Progetto di Ingegneria del Software:
Prova Finale

Nome: Samuele Dario
Cognome: Scamozzi
Matricola: 986209
Codice Persona: 10772778
E-mail istituzionale: samueledario.scamozzi@mail.polimi.it
Voto target: dal 20 al 25

Indice

Progetto

Applicazione per la Gestione degli Eventi in una Sagra Paesana

Introduzione: Ambito del Progetto

L'applicazione è progettata per la gestione completa degli eventi in una sagra paesana, offrendo agli organizzatori e agli utenti una piattaforma centralizzata per la pianificazione, la partecipazione e la gestione degli eventi tipici di una festa paesana. Questo include la gestione di ~~pranzi/cene con prodotti tipici, lotterie, prenotazioni tavoli,~~ servizio bar, acquisti di prodotti da portare via e soprattutto la gestione di eventi di vario tipo. L'obiettivo principale è fornire un'esperienza utente intuitiva e completa, semplificando la partecipazione agli eventi e migliorando la pianificazione e la gestione per gli organizzatori.

Obiettivi Principali

L'obiettivo del progetto è lo sviluppo di un'applicazione web adatta all'uso in una sagra di paese, per fornire una gestione più digitalizzata di questo tipo di eventi; il sistema prevede la creazione di tre livelli di utenti con differenti autorizzazioni, che in ordine di importanza decrescente sono: l'organizzatore, il responsabile bar/venditore e l'utente registrato. Viene mostrato ora il focus di questo progetto, che sarà:

1) Implementare un sistema CRUD per la gestione degli eventi, parte centrale del progetto, consentendo agli utenti di leggere e partecipare agli eventi, mentre agli organizzatori di creare, aggiornare ed eliminare eventi.

2) Introdurre funzionalità aggiuntive che aggiungono complessità alla gestione dei dati nel "database", offrendo un'esperienza più ricca agli utenti, come la possibilità di fare acquisti di prodotti e prodotti da portare via, prodotti che verranno aggiunti, modificati ed eliminati dai responsabili bar/venditori; questi responsabili, inoltre, saranno coloro assegnati alla gestione degli ordini eseguiti dagli utenti, avendo la possibilità di completare i prodotti fino al termine dell'ordine, che sarà possibile solamente dopo la preparazione di tutti i prodotti e il ritiro dei prodotti da portare via da parte dell'utente che ha eseguito l'ordine.

3) Dare la possibilità agli utenti di visualizzare e gestire una propria libreria personale, contenente gli eventi a cui partecipa e gli ordini eseguiti, separandoli fra ordini in corso (che può gestire ritirando i prodotti da portare via) ed ordini completati.

4) Seguire rigorosamente i requisiti forniti, inclusi un'architettura client/server seguendo il pattern MVC e altri due pattern documentati (il Factory Pattern e l'Observer Pattern).

5) Sviluppare l'applicazione con un approccio orientato agli oggetti e assicurare una copertura completa dei test di unità utilizzando JUnit.

6) Fornire documentazione completa per il progetto, inclusi dettagli sull'architettura, l'implementazione dei pattern, i test effettuati e le istruzioni per l'utilizzo dell'applicazione.

Funzionalità per Utente

1. **Utenti Registrati:**
   1) Creazione di un account utente per accedere all'applicazione.
   2) Visualizzazione degli eventi disponibili nella sagra.
   3) Visualizzazione e acquisto di prodotti~~, oltre che di pranzi e cene~~.
   ~~4) Partecipazione alla lotteria di paese.~~
   5) Visualizzazione e gestione di una libreria personale.
2. **Organizzatori dell'Evento:**
   1) Creazione, visualizzazione, modifica, eliminazione degli eventi e dei dettagli.
   ~~2) Creazione e gestione della lotteria.~~
   3) Monitoraggio delle partecipazioni agli eventi.
3. **Responsabili del Servizio Bar e Venditori:**
   1) Visualizzazione e gestione degli ordini dei clienti per il servizio bar.
   2) Visualizzazione e gestione degli acquisti di prodotti da portare via.
   3) Controllo del magazzino e rifornimento dei prodotti (gestione del magazzino).

Tecnologie Utilizzate

Per lo sviluppo di questo progetto, verranno utilizzate tecnologie moderne e adatte allo scopo, inclusi linguaggi di programmazione come Java per il backend; il progetto, verrà dunque sviluppato per essere un applicativo. Per la gestione del database, verrà utilizzato il pattern Singleton per simulare un database contenente dati reali (l'obbiettivo finale, sarebbe utilizzare un sistema di persistenza dei dati come MySQL), si adotterà il pattern MVC e, oltre al pattern MVC, si seguiranno anche due pattern aggiuntivi che sono il Factory e l'Observer. Per i test di unità, verrà impiegato JUnit. Per tutti i Diagram è stato Visual Paradigm.

Comportamento dell'Applicazione

L'applicazione sarà composta da diverse sezioni, ognuna con funzionalità specifiche:

1) Gestione degli Eventi: Gli organizzatori potranno creare, visualizzare, modificare ed eliminare gli eventi tramite un'interfaccia intuitiva. Sarà possibile specificare dettagli come la data, l'orario, il luogo e le attività previste per ciascun evento.

2) Prenotazione dei Tavoli e Servizio Bar: Gli utenti potranno prenotare tavoli per i pranzi/cene e gestire gli ordini al servizio bar direttamente tramite l'applicazione. Questo includerà la visualizzazione dei tavoli disponibili, la selezione dei posti desiderati e la conferma delle prenotazioni.

3) Lotterie di Paese: Sarà possibile partecipare alle lotterie di paese tramite l'applicazione, acquistando biglietti virtuali e visualizzando i premi in palio. Gli organizzatori potranno gestire le lotterie, aggiungendo premi, impostando le quote di partecipazione e monitorando le vendite dei biglietti.

4) Acquisti di Prodotti da Portare Via: Gli utenti potranno esplorare una selezione di prodotti tipici disponibili per l'acquisto ed effettuare ordini direttamente

dall'applicazione. Sarà possibile visualizzare i prodotti disponibili, aggiungerli al carrello e completare l'acquisto tramite un processo di checkout intuitivo.

Fascia di Voto

La fascia di voto mirata per questo progetto è compresa tra 20 e 25, tenendo conto della complessità delle funzionalità implementate (la parte di progetto sviluppata con approccio ad oggetti), della solidità dell'architettura e dell'efficacia dei test di unità, oltre che della documentazione.

Requisiti per il Sistema Sviluppato

I requisiti funzionali sono stati essenziali per delineare le fasi di sviluppo, nonché per controllare l'aderenza di quanto sviluppato all'idea iniziale. Sono suddivisi come segue per una maggiore leggibilità.

1) **Visitatore Anonimo:**
    - Il visitatore anonimo può accedere solo alla funzionalità di registrazione.
    - Registrazione e Utenza:

o Ogni visitatore può creare o accedere ad un account fornendo un indirizzo e-mail valido (quindi, contenente almeno dei caratteri, una chiocciola, altri caratteri, un punto e altri caratteri ancora) e una password.
o Ogni visitatore può collegare al suo account, se decide di farlo, le informazioni della sua carta di credito.

- Non sono previsti ruoli specifici da assegnare agli utenti.
- Non è necessario un ruolo di amministratore o un'area amministrativa per gli utenti.

2) **Utente Registrato:**
    - Gli utenti registrati possiedono una libreria personale, inizialmente vuota.
    - Visualizzazione degli Eventi:

o Gli utenti possono cercare e visualizzare gli eventi disponibili nella sagra paesana.
o Gli utenti possono scegliere di partecipare agli eventi disponibili nella sagra paesana.

- Acquisto di Prodotti e Prodotti da Portare Via:

o Gli utenti possono esplorare e acquistare prodotti tipici disponibili per l'acquisto, e monitorare il loro acquisto nella propria libreria personale.
o Nel momento in cui un utente non ha ancora inserito i dati della sua carta, gli vengono richiesti: se non li inserisce, annulla l'acquisto.

- Gestione della Propria Lista di Eventi:

o Gli utenti, selezionando gli eventi e facendo acquisti, possono visualizzare, rimuovere e gestire gli eventi nella propria libreria personale, che sarà una lista di eventi, ma anche gestire gli acquisti, che sarà una lista di ordini eseguiti dall'utente.

3) **Organizzatore:**
    - Creazione degli Eventi:

o Gli organizzatori possono creare nuovi eventi inserendo dettagli come data, orario, luogo e attività.

- Gestione dei Dettagli degli Eventi:

o Gli organizzatori possono modificare e aggiornare i dettagli degli eventi creati.
o Gli organizzatori possono eleminare degli eventi creati.

- Monitoraggio delle Partecipazioni:

o Gli organizzatori possono monitorare le partecipazioni agli eventi da parte degli utenti per una migliore pianificazione e gestione.

4) **Responsabile Bar/Venditore:**
    - Gestione degli Ordini dei Clienti:

o Il responsabile del servizio bar/venditore può visualizzare e gestire gli ordini dei clienti per il servizio bar/per la vendita.

- Gestione degli Acquisti di Prodotti da Portare Via:

o Il responsabile/venditore può gestire gli acquisti di prodotti da portare via effettuati dai clienti nei vari ordini.

- Controllo del Magazzino e Rifornimento dei Prodotti:

o Il responsabile/venditore può controllare lo stato del magazzino e assicurare il rifornimento dei prodotti necessari.
o Il responsabile/venditore può aggiungere, modificare ed eliminare i prodotti presenti in magazzino.

Lista per i casi d'uso principali

Viene presentata questa lista per i casi d'uso principali:

- **Registrazione**

o Un visitatore anonimo crea un account fornendo un'e-mail valida, una password e a sua scelta le informazioni della sua carta.

- **Ricerca e Aggiunta di Eventi**

o Un utente registrato visualizza e cerca gli eventi disponibili nella sagra paesana.
o L'utente aggiunge gli eventi desiderati alla propria lista personale.
o Un organizzatore aggiunge, rimuove e gestisce gli eventi.

- **Gestione degli Eventi**

o Un utente registrato visualizza e gestisce i dettagli degli eventi nella propria lista personale (cancellazione).
o Un organizzatore monitora e le partecipazioni agli eventi.

- **Ordini e Magazzino**

o Un utente visualizza ed acquista prodotti, anche da portare via.
o Un responsabile del servizio bar/un venditore gestisce gli ordini dei clienti, fra cui i prodotti da portare via e monitora il magazzino, aggiungendo, modificando ed eliminando i prodotti presenti.

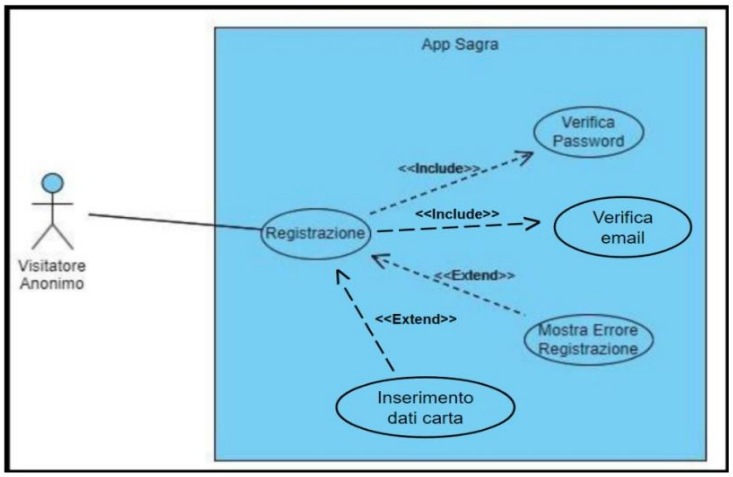
Use Case Diagram

I diagrammi Use Case mostrati nelle immagini permettono di visualizzare tutte le funzionalità a disposizione del sistema, raggruppando i requisiti funzionali dell'applicazione per ogni attore e le loro relazioni. Vengono mostrati 4 diagrammi per i 4 attori, Visitatore Anonimo, Utente Registrato, Responsabile Bar/Venditore e Organizzatore, collegati a tutte le azioni che possono svolgere. Le altre relazioni di dipendenza che si vedono nella rappresentazione sono:

- **Include**, che si trova, ad esempio, su tutte le azioni di Aggiungi, Modifica ed Elimina, in quanto sono azioni che vengono incluse dall'azione di Gestione.

- **Extend**, che si trova, ad esempio, su Mostra Errore Registrazione, poiché l'azione che l'utente deve svolgere è la Registrazione, che per estensione in determinate condizioni, richiede l'invio del messaggio.

1) Visitatore Anonimo



Il Visitatore Anonimo è qualsiasi persona che entra nell'applicazione, quindi qualsiasi sia il ruolo di chi sta accedendo, dovrà sempre passare per la Registrazione:

- Il Visitatore può accedere solo alla funzionalità di Registrazione, la quale include una Verifica E-mail e una Verifica Password.

o Finché il Visitatore non inserisce la password corretta, il sistema reagirà con Mostra Errore Registrazione e il Visitatore non potrà accedere al proprio account come Utente Registrato (oppure il ruolo di quel visitatore).

o Se è la prima volta che quella e-mail viene utilizzata, questo controllo viene meno e viene chiesto se creare un nuovo Utente Registrato.

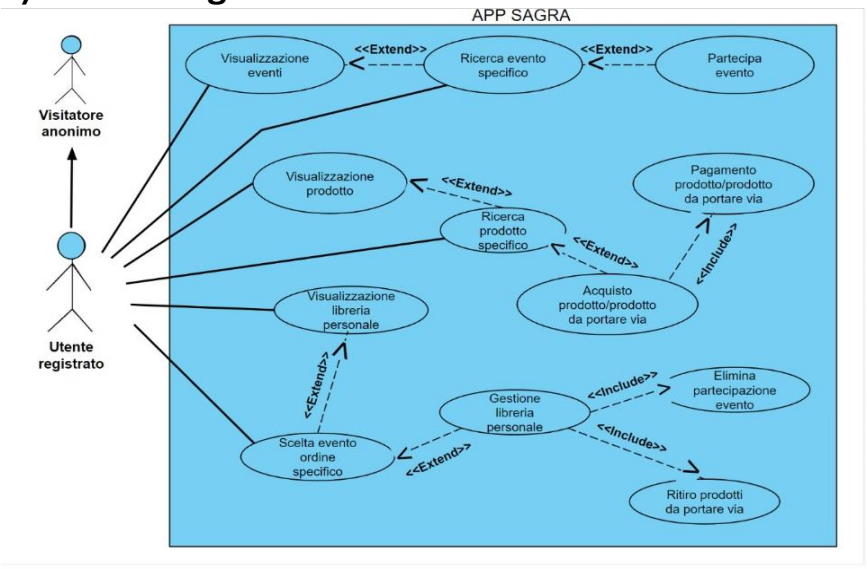
- Ogni Visitatore può creare un account fornendo un indirizzo e-mail non ancora utilizzato e una password.

o Non esiste un metodo per creare un account come ResponsabileBarVenditore oppure come Organizzatore, perciò le e-mail utili per queste classi sono impostate manualmente nella lista.

o Ogni volta che viene creato un nuovo utente, viene chiesto al nuovo utente se inserire i propri dati della carta di credito.

- Non sono previsti ruoli specifici da assegnare a chi accederà con un account di Utente Registrato.
- Non è necessario nessun ruolo di amministratore o un'area amministrativa per gli Utenti.

2) Utente Registrato



Dopo l'accesso ad un account, il Visitatore Anonimo passerà allo stato di Utente Registrato, ovvero colui che usufruisce dei servizi dell'app come attore principale, il quale non dovrà gestire nessun aspetto da amministratore; l'Utente potrà accedere alle seguenti funzionalità:

- Visualizzazione Eventi: Gli Utenti Registrati possono visualizzare gli eventi creati dagli Organizzatori.

o Questa funzionalità viene estesa con uno strumento di Ricerca Evento; infatti, gli Utenti potranno visualizzare degli eventi specifici, utilizzando uno strumento per la ricerca mirata.

- Partecipazione Eventi: Gli Utenti Registrati, dopo la ricerca di un evento specifico, possono scegliere di parteciparci.

o Questa partecipazione permette all'utente di visualizzare l'evento nella propria libreria personale.

- Visualizzazione Prodotti: Gli Utenti Registrati possono visualizzare i prodotti creati dai Responsabili Bar/Venditori.

o Questa funzionalità viene estesa con la ricerca di un prodotto specifico, al quale è collegato uno strumento di Acquisto Prodotti, al quale gli Utenti potranno anche accedere direttamente.

