Politecnico di Torino

POLITECNICO DI TORINO

DIPARTIMENTO DI INGEGNERIA GESTIONALE E DELLA PRODUZIONE

Corso di Laurea Triennale in Ingegneria Gestionale
Classe L8 - Ingegneria dell'Informazione

Tesi di Laurea

Gestione del guardaroba e generazione automatica di outfit personalizzati

Relatore:
Prof. Giuseppe Bruno Averta

Candidato:
Marica Zocco 345322

Anno Accademico 2024/2025

Sommario

1 - Proposta di progetto

1.1 Studente proponente

s345322 Zocco Marica

1.2 Titolo della proposta

Gestione del guardaroba e generazione automatica di outfit personalizzati

1.3 Descrizione del problema proposto

Quotidianamente molte persone si trovano a dover scegliere come vestirsi senza una guida chiara, finendo spesso per trascurare alcuni capi del proprio guardaroba o ricorrere sempre agli stessi abbinamenti. La mancanza di organizzazione porta anche a dimenticare cosa si possiede realmente.

Si vuole offrire all'utente un'applicazione che consenta di catalogare digitalmente i propri capi di abbigliamento e suggerire combinazioni di outfit sulla base di filtri e preferenze indicate dall'utente stesso.

1.4 Descrizione della rilevanza gestionale del problema

L'idea è rendere più semplice la gestione di ciò che indossiamo ogni giorno, valorizzando al meglio i vestiti che abbiamo già nell'armadio. Organizzando gli indumenti in gruppi e imparando a combinarli tra loro, diventa facile creare outfit adatti a qualsiasi occasione, senza mettere in disordine l'armadio e soprattutto risparmiando tempo. Inoltre, in questo modo si adotta un approccio più consapevole e sostenibile, infatti si avranno meno acquisti impulsivi e più scelte mirate, evitando di accumulare capi inutilizzati.

Questa applicazione può essere un valido alleato anche per i Personal Shopper digitali, aiutandoli a offrire un servizio più rapido ed efficace.

1.5 Descrizione dei data-set per la valutazione

Non avendo trovato online un dataset utile a questo caso, si prevede di costruirlo simulando un guardaroba reale composto da una serie di capi d'abbigliamento definiti con le seguenti caratteristiche: id, tipo, sottotipo, colore, stagione, occasione, marca.

L'applicazione potrà anche supportare l'inserimento e la rimozione manuale dei vestiti da parte dell'utente, quindi i dati non saranno statici.

Invece per quanto riguarda la combinazione dei colori, inizialmente avevo pensato di utilizzare un dataset già esistente chiamato "Color Clothes Matcher" dal sito "Kaggle" ma ho notato che non contiene tutti i colori ed inoltre sono in formato esadecimale. Dunque vorrei costruire una tabella con tutti i colori principali definiti con id e nome, mentre un'altra che si occuperà di collegare gli id dei colori abbinati. Ovviamente ogni colore sarà abbinato anche a sé stesso.

Per trovare gli abbinamenti giusti mi baserò sulle regole della teoria del colore e sulle combinazioni più comuni nell'abbigliamento e moda, prendendo spunto da questi siti:

- https://ivyandpearlboutique.com/blogs/fashion-howto/fashion-colors-matching-clothing-colors-using-color-wheel
- https://wonder-wardrobe.com/blog/5-color-outfit-matching-methods
- https://www.masterclass.com/articles/how-to-match-clothes-using-the-color-wheel
- https://www.canva.com/colors/color-wheel/

1.6 Descrizione preliminare degli algoritmi coinvolti

Un algoritmo si occuperà di memorizzare e rimuovere i capi d'abbigliamento scelti dall'utente all'interno del database. Il collegamento avverrà tramite pattern DAO.

In seguito verrà creato un grafo in cui ogni vertice rappresenta un capo, mentre gli archi tra i vertici rappresentano gli abbinamenti. Sarà un grafo non pesato in quanto tutti gli abbinamenti sono allo stesso livello e non orientato perché l'abbinamento tra un capo ed un altro è reciproco.

Successivamente, si applica un algoritmo ricorsivo per determinare tutti gli outfit possibili in base alle preferenze selezionate dall'utente e in base alle combinazioni dei colori memorizzate nel dataset.

L'outfit dovrà essere formato da un solo capo per la parte superiore, uno della parte inferiore e un paio di scarpe oppure un abito intero e un paio di scarpe. Nel caso in cui l'utente ha selezionato una stagione diversa dall'estate allora dovrà essere presente anche un capospalla.

1.7 Descrizione preliminare delle funzionalità previste per l'applicazione software

L'interfaccia sarà suddivisa in tre parti.

Nella prima parte l'utente potrà aggiungere un capo d'abbigliamento al proprio guardaroba inserendo manualmente tutte le caratteristiche. In basso ci sarà un Button che azionerà l'algoritmo e un Label per mostrare gli avvisi.

Nella seconda parte l'utente potrà selezionare il capo d'abbigliamento, tramite una Choicebox, che vuole eliminare dal guardaroba. In basso, anche in questo caso, un Button che aziona l'algoritmo e un Label per i messaggi.

La terza parte sarà dedicata alla creazione degli outfit. L'utente dovrà selezionare i filtri ovvero la stagione, l'occasione e se vuole anche il colore; successivamente tramite un Button l'algoritmo genererà tutti le combinazioni possibili e mostrerà i risultati in una TextArea.

2 - Descrizione dettagliata del problema affrontato

Nel contesto attuale, caratterizzato da una vita frenetica e da un'attenzione sempre maggiore all'immagine personale, organizzare in maniera efficiente il proprio guardaroba non è soltanto una questione di comodità, ma anche di valorizzazione del proprio stile e delle risorse già disponibili. L'utilità di un'applicazione dedicata alla gestione del guardaroba deriva dalla crescente esigenza di ottimizzare il tempo e di semplificare la scelta quotidiana degli abiti.

Organizzare manualmente il guardaroba e pianificare outfit per ogni occasione implica un notevole impegno cognitivo, infatti decidere cosa indossare richiede la considerazione di molteplici fattori: il clima e quindi scegliere abbinamenti adeguati alla stagione, il tipo di occasione se richiede un abbigliamento formale o sportivo, la combinazione armonica dei colori. Questo processo, seppur comune, può risultare dispendioso in termini di tempo e spesso porta gli utenti a ripetere gli stessi outfit o a trascurare parte del proprio guardaroba.

L'applicazione offre una soluzione a questo problema digitalizzando il guardaroba con un sistema semplice e centralizzato. Gli utenti possono aggiungere i nuovi acquisti oppure eliminare i capi non più utilizzati all'interno del database personale, mantenendo in questo modo un archivio digitale sempre coerente con il proprio guardaroba reale. Si tratta quindi di uno strumento affidabile e sempre aggiornato, capace di adattarsi ai cambiamenti dello stile personale o delle necessità pratiche.

Un ulteriore vantaggio è la capacità di suggerire abbinamenti che l'utente difficilmente immaginerebbe da solo. L'algoritmo utilizzato si basa sulle regole della teoria dei colori e su principi di armonia cromatica, in modo tale che sia in grado di generare combinazioni originali e inaspettate che risultano sempre esteticamente gradevoli. Questo permette non solo di evitare ripetitività, ma anche di valorizzare ogni singolo capo presente nel guardaroba.

Di conseguenza, esso diventa uno strumento che stimola la creatività personale e offre agli utenti la possibilità di vestirsi ogni giorno in modo diverso, senza dover possedere competenze da esperti di moda o di abbinamenti cromatici.

Questo servizio contribuisce anche a una gestione più sostenibile e consapevole del guardaroba. Infatti attraverso l'ottimizzazione degli abbinamenti, l'applicazione incoraggia l'utilizzo completo dei capi già posseduti, riducendo gli acquisti e promuovendo un approccio più equilibrato al consumo della moda.

Oltre a semplificare la vita quotidiana, permette di preparare in anticipo una selezione di abiti per un impegno importante, un evento o per una vacanza, riducendo lo stress organizzativo e garantendo coerenza estetica.

L'applicazione non è pensata solo per utenti privati ma anche per professionisti della moda, come Personal Shopper, stilisti o consulenti d'immagine, che necessitano di uno strumento organizzativo e creativo per proporre ai propri clienti abbinamenti mirati e personalizzati. A questo punto diventa anche una risorsa professionale, capace di integrare le competenze umane con il supporto tecnologico.

In conclusione, il software applicativo non punta a sostituire completamente il gusto personale, ma a fornire semplicemente un supporto pratico che riduce lo sforzo decisionale quotidiano, aumenta la varietà di scelte e valorizza al meglio i capi posseduti. Esso rappresenta una soluzione concreta e accessibile per chi cerca un modo più rapido, ordinato ed efficace di gestire il proprio guardaroba, oltre che uno strumento innovativo per chi opera professionalmente nel settore della moda e dello styling.

3 - Descrizione del data-set utilizzato per l'analisi

Come indicato nella proposta, il dataset utilizzato è stato realizzato manualmente. Lo scopo principale era simulare un guardaroba reale, perciò sono stati selezionati alcuni capi d'abbigliamento basandosi sulle tendenze di moda attuali. Attualmente il dataset comprende un totale di 170 capi, gli indumenti in maggioranza sono quelli della parte superiore e della parte inferiore, mentre in quantità minore ci sono abiti interi, capospalla e scarpe.

