# Università degli Studi di Salerno
**Corso di Ingegneria del Software**

# Pop!x
# Object Design Document
# Versione 3.0


Data: 15/12/2024

**Coordinatore del progetto:**

| Nome | Matricola |
|---|---|
| Scaparra Daniele Pio | 0512116260 |

**Partecipanti:**

| Nome | Matricola |
|---|---|
| Scaparra Daniele Pio | 0512116260 |
| Bonagura Grazia | 0512116167 |
| Nappi Antonio | 0512117391 |
| Nardiello Raffaele | 0512118666 |

| | |
|---|---|
| **Scritto da:** | Scaparra Daniele Pio, Bonagura Grazia, Nappi Antonio, Nardiello Raffaele |

# Revision History

| Data | Versione | Descrizione | Autore |
|---|---|---|---|
| 15/12/2024 | 1.0 | Prima versione dell'Object Design Document | Scaparra Daniele Pio, Bonagura Grazia, Nappi Antonio, Nardiello Raffaele |
| 26/12/2024 | 1.1 | Ristrutturato l'Object Design Document | Scaparra Daniele Pio, Bonagura Grazia, Nappi Antonio, Nardiello Raffaele |
| 27/12/2024 | 1.2 | Creata interfaccia del servizio di autenticazione | Bonagura Grazia, Scaparra Daniele Pio |
| 27/12/2024 | 1.3 | Aggiunto servizio di sicurezza | Antonio Nappi, Nardiello Raffaele |
| 28/12/2024 | 2.0 | Versione di rilascio | Scaparra Daniele Pio, Bonagura Grazia, Nappi Antonio, Nardiello Raffaele |
| 06/01/2025 | 3.0 | Versione finale ODD | Scaparra Daniele Pio, Bonagura Grazia, Nappi Antonio, Nardiello Raffaele |
| | | | |

# Indice

**1 Introduzione..... 4**
1.1 Object Design Trade-offs..... 4
1.2 Linee guida per la documentazione dell'interfaccia..... 4
1.3 Definizioni, acronimi, abbreviazioni..... 4
1.4 Riferimenti..... 5
**2 Packages..... 5**
2.1 Struttura dei pacchetti..... 5
2.2 Struttura grafica dei packages nel sistema..... 5
**3 Interfacce delle classi..... 7**
AuthenticationService..... 7
SecurityService..... 9
CartService..... 12
OrderService..... 14
CatalogService..... 16
DataStorage (User)..... 18
Data Storage (Order)..... 19
Data Storage (Cart)..... 20
Data Storage (Catalog)..... 20
UserView..... 21
CatalogView..... 23
OrderView..... 26
CartView..... 27
**4. Design Patterns..... 28**
4.1 Model-View-Controller (MVC)..... 28
4.2 DAO (Data Access Object)..... 28
**5 Glossario..... 29**

# 1 Introduzione

## 1.1 Object Design Trade-offs

Il design di Pop!x si basa sul pattern MVC (Model-View-Controller) per garantire la separazione delle responsabilità.

- ***Buy vs Build***: Sono stati utilizzati framework e librerie open-source, come Bootstrap per il front-end e Tomcat per il server web, riducendo i tempi di sviluppo.
- ***Memory Space vs Response Time***: Sono stati ottimizzati i DAO per minimizzare le query al database e migliorare i tempi di risposta, accettando un maggiore utilizzo della memoria per memorizzare cache locali.

## 1.2 Linee guida per la documentazione dell'interfaccia

Le seguenti convenzioni sono state adottate per uniformare il design delle interfacce:

- **Naming:**
  - Classi con nomi al singolare e descrittivi (es. Prodotto, Ordine).
  - Metodi con frasi verbali che indicano l'azione svolta (es. aggiungiProdotto, validaPagamento).
  - Campi e parametri con nomi sostantivi (es. prezzo, idUtente).
- **Gestione degli errori:**
  - Gli errori sono segnalati tramite eccezioni.
  - Le operazioni su collezioni restituiscono oggetti di tipo List o Map robusti rispetto alla modifica degli elementi.
- **Conformità:** Tutte le interfacce devono essere documentate con JavaDoc e seguire il principio di interfacce minime.

## 1.3 Definizioni, acronimi, abbreviazioni

- **DAO:** Data Access Object, utilizzato per separare la logica di accesso ai dati.
- **DTO:** Data Transfer Object, per trasferire dati tra layer.
- **MVC:** Model-View-Controller, un pattern architetturale.

## 1.4 Riferimenti

- Riferimento al Requirements Analysis Document (RAD).
- Riferimento al System Design Document (SDD).
- Riferimento al Problem Statement Document.

# 2 Packages

## 2.1 Struttura dei pacchetti

Il sistema è suddiviso nei seguenti pacchetti:

1. **Presentation Layer:**
    - Pacchetto: com.popx.presentazione
    - Componenti principali: VistaUtente, VistaCatalogo, VistaCarrello, VistaOrdine.
2. **Service Layer:**
    - Pacchetto: com.popx.servizio
    - Componenti principali: ServizioAutenticazione, ServizioCheckout, ServizioProdotti.
3. **Persistence Layer:**
    - Pacchetto: com.popx.persistenza
    - Componenti principali: UtenteDAO, ProdottoDAO, OrdineDAO.

