Sommario

Software design

Nel contesto del progetto di gestione automatizzata dei parcheggi urbani, è cruciale scegliere un pattern che consenta di gestire efficacemente la complessità delle interazioni tra i vari componenti, come i sensori, i moduli di pagamento e il sistema di notifiche.

Pattern Selezionato: Observer

Il sistema deve notificare in tempo reale i cambiamenti di stato di un parcheggio (es. "libero" o "occupato") a vari componenti:

- Il modulo di notifiche per avvisare gli utenti (residenti o visitatori occasionali) di violazioni o scadenze.
- Il sistema di dashboard per aggiornare la visualizzazione dello stato dei parcheggi disponibili.
- Altri moduli che richiedono informazioni aggiornate in base alle modifiche dello stato del parcheggio.

La difficoltà consiste nel mantenere un sistema decoupled (a basso accoppiamento) in cui le modifiche in un modulo non influiscano direttamente sugli altri.

Soluzione

Il pattern **Observer** definisce una relazione uno-a-molti tra un soggetto (subject) e i suoi osservatori (observers). Quando il soggetto subisce una modifica di stato, notifica automaticamente tutti gli osservatori registrati.

Nel contesto del progetto:

- **Soggetto**: Il modulo di gestione dei sensori di parcheggio.
- **Osservatori**: I moduli che necessitano di aggiornamenti, come il sistema di notifiche, la dashboard e il sistema di gestione delle multe.

Questa configurazione assicura che ogni cambiamento nel sistema dei sensori venga propagato istantaneamente ai componenti interessati senza creare dipendenze dirette.

Applicazione nel progetto

1. **Sensori di parcheggio**:
   - Ogni parcheggio è dotato di sensori che rilevano la presenza di un veicolo. Questi sensori rappresentano il soggetto che genera eventi (es. “occupato” o “libero”).
   - Quando un sensore rileva un cambiamento, notifica gli osservatori registrati.
2. **Sistema di notifiche**:
   - Riceve aggiornamenti dallo stato dei sensori. Ad esempio, se un veicolo è parcheggiato senza pagamento, invia un avviso all’utente.
3. **Dashboard**:
   - Mostra in tempo reale la disponibilità dei parcheggi basandosi sugli aggiornamenti provenienti dai sensori.
4. **Gestione delle multe**:
   - Quando un sensore segnala che un veicolo è in violazione (es. parcheggio senza pagamento), questo modulo genera automaticamente una multa.

Conseguenze dell’applicazione

Vantaggi:

- **Modularità**: I moduli sono indipendenti tra loro, il che facilita l’aggiunta di nuovi osservatori (es. un modulo di analisi dei dati) senza alterare il sistema esistente.
- **Scalabilità**: La struttura consente di aggiungere sensori o nuove funzionalità senza modificare i componenti già implementati.
- **Manutenzione ridotta**: Le modifiche nel soggetto non richiedono aggiornamenti negli osservatori e viceversa.

Svantaggi:

- **Complessità aggiuntiva**: Può essere difficile tracciare quali osservatori sono registrati, specialmente in sistemi con molti moduli.
- **Prestazioni**: Un numero elevato di notifiche simultanee potrebbe rallentare il sistema.

Conclusione

Il pattern **Observer** è particolarmente adatto al sistema di gestione dei parcheggi urbani, poiché permette di creare un'architettura flessibile e reattiva. Ogni componente può concentrarsi sulla propria funzionalità, delegando al soggetto il compito di coordinare gli aggiornamenti. Questo approccio garantisce una gestione efficiente e scalabile del sistema.