- Acquisto Prodotti: Gli Utenti Registrati possono acquistare i prodotti presenti.

o Questa funzionalità include il Pagamento Prodotti, poiché necessario, e viene estesa con una specifica da fare, ovvero il flag "da portare via"; infatti, gli Utenti potranno decidere se acquistare prodotti da consumare sul posto, oppure se acquistare prodotti da portare via.
o L'Acquisto Prodotti e Prodotti da Portare Via, include il Pagamento di tutti i Prodotti, e per i Prodotti da Portare Via è presente la possibilità di ritirarli dalla libreria personale; tuttavia, il ritiro dei prodotti avviene solamente dopo il completamento della loro preparazione.

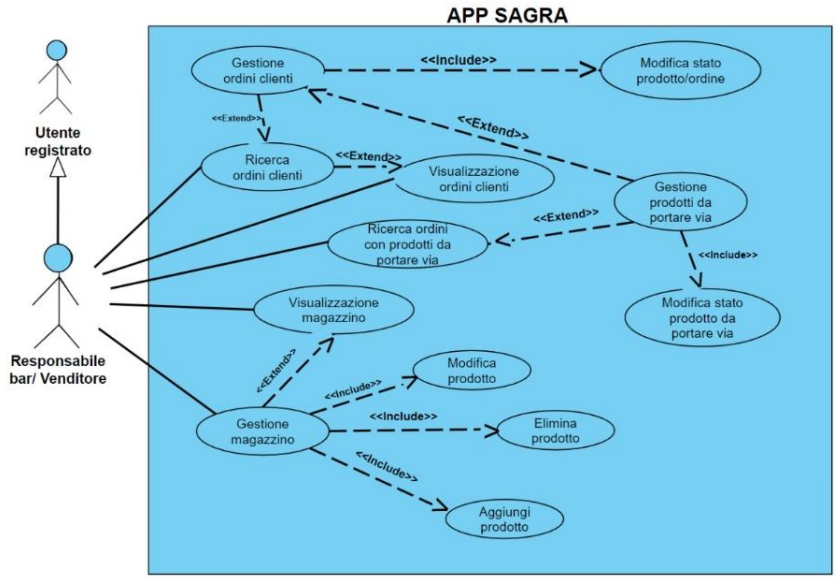
- Visualizzazione Libreria Personale: Gli Utenti Registrati possono visualizzare la propria libreria personale, che sarà già presente alla creazione di un nuovo Utente Registrato, ma vuota.

o Questa funzionalità viene estesa con uno strumento di Gestione Libreria Personale, al quale gli Utenti potranno direttamente accedere.

- Gestione Libreria Personale: Gli Utenti Registrati possono gestire la propria libreria personale.

o Questa funzionalità include l'Eliminazione della partecipazione da Eventi e la Scelta di gestire gli Ordini ritirando i prodotti da portare via.

3) Responsabile Bar/Venditore



Un account deve essere inserito manualmente per avere il ruolo come Responsabile Bar/Venditore e dunque poter accedere a funzionalità di gestione importanti; infatti, ha il compito di gestire una parte importante dell'applicazione, ovvero i Prodotti, per cui saranno a stretto contatto sia con gli Utenti (ma anche gli altri ruoli) in quanto clienti, che con gli Organizzatori in quanto tali. Il Responsabile Bar/Venditore potrà accedere alle seguenti funzionalità (oltre a quelle di un Utente Registrato):

- Visualizzazione Ordini Clienti: I Responsabili Bar/Venditori possono visualizzare gli ordini dei clienti, che saranno gli Utenti o chiunque esegue l'acquisto.

o Questa funzionalità viene estesa con uno strumento di selezione dell'ordine da gestire e di Gestione Ordini Clienti, al quale i Responsabili Bar/Venditori potranno direttamente accedere.

- Gestione Ordini Clienti: I Responsabili Bar/Venditori, dopo aver selezionato un ordine specifico, possono gestire gli ordini presenti fatti dagli utenti.

o Questa funzionalità include la possibilità di Modifica dello stato dei prodotti poiché necessario, e viene estesa con uno strumento di Gestione Prodotti non da Portare Via e gestione prodotti da portare via; infatti, i Responsabili Bar/Venditori saranno coloro che dovranno occuparsi della gestione degli ordini, che vuol dire preparare i prodotti e completare un ordine (al suo completamento, verrà rimosso in automatico dagli ordini attivi).

- Gestione Prodotti da Portare Via (e non): I Responsabili Bar/Venditori possono gestire i prodotti da portare via (e non) acquistati dagli utenti nei vari ordini.

o Questa funzionalità include la Modifica dello stato del Prodotto da Portare Via (e non) poiché necessaria; infatti, i Responsabili Bar/Venditori saranno coloro che dovranno occuparsi della gestione degli ordini, anche per quanto riguarda i prodotti da portare via.
o Per il completamento dell'Ordine, bisognerà attendere il ritiro dei prodotti da portare via da parte del cliente che ha eseguito l'ordine e la preparazione di tutti i prodotti.

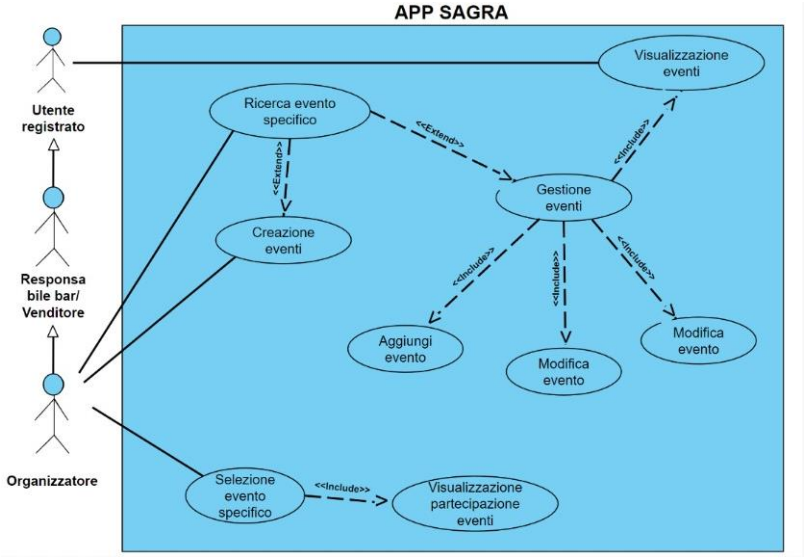
- Visualizzazione Magazzino: I Responsabili Bar/Venditori possono visualizzare il magazzino.

o Questa funzionalità viene estesa con uno strumento di Gestione Magazzino, al quale i Responsabili Bar/Venditori potranno direttamente accedere.

- Gestione Magazzino: I Responsabili Bar/Venditori possono gestire il magazzino.

o Questa funzionalità include l'Aggiunta, il Modifica, l'Elimina Prodotto poiché necessari.

4) Organizzatore



Un account deve essere inserito manualmente per avere il ruolo come Organizzatore e dunque poter accedere alle funzionalità di gestione più importanti, infatti, ha il compito di generare e gestire il fulcro dell'applicazione, ovvero gli Eventi, dove ognuno si può occupare di qualsiasi evento presente; inoltre, ha il compito di gestire anche i prodotti come i Responsabili Bar/Venditori, poiché ruolo di responsabilità. L'Organizzatore potrà accedere alle seguenti funzionalità (oltre a quelle di un Utente Registrato e di un Responsabile Bar/Venditore):

- Creazione Eventi: Gli Organizzatori possono creare eventi.

o Questa funzionalità permette la creazione di Eventi, con tutti i loro dettagli.

o Viene estesa con uno strumento di Gestione Eventi, al quale gli Organizzatori potranno direttamente accedere.

- Gestione Eventi: Gli Organizzatori, dopo aver selezionato un evento specifico, possono gestire gli eventi creati.

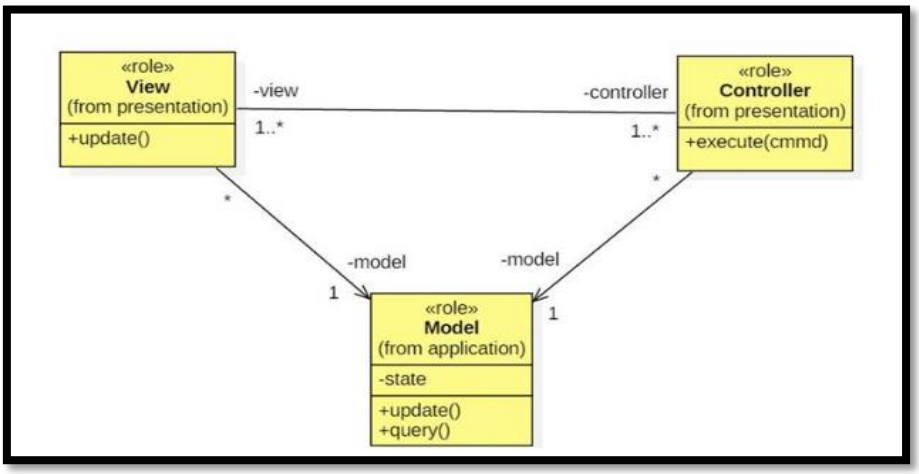
o Questa funzionalità include l'Aggiungi, il Modifica, l'Elimina Evento poiché necessari; infatti, gli Organizzatori saranno coloro che dovranno occuparsi della creazione e gestione degli eventi.

o L'opzione di modifica Evento permette di modificare i dettagli singoli di un evento specifico; questo, serve a dare profondità ad un evento, in cui sono presenti dettagli come data, ora di inizio e di fine, via.

- Visualizzazione Partecipazioni Eventi: Gli Organizzatori possono visualizzare le partecipazioni degli utenti agli eventi.

o Questa funzionalità serve per gestire fisicamente lo spazio necessario per l'evento e aiuta nella gestione di quest'ultimo.

Design Pattern: MVC



Il pattern MVC (Model-View-Controller) è un design pattern (per la precisione, pattern architetturale) largamente utilizzato nella programmazione orientata agli oggetti (OOP); esso, suddivide un'applicazione in tre componenti distinte: Model, View e Controller, ognuna con un ruolo specifico e ben definito. Questa suddivisione permette di separare la logica di business, la gestione della presentazione e il controllo del flusso dell'applicazione.

- **Model**

Il Model è la componente centrale del pattern MVC; il suo ruolo principale è quello di gestire i dati e la logica di business dell'applicazione. Questa componente mantiene lo stato dell'applicazione e si occupa di interagire con il database o altre fonti di dati esterne; la gestione delle operazioni di creazione, lettura, aggiornamento e cancellazione (CRUD) dei dati avviene all'interno del Model. Una sua caratteristica fondamentale consiste nell'essere indipendente dalla View, il che significa che può essere riutilizzato in diverse interfacce utente senza modifiche.

- **View**

La View è la componente responsabile della presentazione dei dati agli utenti; questa riceve le informazioni dal Model e le visualizza in un formato comprensibile e interattivo. La View è strettamente legata all'interfaccia utente e si occupa esclusivamente della rappresentazione grafica dei dati, senza incorporare logica di business; la sua responsabilità è quella di riflettere i cambiamenti di stato del Model e di aggiornare l'interfaccia utente in base a questi cambiamenti.

- **Controller**

Il Controller agisce come intermediario tra il Model e la View; la sua funzione principale è quella di ricevere e gestire gli input degli utenti, interpretando le loro azioni e decidendo quali operazioni eseguire sul Model. Il Controller può aggiornare il Model in base alle interazioni dell'utente e dopo richiedere alla View di aggiornarsi per riflettere i nuovi dati; questa separazione permette di mantenere una chiara distinzione tra il controllo del flusso dell'applicazione e la gestione della logica di business.

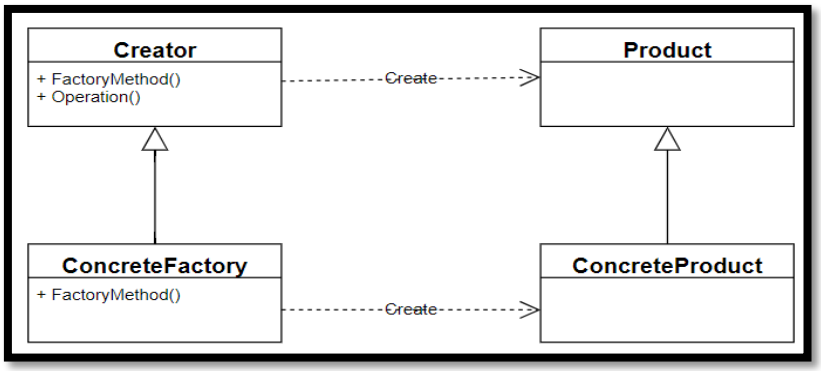
- **Funzionamento del Pattern MVC**

Nella pratica, quando un utente interagisce con l'interfaccia utente (Vista), il Controller cattura l'input e decide quale azione intraprendere: se l'azione richiede una modifica dei dati, il Controller interagisce con il Model per aggiornare lo stato dell'applicazione. Una volta che il Model è stato modificato, il Controller informa la View, che aggiorna la sua presentazione per riflettere i nuovi dati; questo ciclo di interazione continua, mantenendo una chiara separazione tra la logica di business, la presentazione e il controllo del flusso.

- **Vantaggi dell'utilizzo del pattern MVC**

L'adozione del pattern MVC in un'applicazione presenta vari vantaggi; infatti, facilita la manutenzione del codice, poiché la separazione delle responsabilità rende più semplice apportare modifiche a una parte del sistema senza influire sulle altre. Poi, la modularità che offre permette di riutilizzare i componenti, migliorando l'efficienza dello sviluppo e riducendo il rischio di errori; infine, l'MVC favorisce l'espandibilità del sistema, consentendo di aggiungere nuove funzionalità senza compromettere l'integrità dell'applicazione esistente.
In conclusione, questo pattern è uno strumento potente per organizzare e strutturare le applicazioni in modo modulare e manutenibile, garantendo una chiara separazione tra la logica di business, la presentazione e il controllo dell'applicazione.

Design Pattern: Factory



Il Factory Pattern è anche lui un design pattern (per la precisione, pattern creazionale) ampiamente utilizzato nella programmazione orientata agli oggetti; il cuore del pattern, risiede in un oggetto chiamato "Factory", che è responsabile della creazione di oggetti senza specificare la classe esatta dell'oggetto da istanziare. Questo pattern fornisce un'interfaccia per la creazione di oggetti, lasciando alle sottoclassi il compito di determinare quale classe concreta istanziare.

- **Factory**

Come già specificato, il Factory è l'elemento centrale del pattern; il suo scopo è quello di fornire un metodo di creazione che nasconde i dettagli di istanziazione degli oggetti. Invece di creare gli oggetti direttamente utilizzando il costruttore di una classe, si utilizza il Factory per ottenere un'istanza

dell'oggetto desiderato; questo approccio permette di centralizzare la logica di creazione degli oggetti, facilitando eventuali modifiche o estensioni in futuro.

- **Prodotto**

Il Prodotto rappresenta l'interfaccia, o la classe astratta, che definisce il tipo di oggetto che verrà creato; esso, stabilisce le operazioni comuni che tutte le sue implementazioni concrete devono fornire. Le classi che implementano il Prodotto concretizzano le specifiche caratteristiche e comportamenti degli oggetti creati dal Factory; questo, permette di trattare tutti i prodotti creati in modo uniforme, indipendentemente dalla loro implementazione concreta.

- **ConcreteProduct**

ConcreteProduct è la classe concreta che implementa l'interfaccia Prodotto; ognuno di essi rappresenta una specifica implementazione del prodotto, con le proprie caratteristiche e comportamenti distinti. Il Factory Pattern permette di creare oggetti di queste classi concrete senza esporre il loro costruttore direttamente nel codice client, mantenendo così un basso accoppiamento.

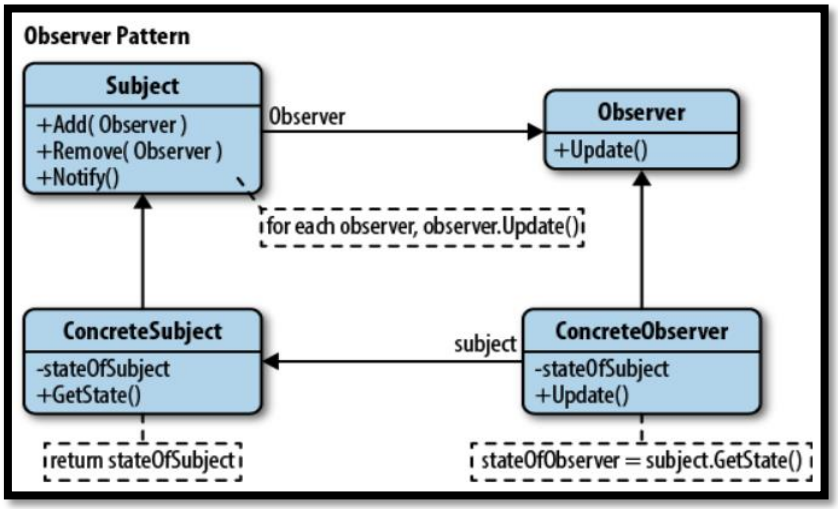
- **Funzionamento del Factory Pattern**

Nel momento in cui viene utilizzato questo pattern, quando un client necessita di un oggetto, esso non invoca direttamente il costruttore della classe concreta, ma si affida al Factory; questo, basato su parametri o condizioni specifiche, decide quale classe concreta istanziare e restituisce un'istanza dell'oggetto richiesto. Questo meccanismo è utile quando il processo di creazione degli oggetti è complesso o dipende da variabili che non sono note in anticipo.
Per esempio, nel progetto attuale, si potrebbe avere il Factory di Prodotto che crea diverse tipologie di prodotto (Cibo, Bere, Vestiti) in base ai parametri forniti dal client; il client non deve conoscere i dettagli di come ogni prodotto viene creato; sa solo che può richiedere un prodotto al Factory, che si occuperà di restituire l'oggetto corretto.