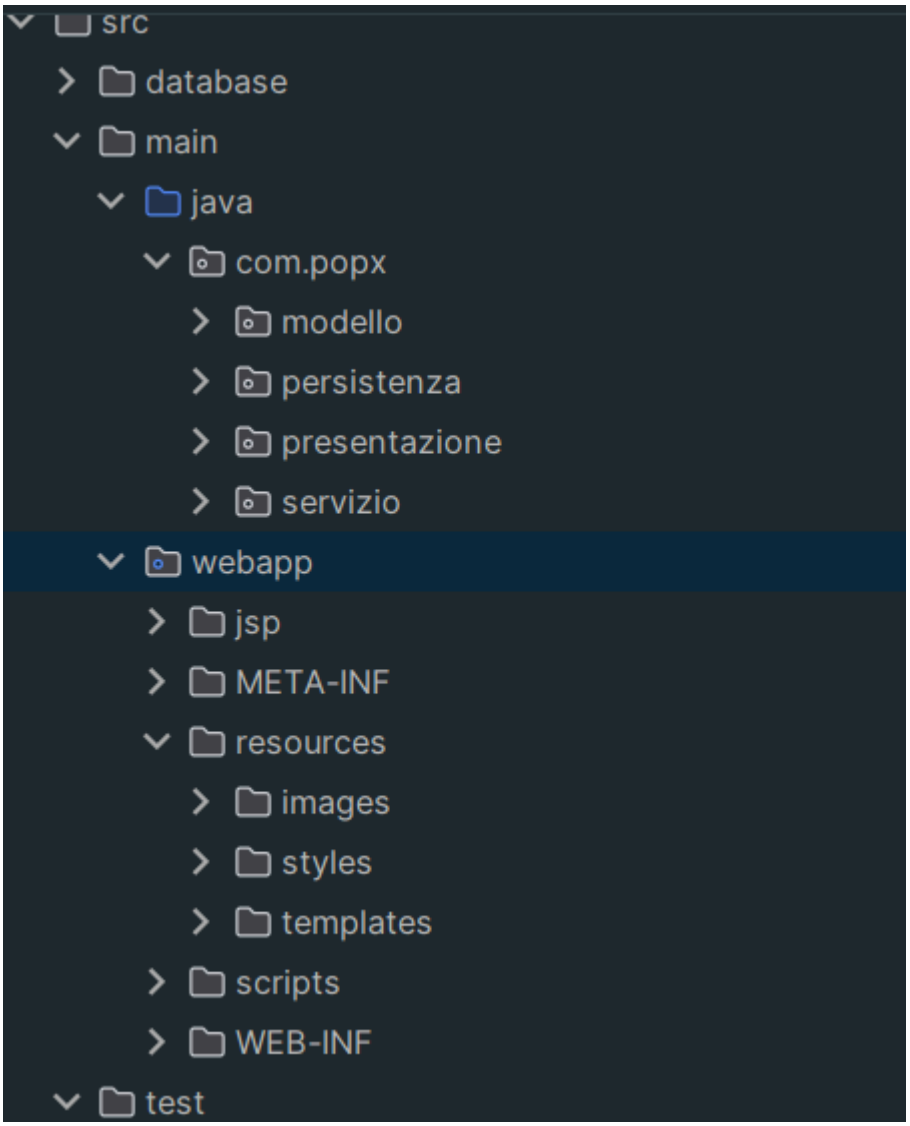
Ogni pacchetto è strutturato per ridurre le dipendenze e facilitare il riuso del codice.

## 2.2 Struttura grafica dei packages nel sistema

In questa sezione è descritta la struttura organizzativa dei files all'interno del progetto.
Si tratta di un'organizzazione di un progetto Java Web che utilizza Maven come gestore delle dipendenze.
Si può notare come il modello della suddivisione dei sottosistemi individuata nel documento di System Design viene pienamente rispettata andando ad essere implementata dai packages presentazione (Persistence Layer), servizio (Service Layer), persistenza (Persistence Layer).
All'interno di ognuno di questo package verranno realizzati i componenti indicati nel component diagram e rispetteranno la funzionalità loro assegnata.
Abbiamo anche i file e directory di Maven, come quello di target, il pom.xml e vari file di

configurazioni, esattamente come dovrebbe essere in un progetto che usa le tecnologie descritte (Java servlets, JSP, Bootstrap, Maven, HTML, CSS)

Il progetto si presenta così nella sua struttura:




Abbiamo i package per il modello (file dei Java Beans che rappresentano le entità definite nel class diagram), persistenza (le servlet che servono per interagire con le pagine JSP e mostrare il contenuto), presentazione (le interfacce dei Dao e l'implementazione dei Dao che servono per fornire i metodi usati da servlet e JSP).
Poi abbiamo il webapp, dove ci sono tutti contenuti per la parte web, come le pagine jsp, il META-INF e WEB-INF per le configurazioni, la folder resources contenente immagini, stili css e i templates (header e footer che sono presenti e riutilizzati praticamenti in ogni pagina).

## 3 Interfacce delle classi

Per ogni package verrà fornita in questa sezione le interfacce pubbliche e i relativi metodi principali.

| **Interfaccia** | **AuthenticationService** |
|---|---|
| Descrizione | AuthenticationService fornisce il servizio di autenticazione degli utenti, verifica dei ruoli e controllo delle credenziali. |
| Metodi | +login(String email, String password) : user |
| | +logout(String email) : void |
| | +getRuolo(String email) : String |
| Invariante di classe | -Gli utenti non autenticati non devono avere ruoli accessibili<br>-Consistenza del ruolo |

Elenco, descrizione, spiegazioni metodi

| **Nome Metodo** | +login(String email, String password) : user |
|---|---|
| Descrizione | Questo metodo permette di effettuare il login, controllando le credenziali e creando la sessione per l'utente. |
| Pre-condizioni | *context AuthenticationService::login(email: String, password: String) : User*<br>**pre:**<br>not email.isEmpty() and not password.isEmpty()<br>and self.isEmailValid(email) |
| Post-condizioni | *context AuthenticationService::login(email: String, password: String) : User*<br>**post:** |

| | |
|---|---|
| | result <> null implies<br>result.email = email and<br>self.verifyPassword(password, result.getHashedPassword()) = true<br>and<br>result.isAuthenticated = true<br>and<br>self.sessions->includes(result) |
| Invarianti | context User<br>inv: self.hashedPassword->notEmpty()<br>and self.hashedPassword = hash(self.hashedPassword) |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email e la password non devono essere vuote.
  - L'email deve essere valida secondo un criterio definito nel metodo isEmailValid.
- **Post-condizioni:**
  - Se il risultato (result) non è null, significa che il login è avvenuto con successo.
    Inoltre:
    - L'utente restituito dal metodo deve avere l'email corrispondente a quella fornita in ingresso.
    - L'utente deve risultare autenticato. Questa condizione aggiorna lo stato dell'utente indicando che ha effettuato correttamente il login.
  - L'utente autenticato deve essere aggiunto alla collezione di sessioni attive.
- **Invarianti:**
  - Ogni utente deve avere una password salvata come valore hashato e non in chiaro.

