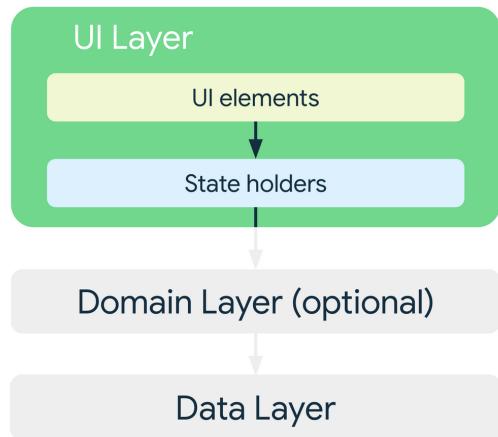


Figure 7: UI Layer

### 3.3 Repository Pattern

Il Data Layer rappresenta il livello di accesso ai dati gestiti dall'applicativo. Esso è stato modellato come suggerito dalla documentazione Android e consiste in un'applicazione del pattern **Repository**.

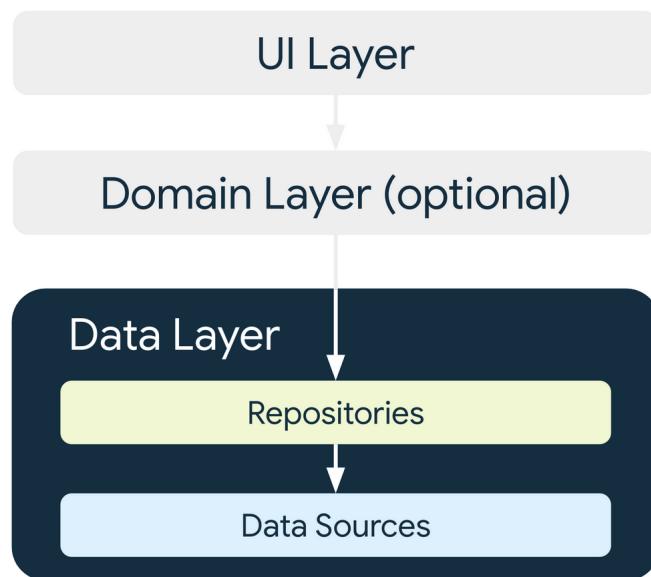


Figure 8: Data Layer

Quest'ultimo indica di creare una classe Repository per ciascun tipo di dato. La responsabilità di tale classe è quella di fornire un sistema di accesso ad una determinata tipologia di dato, permettendo che i livelli superiori possano accedervi come ad una collezione che si trova nella memoria. La classe repository può accedere alle fonti di dati tramite dei **Data Source**, ossia delle classi che rappresentano le varie Source of Truth per ciascuna tipologia di dato e mette a disposizione operazioni CRUD (Create - Read - Update - Delete).

### 3.4 Data Source, Room, DAO

Per soddisfare le richieste della traccia assegnata, si è rivelato necessario introdurre un sistema di persistenza dei dati rappresentato da un database locale, che definisce la Source of Truth per le varie tipologie di dato. Il database locale viene gestito tramite la libreria Room, consigliata dalla documentazione Android, che permette di creare ed organizzare un database SQLite sul dispositivo che eseguirà l'applicativo. Si tratta di una base di dati relazionale a cui è possibile accedere mediante i DAO (Data Access Objects), ossia delle interfacce che specificano le query per interagire con i dati. Per questa ragione è stato modellato un diagramma entità-relazione per definire la struttura del DB. Tale diagramma è stato successivamente convertito nel rispettivo modello relazionale ed infine implementato.