Invece i colori e gli abbinamenti sono stati selezionati in base alle regole della teoria del colore e alle combinazioni più comuni nel mondo della moda. In totale i colori scelti sono 18 mentre gli abbinamenti 238, considerando che ogni colore è stato abbinato anche a sé stesso e l'abbinamento di ogni coppia di colori è reciproca. Tutte le combinazioni selezionate sono:

- Nero: Bianco, grigio, beige, bordeaux, avorio, fucsia, rosso, giallo, arancione, viola.
- Bianco: Nero, grigio, blu, beige, azzurro, avorio, rosso, rosa, senape, fucsia, verde, viola, arancione, giallo, lilla.
- Grigio: Bianco, nero, blu, rosa, lilla, avorio, bordeaux, fucsia, senape, verde, arancione, azzurro, beige, marrone, rosso, giallo, viola.
- Blu: Bianco, grigio, beige, avorio, bordeaux, senape, rosa, arancione, fucsia, azzurro, giallo, lilla, rosso, viola.
- Azzurro: Bianco, grigio, beige, rosa, lilla, avorio, senape, rosso, fucsia, blu, verde, marrone, giallo, bordeaux.
- Beige: Bianco, nero, grigio, blu, rosa, bordeaux, azzurro, verde, senape, arancione, marrone, avorio, giallo.
- Marrone: Beige, avorio, grigio, verde, senape, bordeaux, arancione, azzurro, rosa, giallo.
- Avorio: Nero, bianco, beige, grigio, marrone, rosa, azzurro, bordeaux, blu, rosso, fucsia, verde, lilla, senape.
- Rosso: Nero, bianco, grigio, blu, avorio, lilla, arancione, azzurro, rosa, fucsia.
- Rosa: Grigio, bianco, beige, lilla, avorio, blu, rosso, fucsia, verde, azzurro, marrone, viola, senape, bordeaux.
- Fucsia: Nero, bianco, grigio, blu, avorio, arancione, viola, rosso, azzurro, rosa, giallo, verde, lilla.
- Giallo: Bianco, nero, grigio, blu, beige, azzurro, marrone, viola, fucsia, verde, lilla.
- Verde: Bianco, beige, avorio, lilla, azzurro, rosa, arancione, fucsia, giallo, grigio, bordeaux, senape, marrone.
- Arancione: Nero, beige, bianco, grigio, marrone, blu, fucsia, viola, verde, rosso.

- Viola: Grigio, nero, bianco, rosa, lilla, blu, arancione, giallo, fucsia.
- Lilla: Bianco, grigio, rosa, azzurro, blu, avorio, giallo, fucsia, verde, rosso, viola, senape.
- Senape: Bianco, beige, grigio, blu, marrone, bordeaux, avorio, lilla, rosa, verde, azzurro.
- Bordeaux: Nero, grigio, beige, blu, avorio, marrone, senape, rosa, azzurro, verde.

Il dataset è formato da tre tabelle. La tabella "capo" ha l'identificatore id del capo come chiave primaria, fornendo così un riferimento univoco per ogni vestito presente nel dataset. Alla stessa maniera, nella tabella “colore”, si ha un riferimento univoco per ogni colore. Mentre, la tabella "abbinamento" contiene tutti gli abbinamenti possibili tra i colori della tabella "colore". La chiave primaria è costituita dall'ID del colore e dall’ID dell’abbinato, in modo tale da non avere due volte la stessa coppia di colori.

Questa organizzazione dei dati fornisce una struttura chiara, consentendo un facile accesso alle informazioni necessarie per la creazione degli abbinamenti. Di seguito viene mostrata la struttura del dataset.

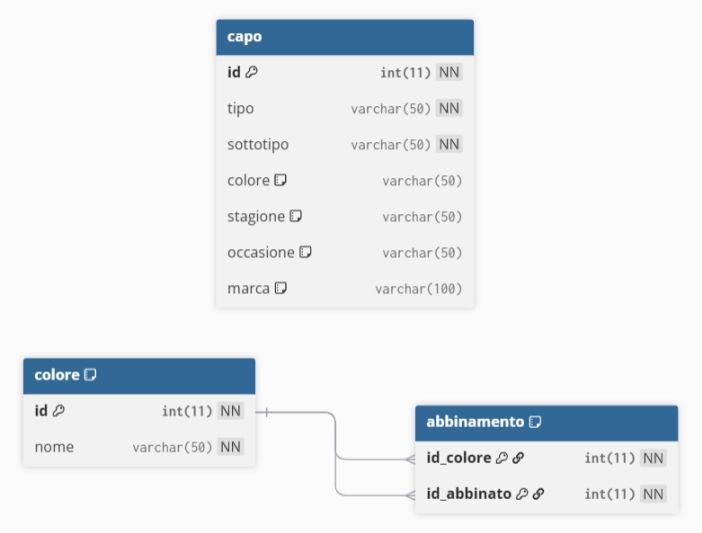


Figura 1 Diagramma ER della struttura del dataset “guardaroba”

3.1 Tabella sui capi d’abbigliamento

La tabella "capo" racchiude le informazioni sui capi d’abbigliamento inclusi nel dataset. La prima colonna è rappresentata dall’identificativo dell’abito (id), un codice univoco che si incrementa automaticamente con l'aggiunta di nuovi capi e che assume il ruolo di chiave primaria.

#	Nome	Tipo di dati	Lunghezza/set	Permetti NULL	Predefinito
1	**id**	**INT**	**11**	☐	**AUTO_INCREMENT**
2	tipo	VARCHAR	50	☐	Nessun valore predefinito
3	sottotipo	VARCHAR	50	☐	Nessun valore predefinito
4	colore	VARCHAR	50	☑	NULL
5	stagione	VARCHAR	50	☑	NULL
6	occasione	VARCHAR	50	☑	NULL
7	marca	VARCHAR	100	☑	NULL

Figura 2 Schema tabella “capo”

La struttura della tabella comprende:

- Il campo "tipo" indica la parte Superiore, Inferiore, Intero, Capospalla o Scarpe;
- Il campo “sottotipo” specifica il tipo d’abbigliamento come ad esempio Maglietta, Camicia, Pantaloni, Tuta, Giacca, Cappotto, Abito lungo, Stivali, Sandali, ecc.;
- Il campo “colore” contiene il nome del colore;
- Il campo “stagione” può essere Inverno, Estate oppure Autunno/Primavera;
- Il campo “occasione” può assumere i valori di Formale, Sportivo o Casual;
- Il campo “marca” contiene il nome della marca.

Di seguito vengono riportate le prime righe della tabella, mostrando così qualche esempio di dati.

1 id	2 tipo	3 sottotipo	4 colore	5 stagione	6 occasione	7 marca
1	Capospalla	Giacca	Nero	Autunno/Primavera	Formale	Zara
2	Capospalla	Piumino	Blu	Inverno	Casual	North Face
3	Capospalla	Trench	Beige	Autunno/Primavera	Casual	Burberry
4	Capospalla	Piumino	Azzurro	Inverno	Sportivo	H&M
5	Capospalla	Cappotto	Marrone	Inverno	Formale	Calliope
6	Capospalla	Blazer	Verde	Autunno/Primavera	Formale	Stradivarius

Figura 3 Esempio dati inseriti nella tabella “capo”

3.2 Tabella dei colori

La tabella "colore" comprende tutti i principali colori che possono avere i vari capi d’abbigliamento contenuti nella tabella “capo”. La prima colonna contiene l’identificativo id, chiave primaria che rende ogni colore univoco e si incrementa automaticamente nel caso in cui vengono aggiunti nuovi colori. La seconda colonna, ovvero il campo “nome”, contiene il nome del colore.

#	Nome	Tipo di dati	Lunghezza/set	Permetti NULL	Predefinito
1	**id**	**INT**	**11**	☐	**AUTO_INCREMENT**
2	nome	VARCHAR	50	☐	Nessun valore predefinito

Di seguito vengono riportate le prime righe della tabella, mostrando così qualche esempio di dati.

1 id	2 nome
14	Arancione
8	Avorio
5	Azzurro
6	Beige
2	Bianco
4	Blu

Figura 5 Esempio dati inseriti nella tabella “colore”

3.3 Tabella degli abbinamenti

La tabella "abbinamento" comprende tutti gli abbinamenti possibili tra i colori presenti nella tabella “colore”. Essa è stata costruita seguendo una relazione molti-a-molti, infatti ogni colore ha più abbinamenti. La chiave primaria è costituita da “id_colore” e “id_abbinato”, in questo modo non può esistere due volte la stessa coppia di colori.

#	Nome	Tipo di dati	Lunghezza/set	Permetti NULL	Predefinito
1	**id_colore**	**INT**	**11**	☐	Nessun valore predefinito
2	**id_abbinato**	**INT**	**11**	☐	Nessun valore predefinito

Figura 6 Schema tabella "abbinamento"

Di seguito vengono riportate le prime righe della tabella, mostrando così qualche esempio di dati.

1 id_colore	2 id_abbinato
1	1
2	1
3	1
6	1
8	1
9	1

Figura 7 Esempio dati inseriti nella tabella "abbinamento"

4 - Descrizione ad alto livello delle strutture dati e degli algoritmi utilizzati

4.1 Strutture dati utilizzate

Il progetto è stato implementato utilizzando il linguaggio di programmazione Java, seguendo due fondamentali pattern architetturali: il Model-View-Controller (MVC) e il Data-Access-Object (DAO).