| **Nome Metodo** | +logout(String email, String password) : void |
|---|---|
| Descrizione | Questo metodo permette di fare il logout, cancellando la sessione dell'utente. |
| Pre-condizioni | *context AuthenticationService::logout(email: String, password: String) : void*<br>**pre:**<br>not email.isEmpty() and not password.isEmpty()<br>and self.sessions->exists(s \| s.user.email = email and s.user.isAuthenticated = true)<br>and self.verifyPassword(password, self.getUserByEmail(email).hashedPassword) = true |
| Post-condizioni | *context AuthenticationService::logout(email: String, password: String) : void*<br>**post:**<br>self.sessions->forAll(s \| s.user.email <> email) |
| Invarianti | context AuthenticationService<br>inv: self.users->select(u \| u.isAuthenticated = true)->forAll(u \| self.sessions->exists(s \| s.user = u)) |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email e la password non devono essere vuote.

- ➢ Deve esistere una sessione attiva per l'utente corrispondente all'email, e l'utente deve essere autenticato.
- ➢ La password fornita deve essere verificata con l'hash salvato per l'utente.

❖ **Post-condizioni:**
- ➢ Dopo l'esecuzione del metodo logout, non devono esistere sessioni attive associate all'utente corrispondente all'email fornita.

❖ **Invarianti:**
- ➢ Se un utente è autenticato (isAuthenticated = true), deve esistere una sessione attiva associata a quell'utente.

| **Nome Metodo** | +getRuolo(String email) : String |
|---|---|
| Descrizione | Questo metodo permette di ottenere il ruolo e riconoscere che ruolo l'utente che è autenticato |
| Pre-condizioni | *context AuthenticationService::getRuolo(email: String) : String*<br>**pre:**<br>not email.isEmpty()<br>and self.sessions->exists(s \| s.user.email = email and s.user.isAuthenticated = true) |
| Post-condizioni | *context AuthenticationService::getRuolo(email: String) : String*<br>**post:**<br>result = self.users->select(u \| u.email = email).role |
| Invarianti | context User<br>inv: not self.role.isEmpty() |

**Spiegazioni su:**

❖ **Pre-condizioni:**
- ➢ L'email fornita non deve essere vuota.
- ➢ Deve esistere una sessione attiva associata a un utente con quell'email, e l'utente deve essere autenticato.

❖ **Post-condizioni:**
- ➢ Il risultato (result) deve corrispondere al ruolo dell'utente con l'email specificata.

❖ **Invarianti:**
- ➢ Ogni utente deve avere un ruolo assegnato e questo ruolo non deve essere vuoto.

| **Interfaccia** | **SecurityService** |
|---|---|
| Descrizione | SecurityService fornisce il servizio per la protezione dei dati e la gestione degli accessi con tecniche come l'hashing delle password |
| Metodi | +hashPassword(String password) : String |
| | +verifyPassword(String insertedPassword, hashedPassword) : boolean |
| | +getHashedPassword(String email) : String |

| | |
|---|---|
| Invariante di classe | -Ogni utente deve avere una password hashata<br>-Le password hashate devono essere memorizzate correttamente<br>-Il risultato dell'hash deve essere una stringa non vuota |

Elenco, descrizione, spiegazioni metodi

| **Nome Metodo** | +hashPassword(String password) : String |
|---|---|
| Descrizione | Questo metodo permette di fare l'hashing della password inserita con l'algoritmo bycript. |
| Pre-condizioni | *context SecurityService::hashPassword(password: String) : String*<br>**pre:**<br>not password.isEmpty() |
| Post-condizioni | *context SecurityService::hashPassword(password: String) : String*<br>**post:**<br>result <> ""<br>and result = hash(password) |
| Invarianti | context SecurityService::hashPassword(password: String) : String<br>inv:<br>not result.isEmpty()<br>and result = hash(password) |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Assicura che la password non sia vuota.
- **Post-condizioni:**
  - Garantisce che il risultato sia una stringa non vuota e che rappresenti la versione hashata della password..
- **Invarianti:**
  - L'output del metodo hashPassword non può mai essere vuoto, quindi l'invariante assicura che il risultato sia sempre una stringa valida.
  - La funzione di hash deve restituire lo stesso valore hash ogni volta che viene chiamata con la stessa password, il che è garantito dalla definizione della funzione di hash.

| **Nome Metodo** | +verifyPassword(String insertedPassword, hashedPassword) : boolean |
|---|---|
| Descrizione | Questo metodo verifica se la password inserita corrisponde alla password hashata memorizzata.. |
| Pre-condizioni | *context SecurityService::verifyPassword(insertedPassword: String, hashedPassword: String) : boolean*<br>**pre:**<br>not insertedPassword.isEmpty() and not hashedPassword.isEmpty() |
| Post-condizioni | *context SecurityService::verifyPassword(insertedPassword: String, hashedPassword: String) : boolean* |

| | |
|---|---|
| | **post:**<br>result = (hash(insertedPassword) = hashedPassword) |
| Invarianti | context SecurityService::verifyPassword(insertedPassword: String, hashedPassword: String) : boolean<br>inv:<br>not insertedPassword.isEmpty() and not hashedPassword.isEmpty()<br>and result = (hash(insertedPassword) = hashedPassword) |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Assicura che entrambe le password (inserita e hashata) siano non vuote.
- **Post-condizioni:**
  - Verifica che l'hash della password inserita corrisponda alla password hashata memorizzata, restituendo true o false in base alla corrispondenza.
- **Invarianti:**
  - L'invariante assicura che entrambe le password non siano vuote, poiché non ha senso verificare una password vuota.
  - La verifica del risultato deve essere coerente con l'operazione di hashing: result è true solo se l'hash della password inserita è uguale a quella memorizzata nel sistema.