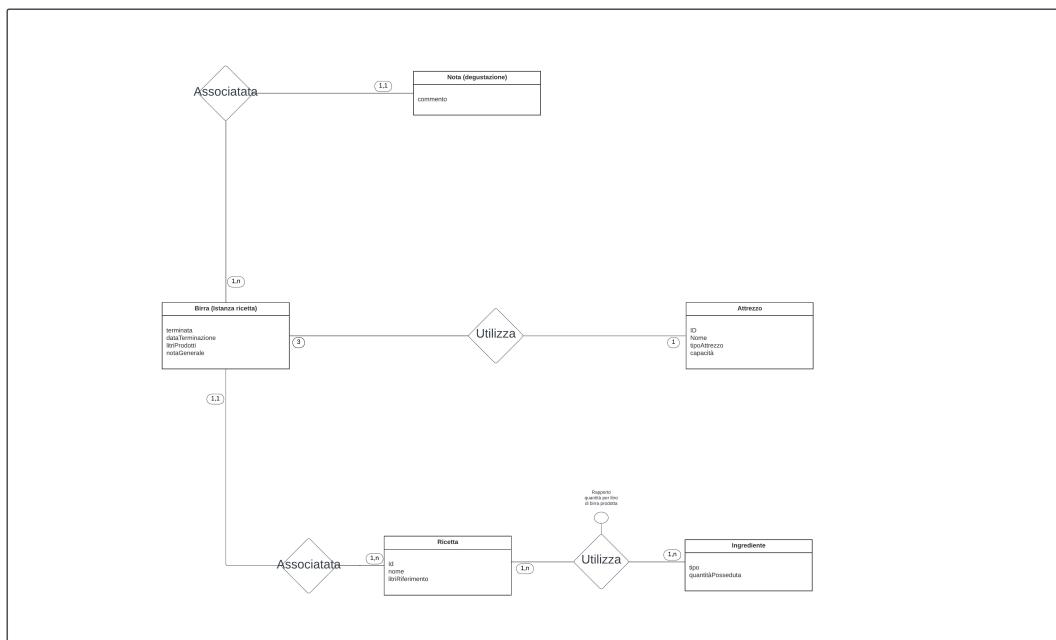


Figure 9: Diagramma ER

### 3.5 Diagramma dell'architettura

Il seguente diagramma rappresenta l'architettura dell'applicativo realizzato in relazione alla clean architecture di Android. A Livello UI, ciascun package rappresenta un caso di utilizzo e contiene le relative Activity, Fragment e ViewModel. Il package Util, invece contiene classi di "supporto" per la logica UI, in modo tale che non sia duplicata tra le varie classi.

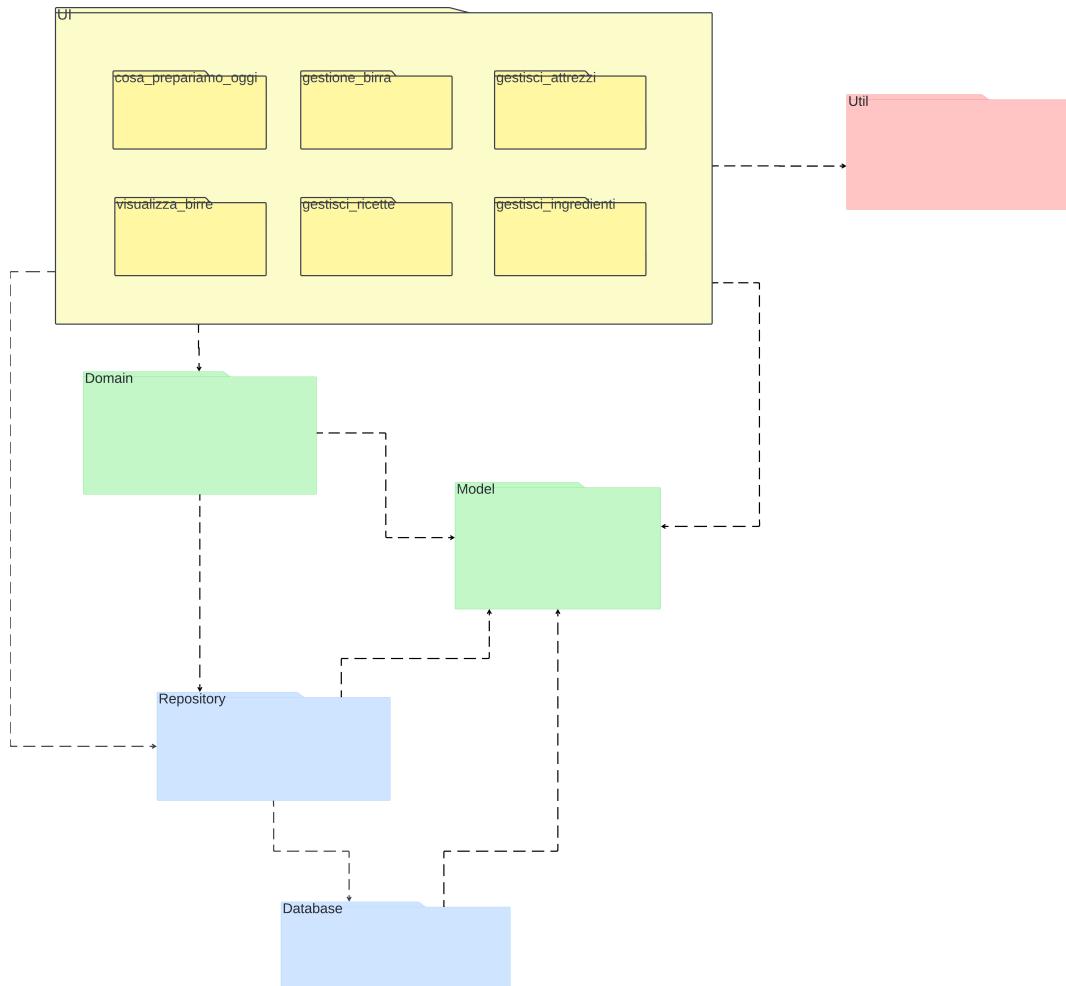


Figure 10: Diagramma dell'architettura

### 3.6 Diagrammi di sequenza (SD)

In questa sezione vengono rappresentati gli SD che sono stati realizzati. Anche in questo caso, sono stati prodotti i Sequence Diagram relativa alle sequenze di operazioni di sistema più complesse, in modo tale da analizzarle dettagliatamente per capire la sequenza di metodi e classi coinvolte.