- **Vantaggi dell'utilizzo del Factory Pattern**

L'adozione del Factory Pattern offre numerosi vantaggi, tra cui:
o Basso accoppiamento: Il client non è direttamente legato alle classi concrete, permettendo una maggiore flessibilità nel cambiare la classe concreta senza influenzare il codice client.
o Centralizzazione della logica di creazione: La logica di creazione degli oggetti è centralizzata nel Factory, facilitando la gestione e la modifica del processo di creazione.
o Facilità di manutenzione ed estensione: Siccome la logica di creazione è isolata, è più semplice aggiungere nuove classi concrete o modificare quelle esistenti senza impattare il codice client.
In conclusione, il Factory Pattern è un buon strumento per la creazione di oggetti, specialmente in scenari in cui il processo di creazione è complesso o variabile; esso, favorisce un codice più modulare, manutenibile e facilmente estendibile, rendendo l'applicazione più robusta e flessibile nel lungo termine.

Design Pattern: Observer



L'Observer Pattern è anch’esso un design pattern (per la precisione, pattern comportamentale) adottato nella programmazione orientata agli oggetti; il suo scopo principale è quello di definire una relazione uno-a-molti tra gli oggetti, dove un oggetto "subject" notifica automaticamente una serie di "observers" ogni volta che il suo stato subisce un cambiamento. Questo meccanismo permette di mantenere gli observers sincronizzati senza che essi debbano richiedere costantemente aggiornamenti al subject.

- **Subject**

Il subject è il principale oggetto che mantiene lo stato che può interessare a più observers; quando il suo stato cambia, il subject si occupa di notificare tutti gli observers registrati, evitando così che debbano monitorare attivamente lo stato. Il subject contiene metodi per aggiungere, rimuovere e notificare gli observers, gestendo così la lista di oggetti interessati a ricevere aggiornamenti.

- **Observer**

Gli observers sono gli oggetti che vogliono essere informati del cambiamento di subject; essi implementano un'interfaccia comune che definisce un metodo di aggiornamento, il quale viene chiamato dal subject quando avviene una modifica. Questo metodo permette all'observer di reagire in modo appropriato ai cambiamenti, potendo aggiornare, per esempio, la propria interfaccia utente, oppure potendo eseguire altre azioni necessarie.

- **Interazione tra Subject e Observers**

Quando vengono utilizzati nella pratica, gli observers si registrano presso il subject attraverso un metodo fornito da quest'ultimo; nel momento in cui si verifica un cambiamento nello stato del subject, esso invoca il metodo di aggiornamento su ciascun observer registrato, passando le informazioni necessarie. Questo meccanismo assicura che tutti gli observers siano costantemente allineati con lo stato attuale del subject, senza dover controllare manualmente eventuali modifiche.

- **Applicazioni Tipiche**

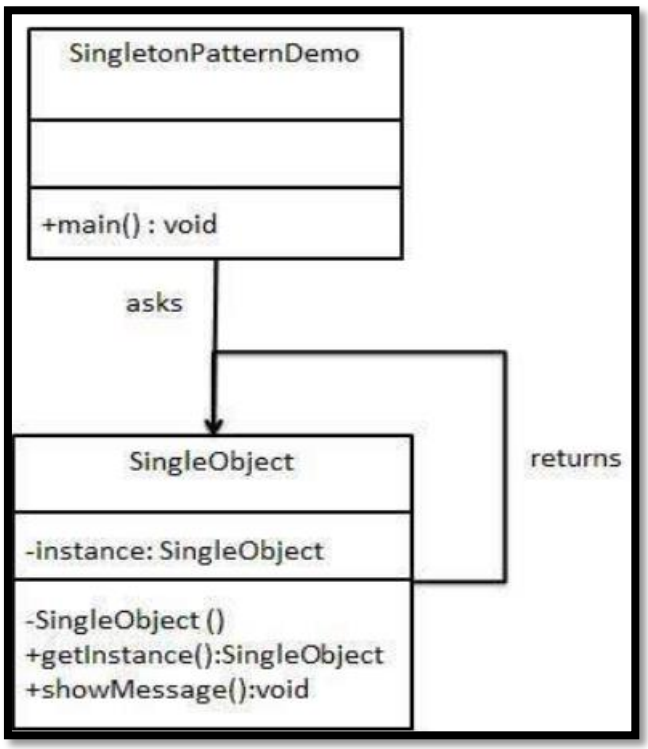
L'Observer Pattern è frequentemente utilizzato in applicazioni dove è essenziale mantenere coerenza tra diverse componenti in risposta a eventi o modifiche di stato; se si vuole fare un esempio con il progetto, si potrebbe pensare ad un eventuale interfaccia utente grafica, dove l'aggiornamento di un possibile modello di dati deve riflettersi automaticamente in più viste, come tabelle, grafici o pannelli di controllo (al momento attuale, aggiornano solamente l’unica vista a loro collegata, che viene stampata poi in console). In questo caso, il modello di dati agirebbe come subject, mentre le varie viste fungerebbero da observers.

- **Vantaggi del Pattern Observer**

o Uno dei principali vantaggi dell'Observer Pattern è la sua capacità di ridurre il livello di accoppiamento tra subject e observers; poiché gli observers non hanno bisogno di conoscere i dettagli interni del subject, ma solo l'interfaccia che esso espone, il codice risulta più modulare e facile da manutenere. Inoltre, facilita l'estensibilità, permettendo di aggiungere nuovi observers senza dover modificare il subject o gli altri observers esistenti.

In sintesi, l'Observer Pattern è una soluzione efficace per gestire la comunicazione tra oggetti in modo decentrato e reattivo; la sua capacità di mantenere gli observers aggiornati rispetto ai cambiamenti del subject lo rende ideale in scenari dove la coerenza dei dati è fondamentale. Utilizzando questo pattern, si ottiene un sistema più flessibile, modulare e mantenibile.

Design Pattern: Singleton



Il Singleton Pattern, come tutti i precedenti, è un design pattern (per la precisione, pattern creazionale) presente e sfruttato nella programmazione orientata agli oggetti; il fulcro del pattern è un oggetto, chiamato "Singleton", che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Questo pattern è particolarmente utile quando è necessario un controllo rigoroso sulla creazione di un singolo oggetto che deve essere condiviso attraverso l'intera applicazione.

- **Singleton**

Il Singleton è l'elemento centrale del pattern; la sua principale responsabilità, è quella di assicurare che non esistano più istanze della classe a cui appartiene. Questo, viene realizzato mediante la restrizione dell'accesso al costruttore della classe, combinata con un metodo pubblico che restituisce l'unica istanza esistente: se l'istanza non esiste ancora, il metodo la crea; altrimenti, restituisce l'istanza esistente. Questo approccio garantisce che l'oggetto venga creato solo una volta durante il ciclo di vita dell'applicazione.

- **Punto di Accesso Globale**

Il Singleton fornisce un punto di accesso globale all'istanza unica della classe; questo significa che l'istanza può essere utilizzata da qualsiasi parte dell'applicazione senza doverla passare come parametro. Il fatto che la gestione dell'istanza unica sia centralizzata semplifica il codice e riduce la possibilità di errori legati alla creazione multipla di oggetti; in altre parole, il Singleton agisce come un gestore globale per un oggetto specifico.

- **Thread-Safety**

In ambienti multithreaded, l'implementazione del Singleton garantisce la sicurezza del thread, ovvero la thread-safety, per evitare che più thread possano creare istanze multiple contemporaneamente; questo può essere ottenuto attraverso varie tecniche, come l'uso di blocchi sincronizzati o l'inizializzazione pigra. Questi approcci assicurano che l'istanza del Singleton venga creata in modo sicuro anche in un contesto con più thread.

- **Funzionamento del Singleton Pattern**

Quando viene sfruttato, nel momento in cui un client richiede l'istanza del Singleton, invoca un metodo, tipicamente denominato getInstance() o simile; se l'istanza non è ancora stata creata, il metodo la crea e la memorizza per gli accessi futuri. Viceversa, se l'istanza esiste già, viene

semplicemente restituita. Questo meccanismo garantisce che l'oggetto venga creato una sola volta e che tutte le parti dell'applicazione condividano la stessa istanza.
Nell'applicazione attuale, per esempio, si utilizza Singleton per controllare l'accesso ai prodotti che sono condivisi; in questo modo, si gestisce la condivisione della risorsa, che si trova nel suo "database".

- **Vantaggi dell'utilizzo del Singleton Pattern**

L'adozione del Singleton Pattern offre diversi vantaggi:
o Controllo sull'istanza unica: Garantisce che esista una sola istanza di una classe, il che è essenziale per risorse condivise come database o registri di configurazione.
o Accesso globale: Fornisce un punto di accesso centralizzato, semplificando l'accesso all'istanza e riducendo la complessità del codice.
o Risparmio di risorse: Poiché l'istanza viene creata solo quando è effettivamente necessaria, si risparmiano risorse di sistema.
In conclusione, il Singleton Pattern è uno strumento essenziale per situazioni in cui è necessario garantire che una classe abbia una sola istanza e che questa istanza sia accessibile in modo globale; la sua applicazione corretta contribuisce a rendere il codice più robusto, modulare e facile da mantenere, soprattutto in scenari che richiedono la condivisione di risorse comuni.

Design Pattern: Altri Pattern in breve

1 - Design Pattern: Template Method

Il Template Method è un design pattern utilizzato nella OOP per definire la struttura generale di un algoritmo, lasciando ai sottotipi la possibilità di implementare alcuni dettagli; il pattern prevede un metodo in una classe base che chiama altri metodi, alcuni dei quali possono essere definiti nelle classi derivate. Questo approccio permette di riutilizzare codice comune mantenendo flessibile l'implementazione dei dettagli specifici; in questo modo, è possibile estendere il comportamento di un algoritmo senza alterarne la struttura di base.

2 - Design Pattern: Ereditarietà

L'ereditarietà è un principio fondamentale dell'OOP che consente la creazione di nuove classi basate su classi esistenti, riutilizzando e ampliando il comportamento definito nella classe madre; una classe derivata eredita i campi e i metodi della classe madre, permettendo di aggiungere o modificare funzionalità. Questo meccanismo supporta la riusabilità del codice e la creazione di gerarchie di classi, facilitando l'organizzazione del codice e la sua estensibilità.

3 - Design Pattern: Iterator

L'Iterator è un design pattern che fornisce un modo per accedere agli elementi di una collezione senza esporre la sua struttura interna; il pattern definisce un'interfaccia che consente di attraversare una collezione in maniera sequenziale. Gli iteratori sono comunemente utilizzati per astrarre il processo di scorrimento delle collezioni, come liste o insiemi, e offrono metodi per ottenere l'elemento successivo o verificare se ci sono altri elementi da visitare, rendendo il codice più flessibile e indipendente dalla struttura della collezione.

4 - Design Pattern: Facade

Il Facade è un design pattern che fornisce un'interfaccia semplificata per un insieme complesso di classi o un intero sottosistema; il pattern mira a ridurre la complessità dell'interazione con un sistema, offrendo un unico punto di accesso alle funzionalità principali. La classe Facade si occupa di nascondere i dettagli di implementazione delle classi sottostanti, rendendo più facile l'uso e la comprensione del sistema per gli utenti esterni; questo è particolarmente utile per migliorare la manutenibilità e la leggibilità del codice.

5 - Design Pattern: State

Il pattern State è utilizzato per permettere a un oggetto di cambiare il suo comportamento quando cambia il suo stato interno; questo pattern definisce una serie di stati che l'oggetto può assumere,

ognuno dei quali implementa un comportamento specifico. Il contesto in cui l'oggetto si trova delega il lavoro allo stato corrente, adattando così il comportamento dinamicamente; questo approccio facilita l'estensione e la manutenzione del codice, riducendo il bisogno di lunghe istruzioni condizionali.

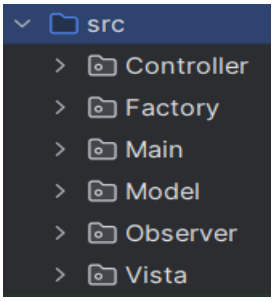
6 - Design Pattern: Strategy

Il pattern Strategy consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili; il pattern separa il comportamento dall'oggetto che lo utilizza, consentendo di modificare l'algoritmo senza alterare il contesto in cui viene eseguito. Questo pattern è utile quando si desidera poter scegliere tra diverse strategie di esecuzione in modo dinamico, migliorando la flessibilità e la riusabilità del codice.

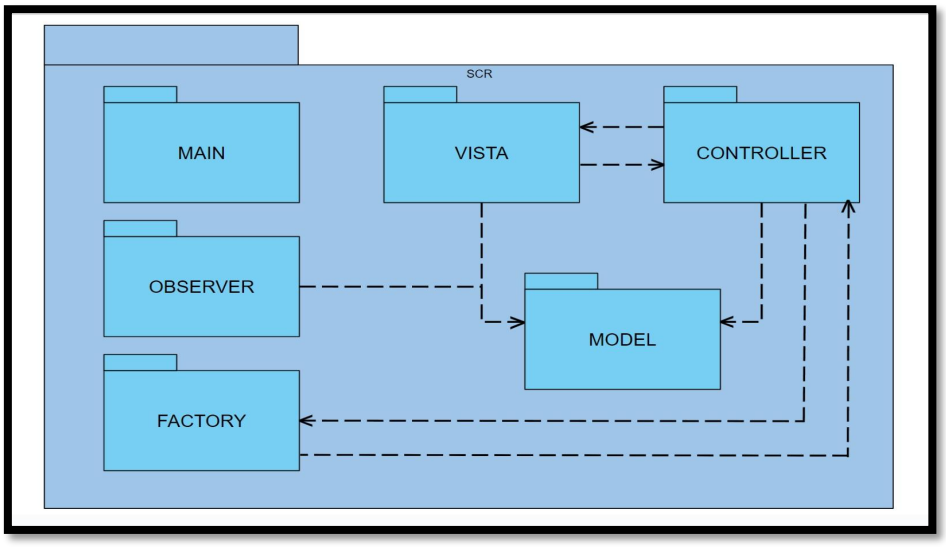
7 - Design Pattern: Factory Method

Il Factory Method è un design pattern che fornisce un'interfaccia per creare oggetti in una superclasse, permettendo alle sottoclassi di modificare il tipo di oggetto che sarà creato; infatti, invece di chiamare un costruttore direttamente, il pattern delega la creazione dell'oggetto a un metodo definito nella classe base. Questo approccio promuove l'estensibilità e permette di centralizzare la logica di creazione, facilitando l'aggiornamento e la manutenzione del codice.

Class Diagram: Packages



I Class Diagram dei packages sono utili visto l'elevato numero di classi utilizzate in questo progetto e dato il numero di metodi che sono necessari per ognuno di essi; è stato deciso di mostrare in primo luogo come interagiscono tra di loro i differenti pacchetti. Di seguito il package diagram del progetto.



Questi package contengono tutto il backend del programma, che nel caso di questo progetto è l'interezza del codice; la struttura di tutti i pacchetti contiene il tipo di classe che gli dà il nome, che nel caso specifico del Main è eseguibile, seguita dagli eventuali supporti, come il Model, gli Observer, la Vista, il Controller e il Factory.
Si ricorda che il progetto segue il **pattern MVC (Model-View-Controller)**, includendo anche l'implementazione di due ulteriori pattern di design: **Observer** e **Factory**.

1. Package: Controller

- Descrizione: Il package Controller contiene le classi che gestiscono la logica di controllo dell'applicazione; questo livello si occupa di ricevere input dall'utente, elaborare tali input, aggiornare lo stato del Model e fornire feedback appropriato alla Vista.
- Responsabilità:

o Intermediario tra Model e View.
o Gestione delle azioni e comandi ricevuti dall'interfaccia utente.

- Classi Principali: Le classi presenti in questo package definiscono i metodi necessari per gestire i flussi operativi dell'applicazione.