Diagrammi uml:

Diagramma 1: Class Diagram del Sistema di Parcheggio

Questo diagramma delle classi rappresenta la struttura statica del sistema di parcheggio. Le principali componenti del sistema sono:

1. **Gestione Utenti**: Contiene le classi correlate agli utenti, inclusi:
   - **Residente** e **Visitatore Occasionale**, che estendono la classe base **Utenti**.
   - **Abbonamento**, che è associato ai residenti o ad altre categorie di utenti.
2. **Gestione Parcheggi**: Include le classi:
   - **Sensore**, responsabile per rilevare i veicoli.
   - **Sistema Parcheggio**, che gestisce l'infrastruttura generale del parcheggio.
3. **Gestione Pagamenti**: Contiene la classe **Sistema Pagamenti**, che si occupa dell'elaborazione delle transazioni economiche per il parcheggio.

Le frecce tratteggiate indicano le relazioni tra i diversi moduli e componenti del sistema.

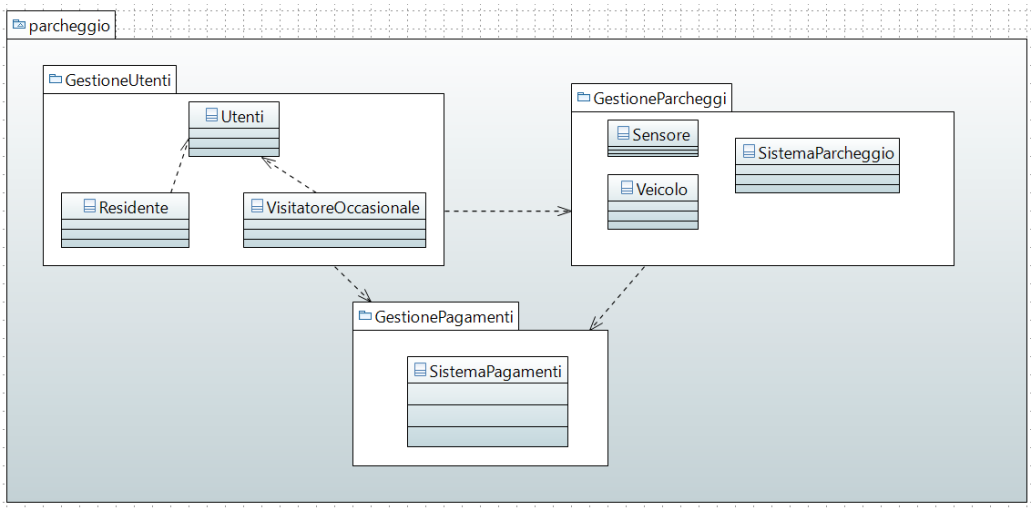
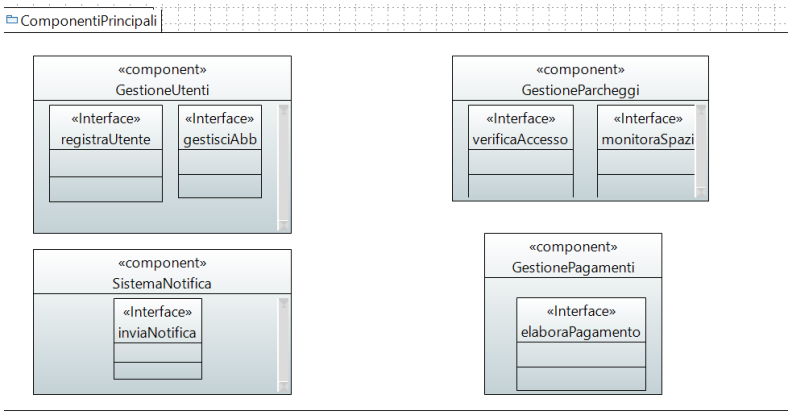


Diagramma 2: Component Diagram del Sistema

Questo diagramma dei componenti descrive le principali unità funzionali del sistema e le loro interfacce. I componenti principali sono:

1. **Gestione Utenti**:
   - Interfaccia registraUtente per registrare nuovi utenti.
   - Interfaccia gestisciAbb per gestire gli abbonamenti degli utenti.
2. **Gestione Parcheggi**:
   - Interfaccia verificaAccesso per controllare l'accesso al parcheggio.
   - Interfaccia monitoraSpazi per monitorare gli spazi disponibili.
3. **Gestione Pagamenti**:
   - Interfaccia elaboraPagamento per gestire i pagamenti dei visitatori occasionali o altre transazioni.
4. **Sistema Notifica**:
   - Interfaccia inviaNotifica per inviare notifiche di problemi o multe in caso di irregolarità.

