UNIVERSITÀ
degli STUDI
di CATANIA

Ingegneria del Software
Corso di laurea Magistrale in Ingegneria Informatica

AutoParking

Conclusioni

Aurora Tallarita- Giulio Samperi

Sommario

Introduzione

In questo capitolo finale si discuteranno aspetti implementativi non trattati durante l'esposizione delle iterazioni. Molti dei temi illustrati in seguito sono stati considerati fin da subito durante lo sviluppo iterativo e affinati durante le varie iterazioni.
Si consideri che per lo sviluppo del software come linguaggi sono stati utilizzati C# per il client, C++ per il server e Python per il client destinato all'amministratore.

Database e persistenza

Uno dei requisiti principali della nostra applicazione è la persistenza dei dati, essendo un parcheggio è importante che l'utente possa salvare i propri dati utente e del veicolo per effettuare il parcheggio e riprendere il veicolo.
Per implementare la persistenza è stato utilizzato un Database relazionale sul quale si appoggia la nostra applicazione, di seguito ne riportiamo il diagramma Entità/Relazioni:

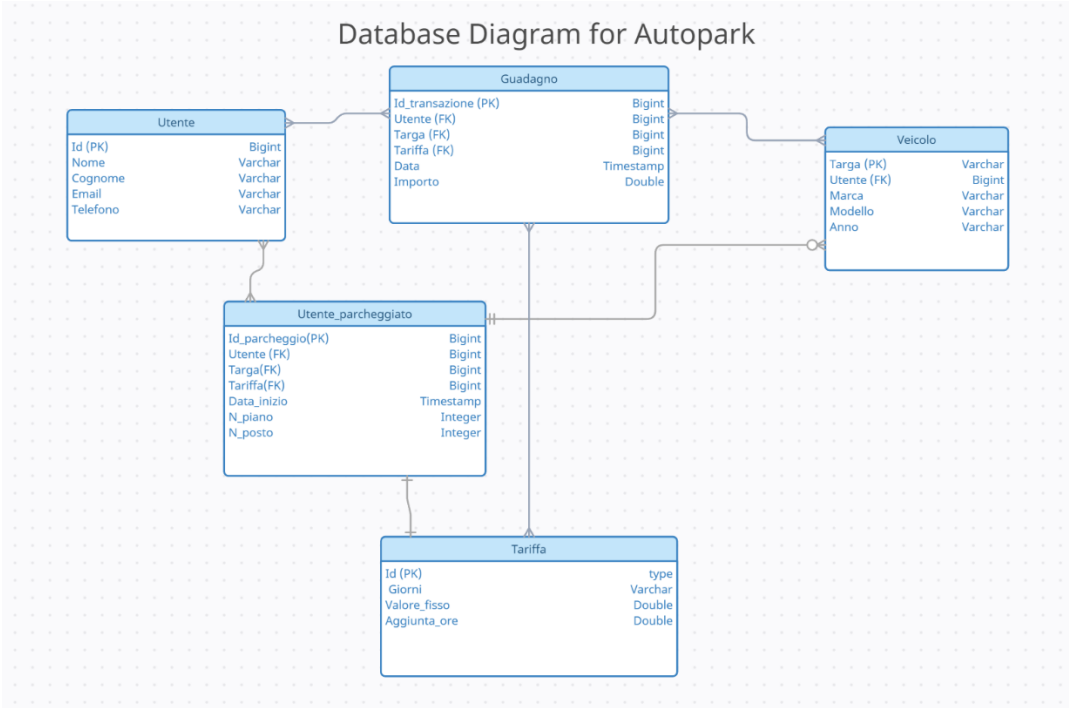


Figura 1 Diagramma ER

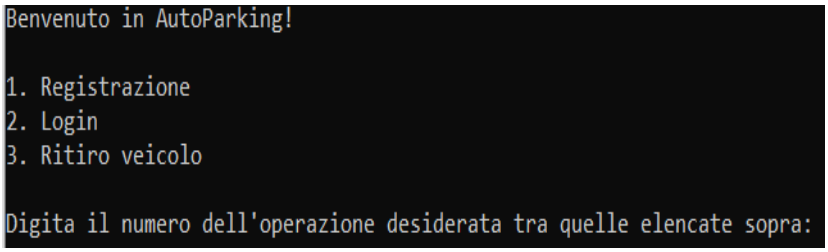
Per la gestione del database è stato utilizzato *MySQL*, per interfacciare il server della nostra applicazione al database è stata usata la libreria *mysql-connector-cpp-8.1,* un wrapper del protocollo jdbc scritto interamente in c++, usato dalla classe Dbhandler.