2. Package: Factory

- Descrizione: Il package Factory implementa il pattern Factory, che fornisce un'interfaccia per creare oggetti in una superclasse, consentendo alle sottoclassi di decidere quale oggetto istanziare. In questo caso, il pattern viene applicato ad alcune delle classi contenute nel Model, senza però utilizzarlo appieno; l'utilizzo potrebbe essere sviluppato tramite aggiornamenti futuri sul progetto.
- Responsabilità:

o Creazione centralizzata delle istanze delle classi del Model.
o Incapsulamento della logica di istanziazione, migliorando la manutenibilità del codice.

- Classi Principali: Contiene una classe Factory principale che include i metodi per creare istanze specifiche delle classi del Model.

3. Package: Main

- Descrizione: Il package Main contiene la classe principale eseguibile dell'applicazione, da cui si può avviare l'intero programma.
- Responsabilità:

o Avvio dell'applicazione.
o Inizializzazione dei componenti principali (Controller, Model, View).
o Gestione del ciclo di vita principale dell'applicazione.

- Classi Principali: Una classe Main che contiene il metodo main, punto di ingresso dell'applicazione.

4. Package: Model

- Descrizione: Il package Model rappresenta il livello di gestione dei dati dell'applicazione; contiene classi che rappresentano i dati e la logica del software.
- Responsabilità:

o Definizione delle strutture dati.
o Gestione dello stato e della logica applicativa.
o Comunicazione con il Controller per aggiornamenti e richieste di dati.

- Classi Principali: Contiene classi che definiscono la logica del software, i dati e le entità dell'applicazione.

5. Package: Observer

- Descrizione: Il package Observer implementa il pattern Observer, che permette a un oggetto di notificare automaticamente uno o più oggetti quando il suo stato cambia; questo pattern viene applicato a determinate classi del package Vista.
- Responsabilità:

o Consentire l'aggiornamento automatico delle Viste in risposta ai cambiamenti di stato nel Model.
o Migliorare la modularità e la separazione delle responsabilità tra i componenti.

- Classi Principali: Include le classi Observer, Subject (questa classe è stata utilizzata in maniera semplice per questioni di tempo e facilità di utilizzo), oltre alle specifiche implementazioni del metodo per le classi Vista che necessitano di essere osservate.

6. Package: Vista

- Descrizione: Il package Vista contiene le classi responsabili dell'interfaccia utente e della presentazione visiva dell'applicazione; queste classi si occupano di rendere i dati del Model visibili e interattivi per l'utente.
- Responsabilità:

o Presentazione dei dati all'utente.

- Classi Principali: Contiene le classi che gestiscono le diverse parti dell'utente e le loro interazioni.

L'architettura è dunque strutturata seguendo il pattern MVC, che consente una chiara separazione tra la logica di business, la logica di controllo e la presentazione dei dati; L'utilizzo dei pattern Factory e Observer permette di migliorare ulteriormente la flessibilità e la manutenibilità del codice. Il package Main serve come punto centrale di avvio, coordinando l'interazione tra i diversi componenti.

Class Diagram: Controller

Il package Controller è parte dell'architettura MVC (Model-View-Controller) del progetto; infatti, contiene le classi responsabili della logica di controllo, ossia la gestione dell'interazione tra il modello (Model) e la vista (View). Il package include tre classi principali: EventoController, ProdottoController e RegistrazioneController; ognuna di queste classi è incaricata di controllare una specifica parte dell'applicazione, gestendo la logica relativa agli eventi, ai prodotti e alla registrazione degli utenti.

1. Classe EventoController

- Descrizione: La classe EventoController gestisce la logica di controllo per gli eventi. Ha la responsabilità di gestire gli eventi disponibili nell'applicazione, ma si occupa anche della comunicazione con altre due componenti: la vista, che visualizza gli eventi; gli osservatori, che monitorano i cambiamenti relativi agli eventi.
- Attributi:

o private static EventoController instance: Implementa il pattern Singleton per garantire l'esistenza di una sola istanza di EventoController.
o private VistaEvento vistaEvento: Riferimento alla vista associata agli eventi.
o private List<Evento> listaEventi: Lista degli eventi gestiti dall'applicazione.
o private List<EventoOsservatore> osservatori: Lista degli osservatori registrati per ricevere notifiche sui cambiamenti negli eventi.

- Metodi:

o public static EventoController getInstance(): Restituisce l'istanza unica di EventoController.
o public void setVistaEvento(VistaEvento vistaEvento): Imposta la vista associata agli eventi.
o public void aggiungiOsservatore(EventoOsservatore osservatore): Aggiunge un osservatore alla lista.
o public void notificaOsservatori(String messaggio): Notifica tutti gli osservatori di un cambiamento.
o public void aggiungiEvento(Evento evento): Aggiunge un nuovo evento alla lista.
o public List<Evento> getListaEventi(): Restituisce la lista degli eventi.
o public Evento trovaEvento(String nomeEvento): Trova un evento tramite una ricerca sul nome.
Pattern Utilizzati:

- Singleton: Garantisce che ci sia solo un'istanza di EventoController.
- Observer: Permette alla classe di notificare i cambiamenti agli osservatori registrati.

2. Classe ProdottoController

- Descrizione: La classe ProdottoController gestisce la logica di controllo per i prodotti disponibili nell'applicazione; si occupa della comunicazione tra il modello dei prodotti e la vista che li presenta, oltre a gestire gli osservatori che monitorano i cambiamenti relativi ai prodotti.
- Attributi:

o private static ProdottoController instance: Implementa il pattern Singleton per garantire l'esistenza di una sola istanza di ProdottoController.
o private List<Prodotto> listaProdotti: Lista dei prodotti gestiti dall'applicazione.
o private List<ProdottoOsservatore> osservatori: Lista degli osservatori registrati per ricevere notifiche sui cambiamenti nei prodotti.

- Metodi:

o public static ProdottoController getInstance(): Restituisce l'istanza unica di ProdottoController.
o public void aggiungiOsservatore(ProdottoOsservatore osservatore): Aggiunge un osservatore alla lista.
o public void notificaOsservatori(String messaggio): Notifica tutti gli osservatori di un cambiamento.
o public List<Prodotto> getListaProdotti(): Restituisce la lista dei prodotti.
o public Prodotto trovaProdotto(String nomeProdotto): Trova un prodotto tramite una ricerca sul nome.
Pattern Utilizzati:

- Singleton: Garantisce che ci sia solo un'istanza di ProdottoController.
- Observer: Permette alla classe di notificare i cambiamenti agli osservatori registrati.

3. Classe RegistrazioneController

- Descrizione: La classe RegistrazioneController gestisce la logica di controllo per la registrazione degli utenti; ha la responsabilità di occuparsi del processo di registrazione, inclusa la validazione dell'e-mail, la creazione di nuovi utenti e la notifica degli osservatori di cambiamenti relativi alla registrazione.
- Attributi:

o private VistaRegistrazione vista: Riferimento alla vista associata alla registrazione.

o private Registrazione registra: Riferimento alla classe Registrazione utilizzata per gestire i dati relativi alla registrazione, ottenuta tramite una RegistrazioneFactory.
o private List<RegistrazioneOsservatore> osservatori: Lista degli osservatori registrati per ricevere notifiche sui cambiamenti nella registrazione.

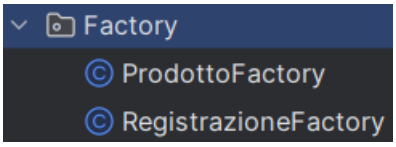
- Metodi:

o public RegistrazioneController(VistaRegistrazione vista): Costruttore che inizializza l'istanza di Registrazione tramite la RegistrazioneFactory e la lista di osservatori.
o public void registrazioneUtente(String e-mail, String password): Gestisce il processo di registrazione degli utenti, inclusa la validazione dell'e-mail e la creazione di nuovi utenti.
o public void setVista(VistaRegistrazione vista): Imposta la vista associata alla registrazione.
o public void aggiungiOsservatore(RegistrazioneOsservatore osservatore): Aggiunge un osservatore alla lista.
o public void notificaOsservatori(String messaggio): Notifica tutti gli osservatori di un cambiamento.
Pattern Utilizzati:

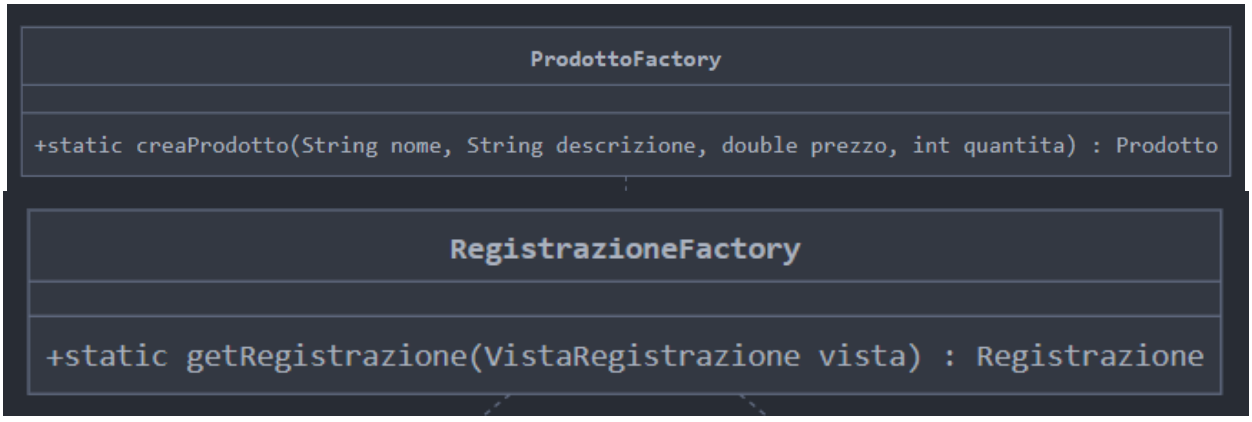
- Factory: La classe RegistrazioneController utilizza una RegistrazioneFactory per ottenere un'istanza della classe Registrazione, che incapsula la logica di creazione degli oggetti.
- Observer: Consente alla classe di notificare i cambiamenti agli osservatori registrati.

Il package Controller rappresenta dunque il livello di controllo dell'applicazione, gestendo la logica operativa e coordinando l'interazione tra il Model e la View; le classi principali del package, EventoController, ProdottoController e RegistrazioneController, implementano pattern di design che sono il **Singleton**, il **Factory** e l'**Observer**, assicurando così un'architettura solida, modulare e facilmente manutenibile. Questi pattern non solo organizzano il codice in modo chiaro e logico, ma facilitano anche l'estensibilità del sistema, rendendo semplice l'aggiunta di nuove funzionalità o la modifica di quelle esistenti.

Class Diagram: Factory



Il package Factory è progettato per centralizzare la creazione di determinati oggetti all'interno del sistema, secondo il pattern Factory; questo pattern di design è utilizzato per fornire un'interfaccia per la creazione di oggetti, delegando la logica di istanziazione alle classi factory. Il package Factory di questo progetto contiene due classi principali: ProdottoFactory e RegistrazioneFactory; queste, gestiscono rispettivamente la creazione di istanze della classe Prodotto e della classe Registrazione.



1. Classe ProdottoFactory

- Descrizione: La classe ProdottoFactory è responsabile della creazione di nuovi oggetti di tipo Prodotto; questa, incapsula la logica di creazione, fornendo un metodo statico che permette di istanziare un oggetto Prodotto con i parametri forniti.

- Attributi: La classe ProdottoFactory non ha attributi poiché si tratta di una factory che utilizza un metodo statico.
- Metodi:

o public static Prodotto creaProdotto(String nome, String descrizione, double prezzo, int quantita): Questo metodo statico accetta i parametri necessari per creare un nuovo oggetto Prodotto, restituendo l'istanza creata; i parametri includono il nome del prodotto, una descrizione, il prezzo e la quantità disponibile.

Pattern Utilizzati:

- Factory: La classe ProdottoFactory implementa il pattern Factory, che permette di centralizzare e semplificare la creazione degli oggetti Prodotto; questo approccio consente una maggiore flessibilità e facilita la gestione delle istanze, poiché la logica di creazione è isolata e può essere modificata senza impattare il resto del sistema.

2. Classe RegistrazioneFactory

- Descrizione: La classe RegistrazioneFactory è utilizzata per ottenere un'istanza di Registrazione, che gestisce la registrazione degli utenti; a differenza di ProdottoFactory, questa factory non crea direttamente una nuova istanza, ma utilizza un metodo statico per ottenere un'istanza già esistente di Registrazione, seguendo il pattern Singleton implementato in Registrazione.
- Attributi: Come ProdottoFactory, RegistrazioneFactory non possiede attributi poiché opera tramite un metodo statico.
- Metodi:

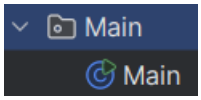
o public static Registrazione getRegistrazione(VistaRegistrazione vista): Questo metodo statico riceve un parametro VistaRegistrazione e lo utilizza per ottenere l'istanza di Registrazione attraverso il metodo getInstance della classe Registrazione; questo metodo, assicura che venga sempre utilizzata la stessa istanza di Registrazione, rispettando il pattern Singleton.

Pattern Utilizzati:

- Factory: La classe RegistrazioneFactory segue il pattern Factory, delegando la logica di istanziazione alla classe stessa; questo, permette di mantenere la creazione dell'oggetto Registrazione centralizzata e di gestire l'eventuale complessità dell'istanziazione in un unico punto.
- Singleton: Anche se il pattern Singleton è implementato direttamente nella classe Registrazione, la RegistrazioneFactory facilita l'accesso a questa unica istanza, contribuendo a mantenere il controllo centralizzato sulle istanze della classe.

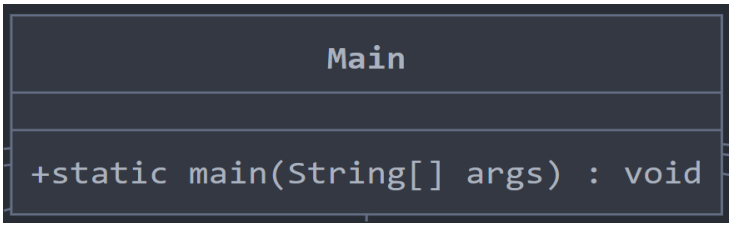
Il package Factory contribuisce in maniera importante nell'architettura del progetto, centralizzando e semplificando la creazione di oggetti attraverso il pattern Factory; le classi ProdottoFactory e RegistrazioneFactory assicurano che la logica di creazione degli oggetti sia ben organizzata e facilmente manutenibile. Inoltre, l'uso del pattern Singleton, combinato con la RegistrazioneFactory, garantisce che esista una singola istanza della classe Registrazione; questo, contribuisce a prevenire possibili incoerenze nel sistema. Infine, questi pattern rendono il codice più flessibile e facilitano la gestione delle istanze, migliorando la robustezza e la modularità dell'intero progetto.

Class Diagram: Main



Il package Main rappresenta il punto di ingresso del programma; esso, contiene la classe Main, che include il metodo main(String[] args) necessario per avviare l'esecuzione del programma. La classe Main è responsabile dell'inizializzazione dei componenti principali del sistema, come i controller e le viste, ma anche dell'impostazione delle interazioni iniziali con l'utente; la struttura di questa classe

riflette l'organizzazione complessiva del programma e mostra come i vari componenti collaborano tra loro per fornire la funzionalità desiderata.



1. Classe Main

- Descrizione: La classe Main è l'unica classe presente nel package Main; la sua funzione è quella di avviare l'applicazione, configurare i controller e le viste, gestire il ciclo di vita dell'interazione utente-programma.
- Attributi: La classe Main non definisce attributi specifici, ma all'interno del metodo main vengono creati e gestiti vari oggetti necessari per l'esecuzione del programma.
- Metodi:

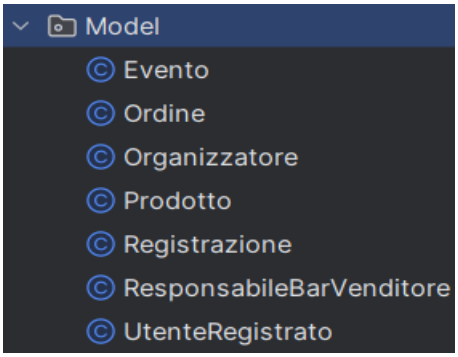
o public static void main(String[] args): Questo è il metodo principale del programma, che viene eseguito all'avvio; al suo interno, vengono create le istanze di vari controller (RegistrazioneController, EventoController, ProdottoController) e delle relative viste (VistaRegistrazione, VistaEvento, VistaProdotto). Inoltre, viene configurato il pattern Observer, aggiungendo le viste come osservatori dei rispettivi controller; il metodo contiene anche un loop che gestisce l'interazione con l'utente, permettendogli di registrarsi o di accedere al sistema, consendogli di decidere quando terminare il programma.