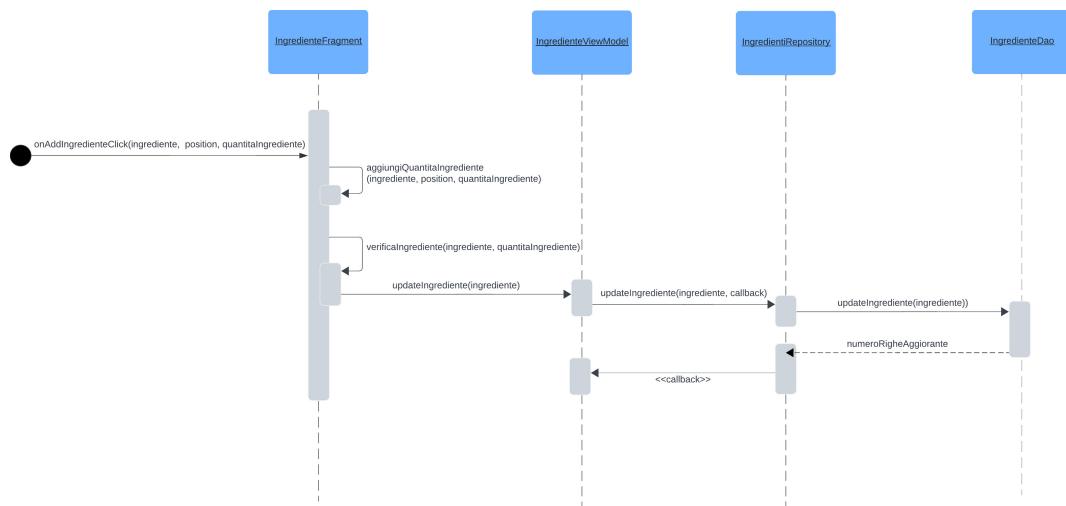


Figure 11: SD - aggiungi ingrediente

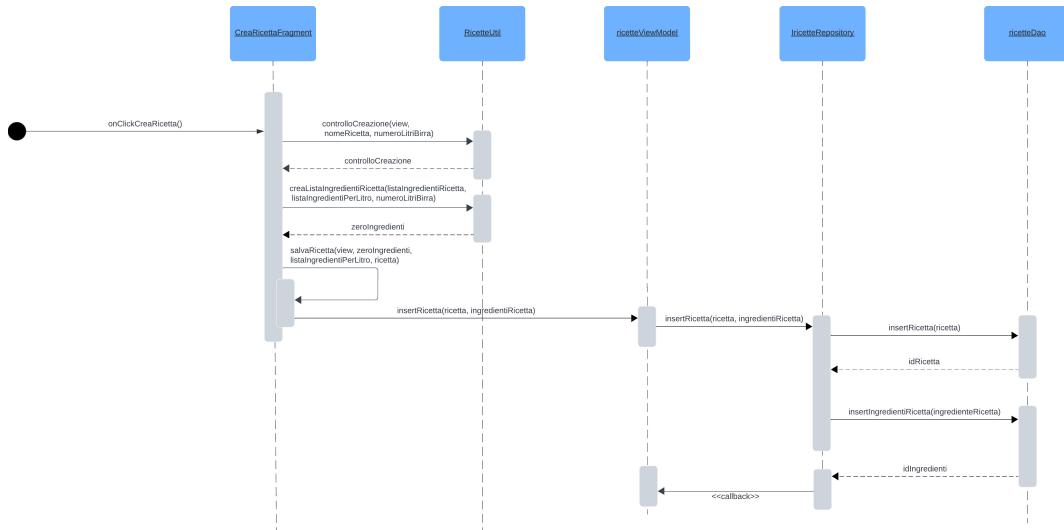


Figure 12: SD - crea ricetta

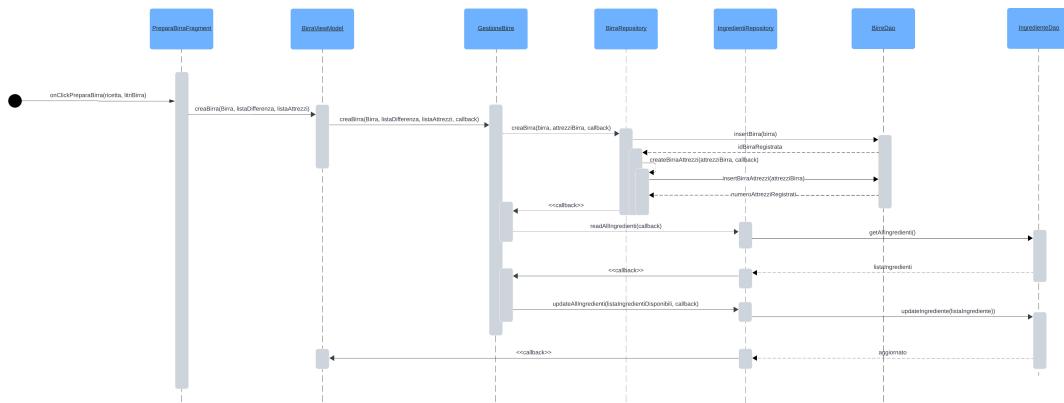


Figure 13: SD - prepara birra

### 3.7 Diagramma delle classi software

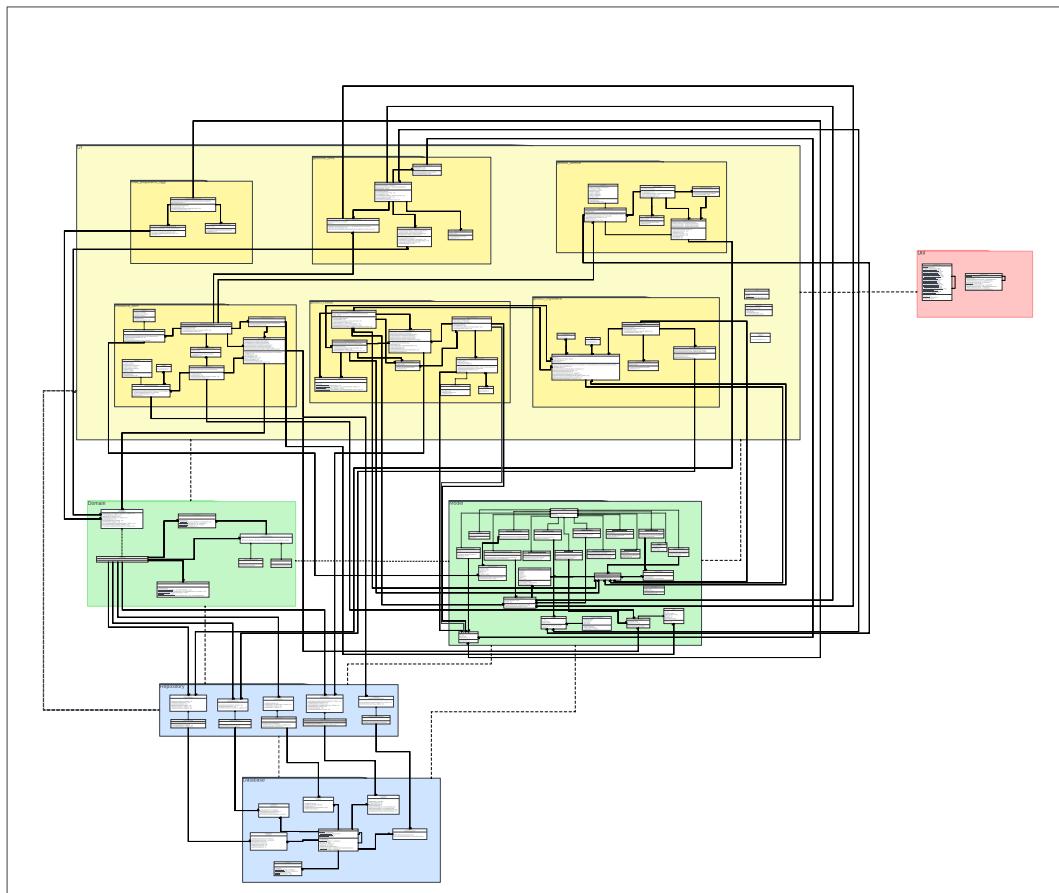


Figure 14: Diagramma delle classi

### 3.8 Diagrammi degli stati ed attività

Il diagramma degli stati è stato utilizzato per rappresentare gli stati in cui si può trovare un attrezzo, che rappresenta uno degli oggetti con più stati nel sistema.



Figure 15: Diagramma degli stati

Il risultato prodotto definisce quattro stati fondamentali:

1. **Libero.** L'attrezzo non viene utilizzato in nessuna produzione, di conseguenza può essere assegnato ad una birra, modificato e cancellato.
2. **Assegnato ad una birra.** L'attrezzo è utilizzato in una produzione, quindi non può essere modificato e cancellato.
3. **Aggiornamento.** L'attrezzo si trova nello stato di modifica, quindi non può essere utilizzato in nessuna produzione, ma può essere cancellato.
4. **Cancellato.** L'attrezzo è stato eliminato, quindi non può più essere utilizzato in nessuna produzione non può essere modificato. Lo stato è un pozzo, perché non è possibile ripristinare un attrezzo dopo la cancellazione.

Il diagramma di attività, invece, è stato utilizzato per descrivere il **flusso di attività** dell'operazione di creazione della ricetta, poichè è stata individuata come una tra le operazioni con i flussi più complessi.