| **Nome Metodo** | +getHashedPassword(String email) : String |
|---|---|
| Descrizione | Questo metodo restituisce la versione hashata della password associata a un determinato utente identificato dall'email. |
| Pre-condizioni | *context SecurityService::getHashedPassword(email: String) : String*<br>**pre:**<br>not email.isEmpty()<br>and self.users->exists(u \| u.email = email) |
| Post-condizioni | *context SecurityService::getHashedPassword(email: String) : String*<br>**post:**<br>result = self.users->select(u \| u.email = email).hashedPassword |
| Invarianti | context SecurityService::getHashedPassword(email: String) : String<br>inv:<br>not email.isEmpty()<br>and self.users->exists(u \| u.email = email)<br>and not result.isEmpty()<br>and result = self.users->select(u \| u.email = email).hashedPassword |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Assicura che l'email non sia vuota e che esista un utente con quell'email.
- **Post-condizioni:**
  - Restituisce la password hashata dell'utente corrispondente all'email.
- **Invarianti:**

- ➢ L'invariante garantisce che l'email passata come parametro non sia vuota e che esista un utente con quella email nel sistema.
- ➢ Inoltre, la password hashata deve essere non vuota e deve corrispondere a quella memorizzata per l'utente specificato.

| **Interfaccia** | **CartService** |
|---|---|
| Descrizione | ManagementService fornisce il servizio per gestire il carrello, permettendo operazioni come aggiungere, rimuovere e modificare articoli (prodotti). |
| Metodi | +aggiungiItem(Prodotto p) : boolean |
| | +rimuoviItem(Prodotto p) : void |
| | +modificaItem(Prodotto p, int quantità) : boolean |
| Invariante di classe | -Ogni prodotto nel carrello deve essere un prodotto valido<br>-Ogni prodotto nel carrello deve avere una quantità positiva<br>-Ogni prodotto può esistere una sola volta nel carrello |

| **Nome Metodo** | +aggiungiItem(Prodotto p) : boolean |
|---|---|
| Descrizione | Questo metodo aggiunge un prodotto al carrello. Se il prodotto è già presente nel carrello, può aumentare la sua quantità o, in caso contrario, aggiungerlo come nuovo articolo. |
| Pre-condizioni | context ManagementCartService::aggiungiItem(p: Prodotto) : boolean<br>**pre:**<br>p <> null and self.carrello->exists(item \| item.prodotto = p) implies p.isDisponibile = true |
| Post-condizioni | context ManagementCartService::aggiungiItem(p: Prodotto) : boolean<br>**post:**<br>if self.carrello->exists(item \| item.prodotto = p) then<br>result = true<br>else<br>result = false |
| Invarianti | // |

**Spiegazioni su:**

- ❖ **Pre-condizioni:**
  - ➢ Il prodotto non deve essere null e deve essere valido (disponibile nel sistema).
- ❖ **Post-condizioni:**

➢ Se il prodotto è già presente nel carrello, l'aggiunta deve riuscire, altrimenti l'operazione restituisce false

| **Nome Metodo** | +rimuoviItem(Prodotto p) : void |
|---|---|
| Descrizione | Questo metodo rimuove un prodotto dal carrello. Se il prodotto non è presente, non viene eseguita alcuna modifica. |
| Pre-condizioni | context ManagementCartService::rimuoviItem(p: Prodotto) : void<br>**pre**:<br>p <> null and self.carrello->exists(item \| item.prodotto = p) |
| Post-condizioni | context ManagementCartService::rimuoviItem(p: Prodotto) : void<br>**post:**<br>not self.carrello->exists(item \| item.prodotto = p) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - ➢ Il prodotto deve essere presente nel carrello e non può essere null.
- **Post-condizioni:**
  - ➢ Una volta che il prodotto è stato rimosso, non deve più essere presente nel carrello.

| **Nome Metodo** | +modificaItem(Prodotto p, int quantità) : boolean |
|---|---|
| Descrizione | Questo metodo modifica la quantità di un prodotto nel carrello. Se la quantità è valida (positiva), viene aggiornata; altrimenti, l'operazione non viene eseguita. |
| Pre-condizioni | context ManagementCartService::modificaItem(p: Prodotto, quantità: Integer) : boolean<br>**pre**:<br>p <> null and self.carrello->exists(item \| item.prodotto = p) and quantità > 0 |
| Post-condizioni | context ManagementCartService::modificaItem(p: Prodotto, quantità: Integer) : boolean<br>**post:**<br>if self.carrello->exists(item \| item.prodotto = p)<br>then<br>result = true<br>else<br>result = false |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - ➢ Il prodotto non deve essere null, deve essere presente nel carrello e la quantità deve essere positiva.
- **Post-condizioni:**
  - ➢ Se la modifica della quantità è riuscita, il metodo restituisce true, altrimenti false.

| **Interfaccia** | **OrderService** |
|---|---|
| Descrizione | OrdersService fornisce il servizio per la gestione degli ordini, tra cui la creazione e aggiornamento dello stato. |
| Metodi | +createOrder(Collection<Prodotto> p, User u, Date d, int total) : Order |
| | +modifyStatus(Order d) : boolean |
| | +getStatus(Order d): String |
| Invariante di classe | -Gli ordini devono avere stati validi<br>-Gli ordini devono avere un totale positivo<br>-Gli ordini devono essere associati a utenti validi |

| **Nome Metodo** | +createOrder(Collection<Prodotto> p, User u, Date d, int total) : Order |
|---|---|
| Descrizione | Questo metodo crea un nuovo ordine basato su una lista di prodotti, un utente e una data. |
| Pre-condizioni | context OrdersService::createOrder(p: Collection(Prodotto), u: User, d: Date, total: Integer) : Order<br>**pre**:<br>not p->isEmpty()<br>and u.isAuthenticated()<br>and d >= Date::now()<br>and total > 0 |
| Post-condizioni | context ManagementCartService::modificaItem(p: Prodotto, quantità: Integer) : boolean<br>**post:**<br>result.products = p<br>and result.user = u<br>and result.date = d<br>and result.total = total |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - La collezione di prodotti non può essere vuota o null.
  - L'utente deve essere valido e autenticato.
  - La data non può essere nel passato.
  - Il totale deve essere maggiore di zero.
- **Post-condizioni:**
  - Un nuovo ordine viene creato con i dettagli forniti.
  - L'ordine è associato all'utente e contiene i prodotti specificati.