Pattern Utilizzati:

- Singleton: Sebbene non sia implementato direttamente nella classe Main, essa utilizza il pattern Singleton per ottenere istanze uniche di alcune classi, come VistaEvento, VistaProdotto, EventoController e ProdottoController; questo, garantisce che ci sia un'unica istanza di queste classi durante l'intera esecuzione del programma.
- Observer: La classe Main configura il pattern Observer collegando le viste ai controller come osservatori; questo, consente alle viste di aggiornarsi automaticamente quando si verificano cambiamenti nei rispettivi controller.

Il package Main, nonostante contenga una sola classe, svolge un ruolo cruciale nel sistema, poiché svolge un ruolo da orchestratore dell'avvio e dell'inizializzazione dell'applicazione; la classe Main, non solo imposta l'ambiente di esecuzione configurando le interazioni tra le diverse componenti, ma utilizza anche vari pattern di design, come Singleton e Observer, per garantire che il programma funzioni in modo corretto ed efficiente. Questi, assicurano che vi sia coerenza nello stato del programma e che le modifiche nei dati siano riflettute in maniera rapida nelle interfacce utente, rendendo il sistema robusto e reattivo alle azioni degli utenti.

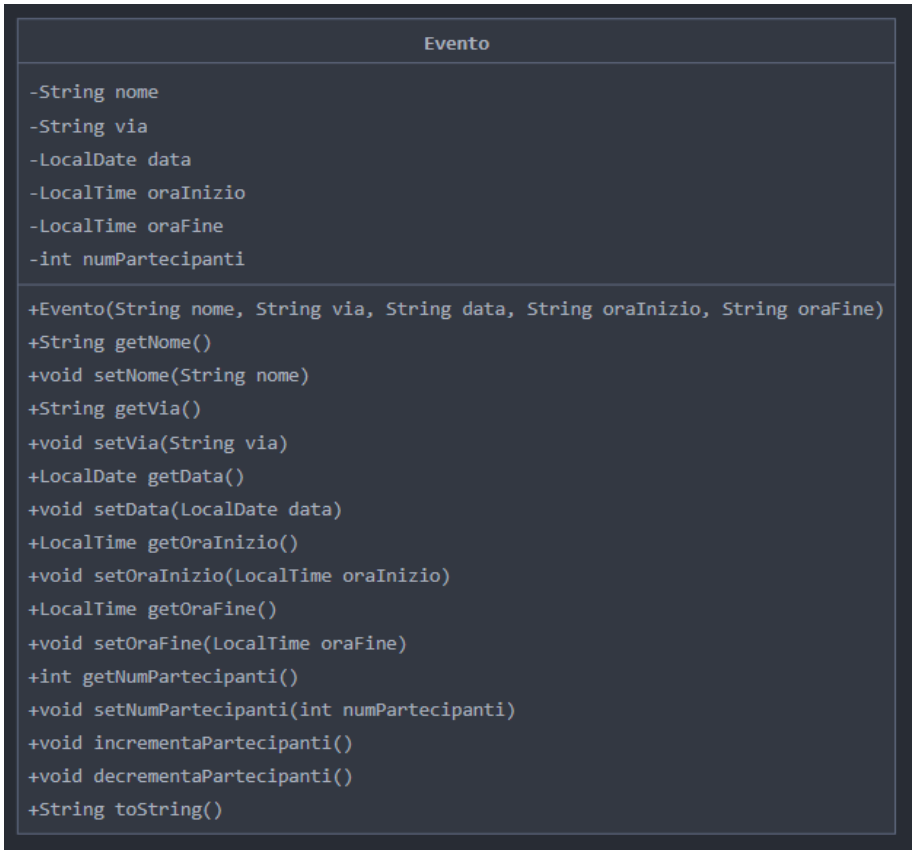
Class Diagram: Model



Il package Model è il cuore del progetto, infatti, rappresenta il livello di gestione dei dati e implementa i vari Model per il pattern MVC; in questo progetto, esso contiene tutte le classi principali con i loro

attributi e metodi principali. Esse, rappresentano i dati e la logica del software e sono: Evento, Ordine, Prodotto, Registrazione, UtenteRegistrato, ResponsabileBarVenditore e Organizzatore; queste, definiscono appunto gli attributi e i metodi principali che le classi utilizzano per fare le operazioni principali, come le operazioni CRUD, ma anche le strutture dati, la logica applicativa del software, le entità dell'applicazione e la comunicazione con i vari Controller per gli aggiornamenti e le richieste di dati.

1. Evento



- Descrizione:

La classe Evento rappresenta un evento con tutte le sue proprietà principali, come il nome, il luogo, la data, l'orario di inizio e di fine, e il numero di partecipanti; è progettata per gestire le informazioni relative a un evento e consente di incrementare o decrementare il numero di partecipanti in modo controllato.

- Attributi:

o String nome: Il nome dell'evento.
o String via: La via dove si svolge l'evento.
o LocalDate data: La data dell'evento.
o LocalTime oraInizio: L'ora di inizio dell'evento.
o LocalTime oraFine: L'ora di fine dell'evento.
o int numPartecipanti: Il numero di partecipanti all'evento.

- Metodi:

o Evento(String nome, String via, String data, String oraInizio, String oraFine): Costruttore per inizializzare un nuovo evento.
o String getNome(): Restituisce il nome dell'evento.
o void setNome(String nome): Imposta il nome dell'evento.
o String getVia(): Restituisce la via dell'evento.
o void setVia(String via): Imposta la via dell'evento.
o LocalDate getData(): Restituisce la data dell'evento.
o void setData(LocalDate data): Imposta la data dell'evento.

o LocalTime getOraInizio(): Restituisce l'ora di inizio dell'evento.
o void setOraInizio(LocalTime oraInizio): Imposta l'ora di inizio dell'evento.
o LocalTime getOraFine(): Restituisce l'ora di fine dell'evento.
o void setOraFine(LocalTime oraFine): Imposta l'ora di fine dell'evento.
o int getNumPartecipanti(): Restituisce il numero di partecipanti all'evento.
o void setNumPartecipanti(int numPartecipanti): Imposta il numero di partecipanti all'evento.
o void incrementaPartecipanti(): Incrementa il numero di partecipanti di uno.
o void decrementaPartecipanti(): Decrementa il numero di partecipanti di uno se il numero è maggiore di zero.
o String toString(): Fornisce una rappresentazione stringa dell'evento.

- Pattern Utilizzati:

o Nessun pattern è stato utilizzato in questa classe.

2. Ordine

- Descrizione:

La classe Ordine rappresenta un ordine di acquisto da parte di un utente, con una lista di prodotti, quantità, lo stato “portare via” e il costo totale; è progettata per gestire e tracciare lo stato degli ordini, inclusi i prodotti ritirati e gestiti.

- Attributi:

o UtenteRegistrato utente: L'utente che ha effettuato l'ordine.
o List<Prodotto> prodotti: La lista dei prodotti nell'ordine.
o List<Integer> quantita: La lista delle quantità per ciascun prodotto.
o List<Boolean> portareVia: Indica se i prodotti sono da portare via.
o List<Boolean> prodottiGestiti: Stato di gestione dei prodotti.
o List<Boolean> prodottiRitirati: Stato di ritiro dei prodotti.
o double costoTotale: Il costo totale dell'ordine.
o boolean completato: Indica se l'ordine è stato completato.
o boolean prodottiPortareViaCompletati: Stato di completamento per i prodotti da portare via.

o boolean prodottiNonPortareViaCompletati: Stato di completamento per i prodotti non da portare via.

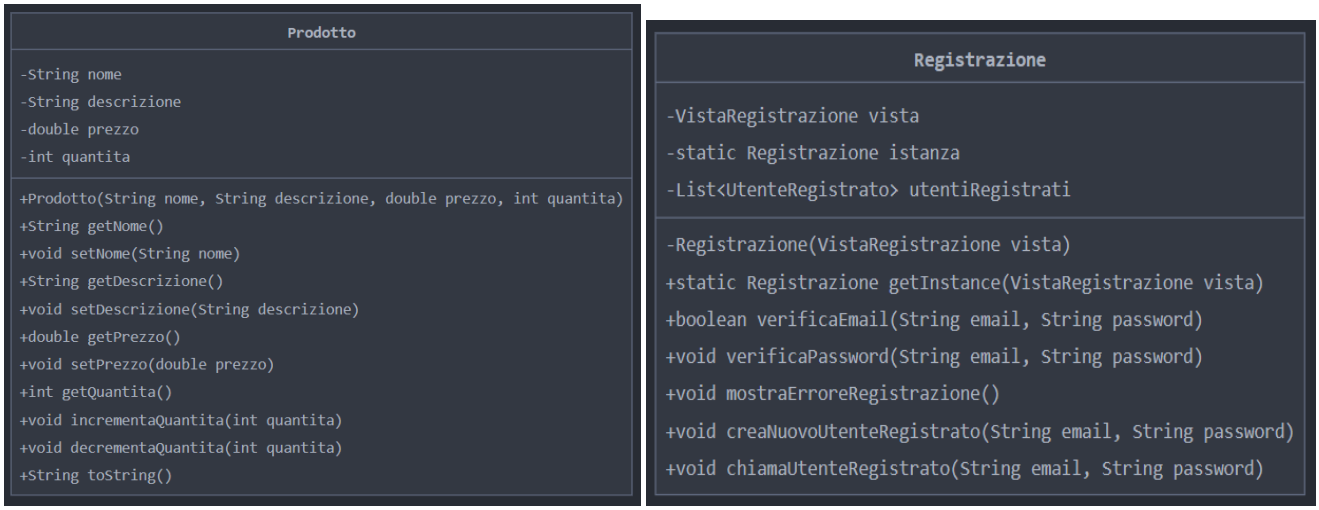
- Metodi:

o Ordine(UtenteRegistrato utente, List<Prodotto> prodotti, List<Integer> quantita, List<Boolean> portareVia, double costoTotale): Costruttore per creare un nuovo ordine.
o UtenteRegistrato getUtente(): Restituisce l'utente che ha effettuato l'ordine.
o List<Prodotto> getProdotti(): Restituisce la lista dei prodotti nell'ordine.
o List<Integer> getQuantita(): Restituisce la lista delle quantità dei prodotti.
o List<Boolean> getPortareVia(): Restituisce la lista delle opzioni di portare via.
o List<Boolean> getProdottiGestiti(): Restituisce la lista degli stati di gestione dei prodotti.
o void setProdottiGestiti(List<Boolean> prodottiGestiti): Imposta la lista degli stati di gestione dei prodotti.
o List<Boolean> getProdottiRitirati(): Restituisce la lista degli stati di ritiro dei prodotti.
o void setProdottiRitirati(List<Boolean> prodottiRitirati): Imposta la lista degli stati di ritiro dei prodotti.
o double getCostoTotale(): Restituisce il costo totale dell'ordine.
o boolean getCompletato(): Restituisce lo stato di completamento dell'ordine.
o void setCompletato(boolean completato): Imposta lo stato di completamento dell'ordine.
o boolean getProdottiPortareViaCompletati(): Restituisce lo stato di completamento per i prodotti da portare via.
o void setProdottiPortareViaCompletati(boolean prodottiPortareViaCompletati): Imposta lo stato di completamento per i prodotti da portare via.
o boolean getProdottiNonPortareViaCompletati(): Restituisce lo stato di completamento per i prodotti non da portare via.
o void setProdottiNonPortareViaCompletati(boolean prodottiNonPortareViaCompletati): Imposta lo stato di completamento per i prodotti non da portare via.
o String toString(): Fornisce una rappresentazione stringa dell'ordine.

- Pattern Utilizzati:

o Nessun pattern è stato utilizzato in questa classe.

3. Prodotto (+ Diagram Registrazione)



- Descrizione:

La classe Prodotto rappresenta un prodotto disponibile per l'acquisto con attributi quali nome, descrizione, prezzo e quantità; è progettata per gestire le informazioni sui prodotti e consentire la gestione delle quantità disponibili.

- Attributi:

o String nome: Il nome del prodotto.

o String descrizione: La descrizione del prodotto.
o double prezzo: Il prezzo del prodotto.
o int quantita: La quantità disponibile del prodotto.

- Metodi:

o Prodotto(String nome, String descrizione, double prezzo, int quantita): Costruttore per creare un nuovo prodotto.
o String getNome(): Restituisce il nome del prodotto.
o void setNome(String nome): Imposta il nome del prodotto.
o String getDescrizione(): Restituisce la descrizione del prodotto.
o void setDescrizione(String descrizione): Imposta la descrizione del prodotto.
o double getPrezzo(): Restituisce il prezzo del prodotto.
o void setPrezzo(double prezzo): Imposta il prezzo del prodotto.
o int getQuantita(): Restituisce la quantità disponibile del prodotto.
o void incrementaQuantita(int quantita): Incrementa la quantità disponibile del prodotto.
o void decrementaQuantita(int quantita): Decrementa la quantità disponibile del prodotto, se sufficiente.
o String toString(): Fornisce una rappresentazione stringa del prodotto.

- Pattern Utilizzati:

o Nessun pattern è stato utilizzato in questa classe.

4. Registrazione

- Descrizione:

La classe Registrazione si occupa della gestione della registrazione e degli utenti registrati all'interno del sistema; implementa il pattern Singleton per garantire che esista solo un'istanza della classe in tutta l'applicazione.

- Attributi:

o VistaRegistrazione vista: Attributo per accettare un parametro di tipo VistaRegistrazione.
o static Registrazione istanza: Attributo statico che contiene l'unica istanza di Registrazione.
o List<UtenteRegistrato> utentiRegistrati: Lista degli utenti registrati.

- Metodi:

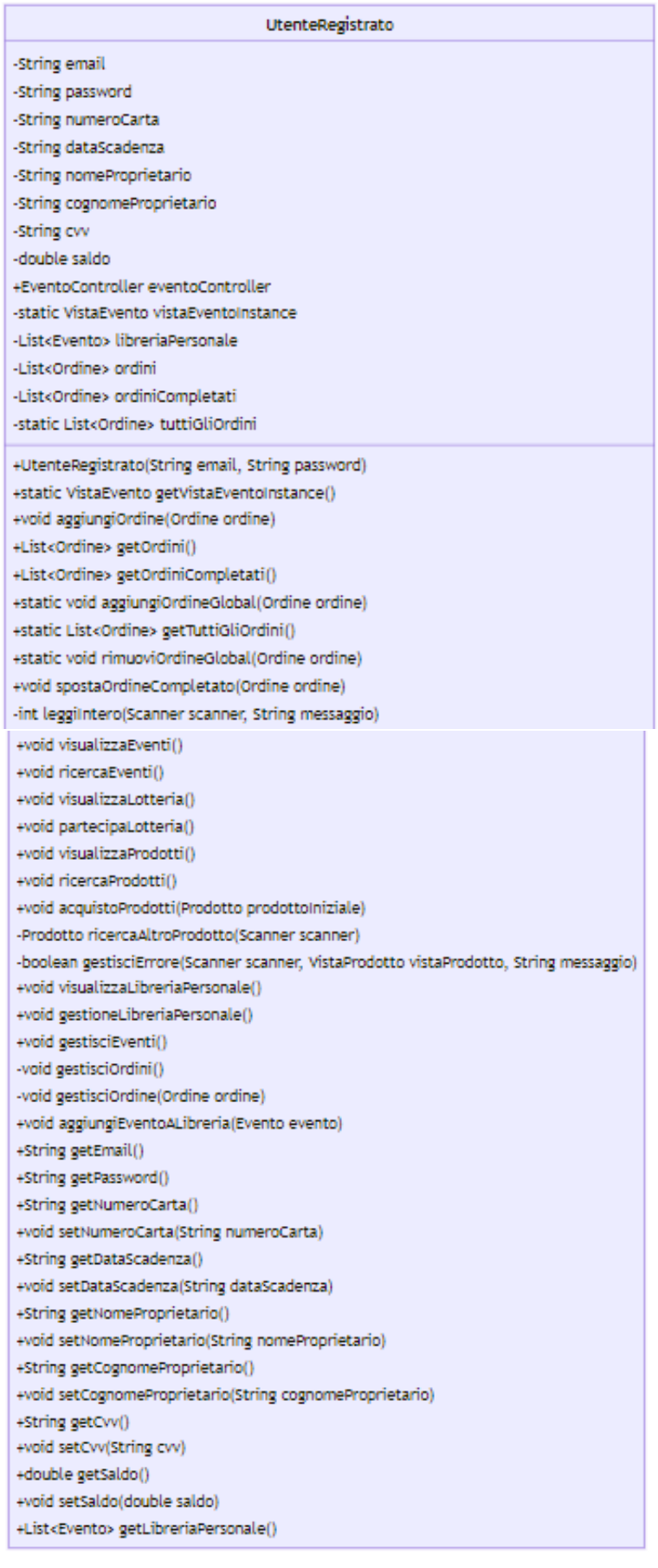
o static Registrazione getInstance(VistaRegistrazione vista): Restituisce l'istanza della classe Registrazione, creandola se non esiste.
o boolean verificaEmail(String email, String password): Verifica se l'email è già registrata.
o void verificaPassword(String email, String password): Verifica se l'email e la password corrispondono a un utente registrato.
o void mostraErroreRegistrazione(): Mostra un messaggio di errore in caso di dati di registrazione incorretti.
o void creaNuovoUtenteRegistrato(String email, String password): Crea un nuovo utente registrato e, se richiesto, aggiunge i dettagli della carta di credito.
o void chiamaUtenteRegistrato(String email, String password): Gestisce le operazioni disponibili per un utente autenticato, come visualizzare eventi, gestire ordini, eccetera.