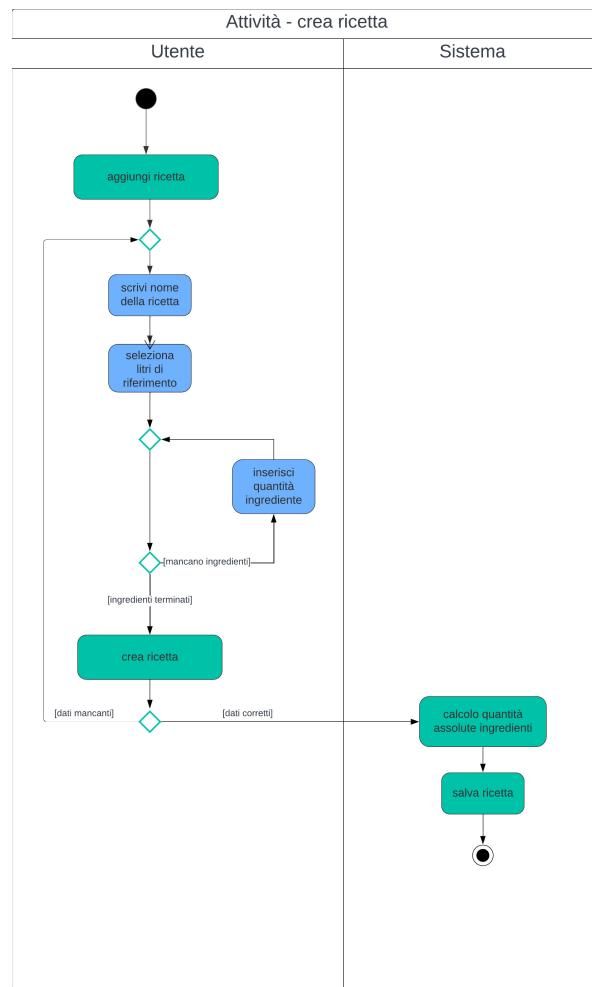


Figure 16: Diagramma di attività - crea ricetta

## 4 Pattern applicati e principi seguiti

Oltre ai già citati pattern MVVM e Repository sono stati applicati, in base al contesto, degli altri pattern che hanno permesso di risolvere problemi implementativi e produrre un architettura più scalabile ed estensibile. L'applicazione di alcuni di questi è richiesta dalle librerie Android: sono stati utilizzati e modificati per implementare correttamente il sistema.

### 4.1 Design pattern

#### 4.1.1 Service Locator

Si tratta di un pattern che permette di gestire in maniera centralizzata le implementazioni concrete di alcuni servizi, permettendo di eliminare la necessità di accoppiamento diretto con il client.

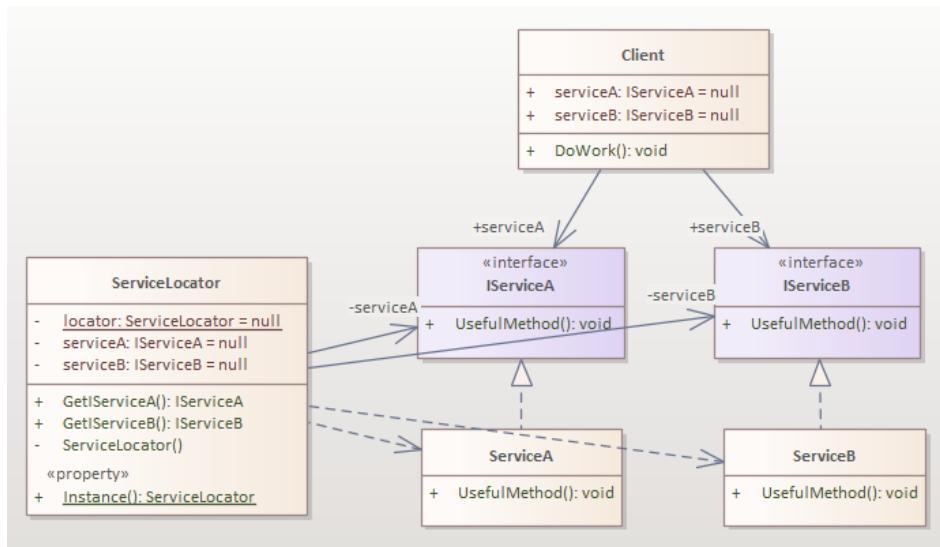


Figure 17: Pattern - Service Locator

Il **ServiceLocator** è una classe la cui responsabilità è quella di mantenere una lista di servizi univoci disponibili per l'applicativo. Le classi che necessitano di tali servizi non sono direttamente dipendenti dalle loro implementazioni concrete, ma definiscono delle interfacce e richiedono al ServiceLocator il servizio di cui necessitano. Nel nostro caso, tale pattern si è rivelato utile, ad esempio, per gestire le dipendenze tra il livello UI ed il Data Layer, in particolare tra ViewModel e Repository.

