Università degli Studi di Salerno
Corso di Ingegneria del Software

Pop!x
Object Design Document
Versione 1.2

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

Indice

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)

```
project-root/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── popx/
│   │   │           ├── presentazione/       // Controller e View
│   │   │           ├── servizio/            // Logica applicativa
│   │   │           ├── persistenza/         // DAO e repository
│   │   │           ├── modello/             // Classi di dominio (es. Prodotto, Ordine)
│   │   │           └── utils/               // Classi delle utilities
│   │   ├── resources/
│   │   │   ├── application.properties       // Configurazioni del progetto
│   │   │   └── templates/                   // Template JSP/HTML
│   │   └── webapp/
│   │       ├── WEB-INF/
│   │       │   ├── jsp/                     // JSP
│   │       │   └── web.xml                  // Configurazione servlet
│   └── test/
│       └── java/                        // Test unitari e di integrazione
├── target/                              // Directory generata da Maven
│   ├── classes/                         // Classi compilate
│   ├── generated-sources/               // Sorgenti generati
│   ├── test-classes/                    // Classi di test compilate
│   ├── popx-ecommerce-1.0-SNAPSHOT.jar  // Artefatto generato
└── pom.xml                              // File Maven principale
```

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 -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* **pre:** not email.isEmpty() and not password.isEmpty() and self.isEmailValid(email)
Post-condizioni	*context AuthenticationService::login(email: String, password: String) : User* **post:** result <> null implies result.email = email and self.verifyPassword(password, result.getHashedPassword()) = true and result.isAuthenticated = true and self.sessions->includes(result)
Invarianti	context User inv: self.hashedPassword->notEmpty()

	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* **pre:** not email.isEmpty() and not password.isEmpty() and self.sessions->exists(s | s.user.email = email and s.user.isAuthenticated = true) and self.verifyPassword(password, self.getUserByEmail(email).hashedPassword) = true
Post-condizioni	*context AuthenticationService::logout(email: String, password: String) : void* **post:** self.sessions->forAll(s | s.user.email <> email)
Invarianti	context AuthenticationService 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* **pre:** not email.isEmpty() and self.sessions->exists(s | s.user.email = email and s.user.isAuthenticated = true)
Post-condizioni	*context AuthenticationService::getRuolo(email: String) : String* **post:** result = self.users->select(u | u.email = email).role
Invarianti	context User 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.

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