- Pattern Utilizzati:

o Singleton: La classe Registrazione utilizza il pattern Singleton per garantire la presenza di solo un'istanza dell'oggetto Registrazione durante l'intera esecuzione dell'applicazione; questo, è utile per centralizzare la gestione degli utenti registrati e delle operazioni di registrazione e login.
L'implementazione del pattern è questa:
o private static Registrazione istanza: L'istanza statica della classe.
o private Registrazione(VistaRegistrazione vista): Costruttore privato per evitare la creazione di istanze multiple.

o public static Registrazione getInstance(VistaRegistrazione vista): Metodo statico per ottenere l'unica istanza della classe.

5. UtenteRegistrato



- Descrizione:

La classe UtenteRegistrato rappresenta un utente registrato nel sistema, che ha la capacità di visualizzare e partecipare a eventi, gestire ordini, interagire con altri elementi dell'applicazione come i prodotti e gestire una personale libreria; di seguito, viene fornita una descrizione degli attributi e dei metodi principali di questa classe:

- Attributi:

o email: Stringa che rappresenta l'indirizzo e-mail dell'utente; viene utilizzato come identificatore univoco dell'utente.
o password: Stringa che rappresenta la password dell'utente; viene utilizzata per l'autenticazione.
o numeroCarta: Stringa che memorizza il numero della carta di credito dell'utente per i pagamenti.
o dataScadenza: Stringa che rappresenta la data di scadenza della carta di credito.
o nomeProprietario: Stringa che contiene il nome del proprietario della carta di credito.
o cognomeProprietario: Stringa che contiene il cognome del proprietario della carta di credito.
o cvv: Stringa che memorizza il codice CVV della carta di credito.
o saldo: Variabile di tipo double che rappresenta il saldo disponibile dell'utente.
o eventoController: Istanza di EventoController utilizzata per gestire e recuperare gli eventi.
o vistaEventoInstance: Istanza statica di VistaEvento, utilizzata per aggiornare la vista associata agli eventi.
o libreriaPersonale: Lista di oggetti Evento, che rappresenta la libreria personale degli eventi a cui l'utente partecipa.
o ordini: Lista di oggetti Ordine che rappresenta gli ordini correnti dell'utente.
o ordiniCompletati: Lista di oggetti Ordine che rappresenta gli ordini completati dall'utente.
o tuttiGliOrdini: Lista statica di Ordine, condivisa tra tutte le istanze di UtenteRegistrato, che contiene tutti gli ordini effettuati nel sistema.

- Metodi:

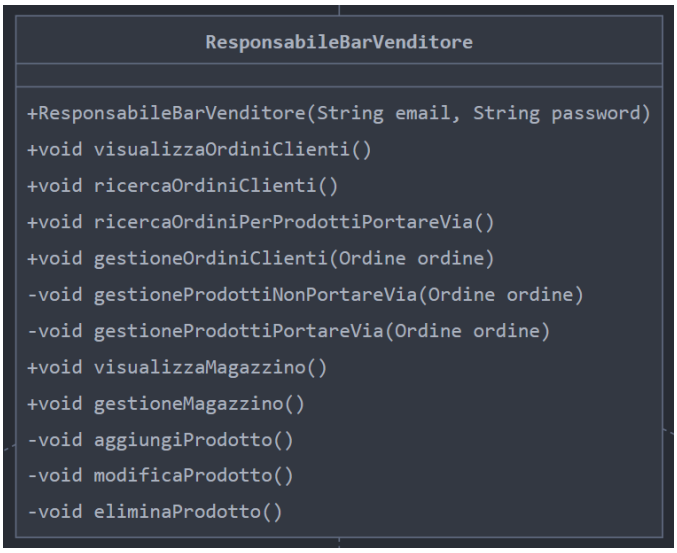
o UtenteRegistrato(String email, String password): Costruttore che inizializza l'utente con e-mail e password, crea le liste di ordini, la libreria personale, e associa il controller degli eventi.
o static VistaEvento getVistaEventoInstance(): Restituisce l'istanza della vista degli eventi.
o void aggiungiOrdine(Ordine ordine): Aggiunge un ordine alla lista degli ordini dell'utente.
o List<Ordine> getOrdini(): Restituisce la lista degli ordini correnti dell'utente.
o List<Ordine> getOrdiniCompletati(): Restituisce la lista degli ordini completati dall'utente.
o static void aggiungiOrdineGlobal(Ordine ordine): Aggiunge un ordine alla lista globale di tutti gli ordini.
o static List<Ordine> getTuttiGliOrdini(): Restituisce la lista globale di tutti gli ordini.
o static void rimuoviOrdineGlobal(Ordine ordine): Rimuove un ordine dalla lista globale di tutti gli ordini.
o void spostaOrdineCompletato(Ordine ordine): Sposta un ordine dalla lista degli ordini correnti a quella degli ordini completati.
o void visualizzaEventi(): Mostra all'utente la lista degli eventi disponibili, permettendo di partecipare o cercare un evento specifico.
o void ricercaEventi(): Permette all'utente di cercare un evento per nome e di partecipare a esso se desiderato.
o void visualizzaProdotti(): Mostra la lista dei prodotti disponibili e permette di cercare un prodotto specifico.
o void ricercaProdotti(): Permette all'utente di cercare un prodotto per nome e di acquistarlo se disponibile.
o void visualizzaLotteria() e void partecipaLotteria(): Metodi placeholder per la gestione della visualizzazione e partecipazione alla lotteria.
o void acquistoProdotti(Prodotto prodottoIniziale): Metodo che gestisce l'intero processo di acquisto, verificando se l'utente ha inserito le informazioni della carta di credito, consentendo la selezione e la

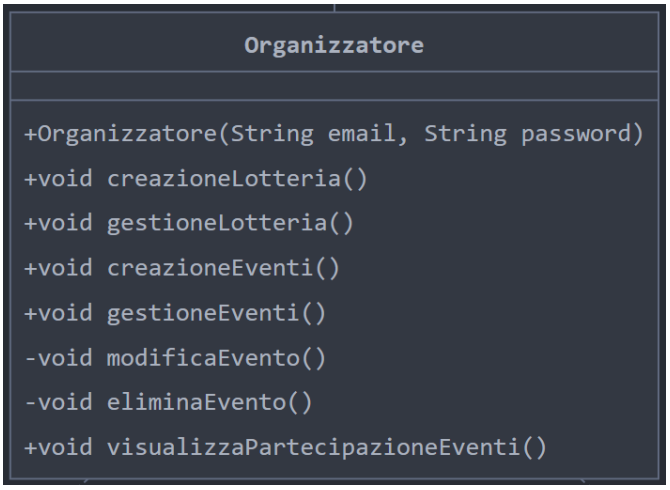
modifica di prodotti, quantità e stato “portare via”, anche aggiornandoli a scelta e gestendo errori e conferme finali prima di completare l'acquisto o annullarlo.
o ricercaAltroProdotto(Scanner scanner): Metodo che consente di cercare un prodotto specifico per nome interno ad acquistoProdotti; utilizza Singleton per VistaProdotto e ProdottoController. Il metodo consente una ricerca iterativa finché l'utente non trova un prodotto disponibile oppure decide di interrompere la ricerca.
o gestisciErrore(Scanner scanner, VistaProdotto vistaProdotto, String messaggio): Gestisce le situazioni in cui un prodotto non è disponibile, permettendo all'utente di decidere se continuare la ricerca o annullare l'acquisto.
o visualizzaLibreriaPersonale(): Visualizzazione di informazioni relative al proprio account, la libreria personale di eventi e lo stato degli ordini, fornendo un riepilogo di tutti gli eventi e gli ordini associati all'utente.
o gestioneLibreriaPersonale(): Interfaccia per gestire la libreria personale dell'utente, consentendo di scegliere se gestire eventi o ordini.
o gestisciEventi(): Gestione di eventi presenti nella libreria personale; è possibile selezionare un evento specifico per eliminarlo, aggiornando di conseguenza il numero di partecipanti all'evento.
o gestisciOrdini(): Gestione degli ordini in corso dell'utente, permettendo di visualizzare e ritirare i prodotti pronti; i prodotti già gestiti e pronti per il ritiro vengono mostrati all'utente, il quale può decidere se ritirarli o lasciarli in sospeso.
o gestisciOrdine(Ordine ordine): Gestione del ritiro dei prodotti all'interno di un ordine specifico; è possibile confermare il ritiro dei prodotti pronti, con il sistema che aggiorna lo stato dell'ordine.
o Metodi Getter e Setter per i vari attributi della classe, come getEmail(), getPassword(), getNumeroCarta(), setNumeroCarta(String numeroCarta), ecc.

- Pattern Utilizzati:

o Singleton: Utilizzato per garantire che solo un'istanza delle classi VistaProdotto e ProdottoController sia attiva durante l'esecuzione.
o Observer: Utilizzato nella gestione degli eventi per notificare cambiamenti, come il numero di partecipanti.
o Iterator: Utilizzato per scorrere collezioni di eventi, ordini e prodotti, facilitando la gestione iterativa.
o Facade: Fornisce un'interfaccia semplice per la gestione delle operazioni dell'utente, nascondendo la complessità sottostante.
o State: Gestisce lo stato degli ordini e dei prodotti, aggiornandoli dinamicamente in base alle azioni dell'utente.
o Strategy: Viene applicato nella gestione delle scelte dell'utente in acquistoProdotti, come l'inserimento di dati, con differenti strategie per validare input e aggiornare lo stato dell'acquisto.
o Factory Method: Viene usato per ottenere istanze di VistaEvento e VistaProdotto attraverso i metodi statici getInstance(), assicurando un'interfaccia centralizzata per la creazione e gestione di queste visualizzazioni.

6. ResponsabileBarVenditore (+ Diagram Organizzatore)





- Descrizione:

La classe ResponsabileBarVenditore rappresenta un'entità che può gestire sia gli ordini dei clienti sia i prodotti disponibili per la vendita; questa, eredita dalla classe UtenteRegistrato, e quindi ne condivide attributi e metodi, aggiungendo funzionalità specifiche per il ruolo di responsabile bar o venditore. Essa collabora con:

o UtenteRegistrato: Come già detto, la classe ResponsabileBarVenditore estende UtenteRegistrato, ereditando quindi anche i metodi di un normale utente.

o Registrazione: Si interfaccia con la classe Registrazione per aggiungere nuovi responsabili e per la gestione dei login.

- Attributi:

o Nessuno specifico: La classe ResponsabileBarVenditore non introduce nuovi attributi rispetto a quelli ereditati da UtenteRegistrato.

- Metodi:

o visualizzaOrdiniClienti(): void: Permette al responsabile di visualizzare tutti gli ordini effettuati dai clienti.

o ricercaOrdiniClienti(): void: Consente di cercare specifici ordini effettuati dai clienti in base a determinati criteri.

o ricercaOrdiniPerProdottiPortareVia(): void: Gestisce gli ordini per i prodotti da portare via, consentendo al responsabile di verificare e processare tali ordini.

o visualizzaMagazzino(): void: Mostra lo stato attuale del magazzino, inclusi i prodotti disponibili.

o gestioneMagazzino(): void: Permette di gestire il magazzino, aggiungendo o rimuovendo prodotti e aggiornando le loro caratteristiche.

- Pattern Utilizzati:

o La classe ResponsabileBarVenditore, come descritto, eredita da UtenteRegistrato. Questo sfrutta il pattern di Ereditarietà per riutilizzare e specializzare comportamenti definiti nella classe base.

o Singleton: La classe Registrazione all'interno del package utilizza il pattern Singleton per assicurare che ci sia una sola istanza di questa classe nell'applicazione. Il pattern Singleton è fondamentale per gestire un'unica registrazione centralizzata di tutti gli utenti, inclusi i responsabili bar e venditori.

7. Organizzatore

- Descrizione:

La classe Organizzatore rappresenta un utente che ha il ruolo di organizzatore all'interno dell'applicazione; estende la classe ResponsabileBarVenditore, ereditando così tutte le proprietà e i metodi dalle classi UtenteRegistrato e ResponsabileBarVenditore. Essa collabora con:

o ResponsabileBarVenditore: La classe Organizzatore, come detto, estende ResponsabileBarVenditire, ereditando quindi i metodi e gli attributi da essa.
o VistaEvento: Si interfaccia con la classe VistaEvento per tenerla aggiornata.

- Attributi:

o Nessuno specifico: La classe Organizzatore non introduce nuovi attributi rispetto a quelli ereditati da ResponsabileBarVenditore e UtenteRegistrato.

- Metodi:

o Organizzatore(String email, String password): Costruttore che inizializza un nuovo oggetto Organizzatore con l'e-mail e la password fornite.
o creazioneLotteria() e gestioneLotteria(): Metodi specifici per creare una nuova lotteria e gestire le lotterie create. L'implementazione è attualmente da completare.
o creazioneEventi(): Creazione di nuovi eventi, in cui viene utilizzata un'istanza della classe VistaEvento per aggiornare la vista e fornire feedback all'utente; durante il processo, l'organizzatore fornisce dettagli come il nome, la via, la data e l'orario di inizio e di fine dell'evento.
o gestioneEventi(): Gestione degli eventi già creati; il metodo offre opzioni per modificare o eliminare un evento esistente e permette di tornare al menu principale:

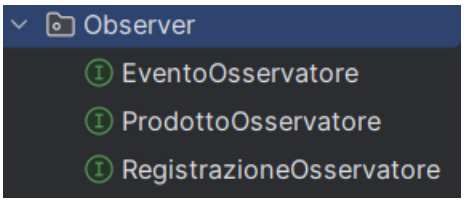
- modificaEvento(): Metodo interno a gestioneEventi() che gestisce la modifica dei dettagli di un evento specifico, come il nome, la via, la data, l'ora di inizio e l'ora di fine.
- eliminaEvento(): Metodo interno a gestioneEventi(), utilizzato per eliminare un evento specifico dopo aver ricevuto conferma dall'organizzatore.

o visualizzaPartecipazioneEventi(): Visualizzazione delle partecipazioni agli eventi creati; mostra la lista degli eventi e permette di selezionare un evento specifico per visualizzare il numero di partecipanti.

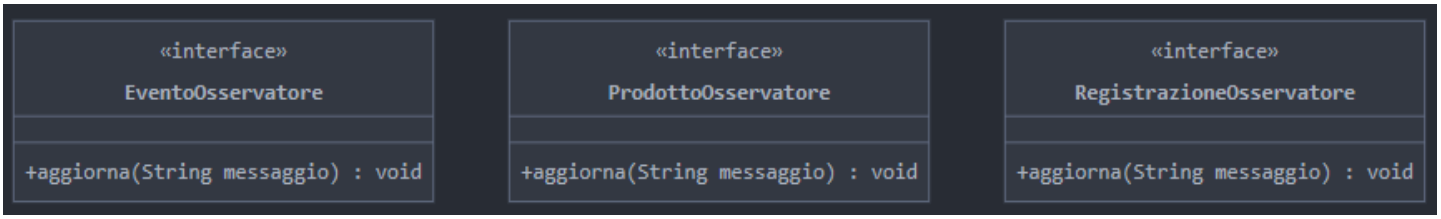
- Pattern Utilizzati:

o La classe Organizzatore, come descritto, eredita da ResponsabileBarVenditore, sfruttando il pattern di Ereditarietà per riutilizzare e specializzare comportamenti definiti nelle classi base.
o Organizzatore utilizza il pattern di progettazione Template Method per definire una sequenza di operazioni comuni per la gestione degli eventi e delle lotterie, permettendo alle sottoclassi di definire implementazioni specifiche di certi passaggi; infatti, questo pattern è applicato nei metodi che richiedono una sequenza fissa di operazioni, con la possibilità di estendere o modificare i singoli passi in sottoclassi specifiche. Viene quindi usato, per esempio, nei metodi gestioneEventi() e creazioneEventi() per fornire un flusso chiaro delle operazioni, lasciando dettagli specifici a metodi interni come modificaEvento() ed eliminaEvento().