#### 4.1.2 Factory Method

Il pattern Factory Method suggerisce di sostituire la chiamata al costruttore di una classe con una verso un tipo speciale di metodo, detto "factory", che si occupa di istanziare oggetti di tale classe.

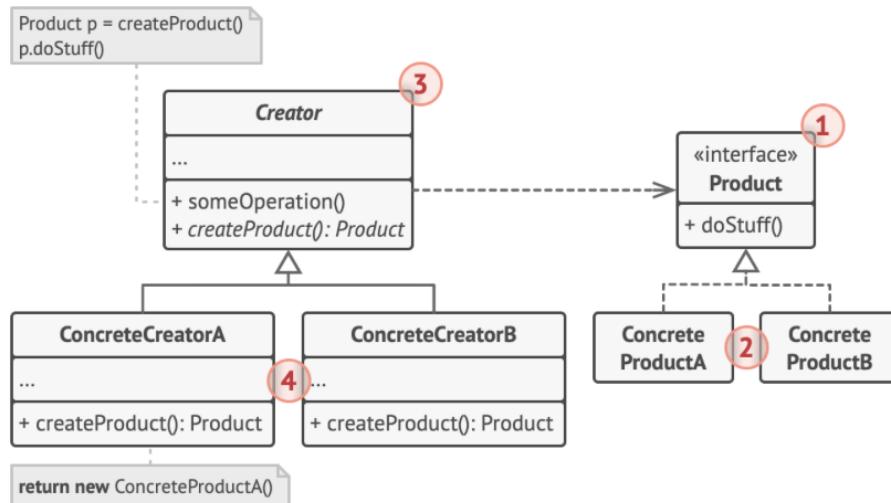


Figure 18: Pattern - Factory Method  
[Shv18]

1. **Product** è un'interfaccia o una classe astratta che definisce le caratteristiche comuni a tutti gli oggetti che la implementano.
2. I **ConcreteProduct** rappresentano le varie implementazioni del **Product**.
3. Il **Creator** è la classe che dichiara il Factory Method per restituire un nuovo oggetto di tipo **Product** (ossia l'interfaccia definita nel punto 1).
4. I **ConcreteCreator** estendono la classe astratta **Creator** e ridefiniscono il Factory Method per restituire le implementazioni dei **ConcreteProduct**.

Il Factory Method permette di ridurre l'accoppiamento tra una classe ed il client che la utilizza. Il client può definire un'interfaccia **Product** e successivamente inizializzarla utilizzando un **ConcreteCreator**. Dato che viene sfruttato il principio di ereditarietà, il client non è "interessato" al tipo di implementazione che gli viene fornito e quindi non ha una dipendenza diretta con il prodotto concreto. Nel nostro caso, questo pattern

si è rivelato utile per gestire correttamente i **ViewModel**. In particolare per istanziare un **ViewModel** in un'activity o fragment è necessario utilizzare il **ViewModelProvider**, ossia una classe messa a disposizione da Android, la cui responsabilità è quella di creare istanze relative alla classe **ViewModel** richiesta. Per poter istanziare un **ViewModel** tramite il suo costruttore è quindi necessario creare la rispettiva classe **Factory** (che estende **ViewModelProvider.Factory**) ed utilizzarla per ottenerne il riferimento.

#### 4.1.3 Observer

Il pattern **Observer** permette di gestire i casi in cui è necessario notificare un insieme di client di un determinato evento che si è verificato nel sistema.

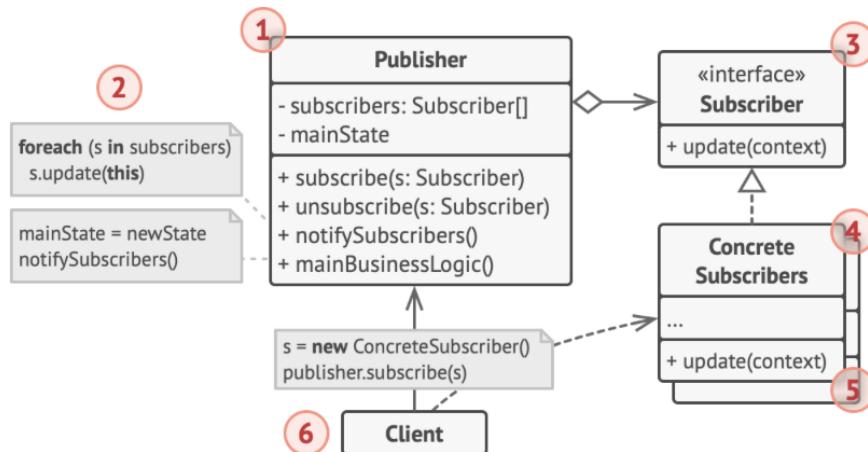


Figure 19: Pattern - Observer  
[Shv18]

1. Il **Publisher** è la classe che gestisce gli eventi a cui i client sono interessati. Mantiene una lista di **subscribers** e mette a disposizione ai client metodi per iscriversi e disiscriversi.
2. Quando si verifica un evento, il publisher **notifica tutti i subscribers**.
3. Un **Subscriber** è un'interfaccia che definisce un metodo **onUpdate()**.
4. Quest'ultima viene implementata dai **ConcreteSubscribers**, i quali ridefiniscono il metodo per eseguire delle azioni in risposta ad un evento.