Questo diagramma mostra come i diversi componenti interagiscono tra loro attraverso interfacce ben definite.



Software testing

Test per il Sistema di Gestione dei Parcheggi Urbani

Per il progetto sono stati eseguiti diversi test per verificare il funzionamento del sistema in base ai requisiti. Ecco una sintesi delle principali attività di testing:

Tecniche Applicate

1. **Black-Box Testing**:
   - Test sulle funzionalità principali come il pagamento tramite QRCode e la gestione delle tariffe aggiuntive per i visitatori occasionali.
2. **White-Box Testing**:
   - Verifica della copertura del codice, in particolare sul modulo di gestione delle notifiche e sulla logica di calcolo delle tariffe.

3. **Fault-Based Testing**:
   - Simulazione di errori come il mancato collegamento al database.
4. **Test-Driven Development (TDD)**:
   - Test scritti prima dello sviluppo, ad esempio per il calcolo delle tariffe. Questo approccio ha permesso di prevenire errori durante lo sviluppo.

Modelli di Testing

- **Demonstration**: Verifica che il sistema soddisfi i requisiti principali, come l'accesso tramite QRCode.
- **Destruction**: Prova di input errati (es. QRCode non valido) per testare la robustezza.
- **Evaluation**: Test mirati per individuare problemi nella fase iniziale.

Testing delle Classi Backend

Il testing delle classi contenute nei pacchetti responsabili della logica di gestione dei parcheggi (es. parking.model e parking.controller) è stato realizzato utilizzando **JUnit**. I test sono stati progettati per verificare:

- Il corretto funzionamento dei metodi di gestione dei pagamenti e degli abbonamenti.
- La validità delle operazioni legate al database simulato (es. associazione del QRCode ai dati del veicolo e calcolo delle tariffe).

Software Maintenance

Il refactoring è una delle attività fondamentali in un progetto software, e la sua scelta è strettamente legata alle necessità di migliorare la qualità del codice e garantirne la manutenibilità.
Nel contesto del sistema di gestione automatizzata dei parcheggi descritto nel documento, il refactoring può essere particolarmente utile per affrontare diversi aspetti critici:

1.Chiarezza e leggibilità del codice

Il progetto richiede che il codice sia scritto seguendo standard condivisi (come quelli Java), mantenendo la semplicità e riducendo la duplicazione, in linea con i principi di Extreme Programming (XP). Il refactoring, applicando tecniche come l'estrazione di metodi o la ristrutturazione di classi complesse, può semplificare il codice rendendolo più leggibile e accessibile a tutto il team, favorendo la proprietà collettiva del progetto.

2. Risoluzione dei "Bad Smells"

Un sistema che integra funzionalità avanzate come sensori, QRCode, e database embedded può facilmente sviluppare "cattivi odori" nel codice, ad esempio metodi troppo lunghi o un'eccessiva dipendenza tra

moduli. Utilizzando il refactoring, è possibile individuare ed eliminare questi problemi per garantire che il codice rimanga modulare e manutenibile.

3. Evoluzione del sistema

Il progetto è destinato a evolversi e adattarsi a cambiamenti eventual, come l'aggiunta di funzionalità per migliorare l'esperienza utente o gestire nuove regole per residenti e visitatori. Il refactoring facilita questa evoluzione garantendo che il codice esistente sia abbastanza flessibile da supportare modifiche senza compromettere la stabilità.

4. Prevenzione dell'entropia

Come indicato nel Capitolo 14, l'entropia del software può portare a una degradazione del sistema nel tempo. Il refactoring agisce come una misura preventiva per mantenere il codice in condizioni ottimali, riducendo i rischi di malfunzionamenti futuri e semplificando l'integrazione di nuove funzionalità.