Class Diagram: Observer



Il package Observer è progettato per implementare il pattern Observer, un pattern di design comportamentale che permette a un oggetto (il soggetto) di notificare automaticamente a una lista di osservatori (oggetti dipendenti) ogni volta che si verifica un cambiamento di stato; in questo progetto, il package Observer contiene tre interfacce principali: EventoOsservatore, ProdottoOsservatore, e RegistrazioneOsservatore. Queste, definiscono il contratto che gli osservatori devono seguire per ricevere notifiche dagli oggetti osservati, come eventi, prodotti e registrazioni.



1. Interfaccia EventoOsservatore

- Descrizione: Questa interfaccia definisce il comportamento che ogni osservatore di eventi deve implementare; gli oggetti che implementano EventoOsservatore possono registrarsi presso un soggetto (come EventoController) per essere notificati quando si verificano cambiamenti rilevanti negli eventi.
- Metodi:

o void aggiorna(String messaggio): Questo metodo viene chiamato dal soggetto per inviare notifiche agli osservatori; il parametro messaggio consente di passare informazioni rilevanti agli osservatori, che possono aggiornare la loro vista o eseguire altre azioni in risposta alla notifica.

2. Interfaccia ProdottoOsservatore

- Descrizione: L'interfaccia ProdottoOsservatore segue lo stesso concetto di EventoOsservatore, ma è specificamente progettata per prodotti; gli osservatori dei prodotti, implementando questa interfaccia, possono ricevere notifiche sui cambiamenti relativi ai prodotti, come l'aggiornamento della quantità o del prezzo.
- Metodi:

o void aggiorna(String messaggio): Simile a EventoOsservatore, questo metodo viene invocato dal soggetto, in questo caso ProdottoController, per notificare gli osservatori sui cambiamenti o aggiornamenti relativi ai prodotti.

3. Interfaccia RegistrazioneOsservatore

- Descrizione: L'interfaccia RegistrazioneOsservatore è progettata per gestire le notifiche relative alla registrazione degli utenti; gli osservatori che implementano questa interfaccia possono essere informati di cambiamenti significativi o aggiornamenti nei processi di registrazione.
- Metodi:

o void aggiorna(String messaggio): Questo metodo permette ai soggetti come RegistrazioneController di notificare agli osservatori eventuali cambiamenti o azioni compiute durante il processo di registrazione, come la registrazione di un nuovo utente.

Pattern Utilizzati

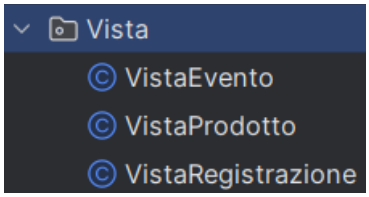
- Observer: Il package Observer è un'implementazione diretta del pattern Observer; questo, viene utilizzato per stabilire una relazione uno-a-molti tra un soggetto e i suoi osservatori, permettendo ai soggetti di notificare automaticamente agli osservatori ogni volta che si verifica un cambiamento di stato. Le interfacce EventoOsservatore, ProdottoOsservatore e RegistrazioneOsservatore definiscono il contratto che gli osservatori devono seguire per ricevere queste notifiche.

Questo pattern è estremamente utile in un contesto dove più componenti (come le viste) devono rimanere sincronizzati con lo stato di uno o più oggetti (come modelli o controller) senza che questi ultimi abbiano una dipendenza rigida dalle viste stesse; ciò, contribuisce a mantenere un'architettura modulare e flessibile, dove le componenti possono essere modificate, aggiunte o rimosse senza dover cambiare il comportamento del soggetto o degli altri osservatori.

Concludendo, il package Observer svolge un ruolo fondamentale nell'architettura del progetto, fornendo un meccanismo flessibile e modulare per gestire le notifiche tra i vari componenti del sistema; le interfacce EventoOsservatore, ProdottoOsservatore e RegistrazioneOsservatore

permettono di implementare facilmente il pattern Observer all'interno del sistema, garantendo che le viste siano sempre aggiornate in modo coerente rispetto alle modifiche che avvengono nei controller. Questo approccio migliora l'estensibilità e la manutenibilità del codice, rendendo il sistema più robusto e adattabile a future evoluzioni.

Class Diagram: Vista



Il package Vista è dedicato alla gestione dell'interfaccia utente e all'aggiornamento delle visualizzazioni in risposta ai cambiamenti nei dati dell'applicazione; esso contiene le classi VistaEvento, VistaProdotto, e VistaRegistrazione, ciascuna delle quali implementa specifiche interfacce di osservazione, permettendo loro di essere notificate dai rispettivi controller quando avvengono cambiamenti. In questo modo, il package Vista aderisce al pattern Observer, che permette di mantenere la coerenza tra il modello dati e l'interfaccia utente.

1. Classe VistaEvento

- Descrizione: La classe VistaEvento gestisce la visualizzazione degli eventi all'interno dell'applicazione; implementa l'interfaccia EventoOsservatore, che le permette di essere notificata dei cambiamenti relativi agli eventi, come ad esempio l'aggiunta o la modifica di un evento.
- Metodi:

o private VistaEvento(): Primo pezzo del pattern Singleton; funge da costruttore privato, utilizzato per evitare l'instanziazione diretta della classe.
o public static VistaEvento getInstance(): Secondo pezzo del pattern Singleton; si tratta di un metodo statico che restituisce l'unica istanza di VistaEvento, creando l'istanza se non è già stata creata.
o @Override public void aggiorna(String messaggio): Metodo che viene chiamato quando la vista deve essere aggiornata; stampa un messaggio che indica l'aggiornamento della vista.

- **Pattern Utilizzati**:

o Singleton: La classe VistaEvento utilizza il pattern Singleton per garantire che esista una sola istanza della vista degli eventi in tutto il sistema; questo è gestito attraverso un costruttore privato e un metodo statico getInstance(), che crea e restituisce l'istanza univoca. Tutto ciò aiuta a mantenere uno stato coerente dell'interfaccia utente e riduce il rischio di incongruenze o conflitti tra istanze multiple.

o Observer: Questo è il pattern centrale utilizzato nel package Vista; implementa l'interfaccia EventoOsservatore, così che la classe possa ricevere notifiche dal controller degli eventi e aggiornare di conseguenza in modo coerente la visualizzazione tramite il metodo aggiorna(String messaggio), che stampa un messaggio per indicare che la vista è stata aggiornata.

2. Classe VistaProdotto

- Descrizione: VistaProdotto gestisce la visualizzazione dei prodotti e le interazioni con essi; come VistaEvento, anche questa classe implementa il pattern Observer, consentendo di essere notificata dal controller dei prodotti riguardo a cambiamenti rilevanti.
- Metodi:

o private VistaProdotto(): Primo pezzo del pattern Singleton; funge da costruttore privato, utilizzato per prevenire la creazione diretta di istanze multiple della classe.
o public static VistaProdotto getInstance(): Secondo pezzo del pattern Singleton; si tratta di un metodo statico che restituisce l'unica istanza di VistaProdotto, creandola se necessario.
o public static void setTornaAlMenu(boolean torna): Metodo statico utilizzato per impostare il flag tornaAlMenu, che indica se l'utente deve tornare al menu principale dopo un'operazione di acquisto.
o public static boolean getTornaAlMenu(): Metodo statico che restituisce il valore del flag tornaAlMenu.
o @Override public void aggiorna(String messaggio): Metodo che viene chiamato per aggiornare la vista dei prodotti, stampando un messaggio che indica che la vista è stata aggiornata.

- **Pattern Utilizzati**:

o Singleton: Anche VistaProdotto utilizza il pattern Singleton per assicurare un'unica istanza della vista dei prodotti, con un costruttore privato e il metodo getInstance() per gestire l'accesso all'istanza; questo, aiuta a mantenere uno stato coerente dell'interfaccia utente e riduce il rischio di incongruenze o conflitti tra istanze multiple.
o Observer: Questo è il pattern centrale utilizzato nel package Vista; implementa l'interfaccia ProdottoOsservatore, permettendo alla classe di ricevere notifiche dal controller dei prodotti e di aggiornare in modo coerente la visualizzazione dei prodotti tramite il metodo aggiorna(String messaggio).
o State Management: Oltre a Singleton e Observer, VistaProdotto include un semplice meccanismo di gestione dello stato attraverso i metodi statici setTornaAlMenu(boolean torna) e getTornaAlMenu(), che sono utilizzati per determinare quando la funzione di acquisto prodotti è terminata, indicando se tornare al menu principale.

3. Classe VistaRegistrazione

- Descrizione: VistaRegistrazione è responsabile della visualizzazione e gestione delle operazioni legate alla registrazione degli utenti; essa implementa l'interfaccia RegistrazioneOsservatore che serve per ricevere aggiornamenti relativi al processo di registrazione.
- Metodi:

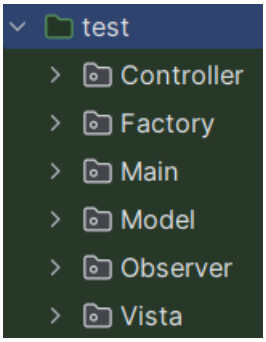
o @Override public void aggiorna(String messaggio): Metodo che viene chiamato quando la vista deve essere aggiornata a seguito di un cambiamento relativo alla registrazione. Questo metodo stampa un messaggio per indicare che la vista è stata aggiornata.

- **Pattern Utilizzati**:

o Observer: Questo è il pattern centrale utilizzato nel package Vista; implementa l'interfaccia RegistrazioneOsservatore, così che questa classe sia in grado di ricevere notifiche dal controller della registrazione e di aggiornare la vista in modo coerente di conseguenza. Il metodo aggiorna(String messaggio) stampa un messaggio per indicare che la vista è stata aggiornata in seguito a una notifica.

Il package Vista ha il compito di separare le responsabilità all'interno dell'architettura dell'applicazione; utilizzando il pattern Observer, le classi in questo package possono ricevere e reagire alle notifiche di cambiamento provenienti dai controller, garantendo che l'interfaccia utente rimanga sincronizzata con il modello di dati sottostante. Il pattern Singleton aggiunge un ulteriore livello di controllo, assicurando che ogni vista abbia una singola istanza, mantenendo così la consistenza dello stato visivo dell'applicazione; questi pattern, combinati fra di loro, permettono una struttura flessibile e facilmente manutenibile, che supporta la modularità e la riusabilità del codice.

Piano di Test: Test d'Unità



Per verificare la corretta funzionalità di ogni modulo del software sono stati implementati numerosi test di unità; questi, mirano a validare il comportamento di singole componenti del sistema in isolamento, garantendo che ciascuna parte funzioni come previsto prima dell'integrazione.
Sono stati eseguiti 149 test, tutti con esito positivo, per una coverage del 43,6%.
I test sono stati implementati usando principalmente un solo framework e librerie specifiche per il testing in ambiente Java:

- JUnit: Framework di testing principale che consente di scrivere ed eseguire test di unità ripetibili e fornisce annotazioni e asserzioni per verificare i risultati attesi; infatti, sono numerose le annotazioni come @Test, tipiche di questo framework, per i metodi di test. JUnit è un framework di testing che, oltre a consentire di scrivere ed eseguire test di unità e di integrazione, si può utilizzare per fare in modo che siano ripetibili per verificare che il codice scritto funzioni come preventivato; infine, fornisce annotazioni e asserzioni ad hoc per testare i risultati attesi.

Metodi e classi tipiche utilizzate per creare e gestire i test:

- @Test: Annotazione JUnit per indicare un metodo di test.
- @BeforeEach: Annotazione JUnit per specificare un metodo da eseguire prima di ogni test.
- assertEquals(): Metodo JUnit per verificare l'uguaglianza di valori.
- assertNotNull(), assertTrue(), assertFalse(): Altri metodi di asserzione JUnit.
- ByteArrayOutputStream e System.setOut(): Usati per catturare l'output della console nei test.

I test di unità forniti coprono un'ampia gamma di funzionalità del sistema; viene dunque riportato di seguito un breve elenco delle funzionalità di ogni classe di test:

- EventoControllerTest: Verifica le operazioni relative agli eventi, come la creazione, la modifica e la ricerca di eventi.
- ProdottoControllerTest: Testa la gestione dei prodotti, incluse le operazioni CRUD e la ricerca.
- RegistrazioneControllerTest: Controlla i processi di registrazione degli utenti.
- ProdottoFactoryTest: Valida la corretta creazione di oggetti Prodotto.
- RegistrazioneFactoryTest: Assicura la corretta istanziazione degli oggetti Registrazione.
- MainTest: Testa il flusso principale dell'applicazione.
- EventoTest, OrdineTest, ProdottoTest, RegistrazioneTest: Verificano il comportamento dei modelli di dati.
- ResponsabileBarVenditoreTest, UtenteRegistratoTest, OrganizzatoreTest: Testano le funzionalità specifiche di questi ruoli utente.
- Test per gli Observer (EventoOsservatoreTest, ProdottoOsservatoreTest, RegistrazioneOsservatoreTest): Controllano il corretto funzionamento del pattern Observer.

- Test per le Viste (VistaEventoTest, VistaProdottoTest, VistaRegistrazioneTest): Verificano la corretta visualizzazione delle informazioni.

I test implementati seguono il Triple A pattern (Arrange-Act-Assert), un approccio strutturato per la scrittura di test di unità:

1. Arrange: Preparazione dell'ambiente di test, inclusa l'inizializzazione degli oggetti necessari e la configurazione delle precondizioni.
2. Act: Esecuzione dell'operazione da testare.
3. Assert: Verifica che il risultato dell'operazione corrisponda all'output atteso.

Il Triple A pattern offre diversi vantaggi:

- Chiarezza: Rende i test più leggibili e facili da comprendere.
- Struttura: Fornisce una struttura coerente per tutti i test, facilitando la manutenzione.
- Isolamento: Aiuta a isolare ciò che viene effettivamente testato, separando la preparazione dall'esecuzione e dalla verifica.
- Documentazione: I test scritti in questo modo fungono anche da documentazione sul comportamento atteso del codice.

Questi test di unità, implementati utilizzando JUnit, forniscono una solida base per garantire la qualità e l'affidabilità del software, permettendo di identificare e correggere rapidamente eventuali regressioni o errori introdotti durante lo sviluppo.

Segue l'immagine della pagina principale del Coverage Report che viene generato, poi anche la prova del successo dei test (L'ordine delle foto è quello della lettura, prima verso destra e poi in basso).