| **Nome Metodo** | +modifyStatus(Order d) : boolean |
|---|---|
| Descrizione | Questo metodo modifica lo stato di un ordine |
| Pre-condizioni | context OrdersService::modifyStatus(d: Order) : boolean<br>**pre**:<br>d <> null<br>and self.orders->includes(d)<br>and d.status.isModifiable() |
| Post-condizioni | context OrdersService::modifyStatus(d: Order) : boolean<br>**post:**<br>if result = true<br>then<br>d.status = d.status.next()<br>endif |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'ordine non deve essere null.
  - L'ordine deve esistere nel sistema.
  - Lo stato attuale dell'ordine deve permettere modifiche.
- **Post-condizioni:**
  - Lo stato dell'ordine viene modificato, se possibile.
  - Restituisce true se la modifica è stata eseguita, false altrimenti.

| **Nome Metodo** | +getStatus(Order d) : String |
|---|---|
| Descrizione | Questo metodo restituisce lo stato attuale di un ordine |
| Pre-condizioni | context OrdersService::getStatus(d: Order) : String<br>**pre**:<br>d <> null<br>and self.orders->includes(d) |
| Post-condizioni | context OrdersService::getStatus(d: Order) : String<br>**post:**<br>result = d.status |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'ordine non deve essere null.
  - L'ordine deve esistere nel sistema.
- **Post-condizioni:**
  - Restituisce lo stato dell'ordine specificato.

| **Interfaccia** | **CatalogService** |
|---|---|
| Descrizione | CatalogService fornisce il servizio per gestire il catalogo, permettendo operazioni come aggiungere, rimuovere e modificare articoli (prodotti). |
| Metodi | +creaProdotto(id: String, price: Double, description: String, brand: String, figure: String, img: Image) : Prodotto |
| | +modificaProdotto(p: Prodotto) : boolean |
| | +rimuoviProdotto(Prodotto p) : boolean |
| Invariante di classe | -Ogni prodotto deve avere id univoco<br>-Ogni prodotto nel carrello deve avere una quantità positiva<br>-Ogni prodotto deve avere prezzo positivo |

| **Nome Metodo** | +creaProdotto(String id, double price, String description, String brand, String figure, Image img) : Prodotto |
|---|---|
| Descrizione | Questo metodo crea un nuovo prodotto con i dettagli forniti. |
| Pre-condizioni | context CatalogService::creaProdotto(id: String, price: Double, description: String, brand: String, figure: String, img: Image) : Prodotto<br>**pre**:<br>not id.isEmpty() and not description.isEmpty() and not brand.isEmpty() and not figure.isEmpty()<br>and price > 0 |
| Post-condizioni | context CatalogService::creaProdotto(id: String, price: Double, description: String, brand: String, figure: String, img: Image) : Prodotto<br>**post:**<br>result <> null<br>and self.catalogo->includes(result) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il prezzo deve essere maggiore di zero.
  - Gli identificativi del prodotto (id, description, brand, figure) non possono essere nulli o vuoti.
- **Post-condizioni:**
  - Il prodotto creato è presente nel catalogo.

| **Nome Metodo** | +modificaProdotto(Prodotto p) : boolean |
|---|---|
| Descrizione | Questo metodo modifica i dettagli di un prodotto esistente. |
| Pre-condizioni | context CatalogService::modificaProdotto(p: Prodotto) : boolean<br>**pre**:<br>p <> null<br>and self.catalogo->includes(p) |
| Post-condizioni | context CatalogService::modificaProdotto(p: Prodotto) : boolean<br>**post:**<br>if result = true<br>then<br>self.catalogo->includes(p)<br>endif |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il prodotto non deve essere null.
  - Il prodotto deve esistere nel catalogo.
- **Post-condizioni:**
  - Il prodotto è aggiornato con i nuovi dettagli.
  - Restituisce true se l'operazione ha successo, false altrimenti.

| **Nome Metodo** | +rimuoviProdotto(Prodotto p) : boolean |
|---|---|
| Descrizione | Questo metodo rimuove un prodotto esistente dal catalogo. |
| Pre-condizioni | context CatalogService::rimuoviProdotto(p: Prodotto) : boolean<br>**pre**:<br>p <> null<br>and self.catalogo->includes(p) |
| Post-condizioni | context CatalogService::rimuoviProdotto(p: Prodotto) : boolean<br>**post:**<br>if result = true<br>then<br>not self.catalogo->includes(p)<br>endif |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il prodotto non deve essere null.
  - Il prodotto deve esistere nel catalogo.
- **Post-condizioni:**
  - Il prodotto non è più presente nel catalogo.
  - Restituisce true se l'operazione ha successo, false altrimenti.

| **Interfaccia** | **DataStorage (User)** |
|---|---|
| Descrizione | Gestisce la memorizzazione e il recupero di informazioni sugli utenti. |
| Metodi | +saveUser(User u): boolean |
| | +getUser(String email): User |
| Invariante di classe | -Ogni utente deve essere identificabile con un'email valida<br>-Ogni utente deve avere le informazioni personali richieste.<br>-Ogni utente deve avere un ruolo conforme ai ruoli definiti nel dominio applicativo. |

| **Nome Metodo** | +saveUser(User u): boolean |
|---|---|
| Descrizione | Questo metodo salva un utente nel database |
| Pre-condizioni | context UserPersistence::saveUser(u: User) : boolean<br>**pre**:<br>u <> null<br>and not u.email.isEmpty()<br>and User.allInstances()->forAll(existing \| existing.email <> u.email)<br>and u.role <> null |
| Post-condizioni | context UserPersistence::saveUser(u: User) : boolean<br>**post:**<br>result = true implies User.allInstances()->includes(u) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'oggetto User non deve essere nullo.
  - L'email dell'utente deve essere valida e unica nel sistema.
  - Il ruolo dell'utente deve essere specificato e valido (ad esempio, "Admin", "User", etc.).
- **Post-condizioni:**
  - Restituisce true se l'utente è stato salvato correttamente.