5. I **ConcreteSubscribers** spesso necessitano di un contesto per rispondere all'evento correttamente.
6. Il **Client** utilizza il pattern definendo separatamente **Publisher** e **Subscribers**.

Nel nostro caso questo pattern è stato applicato per gestire la logica UI in risposta ai dati che vengono registrati nei **ViewModel**. Questi ultimi definiscono dei **MutableLiveData**, che sono dei contenitori di dati osservabili. Quando un **MutableLiveData** viene aggiornato, per esempio a seguito di una modifica al dato in esso contenuto, gli osservatori (nel nostro caso, i fragment) vengono notificati. Questo pattern viene quindi utilizzato per gestire la logica di presentazione dei dati.

#### 4.1.4 Singleton

Il pattern Singleton viene utilizzato per gestire i casi in cui deve esistere una sola istanza di una determinata classe nel sistema.

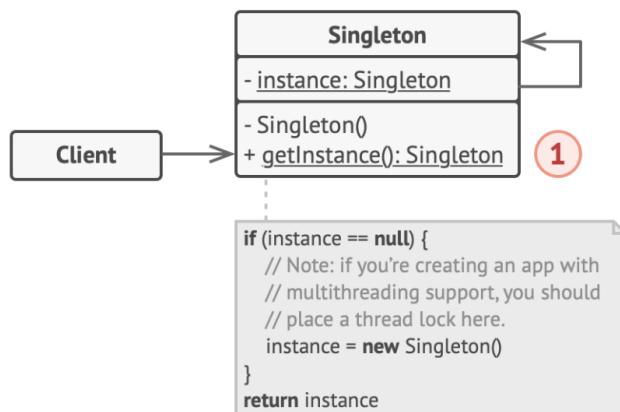


Figure 20: Pattern - Singleton  
[Shv18]

La classe **Singleton** dichiara un metodo **statico** `getInstance()` che restituisce un'istanza della sua stessa classe. Il costruttore di quest'ultima è privato e quindi inaccessibile al di fuori della classe. Nel nostro caso questo pattern si è rivelato utile per gestire classi come il **ServiceLocator** o il **LocalRoomDatabase**, di cui è necessario disporre di una sola istanza nell'intero sistema.

#### 4.1.5 Strategy

Il pattern Strategy viene utilizzato per definire una famiglia di algoritmi (ossia un insieme di algoritmi con lo stesso fine) in classi separate e di scegliere dinamicamente l'algoritmo da utilizzare.

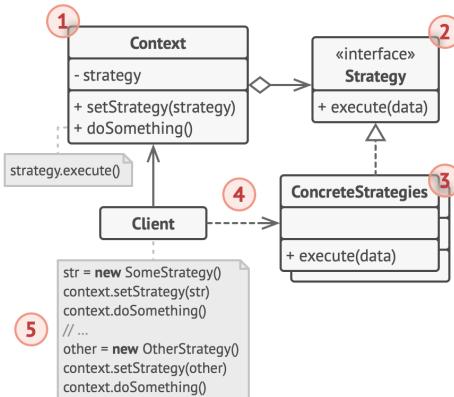


Figure 21: Pattern - Strategy  
[Shv18]

1. Il **Context** mantiene ed utilizza un riferimento ad una **Strategy**.
2. La **Strategy** è un'interfaccia comune a tutte le **ConcreteStrategy**. Essa definisce il metodo che viene utilizzato dal **Context**.
3. Le **Concrete Strategies** sono delle classi che implementano l'interfaccia **Strategy** e definiscono la vera implementazione del metodo utilizzato dal **Context**.
4. Il **Context** esegue i metodi definiti dall'interfaccia **Strategy**, senza sapere che tipo di **Strategy** concreta sia effettivamente istanziata.

Abbiamo utilizzato questo pattern per gestire il problema di ottimizzazione degli attrezzi e degli ingredienti nel caso di utilizzo "Cosa prepariamo oggi". Infatti entrambi gli algoritmi hanno come obiettivo l'identificazione di una ricetta e della quantità di litri che si possono produrre sulla base degli ingredienti a disposizione. L'applicazione di tale pattern ha permesso di ridurre la complessità e le responsabilità delle classe **GestioneBirra** nel Domain Layer.