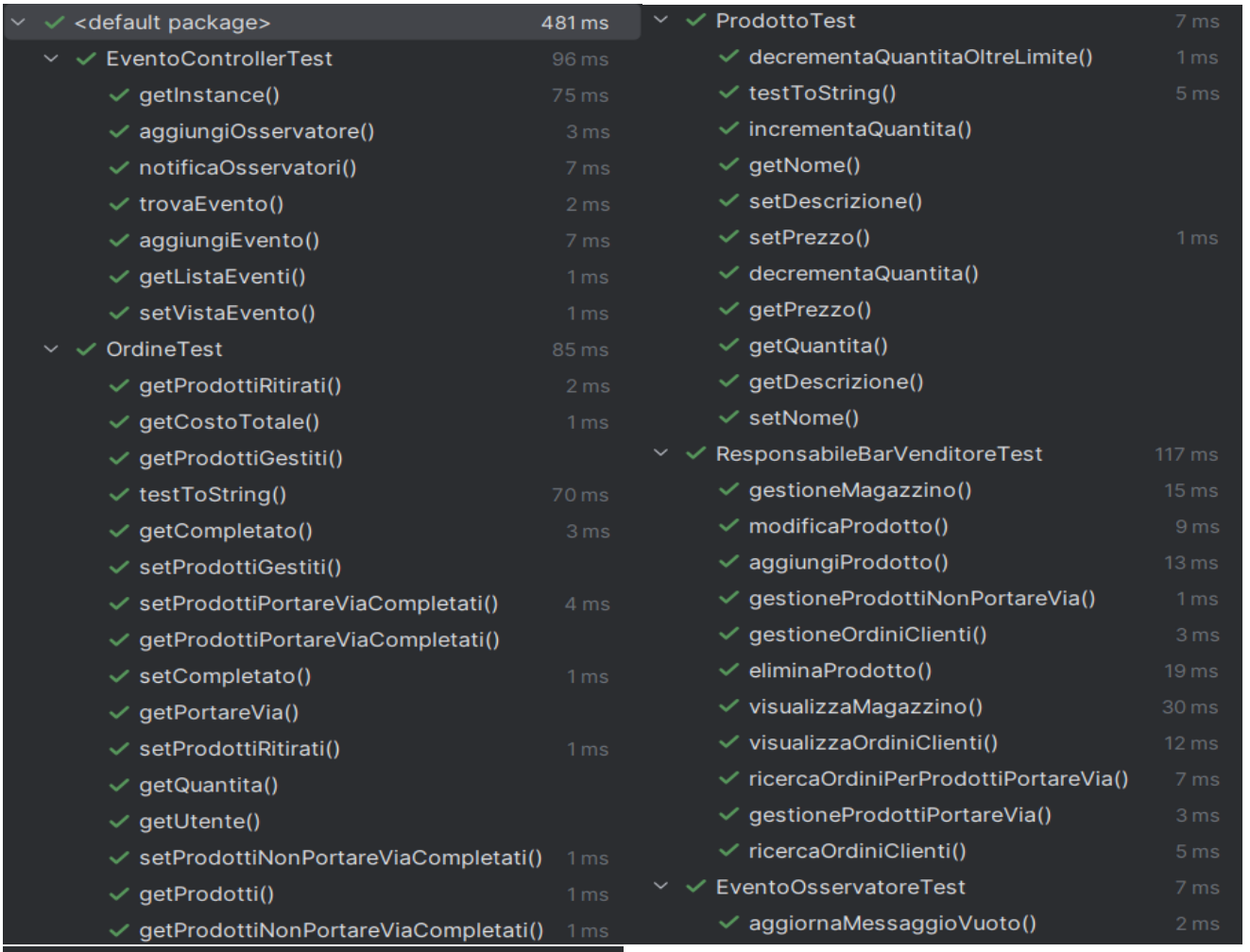
Current scope: all classes

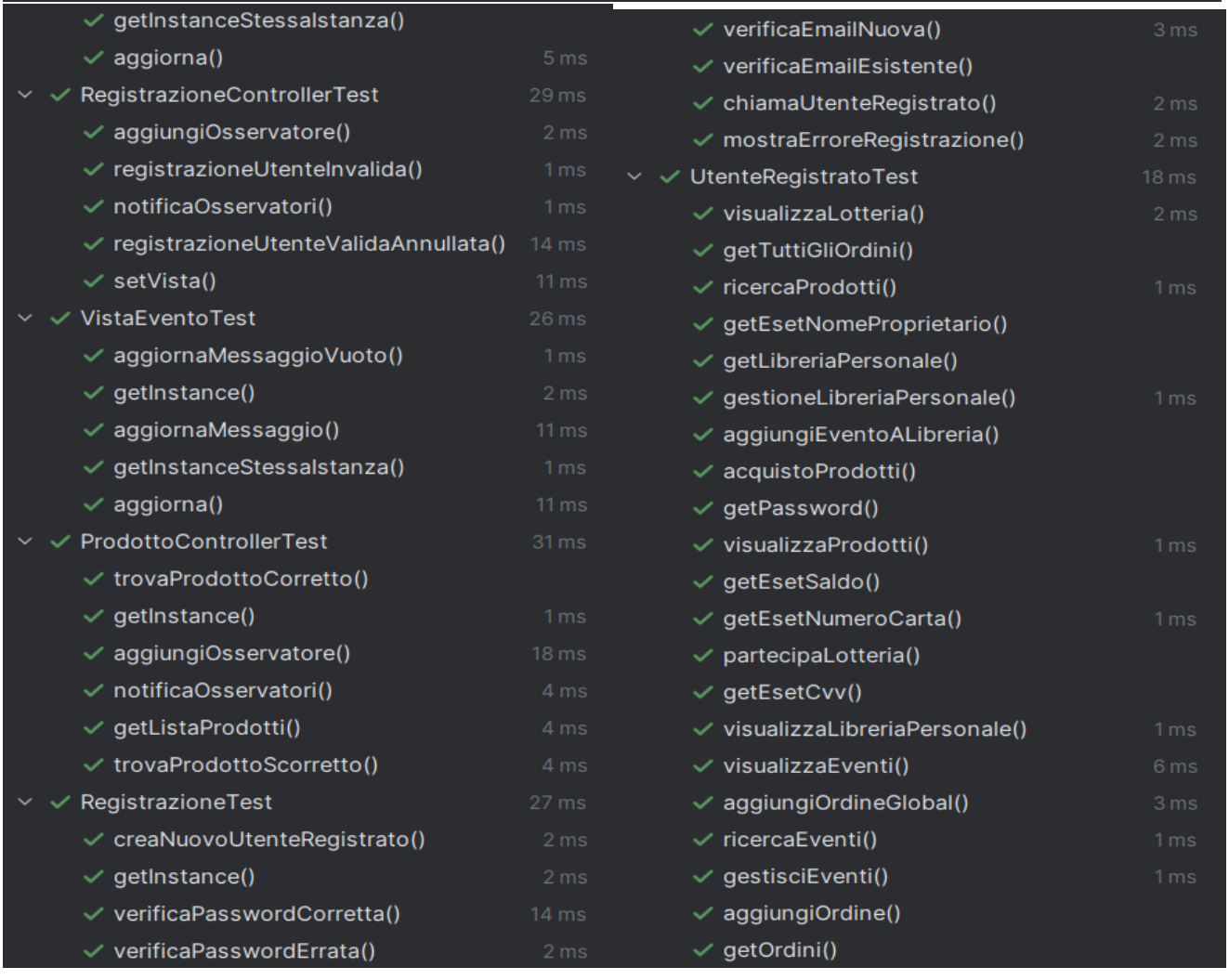
Overall Coverage Summary

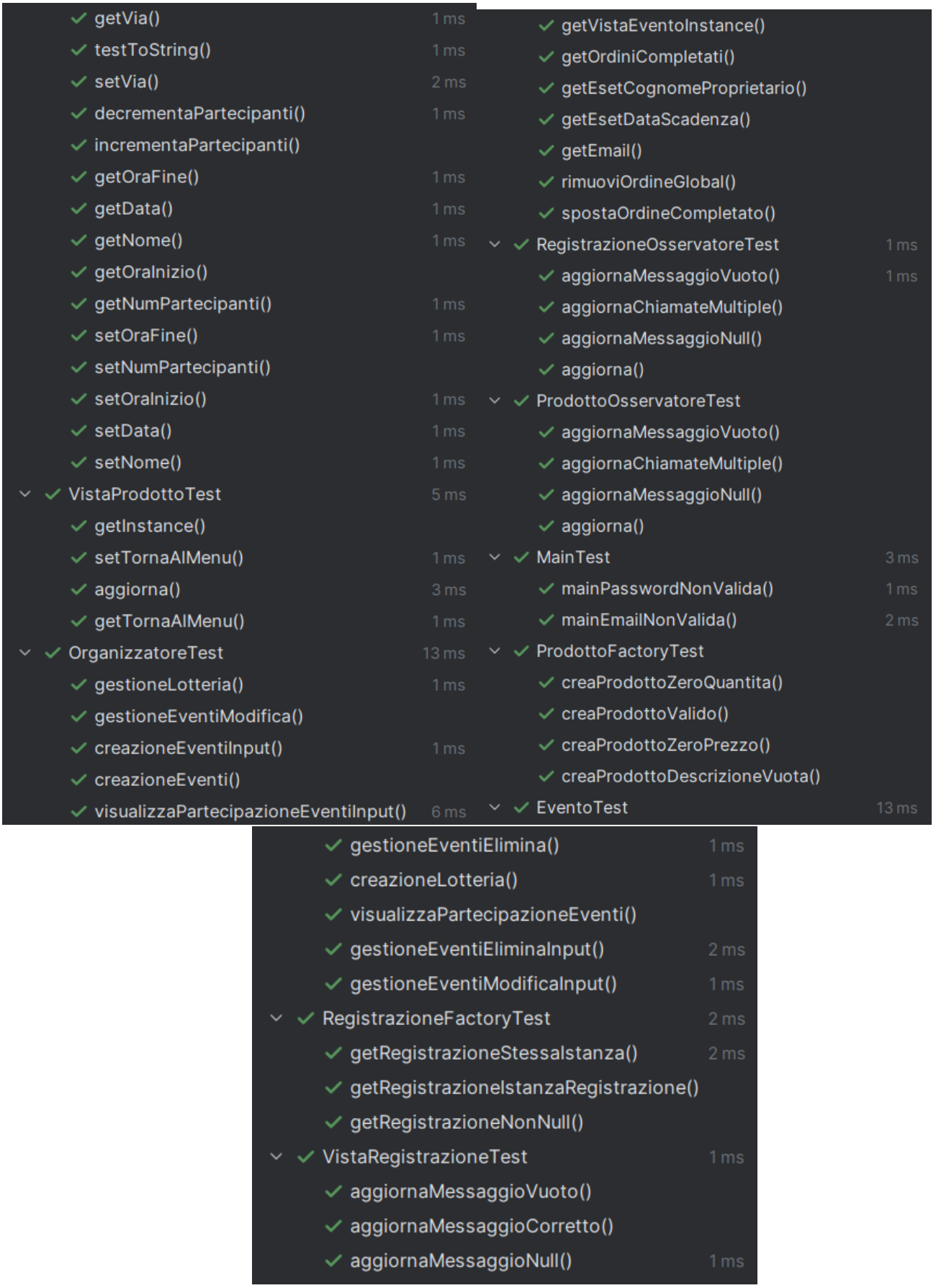
Package	Class, %	Method, %	Branch, %	Line, %
all classes	100% (16/16)	90% (135/150)	25% (173/693)	43,6% (655/1503)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %
Controller	100% (3/3)	100% (20/20)	96,2% (25/26)	91,9% (57/62)
Factory	100% (2/2)	50% (2/4)		50% (2/4)
Main	100% (1/1)	50% (1/2)	83,3% (5/6)	90,6% (29/32)
Model	100% (7/7)	89,4% (101/113)	21,2% (139/657)	39,7% (552/1390)
Vista	100% (3/3)	100% (11/11)	100% (4/4)	100% (15/15)


<default package> 481 ms
EventoControllerTest 96 ms
getInstance() 75 ms
aggiungiOsservatore() 3 ms
notificaOsservatori() 7 ms
trovaEvento() 2 ms
aggiungiEvento() 7 ms
getListaEventi() 1 ms
setVistaEvento() 1 ms
OrdineTest 85 ms
getProdottiRitirati() 2 ms
getCostoTotale() 1 ms
getProdottiGestiti()
testToString() 70 ms
getCompletato() 3 ms
setProdottiGestiti()
setProdottiPortareViaCompletati() 4 ms
getProdottiPortareViaCompletati()
setCompletato() 1 ms
getPortareVia()
setProdottiRitirati() 1 ms
getQuantita()
getUtente()
setProdottiNonPortareViaCompletati() 1 ms
getProdotti() 1 ms
getProdottiNonPortareViaCompletati() 1 ms
ProdottoTest 7 ms
decrementaQuantitaOltreLimite() 1 ms
testToString() 5 ms
incrementaQuantita()
getNome()
setDescrizione()
setPrezzo() 1 ms
decrementaQuantita()
getPrezzo()
getQuantita()
getDescrizione()
setNome()
ResponsabileBarVenditoreTest 117 ms
gestioneMagazzino() 15 ms
modificaProdotto() 9 ms
aggiungiProdotto() 13 ms
gestioneProdottiNonPortareVia() 1 ms
gestioneOrdiniClienti() 3 ms
eliminaProdotto() 19 ms
visualizzaMagazzino() 30 ms
visualizzaOrdiniClienti() 12 ms
ricercaOrdiniPerProdottiPortareVia() 7 ms
gestioneProdottiPortareVia() 3 ms
ricercaOrdiniClienti() 5 ms
EventoOsservatoreTest 7 ms
aggiornaMessaggioVuoto() 2 ms


getInstanceStessaIstanza()
aggiorna() 5 ms
RegistrazioneControllerTest 29 ms
aggiungiOsservatore() 2 ms
registrazioneUtenteInvalida() 1 ms
notificaOsservatori() 1 ms
registrazioneUtenteValidaAnnullata() 14 ms
setVista() 11 ms
VistaEventoTest 26 ms
aggiornaMessaggioVuoto() 1 ms
getInstance() 2 ms
aggiornaMessaggio() 11 ms
getInstanceStessaIstanza() 1 ms
aggiorna() 11 ms
ProdottoControllerTest 31 ms
trovaProdottoCorretto()
getInstance() 1 ms
aggiungiOsservatore() 18 ms
notificaOsservatori() 4 ms
getListaProdotti() 4 ms
trovaProdottoScorretto() 4 ms
RegistrazioneTest 27 ms
creaNuovoUtenteRegistrato() 2 ms
getInstance() 2 ms
verificaPasswordCorretta() 14 ms
verificaPasswordErrata() 2 ms
verificaEmailNuova() 3 ms
verificaEmailEsistente()
chiamaUtenteRegistrato() 2 ms
mostraErroreRegistrazione() 2 ms
UtenteRegistratoTest 18 ms
visualizzaLotteria() 2 ms
getTuttiGliOrdini()
ricercaProdotti() 1 ms
getEsetNomeProprietario()
getLibreriaPersonale()
gestioneLibreriaPersonale() 1 ms
aggiungiEventoALibreria()
acquistoProdotti()
getPassword()
visualizzaProdotti() 1 ms
getEsetSaldo()
getEsetNumeroCarta() 1 ms
partecipaLotteria()
getEsetCvv()
visualizzaLibreriaPersonale() 1 ms
visualizzaEventi() 6 ms
aggiungiOrdineGlobal() 3 ms
ricercaEventi() 1 ms
gestisciEventi() 1 ms
aggiungiOrdine()
getOrdini()



Struttura dei File e Allegati

La consegna è stata eseguita tramite file separati, di cui un file pdf e due cartelle zip; il file pdf “[ING SW PF]SamueleDarioScamozzi_10772778_986209” contiene la documentazione, la cartella zip “ProgettoFinale-Vero” contiene il codice sorgente commentato e i relativi test del programma e la cartella “ProgettoFinale-Vero_CoverageReport” contiene il Coverage Report che viene stampato.

Conclusioni

Il progetto per la gestione degli eventi in una sagra paesana, pur essendo nato in ambito accademico, è stato sviluppato con l'obiettivo di creare un'applicazione potenzialmente funzionale e utile in un contesto reale ma non ampio; nonostante non sia pronta per un rilascio immediato, l'architettura e le funzionalità implementate forniscono una solida base per un'eventuale evoluzione verso un prodotto finito.

La struttura del progetto, basata sul pattern MVC e arricchita dall'implementazione di altri design pattern come Observer, Factory e Singleton, offre una buona flessibilità e modularità; questo, faciliterebbe future espansioni e miglioramenti del sistema, rendendo più agevole l'aggiunta di nuove funzionalità o la modifica di quelle esistenti.

Per trasformare questo progetto in un'applicazione pronta per l'uso in un contesto reale, sarebbero necessari alcuni passaggi fondamentali:

1. Utilizzo di dati reali: Sostituire l'attuale simulazione di dati, come le e-mail o i dati della carta di credito, con dati reali che verrebbero controllati tramite funzioni esterne.
2. Implementazione di un database remoto: Sostituire l'attuale simulazione del database con un sistema di persistenza dei dati robusto e scalabile, come MySQL.
3. Sviluppo di un'interfaccia utente grafica: Creare un frontend user-friendly, potenzialmente utilizzando tecnologie web moderne come React o Flutter per garantire la compatibilità multi-piattaforma.
4. Potenziamento della sicurezza: Implementare misure di cybersecurity per proteggere i dati degli utenti e l'integrità del sistema.
5. Test approfonditi: Condurre test estensivi, inclusi test di integrazione e di carico, per garantire la stabilità e l'affidabilità dell'applicazione in scenari di utilizzo reale; inoltre, ampliare i test di unità.
6. Ottimizzazione per dispositivi mobili: Adattare l'interfaccia e le funzionalità per un uso agevole su smartphone e tablet, considerando l'importanza della mobilità in contesti come le sagre paesane.

Alcuni obiettivi futuri per l'evoluzione del progetto potrebbero includere:

- Implementazione delle funzionalità lasciate vuote per migliorare l'esperienza utente durante gli eventi, oltre che aggiunta di nuove, come la prenotazione dei tavoli, fornire indicazioni dei luoghi e delle informazioni importanti degli eventi come parcheggi, posti disponibili, menù, eccetera.
- Integrazione di un sistema di pagamento online per semplificare le transazioni.
- Sviluppo di un sistema di notifiche push per tenere gli utenti aggiornati sugli eventi e le offerte.
- Creazione di un'API per consentire l'integrazione con altri sistemi o applicazioni di terze parti.
- Implementazione di analisi dei dati per fornire insights agli organizzatori sulle preferenze degli utenti e il successo degli eventi.

In conclusione, questo progetto ha gettato solide basi per un'applicazione potenzialmente utile nel contesto delle sagre paesane; con ulteriori sviluppi e raffinamenti, potrebbe diventare uno strumento prezioso per organizzatori e partecipanti, migliorando significativamente la gestione e l'esperienza di questi eventi tradizionali. Il feedback degli utenti reali sarà cruciale per guidare le future iterazioni e assicurare che l'applicazione soddisfi effettivamente le esigenze del suo target.