Questa scelta ha permesso di realizzare un codice modulare, agevolando la gestione e la manutenzione del sistema. A tal fine, sono stati creati tre distinti packages, ognuno svolgente specifiche funzioni, per una chiara suddivisione delle responsabilità all'interno dell'applicazione. Questa struttura organizzativa contribuisce a rendere il codice più comprensibile.

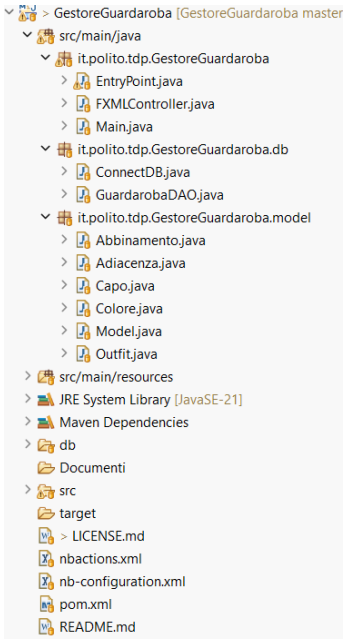


Figura 8 Struttura progetto

4.1.1 Package GestoreGuardaroba

Questo package serve a gestire l'interfaccia utente, presenta la scena all'utente e cattura le sue interazioni. La sua struttura è composta da tre classi distinte:

1. **EntryPoint**: questa classe è responsabile del caricamento iniziale della scena. Si occupa di inizializzare il controller che gestisce la scena, avviare il model e stabilire il collegamento tra il controller e il modello. La classe EntryPoint agisce come punto di ingresso, inizializzando gli elementi chiave necessari per avviare il funzionamento dell'applicazione.
2. **FXMLController**: questa classe ha il compito di raccogliere e gestire tutte le interazioni provenienti dall'utente. Le interazioni raccolte vengono successivamente inoltrate al modello per ottenere risposte specifiche relative al contesto corrente. Inoltre, la classe FXMLController è responsabile dei cambiamenti di stato della scena, garantendo una dinamica interattiva fluida tra l'utente e l'applicazione.
3. **Main**: la classe è fondamentale per l'avvio dell'applicazione. Fornisce il punto di partenza necessario per l'esecuzione del programma, inizializza e mettere in moto l'applicazione.

4.1.2 Package GestoreGuardaroba.db

Questo package serve a stabilire e gestire la connessione con il database e la raccolta dei dati, facilitando il trasferimento dei risultati ottenuti tramite le query al model. È stato appositamente sviluppato per aderire al pattern DAO, garantendo così un accesso limitato e controllato per la lettura e la modifica dei dati. La sua struttura comprende due classi:

1. **GuardarobaDAO**: questa classe svolge la funzione di estrarre dal database tutti i dati desiderati dall'utente mediante l'implementazione di query che estrapolano i dati dal database collegato. Una volta ottenuti i risultati, la classe GuardarobaDAO li passa al modello per ulteriori elaborazioni.

2. **DBConnect**: questa classe è responsabile di stabilire una connessione con il database. Fornisce quindi i mezzi necessari per la gestione delle interazioni tra l'applicazione e il database.

4.1.3 Package GestoreGuardaroba.model

Questo package costituisce il nucleo del progetto, poiché tutte le operazioni di elaborazione dei dati sono centralizzate in questo punto. È responsabile della logica applicativa, svolgendo compiti fondamentali per il corretto funzionamento dell'applicazione. La sua struttura è costituita da sei classi distinte:

1. **Capo**: classe implementata per rispettare il pattern **ORM** (Object-Relational Mapping), corrisponde perciò alla tabella dei capi d'abbigliamento. Definisce l'oggetto Capo ed ha il proprio costruttore*: Capo (id, tipo, sottotipo, colore, stagione, occasione, marca)*. È dotata dei relativi getters e setters per ogni attributo della classe, inoltre sono stati implementati i metodi: *hashCode ()*, *equals ()*, *toString ().*

```
public class Capo {
	private int id;
	private String tipo;
	private String sottotipo;
	private String colore;
	private String stagione;
	private String occasione;
	private String marca;

	public Capo(int id, String tipo, String sottotipo, String colore,
	String stagione, String occasione, String marca) {
		super();
		this.id = id;
		this.tipo = tipo;
		this.sottotipo = sottotipo;
		this.colore = colore;
		this.stagione = stagione;
		this.occasione = occasione;
		this.marca = marca;
	}…
```

2. **Colore**: classe implementata per rispettare il pattern **ORM** (Object-Relational Mapping), corrisponde alla tabella dei colori. Definisce l'oggetto Colore ed ha il proprio costruttore*: Colore (id, nome).* Ha i relativi getters e setters per ogni attributo della classe, inoltre sono stati implementati i metodi: *hashCode ()*, *equals ()* e *toString ().*

```
public class Colore {

	private int id;
	private String nome;

	public Colore(int id, String nome) {
		super();
		this.id = id;
		this.nome = nome;
	}…
```

3. **Abbinamento**: classe implementata per rispettare il pattern **ORM** (Object-Relational Mapping), corrisponde alla tabella degli abbinamenti. Definisce l'oggetto Abbinamento ed ha il proprio costruttore*: Abbinamento (id_colore, id_abbinato).* Ha i relativi getters e setters per ogni attributo della classe.

```
public class Abbinamento {
	private int id_colore;
	private int id_abbinato;

	public Abbinamento(int id_colore, int id_abbinato) {
		super();
		this.id_colore = id_colore;
		this.id_abbinato = id_abbinato;
	}…
```

4. **Adiacenza**: classe implementata per creare gli archi del grafo tra i capi d'abbigliamento. Definisce l'oggetto Adiacenza ed ha il proprio costruttore: *Adiacenza (c1, c2)*. Ha i relativi getters e setters per ogni attributo della classe.

```
public class Adiacenza {
	private Capo c1;
	private Capo c2;

	public Adiacenza(Capo c1, Capo c2) {
		super();
		this.c1 = c1;
		this.c2 = c2;
	}…
```

5. **Outfit**: classe implementata per creare i risultati finali. Definisce l'oggetto Outfit ed ha due costruttori in base se esso è formato da parte superiore e inferiore oppure da solo abito intero: *Outfit (superiore, inferiore, scarpe, capospalla)* e *Outfit (intero, scarpe, capospalla).* Ha i relativi getters per ogni attributo della classe ed è stato implementato il metodo *toString ().*

```
public class Outfit {
	private Capo superiore;
	private Capo inferiore;
	private Capo intero;
	private Capo scarpe;
	private Capo capospalla;

	//Outfit superiore + inferiore
	public Outfit(Capo superiore, Capo inferiore, Capo scarpe, Capo capospalla) {
		super();
		this.superiore = superiore;
		this.inferiore = inferiore;
		this.scarpe = scarpe;
		this.capospalla = capospalla;
	}

	//Outfit intero
	public Outfit(Capo intero, Capo scarpe, Capo capospalla) {
		super();
		this.intero = intero;
		this.scarpe = scarpe;
		this.capospalla = capospalla;
	}…
```

6. **Model**: è il cuore del software, si occupa della gestione di tutti i processi applicativi. Dispone delle strutture di archiviazione dei dati e di tutti i metodi essenziali per elaborare le richieste dell'utente. La sua funzione principale consiste nell'agire come intermediario tra l'applicazione e il database. Contiene tutti codici per la creazione del grafo e l'algoritmo ricorsivo per la creazione di tutte le combinazioni possibili.

4.2 Algoritmi utilizzati

4.2.1 GuardarobaDAO

Nella classe GuardarobaDAO sono contenuti tutti i codici per interrogare il database in modo da

1. Ottenere i dati per popolare le ChoiceBox dell'interfaccia:

```
public List<Capo> getCapi() {
String sql = "SELECT * FROM capo ORDER BY sottotipo"; …
```

per ottenere tutti i capi presenti nel database in ordine alfabetico secondo il sottotipo.

```
public List<String> getSottotipiByTipo(String tipo) {
String sql = "SELECT DISTINCT sottotipo FROM capo WHERE tipo = ? ORDER BY sottotipo"; …
```

per ottenere tutti i sottotipi di un determinato tipo passato come parametro.

```
public List<Colore> getColori() {
String sql = "SELECT id, nome FROM colore"; …
```

per ottenere tutti i colori presenti nel database.

Usati nel FXMController per popolare le seguenti ChoiceBox

```
private ChoiceBox<Colore> CbColore;
```

per selezionare il colore del capo da aggiungere.

```
private ChoiceBox<Colore> CbColoreO;
```

per selezionare il colore degli outfit da creare.

```
private ChoiceBox<String> CbSottotipo;
```

per selezionare il sottotipo del capo da aggiungere.

```
private ChoiceBox<Capo> cbCapo;
```

per selezionare il capo da eliminare.

2. Modificare i dati del database:

```
public boolean esisteCapo(Capo c) {
String sql = "SELECT COUNT(*) AS cnt FROM capo "
             + "WHERE tipo = ? AND sottotipo = ? AND colore = ? "
             + "AND stagione = ? AND occasione = ? AND marca = ?"; …
```

per controllare se il capo passato come parametro è già presente nel database.

```
public boolean aggiungiCapo(Capo c) {
String sql = "INSERT INTO capo (tipo, sottotipo, colore, stagione, occasione, marca) "
              + "VALUES (?, ?, ?, ?, ?, ?)"; …
```

per aggiungere il capo passato come parametro nel database.

```
public boolean eliminaCapo(Capo c) {
String sql = "DELETE FROM capo "
              + "WHERE tipo = ? AND sottotipo = ? "
              + "AND colore = ? AND stagione = ? "
              + "AND occasione = ? AND marca = ?"; …
```

per eliminare il capo passato come parametro dal database.