| **Nome Metodo** | +getUser(String email): User |
|---|---|
| Descrizione | Questo metodo recupera un utente dal database |

| | |
|---|---|
| Pre-condizioni | context UserPersistence::getUser(email: String) : User<br>**pre**:<br>not email.isEmpty()<br>and User.allInstances()->exists(u \| u.email = email) |
| Post-condizioni | context UserPersistence::getUser(email: String) : User<br>**post:**<br>result.email = email<br>and result.role <> null |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email non deve essere vuota.
  - L'utente con l'email specificata deve esistere.
- **Post-condizioni:**
  - Restituisce l'utente con l'email specificata, incluso il ruolo.

| **Interfaccia** | **Data Storage (Order)** |
|---|---|
| Descrizione | Gestisce i dettagli relativi agli ordini, come prodotti, stato, e date. |
| Metodi | +saveOrder(Order o): boolean |
| Invariante di classe | -Ogni ordine deve contenere almeno un prodotto per essere valido.<br>-Ogni ordine deve avere un identificativo unico e non vuoto |

| **Nome Metodo** | +saveOrder(Order o): boolean |
|---|---|
| Descrizione | Questo metodo permette di salvare un ordine |
| Pre-condizioni | context OrderPersistence::saveOrder(o: Order) : boolean<br>**pre**:<br>o <> null<br>and not o.orderId.isEmpty()<br>and Order.allInstances()->forAll(existing \| existing.orderId <> o.orderId) |
| Post-condizioni | context UserPersistence::updatePaymentMethod(u: User, p: Payment) : boolean<br>**post:**<br>result = true implies Order.allInstances()->includes(o) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'oggetto Order non deve essere nullo.

- ➢ L'identificativo dell'ordine (orderId) deve essere unico.

❖ **Post-condizioni:**

- ➢ Restituisce true se l'ordine è stato salvato correttamente.

| **Interfaccia** | **Data Storage (Cart)** |
|---|---|
| Descrizione | Gestisce lo stato del carrello dell'utente, salvando i prodotti selezionati. |
| Metodi | +saveCart(Cart c): boolean |
| Invariante di classe | -Ogni carrello deve essere associato a un utente valido.<br>-Ogni carrello deve avere una lista di articoli valida |

| **Nome Metodo** | +saveCart(Cart c): boolean |
|---|---|
| Descrizione | Questo metodo permette di salvare un ordine |
| Pre-condizioni | context CartPersistence::saveCart(c: Cart) : boolean<br>**pre**:<br>c <> null<br>and c.user <> null<br>and User.allInstances()->includes(c.user) |
| Post-condizioni | context CartPersistence::saveCart(c: Cart) : boolean<br>**post:**<br>result = true implies Cart.allInstances()->includes(c) |
| Invarianti | // |

**Spiegazioni su:**

❖ **Pre-condizioni:**

- ➢ Il carrello non deve essere nullo.
- ➢ Il carrello deve essere associato a un utente valido.

❖ **Post-condizioni:**

- ➢ Restituisce true se il carrello è stato salvato correttamente.

| **Interfaccia** | **Data Storage (Catalog)** |
|---|---|
| Descrizione | Gestisce i dati relativi ai prodotti, come descrizioni, prezzi, stock e altri dettagli. |
| Metodi | +saveProduct(Prodotto p): boolean |

| | |
|---|---|
| Invariante di classe | -Ogni prodotto deve avere un identificativo unico e non vuoto<br>-Ogni prodotto deve avere un prezzo maggiore di zero<br>-Ogni prodotto deve avere una descrizione<br>-Ogni prodotto deve appartenere a una categoria |

| | |
|---|---|
| **Nome Metodo** | +saveProduct(Prodotto p): boolean |
| Descrizione | Questo metodo permette di salvare un prodotto |
| Pre-condizioni | context ProductsPersistence::saveProduct(p: Prodotto) : boolean<br>**pre**:<br>p <> null<br>and not p.id.isEmpty()<br>and p.price > 0<br>and Product.allInstances()->forAll(existing \| existing.id <> p.id) |
| Post-condizioni | context ProductsPersistence::saveProduct(p: Prodotto) : boolean<br>**post:**<br>result = true implies Product.allInstances()->includes(p) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il prodotto non deve essere nullo.
  - Il prodotto deve avere un identificativo univoco (ID).
  - Il prodotto deve avere un prezzo maggiore di zero.
- **Post-condizioni:**
  - Restituisce true se il prodotto è stato salvato correttamente nel sistema.

| **Interfaccia** | **UserView** |
|---|---|
| Descrizione | Gestisce i dati relativi ai prodotti, come descrizioni, prezzi, stock e altri dettagli. |
| Metodi | +register(String email, String username, String password1, String password2): User |
| | +getUser(String email): User |
| Invariante di classe | |

| **Nome Metodo** | +register(String email, String username, String password1, String password2): User |
|---|---|
| Descrizione | Questo metodo permette di effettuare la registrazione |
| Pre-condizioni | context UserView::register(email: String, username: String, password1: String, password2: String) : User<br>**pre**:<br>not email.isEmpty()<br>and User.allInstances()->forAll(u \| u.email <> email)<br>and not username.isEmpty()<br>and password1 = password2 |
| Post-condizioni | context UserView::register(email: String, username: String, password1: String, password2: String) : User<br>**post:**<br>result.email = email<br>and result.username = username<br>and result.password = password1 |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email non deve essere già associata a un altro account.
  - Il nome utente (username) non deve essere già utilizzato.
  - La password1 deve essere uguale alla password2.
- **Post-condizioni:**
  - Restituisce l'utente registrato se l'operazione ha successo.

| **Nome Metodo** | +getUser(String email): User |
|---|---|
| Descrizione | Questo metodo permette di ottenere un utente dalla email. |
| Pre-condizioni | context UserPersistence::getUser(email: String) : User<br>**pre**:<br>not email.isEmpty()<br>and User.allInstances()->exists(u \| u.email = email) |
| Post-condizioni | context UserPersistence::getUser(email: String) : User<br>**post:**<br>result.email = email |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - La precondizione not email.isEmpty() verifica che l'email fornita non sia vuota.
  - User.allInstances()->exists(u | u.email = email) assicura che esista almeno un utente nel sistema con l'email fornita.
- **Post-condizioni:**