## 4.2 Design Principles

Oltre ai principi di design definiti e suggeriti dalla documentazione Android, sono stati utilizzati altri principi come linee guida per progettare un applicativo corretto e scalabile. Abbiamo cercato di rispettare i principi che abbiamo analizzato, cercando di stabilire dei compromessi tra quelli conflittuali. In particolare alcuni hanno avuto maggiore impatto e sono:

- **Single Responsibility Principle (SRP).** Abbiamo cercato di definire dei componenti con una singola responsabilità. Questo ha garantito estendibilità e facilità di riutilizzo delle varie classi.
- **Open Closed Principle (OCP).** Le classi sono aperte all'estensione e chiuse alla modifica. L'aggiunta di nuove funzionalità all'applicativo è semplice e non comporta modifiche importanti del sistema già implementato.
- **Dependency Inversion Principle (DIP).** I livelli inferiori dell'architettura espongono funzionalità ai livelli superiori tramite interfacce.
- **Don't repeat yourself (DRY).** Il principio indica di evitare ripetizione del codice, la cui logica deve essere definita in un punto ben preciso dell'applicativo. Il principio è stato osservato nell'intero applicativo.

## 4.3 Analisi con Understand

È stata effettuata un'analisi del software prodotto utilizzando il tool **Understand**, al fine di riscontrare eventuali antipattern strutturali.

Dall'analisi è stato riscontrato che le classi con la maggior complessità ciclomatica sono quelle relative alla funzionalità "Cosa prepariamo oggi": in particolare, la classe **Ottimizzazione** (contenente la logica per la selezione degli attrezzi e per il calcolo dei litri massimi sulla base di attrezzi ed ingredienti) è la classe avente complessità ciclomatica massima più alta, con un valore pari a 7.

Inoltre, analizzando il grafo delle dipendenze di ciascuna classe, è stato individuato un antipattern legato alla classe **Risultato** del package **Model**: dal grafo si evince facilmente la presenza di un grande numero di dipendenze in entrata provenienti da classi esterne al package **Model**, risultando così in un antipattern di tipo **External Butterflies**; ciò accade perché la classe **Risultato** è utilizzata per astrarre il generico risultato di un'operazione, modellandone così il successo o il fallimento attraverso apposite classi interne.

Attualmente la presenza di questo antipattern non rappresenta un problema, tenendo però in considerazione che una modifica alla classe **Risultato** potrebbe causare un grande impatto sull'intera codebase.

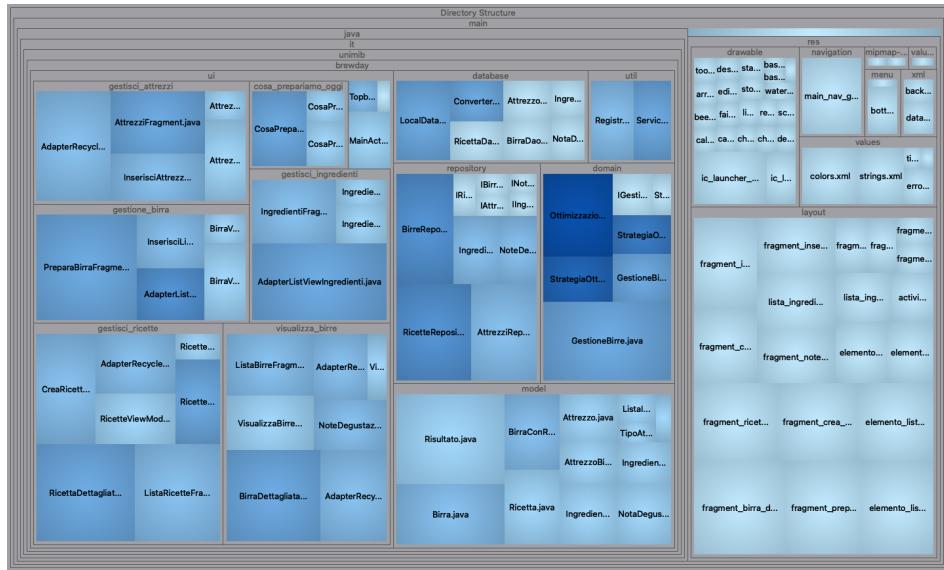


Figure 22: Recap di Understand sulla massima complessità ciclomatica

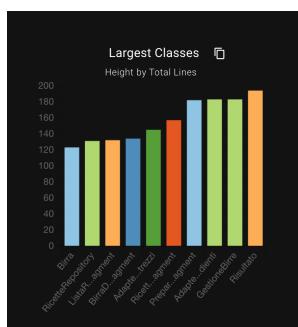


Figure 23: Classi più grandi del progetto

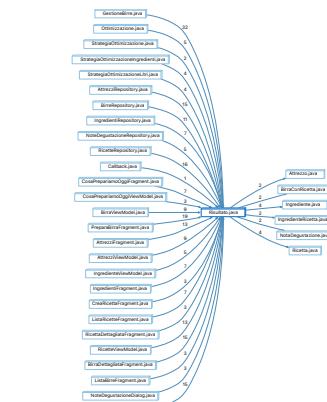


Figure 24: External Butterfly

## Riferimenti bibliografici

[Shv18] Alexander Shvets. Dive into design patterns. *Refactoring. Guru*, 2018.