Usati nel FXMController per i seguenti Button

```
private Button btnAggiungi;

void aggiungiCapo(ActionEvent event) {
String tipo = this.CbTipo.getValue();
String sottotipo = this.CbSottotipo.getValue();
String colore = this.CbColore.getValue().getNome();
```

```
String stagione = this.CbStagione.getValue();
String occasione = this.CbOccasione.getValue();
String marca = this.model.capitalizeMarca(this.TfMarca.getText());

Capo c = new Capo (0, tipo, sottotipo, colore, stagione, occasione, marca);

if(this.model.esisteCapo(c))
	this.txtMessaggioAggiungi.setText(c + "\n è già presente!");
else {
	this.model.aggiungiCapo(c);
	this.txtMessaggioAggiungi.setText(c + "\n è stato aggiunto correttamente!");
	this.aggiornaCapi();
	}
}
```
per aggiungere un capo al database.

```
Private Button btnElimina;

void eliminaCapo(ActionEvent event) {
Capo c = this.cbCapo.getValue();

if(this.model.eliminaCapo(c)) {
	this.txtMessaggioElimina.setText(c +"\n è stato eliminato correttamente!");
	this.aggiornaCapi();
	}
}
```
per rimuovere un capo dal database.

3. Creare il grafo:

```
public void getVertici(String stagione, String occasione, Map<Integer, Capo> idMap) {
String sql = "SELECT * "
		+ "FROM capo "
		+ "WHERE stagione = ? AND occasione = ? "; …
```
per ottenere tutti i vertici del grafo.

Utilizzato nel model per creare il grafo in base ai parametri impostati dall'utente.

4.2.2 Creazione grafo

```
private void creaGrafo(String stagione, String occasione) {
	this.grafo = new SimpleGraph<>(DefaultEdge.class);
	this.idMap = new HashMap<>();

	this.guardarobaDAO.getVertici(stagione, occasione, idMap);
	Graphs.addAllVertices(this.grafo, idMap.values());

	List <Adiacenza> archi = new ArrayList<>();
	this.idColoriMap = new HashMap<>();

	for(Colore c : this.getColori())
		idColoriMap.put(c.getId(), c);

	for (Abbinamento abb : this.getAbbinamenti()) {
	 Colore col1 = idColoriMap.get(abb.getId_colore());
	 Colore col2 = idColoriMap.get(abb.getId_abbinato());

	 for (Capo c1 : idMap.values()) {
	     for (Capo c2 : idMap.values()) {
	         if (c1.getId() < c2.getId()) {

	             //controllo se i colori dei due capi rispettano l'abbinamento
	                 if ((c1.getColore().equalsIgnoreCase(col1.getNome()) &&
	                      c2.getColore().equalsIgnoreCase(col2.getNome())) ||
	                     (c1.getColore().equalsIgnoreCase(col2.getNome()) &&
	                      c2.getColore().equalsIgnoreCase(col1.getNome()))) {
```

```
                        archi.add(new Adiacenza(c1, c2));
                    }
                }
            }
        }
    }

       for(Adiacenza c: archi)
              grafo.addEdge(c.getC1(), c.getC2());
}
```

Il metodo *creaGrafo()* si occupa di aggiungere i vertici e di effettuare un controllo preliminare sugli archi prima di inserirli nel grafo.

I vertici vengono ottenuti dal metodo *getVertici()*, dove vengono prelevati tutti i capi dal database basandosi sulle scelte obbligatorie dell'utente (stagione e occasione).

Successivamente, viene verificato se ogni coppia di vertici è compatibile cromaticamente facendo il paragone con gli abbinamenti dati dal metodo *getAbbinamenti()*. A questo punto vengono aggiunti tutti gli archi al grafo.

4.2.3 Creazione outfit

```
public List<Outfit> creaOutfit(String stagione, String occasione, Colore colore) {
    this.creaGrafo(stagione, occasione);

    List<Outfit> result = new ArrayList<>();
    List<Outfit> result_colore = new ArrayList<>();

    List<Capo> interi = new ArrayList<>();
    List<Capo> superiori = new ArrayList<>();
    List<Capo> inferiori = new ArrayList<>();
    List<Capo> scarpe = new ArrayList<>();
    List<Capo> capispalla = new ArrayList<>();

    for (Capo c : idMap.values()) {
        switch (c.getTipo().toLowerCase()) {
            case "intero": interi.add(c); break;
            case "superiore": superiori.add(c); break;
            case "inferiore": inferiori.add(c); break;
            case "scarpe": scarpe.add(c); break;
            case "capospalla": capispalla.add(c); break;
        }
    }

    boolean estate = stagione.equalsIgnoreCase("estate");

    // Outfit con intero
    for (Capo intero : interi) {
        for (Capo shoe : scarpe) {
            if (grafo.containsEdge(intero, shoe)) {
                if (!estate) {
                    for (Capo spalla : capispalla) {
                        if (grafo.containsEdge(intero, spalla) &&
                            grafo.containsEdge(shoe, spalla)) {
                            result.add(new Outfit(intero, shoe, spalla));
                        }
                    }
                } else {
                    result.add(new Outfit(intero, shoe, null));
                }
            }
        }
    }
```

```
    // Outfit con superiore + inferiore
    for (Capo sup : superiori) {
        for (Capo inf : inferiori) {
            if (grafo.containsEdge(sup, inf)) {
                for (Capo shoe : scarpe) {
                    if (grafo.containsEdge(sup, shoe) && grafo.containsEdge(inf, shoe)) {
                        if (!estate) {
                            for (Capo spalla : capispalla) {
                                if (grafo.containsEdge(sup, spalla) &&
                                    grafo.containsEdge(inf, spalla) &&
                                    grafo.containsEdge(shoe, spalla)) {
                                    result.add(new Outfit(sup, inf, shoe, spalla));
                                }
                            }
                        } else {
                            result.add(new Outfit(sup, inf, shoe, null));
                        }
                    }
                }
            }
        }
    }

    if(colore!= null) { //almeno un capo deve avere il colore selezionato
       for(Outfit o : result) {
              if(this.capoHaColore(o.getCapospalla(), colore) ||
              this.capoHaColore(o.getInferiore(), colore) ||
              this.capoHaColore(o.getIntero(), colore) ||
              this.capoHaColore(o.getScarpe(), colore) ||
              this.capoHaColore(o.getSuperiore(), colore))

              result_colore.add(o);
       }

    return result_colore;
    }

    return result;
}

private boolean capoHaColore(Capo capo, Colore colore) {
       if(capo == null)
              return false;
       else
              return capo.getColore().equalsIgnoreCase(colore.getNome());
}
```

Il metodo *creaOutfit(),* presente nella classe Model, ha lo scopo di creare la lista di tutte le combinazioni possibili da mostrare successivamente nell'interfaccia. Una volta creato il grafo secondo i parametri stagione e occasione selezionati dall'utente; i vertici vengono suddivisi per tipo ovvero superiore, inferiore, intero, scarpe e capospalla.

Inizialmente vengono creati gli outfit formati da abito intero e scarpe e successivamente quelli formati da parte superiore, parte inferiore e scarpe. In entrambi i casi, se la stagione selezionata dall'utente non è "estate", viene aggiunto anche un capospalla.

Ogni combinazione trovata viene salvata come oggetto Outfit nella lista. Se l'utente ha selezionato anche il colore, allora bisogna controllare se per ogni outfit almeno un capo ha il colore scelto. Questa verifica è effettuata tramite il metodo *capoHaColore().*

```
void creaOutfit(ActionEvent event) {
this.txtMessaggioOutfit.clear();

String stagione = this.CbStagioneO.getValue();
String occasione = this.CbOccasioneO.getValue();
Colore colore = this.CbColoreO.getValue();

List<Outfit> outfit=this.model.creaOutfit(stagione, occasione, colore);

if(outfit == null || outfit.isEmpty())
	this.txtMessaggioOutfit.appendText("Non ci sono outfit possibili!");

else {

	this.txtMessaggioOutfit.appendText("Tutti gli outfit con stagione " + stagione + ",
occasione " + occasione);

	if(colore != null)
		this.txtMessaggioOutfit.appendText(", colore " + colore);

	this.txtMessaggioOutfit.appendText(" sono: \n\n");

	for (Outfit o: outfit)
		this.txtMessaggioOutfit.appendText(o.toString() + "\n\n");
	}

//pulisco tutto per una nuova ricerca
this.CbStagioneO.getSelectionModel().clearSelection();
this.CbOccasioneO.getSelectionModel().clearSelection();
this.CbColoreO.getSelectionModel().clearSelection();
this.btnCrea.setDisable(true);
}
```

Il metodo *creaOutfit(),* presente nella classe FXMLController, si occupa di passare i parametri selezionati dall'utente nell'interfaccia al Model e stampare la lista di tutti gli outfit creati dall'algoritmo. Nel caso in cui non è disponibile nessuna combinazione che rispetti le preferenze selezionate dall'utente, allora nella TextArea verrà visualizzato un avviso.