- La postcondizione assicura che l'utente restituito abbia l'email che è stata passata come parametro (result.email = email).

| **Interfaccia** | **CatalogView** |
|---|---|
| Descrizione | Gestisce la visualizzazione del catalogo di prodotti, con supporto a ricerche e filtri. |
| Metodi | +getProducts(): Set<Prodotto> |
| | +getProductById(String id): Prodotto |
| | +getProductsByBrand(String brand): Set<Prodotto> |
| | +getProductsByPrice(boolean ascending): Set<Prodotto> |
| | +getProductsByBrandAndPrice(String brand, boolean ascending): Set<Prodotto> |
| Invariante di classe | -Il catalogo non può essere vuoto<br>-Gli ID dei prodotti devono essere univoci<br>-Il prezzo di ogni prodotto deve essere un valore positivo. |

| **Nome Metodo** | +getProducts(): Set<Prodotto> |
|---|---|
| Descrizione | Questo metodo permette di recuperare tutte i prodotti |
| Pre-condizioni | // |
| Post-condizioni | context CatalogView::getProducts() : Set(Prodotto)<br>**post:**<br>result->notEmpty() |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Nessuna
- **Post-condizioni:**
  - Restituisce l'insieme di tutti i prodotti nel catalogo.

| **Nome Metodo** | +getProductById(String id): Prodotto |
|---|---|

| | |
|---|---|
| Descrizione | Questo metodo permette di ottenere un prodotto dall'id |
| Pre-condizioni | context CatalogView::getProductById(id: String) : Prodotto<br>**pre**:<br>not id.isEmpty()<br>and Product.allInstances()->exists(p \| p.id = id) |
| Post-condizioni | context CatalogView::getProductById(id: String) : Prodotto<br>**post:**<br>result.id = id |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'ID del prodotto deve essere valido e non vuoto.
- **Post-condizioni:**
  - Restituisce il prodotto con l'ID specificato, se esiste.

| **Nome Metodo** | +getProductsByBrand(String brand): Set<Prodotto> |
|---|---|
| Descrizione | Questo metodo permette di ottenere una collezione di prodotti dal brand |
| Pre-condizioni | context CatalogView::getProductsByBrand(brand: String) : Set(Prodotto)<br>**pre**:<br>not brand.isEmpty()<br>and Product.allInstances()->exists(p \| p.brand = brand) |
| Post-condizioni | context CatalogView::getProductsByBrand(brand: String) : Set(Prodotto)<br>**post:**<br>result->forAll(p \| p.brand = brand) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il marchio deve essere valido e non vuoto.
- **Post-condizioni:**
  - Restituisce un insieme di prodotti appartenenti al marchio specificato.

| **Nome Metodo** | +getProductsByPrice(boolean ascending): Set<Prodotto> |
|---|---|
| Descrizione | Questo metodo permette di ottenere una collezione di prodotti in base al criterio di prezzo. |
| Pre-condizioni | // |

| | |
|---|---|
| Post-condizioni | context CatalogView::getProductsByPrice(ascending: Boolean) : Set(Prodotto)<br>**post:**<br>if ascending<br>then<br>result = Product.allInstances()->sortedBy(p \| p.price)<br>else<br>result = Product.allInstances()->sortedBy(p \| p.price)->reverse()<br>endif |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'ordinamento per prezzo deve essere valido (ascendente o discendente).
- **Post-condizioni:**
  - Restituisce i prodotti ordinati per prezzo, in ordine crescente o decrescente a seconda del parametro ascending.

| | |
|---|---|
| **Nome Metodo** | +getProductsByBrandAndPrice(String brand, boolean ascending): Set<Prodotto> |
| Descrizione | Questo metodo permette di ottenere una collezione di prodotti in base al criterio di prezzo e brand |
| Pre-condizioni | context CatalogView::getProductsByBrandAndPrice(brand: String, ascending: Boolean) : Set(Prodotto)<br>**pre**:<br>not brand.isEmpty()<br>and Product.allInstances()->exists(p \| p.brand = brand) |
| Post-condizioni | context CatalogView::getProductsByBrandAndPrice(brand: String, ascending: Boolean) : Set(Prodotto)<br>**post:**<br>result->forAll(p \| p.brand = brand)<br>if ascending<br>then<br>result = result->sortedBy(p \| p.price)<br>else<br>result = result->sortedBy(p \| p.price)->reverse()<br>endif |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Il marchio deve essere valido e non vuoto.
  - L'ordinamento per prezzo deve essere valido (ascendente o discendente).
- **Post-condizioni:**
  - Restituisce i prodotti appartenenti al marchio specificato, ordinati per prezzo (ascendente o discendente a seconda del parametro ascending).

| **Interfaccia** | **OrderView** |
|---|---|
| Descrizione | Gestisce la visualizzazione degli ordini degli utenti e la gestione degli ordini da parte degli admin. |
| Metodi | +getOrderById(String orderId): Order |
| | +getOrdersByUser(String email): Set<Order> |
| | +getAllOrders(): Set<Order> |
| Invariante di classe | -L'ID dell'ordine deve essere unico e non può essere nullo o vuoto.<br>-Gli ordini devono essere associati a un utente esistente.<br>- |

| **Nome Metodo** | +getOrderById(String orderId): Order |
|---|---|
| Descrizione | Questo metodo permette di ottenere un ordine in base all'id |
| Pre-condizioni | context OrderView::getOrderById(orderId: String) : Order<br>**pre**:<br>not orderId.isEmpty()<br>and Order.allInstances()->exists(o \| o.id = orderId) |
| Post-condizioni | context OrderView::getOrderById(orderId: String) : Order<br>**post:**<br>result.id = orderId |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'ID dell'ordine deve essere valido e non vuoto.
- **Post-condizioni:**
  - Restituisce l'ordine con l'ID specificato, se esiste.