4.2.4 Controlli campi interfaccia

```
private void checkCampiAggiungi() {
this.txtMessaggioAggiungi.setText("");
this.btnAggiungi.setDisable(true);

if(this.CbColore.getValue() != null && this.CbOccasione.getValue() != null
&& this.CbSottotipo.getValue() != null && this.CbStagione.getValue() != null
&& this.CbTipo.getValue() != null && !this.TfMarca.getText().trim().isEmpty())

this.btnAggiungi.setDisable(false);
}
```

per abilitare il pulsante "Aggiungi" solo quando l'utente ha compilato tutti i campi.

```
private void aggiornaSottotipi() {
	if(this.CbTipo.getValue() != null) {
		this.CbSottotipo.setDisable(false);
		String tipo = this.CbTipo.getValue();
		List<String> sottotipi = this.model.getSottotipiByTipo(tipo);
	this.CbSottotipo.getItems().clear();
	this.CbSottotipo.getItems().addAll(sottotipi);
	}
 }
```

per aggiornare la lista dei sottotipi nel ChoiceBox quando l'utente seleziona un altro tipo.

```
private void aggiornaCapi() {
//aggiorno la lista nel ChoiceBox quando aggiungo o elimino un capo
this.cbCapo.getItems().clear();
       this.cbCapo.getItems().addAll(this.model.getCapi());
    }
```
per aggiornare la lista dei capi nel ChoiceBox quando l'utente aggiunge o rimuove un capo dal database.

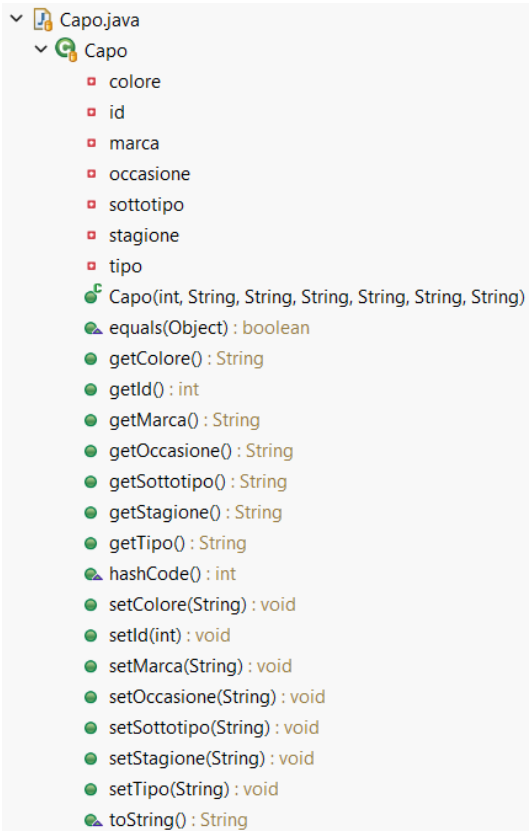
```
private void checkCampoElimina() {
       this.btnElimina.setDisable(true);

if(this.cbCapo.getValue() != null) {
       this.btnElimina.setDisable(false);
       this.txtMessaggioElimina.setText("");
       }
}
```
per abilitare il pulsante "Elimina" solo quando l'utente ha selezionato il capo nel ChoiceBox.

```
private void checkCampiOutfit() {
if(this.CbOccasioneO.getValue() != null && this.CbStagioneO.getValue() != null)
       this.btnCrea.setDisable(false);
}
```
per abilitare il pulsante "Crea" solo quando l'utente ha selezionato i filtri obbligatori.

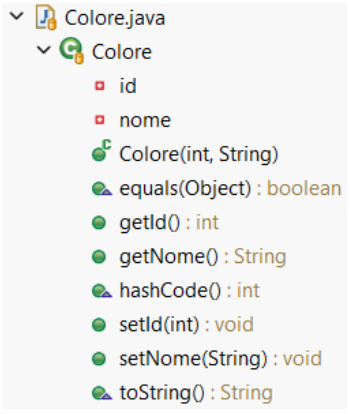
```
public String capitalizeMarca(String m) {
       String[] parole = m.trim().toLowerCase().split("\\s+");

       /*
          trim() --> rimuovo eventuali spazi all'inizio e alla fine
          toLowerCase() --> converto tutto in minuscolo
          split("\\s+") --> divido la stringa in parole usando lo spazio come separatore
       */

       String result = "";
//costruisco la nuova stringa concatenando le parole con un ciclo

       for(String parola : parole) {
              if(!parola.isEmpty())
              result += parola.substring(0,1).toUpperCase() + parola.substring(1) + " ";
       }

       /*
          !parola.isEmpty() --> controllo che non sia vuota nel caso in cui ci sono spazi
          multipli tra le parole
          parola.substring(0,1).toUpperCase() --> trasformo la prima lettera in maiuscolo
          parola.substring(1) --> prendo il resto della parola così com'è in minuscolo
          + " " --> aggiungo uno spazio dopo la parola
       */

       return result.trim(); //trim() --> rimuovo lo spazio dato dall'ultimo + " "
}
```
per convertire il nome della marca con lettera iniziale maiuscola e il resto in minuscolo prima di memorizzarlo nel database.

5 - Diagramma delle classi delle parti principali dell'applicazione

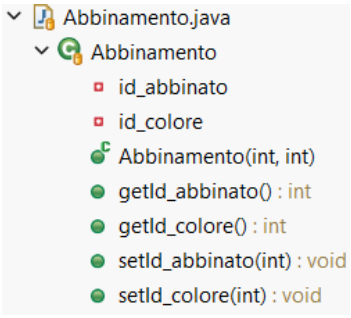
Classe Capo:



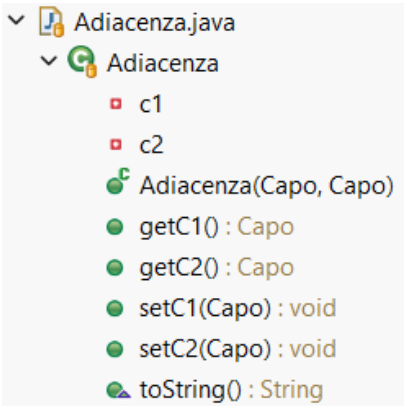
Classe Colore:



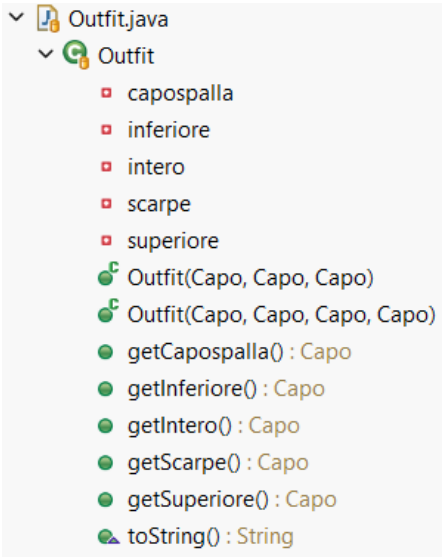
Classe Abbinamento:



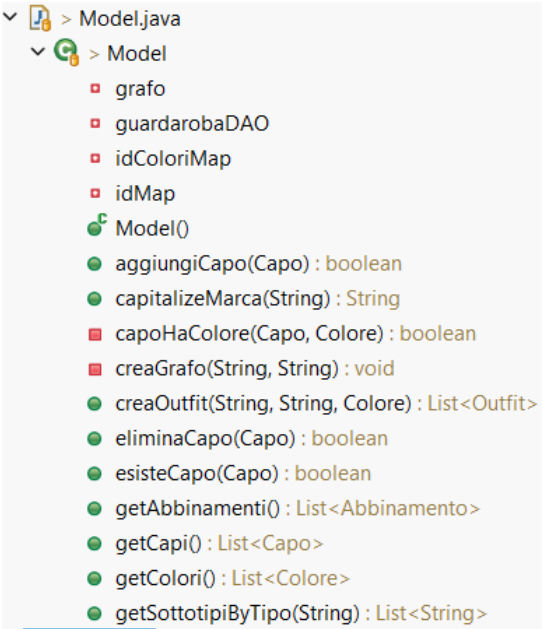
Classe Adiacenza:



Classe Outfit:

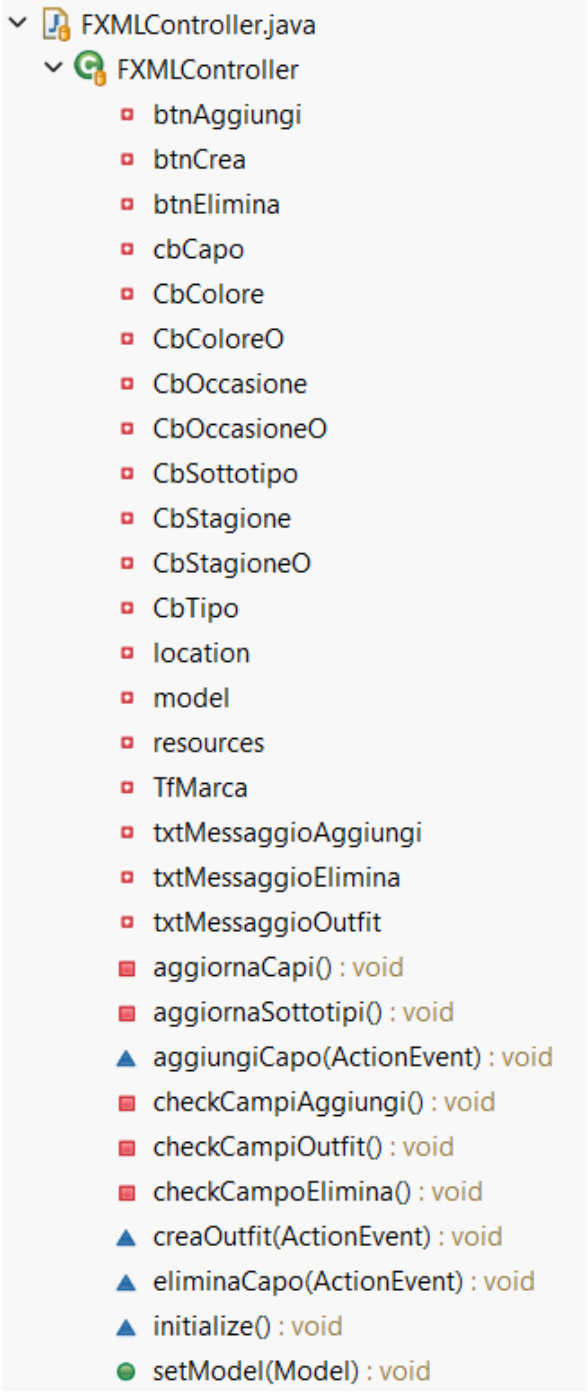


Classe Model:



Classe GuardarobaDAO:

Classe FXMLController:



6 - Alcune videate dell'applicazione e link al video

Come si può vedere dallo schema nella pagina precedente, la classe "FXMLController" gestisce le interazioni con l'utente attraverso una serie di metodi attivati quando l'utente preme i bottoni presenti nell'interfaccia. L'interfaccia è suddivisa in tre parti separate da tre tab: "Aggiungi", "Elimina" e "Crea outfit".

La prima parte permette di aggiungere un capo d'abbigliamento al guardaroba. Il bottone "Aggiungi" si attiva solamente quando l'utente ha compilato tutti i campi. La lista contenuta nel ChoiceBox Sottotipo viene aggiornata ogni qualvolta che l'utente seleziona il Tipo.

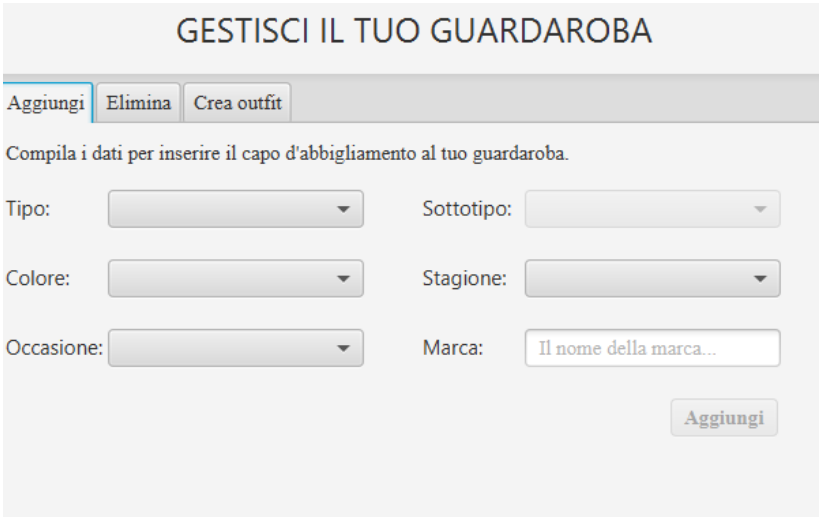


Figura 9 Interfaccia grafica tab "Aggiungi"

La seconda parte è utile per rimuovere un capo d'abbigliamento dal guardaroba. Il bottone "Elimina" si attiva solamente quando l'utente ha selezionato il capo nel rispettivo ChoiceBox.

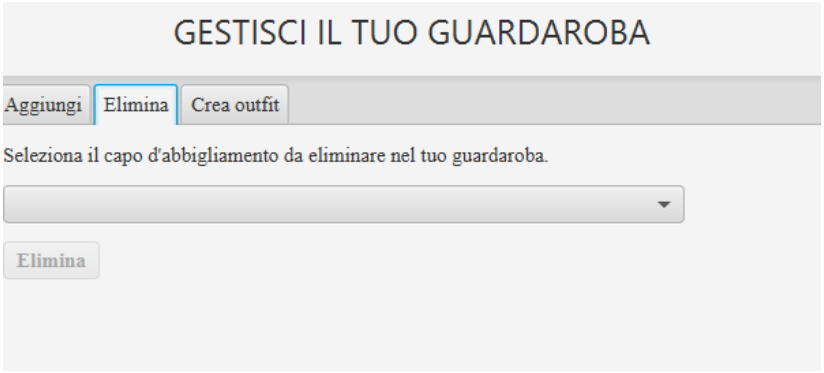


Figura 10 Interfaccia grafica tab "Elimina"

La terza parte è dedicata alla creazione degli outfit. Il bottone "Crea" si attiva solamente quando l'utente ha selezionato i filtri obbligatori.

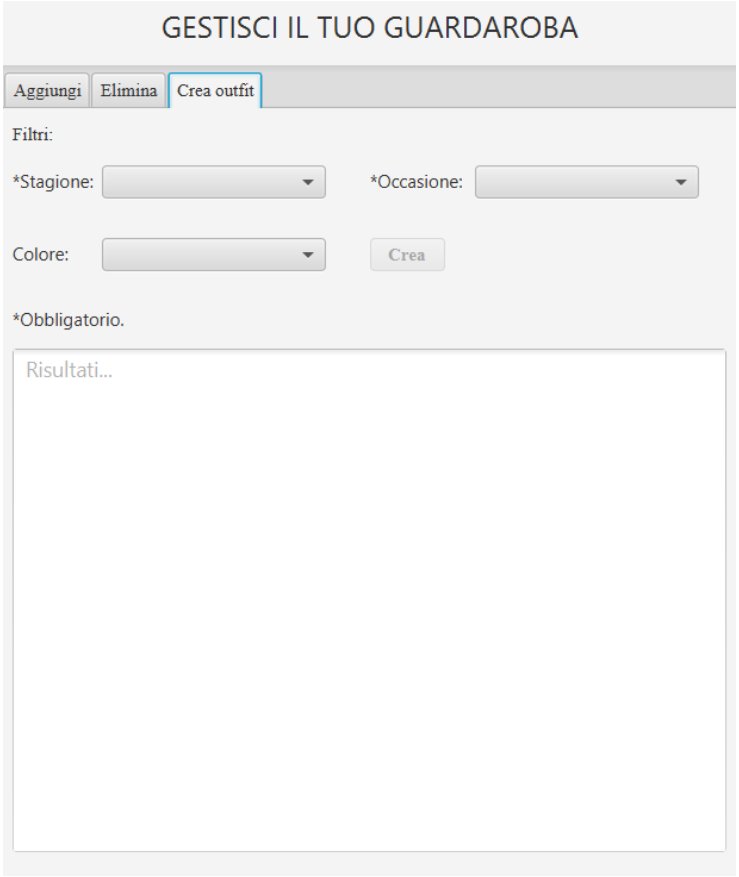


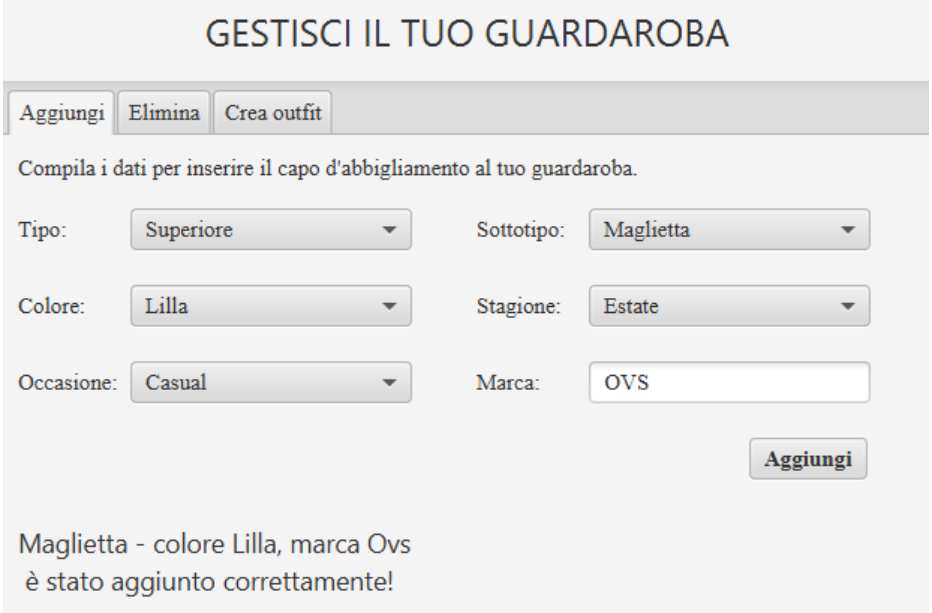
Figura 11 Interfaccia grafica tab "Crea outfit"

LINK per il video illustrativo: https://www.youtube.com/watch?v=A70ayNkHKPw

7 - Tabelle con risultati sperimentali ottenuti

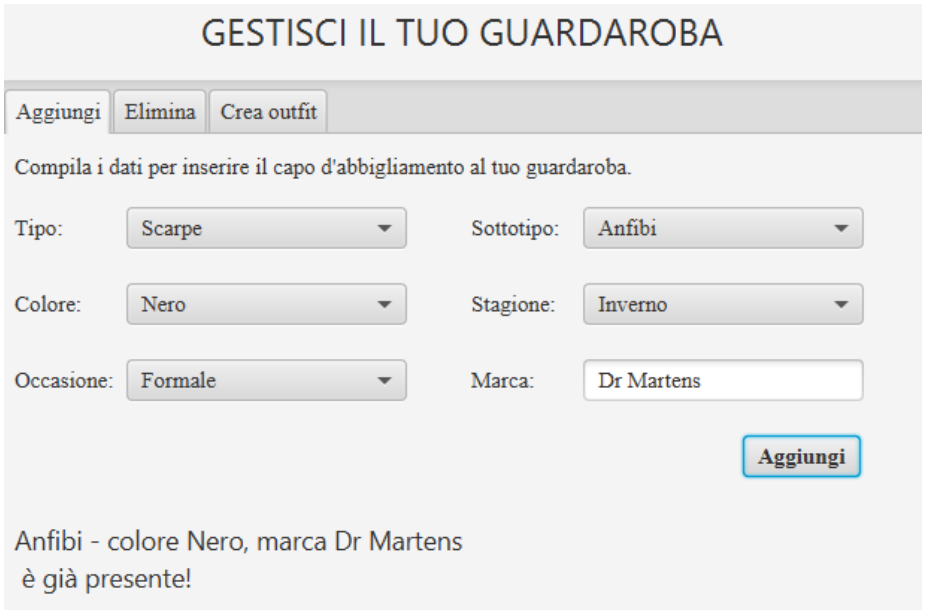
7.1 Schermata 1ª modalità di utilizzo

7.1.1 Esempio 1 – Capo d'abbigliamento aggiunto



L'utente seleziona tutte le caratteristiche del capo d'abbigliamento che vuole aggiungere al guardaroba, a questo punto si abilita il bottone "Aggiungi". Una volta cliccato, si attiva l'algoritmo che, dopo aver effettuato vari controlli sui parametri inseriti dall'utente, aggiunge l'abito al database. Infine, viene mostrato un messaggio in basso nell'interfaccia.

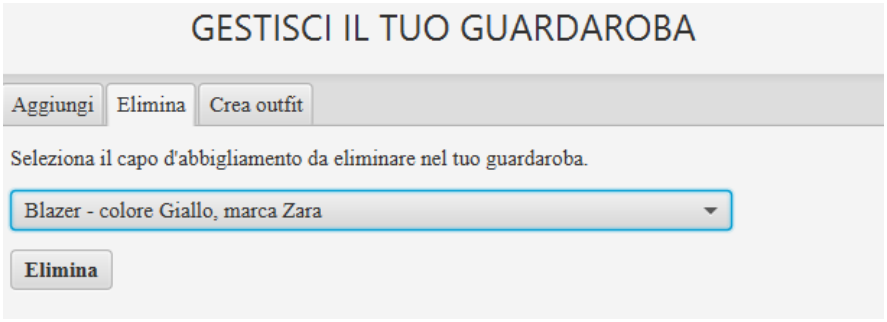
7.1.2 Esempio 2 – Capo d'abbigliamento già presente



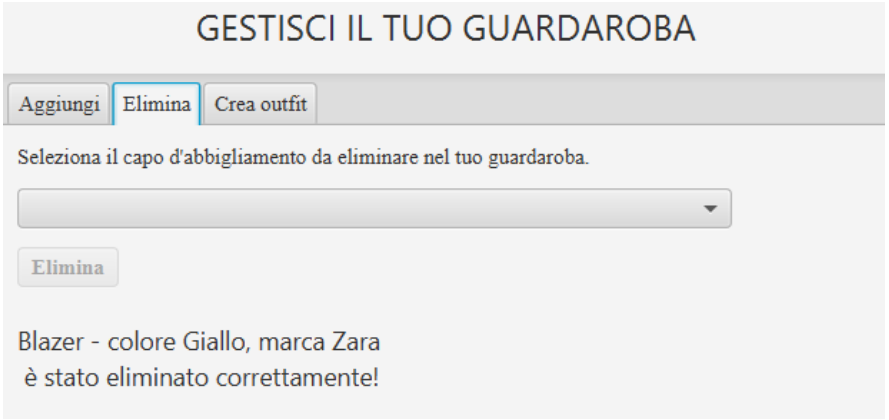
Nel caso in cui il capo d'abbigliamento descritto dall'utente è già presente nel guardaroba, esso non viene aggiunto al database e viene mostrato un avviso nell'interfaccia in basso.

7.2 Schermata 2ª modalità di utilizzo

7.2.1 Esempio 1 – Capo d'abbigliamento eliminato

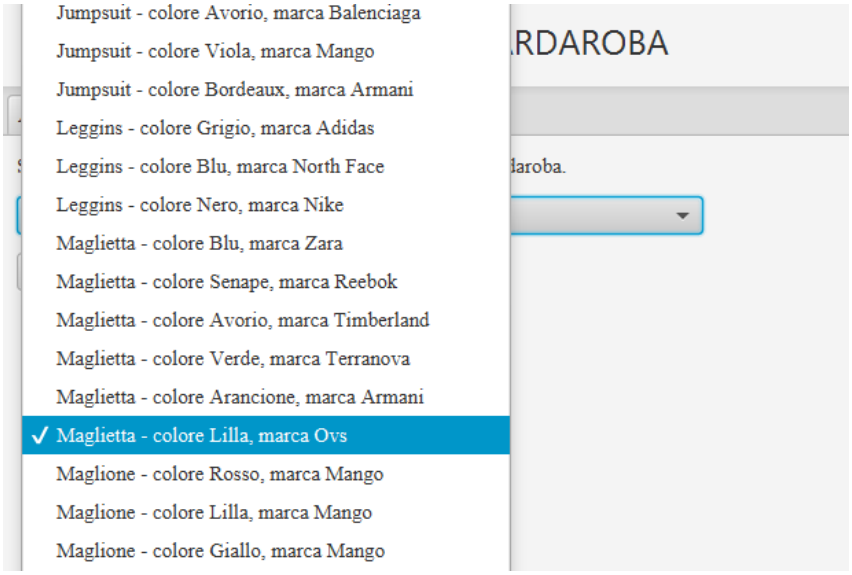


L'utente seleziona il capo d'abbigliamento che vuole rimuovere dal guardaroba, a questo punto il bottone "Elimina" viene abilitato.



Una volta cliccato il pulsante, si attiva l'algoritmo che elimina il capo selezionato dall'utente dal database. Viene mostrato un messaggio in basso nell'interfaccia.

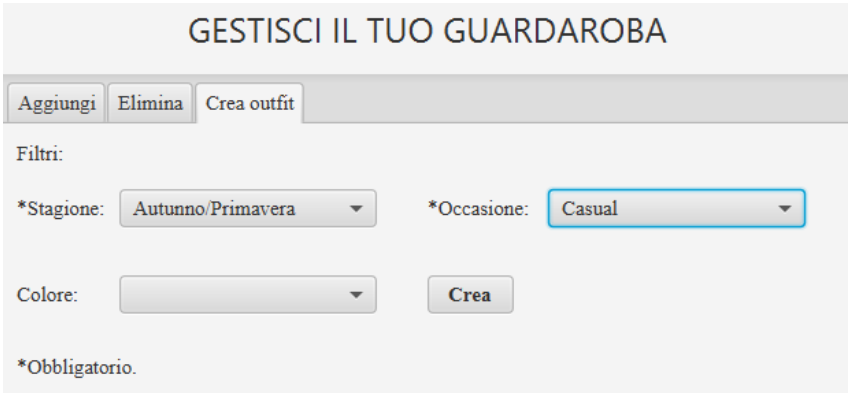
7.2.2 Esempio 2 – Lista aggiornata



La lista viene aggiornata ogni volta che viene aggiunto o rimosso un capo d'abbigliamento. In figura viene mostrato il capo aggiunto nell'esempio 7.1.1.

7.3 Schermata 3ª modalità di utilizzo

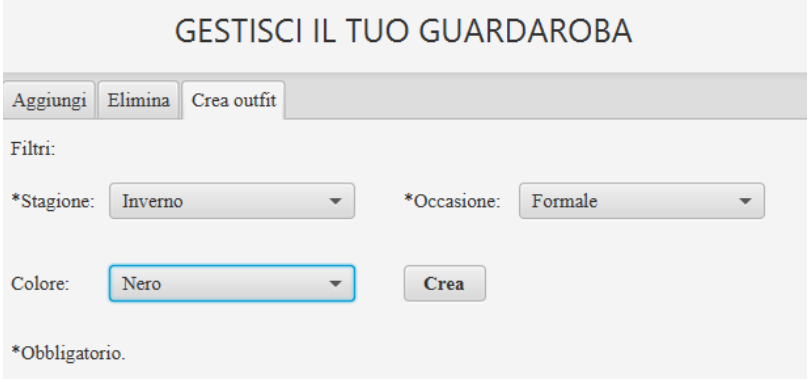
7.3.1 Esempio 1 – Outfit senza colore selezionato

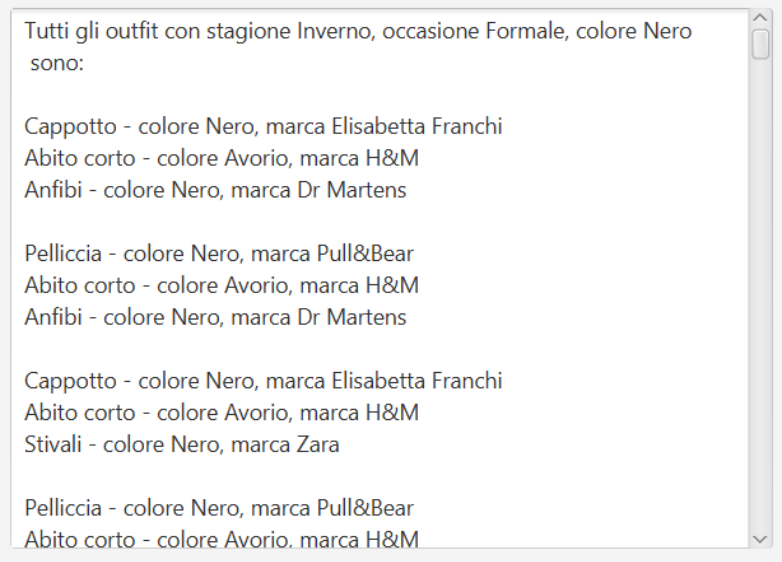


Quando l'utente seleziona la stagione e l'occasione, il pulsante "Crea" viene abilitato. Una volta cliccato, attiva l'algoritmo che crea tutte le combinazioni possibili che hanno i parametri selezionati dall'utente.