| **Nome Metodo** | +getOrdersByUser(String email): Set<Order> |
|---|---|
| Descrizione | Questo metodo permette di ottenere una collezione di ordini in base all'utente |
| Pre-condizioni | context OrderView::getOrdersByUser(email: String) : Set(Order)<br>**pre**:<br>not email.isEmpty()<br>and User.allInstances()->exists(u \| u.email = email) |

| | |
|---|---|
| Post-condizioni | context OrderView::getOrdersByUser(email: String) : Set(Order)<br>**post:**<br>result->forAll(o \| o.user.email = email) |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email dell'utente deve essere valida e corrispondere a un utente esistente.
- **Post-condizioni:**
  - Restituisce un insieme di ordini associati all'utente specificato.

| **Nome Metodo** | +getAllOrders(): Set<Order> |
|---|---|
| Descrizione | Questo metodo permette di ottenere tutti gli ordini del sistema |
| Pre-condizioni | // |
| Post-condizioni | context OrderView::getAllOrders() : Set(Order)<br>**post:**<br>result->notEmpty() |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - Nessuna precondizione specifica.
- **Post-condizioni:**
  - Restituisce tutti gli ordini presenti nel sistema.

| **Interfaccia** | **CartView** |
|---|---|
| Descrizione | Gestisce la visualizzazione e la modifica del carrello degli utenti. |
| Metodi | +getCartByUser(String email): Cart |
| Invariante di classe | -Ogni carrello deve appartenere a un utente valido. |

| **Nome Metodo** | +getCartByUser(String email): Cart |
|---|---|
| Descrizione | Questo metodo permette di ottenere un carrello in base all'email dell'utente. |
| Pre-condizioni | context CartView::getCartByUser(email: String) : Cart<br>**pre**:<br>not email.isEmpty()<br>and User.allInstances()->exists(u \| u.email = email) |
| Post-condizioni | context CartView::getCartByUser(email: String) : Cart<br>**post:**<br>result.user.email = email |
| Invarianti | // |

**Spiegazioni su:**

- **Pre-condizioni:**
  - L'email dell'utente deve essere valida e corrispondere a un utente esistente.
- **Post-condizioni:**
  - Restituisce il carrello dell'utente specificato

# 4. Design Patterns

Per garantire una progettazione robusta, modulare e manutenibile, sono stati utilizzati i seguenti design pattern:

## 4.1 Model-View-Controller (MVC)

- Motivo: Separare le responsabilità tra la logica di presentazione (View), la logica applicativa (Controller), e i dati (Model).
- Applicazione:
  - Model: Classi come Prodotto, Ordine, Carrello e DAO.
  - View: Classi come VistaCarrello, VistaOrdine per l'interfaccia utente.
  - Controller: Servizi come ServizioAutenticazione, ServizioCheckout.
  -

## 4.2 DAO (Data Access Object)

- Motivo: Separare la logica di accesso ai dati dalla logica di business.
- Applicazione:
  - Classi come ProdottoDAO, OrdineDAO, UtenteDAO gestiscono le operazioni di lettura/scrittura sul database.
  - Implementare diverse strategie di pagamento (es. carta di credito, PayPal).
  - Calcolare i costi di spedizione (es. standard, express).

## 5 Glossario

| Termine | Descrizione |
|---|---|
| Autenticazione | Processo di verifica dell'identità di un utente tramite credenziali come email e password. |
| Autorizzazione | Processo che verifica se un utente ha i permessi per eseguire una determinata azione. |
| Cache locale | Memorizzazione temporanea di dati vicino al punto di utilizzo per migliorare le prestazioni. |
| Caching | Tecnica per memorizzare dati temporanei al fine di migliorare le prestazioni del sistema. |
| Collezione | Struttura dati che memorizza più elementi (es. liste, mappe), robusta rispetto alla modifica. |
| Configurazione | Insieme di impostazioni e parametri che definiscono il comportamento di un sistema. |
| Data Access Object (DAO) | Pattern che separa la logica di accesso ai dati dal resto del sistema. |
| Data Transfer Object (DTO) | Oggetto utilizzato per trasferire dati tra diversi livelli del sistema. |
| Eccezione di runtime | Errore che si verifica durante l'esecuzione di un programma e richiede una gestione immediata. |
| Eccezioni | Meccanismo per gestire errori o comportamenti inattesi durante l'esecuzione del codice. |
| Framework | Struttura software che fornisce funzionalità generali per semplificare lo sviluppo di applicazioni. |
| Gestione errori | Strategie per intercettare, segnalare e |

| | |
|---|---|
| | gestire situazioni anomale durante l'esecuzione. |
| Hashing | Processo di conversione di dati (es. password) in una stringa univoca e non reversibile. |
| Invarianti | Condizioni che devono essere sempre vere durante il ciclo di vita di un oggetto o di una classe. |
| Interfaccia | Contratto che definisce i metodi pubblici e il comportamento di una classe. |
| Layer | Livello logico di un'architettura software che separa responsabilità specifiche (es. presentazione, logica, persistenza). |
| Metodo | Funzione associata a un oggetto o a una classe che può essere invocata per eseguire operazioni. |
| Model-View-Controller (MVC) | Pattern che separa la logica di presentazione (View), applicativa (Controller) e dei dati (Model). |
| Pattern di progettazione | Soluzione generica e riutilizzabile per problemi ricorrenti nel design software. |
| Post-condizioni | Condizioni che devono essere vere dopo l'esecuzione di un metodo o di una funzione. |
| Pre-condizioni | Condizioni che devono essere vere prima dell'esecuzione di un metodo o di una funzione. |
| Query | Richiesta di informazioni o operazioni effettuata su un database. |
| Ruoli | Permessi o responsabilità assegnati agli utenti in base alla loro posizione nel sistema. |
| Sicurezza | Misure e tecniche per proteggere dati e accessi da utilizzi non autorizzati o compromissioni. |

| | |
|---|---|
| Sessione | Contesto attivo che rappresenta un utente autenticato e le sue interazioni con il sistema. |
| Set | Struttura dati che memorizza un insieme di elementi unici, senza duplicati |
| Validazione | Processo per verificare che i dati forniti rispettino determinati criteri o requisiti. |