In questo caso il colore non è stato selezionato, dunque tutti gli outfit sono stati creati secondo l'abbinamento dei colori suggerito dalla tabella "Abbinamento" presente nel database.

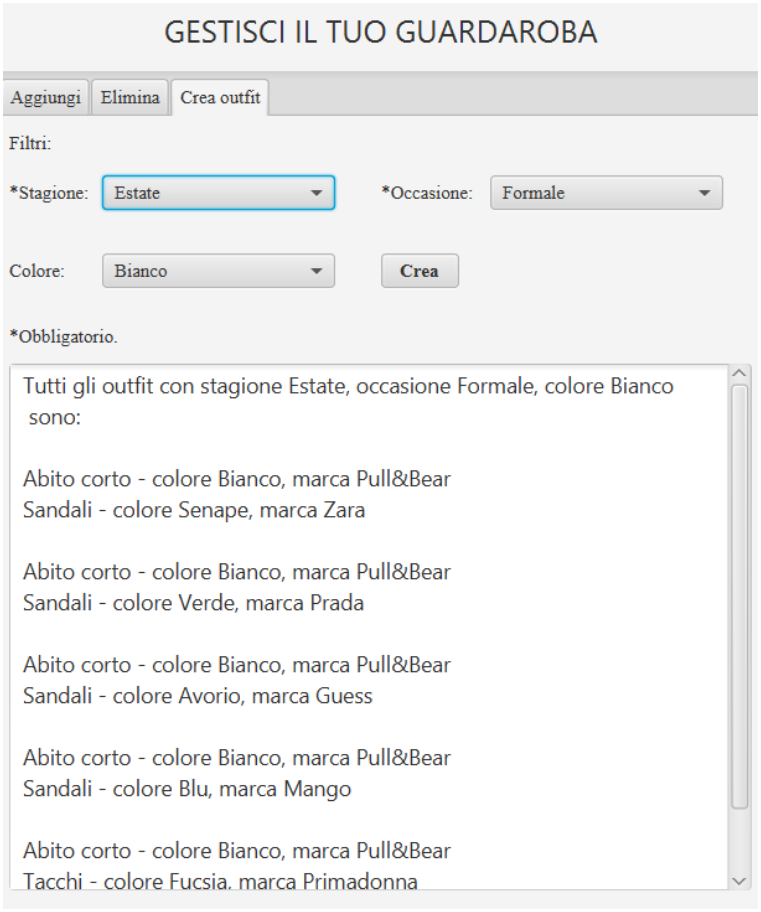
7.3.2 Esempio 2 – Outfit con colore selezionato





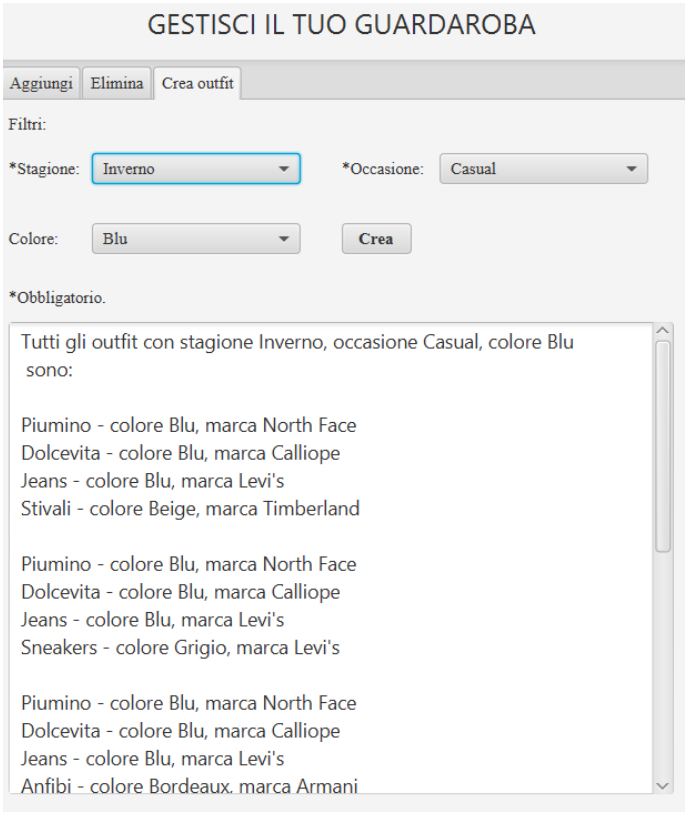
Nel caso in cui è stato selezionato anche il colore, gli outfit devono contenere almeno un capo d'abbigliamento con il colore scelto dall'utente; sempre rispettando gli abbinamenti dati dalla tabella "Abbinamento" presente nel database.

7.3.3 Esempio 3 – Outfit senza capospalla



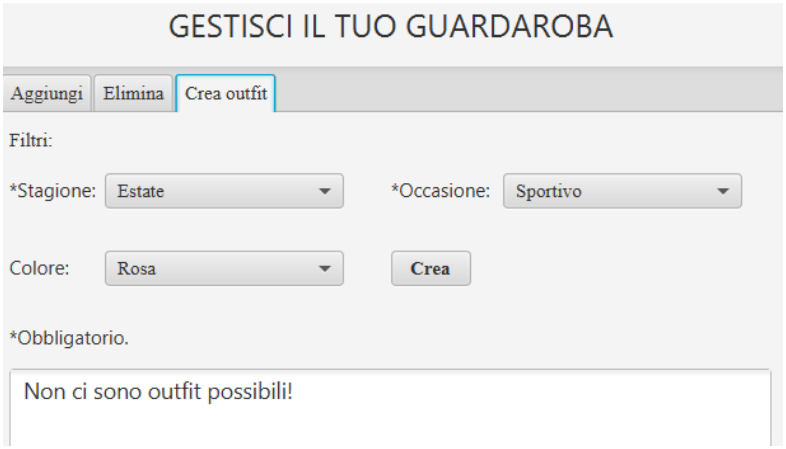
Se la stagione selezionata dall'utente è "Estate", allora tutti gli outfit non devono avere un capospalla.

7.3.4 Esempio 4 – Outfit con capospalla



Se la stagione selezionata dall'utente è "Inverno" oppure "Autunno/Primavera", tutti gli outfit devono comprendere anche un capospalla.

7.3.5 Esempio 5 – Nessun Outfit disponibile



Se l'algoritmo non trova nessuna combinazione possibile che rispetti le preferenze scelte dall'utente, verrà mostrato un avviso nell'interfaccia.

8 - Valutazioni sui risultati ottenuti e conclusioni

Gli algoritmi che permettono di aggiungere e rimuovere un capo d'abbigliamento dal guardaroba, sono abbastanza semplici e veloci. Invece la parte più complessa, che rappresenta il cuore dell'applicazione, è sicuramente la generazione degli outfit.

L'algoritmo che trova gli abbinamenti di colore tra i capi e l'algoritmo che si occupa di formare gli outfit con i diversi componenti, costituiscono la parte più onerosa in termini di risorse utilizzate. I tempi di risposta dipendono dall'input fornito dall'utente e dal numero di capi presenti nel database.

Gli aspetti chiave che influenzano la velocità di esecuzione dell'algoritmo utilizzato sono la numerosità di vertici e archi del grafo; infatti maggiore è il numero di collegamenti possibili, maggiori saranno le combinazioni da esaminare. Questo dipende dai parametri impostati dall'utente, in quanto se quest'ultimo seleziona solamente la stagione e l'occasione, le combinazioni da mostrare saranno più numerose, mentre selezionando anche il colore gli outfit saranno più ridotti. Inoltre un altro aspetto che influenza la velocità di esecuzione è il numero di capi presente nel guardaroba dato che l'utente ha la possibilità di aggiungere abiti e dunque la numerosità dei dati più diventare elevata. Ma, fortunatamente, l'utente può anche eliminare i capi dal guardaroba riducendo in questo modo le dimensioni del grafo e a sua volta le combinazioni da calcolare.

Tuttavia, in generale, l'applicazione offre un'esperienza fluida.