Test

Per il test dell'applicazione, avvenuto progressivamente, è stata utilizzata la libreria NUnit per il codice sviluppato in C# e UnitTest per la parte in Python.
NUnit è una delle librerie più popolari per scrivere test unitari in C# e altri linguaggi di programmazione basati su .NET. È open-source e offre una vasta gamma di funzionalità per aiutare gli sviluppatori a scrivere, eseguire e analizzare i test unitari in modo efficace.
Sono state utilizzate le Le fixture di test che consentono di creare un ambiente di test riproducibile, dove si possono preparare i dati e le condizioni necessarie per eseguire i test, sono state create le classi di test utilizzando l'attributo [TestFixture]. All'interno della classe, abbiamo definito i metodi di test utilizzando l'attributo [Test], questi metodi saranno eseguiti quando NUnit eseguirà i test.
UnitTest è un modulo integrato nel linguaggio di programmazione Python che fornisce un framework per scrivere ed eseguire test, i test vengono organizzati in classi che derivano dalla classe unittest.TestCase. Ogni metodo all'interno di una classe di test ereditata da TestCase rappresenta un test. Si è usato Il decoratore @patch che è particolarmente utile quando si desidera sostituire una funzione o un oggetto reale con un oggetto mock, in modo da poter simulare il comportamento desiderato per il test.
Nel nostro caso, volendo gestire un parcheggio, abbiamo spesso metodi che coinvolgono a loro volta metodi di altre classi che devono essere a loro volta istanziate correttamente. Inoltre alcune classi hanno un certo grado di accoppiamento poiché utilizzate in vari contesti e per queste ragioni la scelta delle classi, dei metodi e dell'ordine in cui testarli è stata effettuata seguendo un approccio dal basso verso l'alto. In particolare alcuni metodi apparentemente complessi non sono stati oggetto di test in modo esplicito.

Implementazione

Alla fine di ogni iterazione è stata implementata una parte di sistema, complessivamente è stato implementato in codice funzionante quasi tutta la parte del sistema, più nel dettaglio sono stati implementati completamente i casi d'uso UC1 (Richiesta parcheggio), UC2 (Ritiro auto), UC3 (Controllo posti occupati in tutti i piani), UC4 (Controllo posti occupati per piano), UC7(Controlla incassi), UC8(Modifica tariffa fissa), UC9(Crea pacchetto ore).
La scelta dei casi d'uso da implementare è ricaduta su quelli più importanti per il funzionamento del software. I casi d'uso mancanti infatti vanno ad aggiungere ulteriori funzionalità accessorie e sono UC5(Chiudi parcheggio) e UC6(Apri parcheggio) che sono più indicativi in un contesto reale e non progettuale.
Per poter accedere ai casi d'uso basta usufruire dei client, che tramite un menu testuale limitato all'uso della console di comando guidano l'utente o



l'amministratore nelle scelte a loro presentate. Una volta avviata l'esecuzione del software sarà possibile scegliere tra Registrazione, Login o Ritiro veicolo.

Figura 2 Menu principale

Nel caso di registrazione verranno richiesti prima i dati dell'utente quali nome, cognome, numero di telefono e password, dopodiché si procederà con la registrazione di un veicolo. Nel caso di Login o dopo aver effettuata la registrazione si accederà ad un menu interno dove sarà possibile effettuare altre scelte.

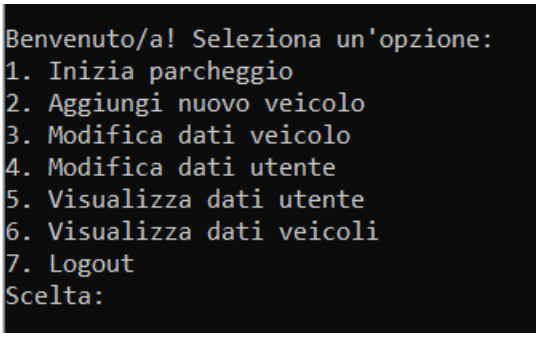


Figura 3 Menu interno

Pattern Gof

Pattern strutturale: Façade

È stato deciso di utilizzare il pattern façade nella classe DbHandler, semplifica l'interazione con il sistema di gestione del database sottostante. Incapsula l'accesso al database, la gestione delle connessioni, l'esecuzione di query e la manipolazione dei risultati. Espone metodi come start, add_user, check_user, add_vehicle, ecc funge da intermediario tra il sottosistema del database e gli altri componenti del sistema.

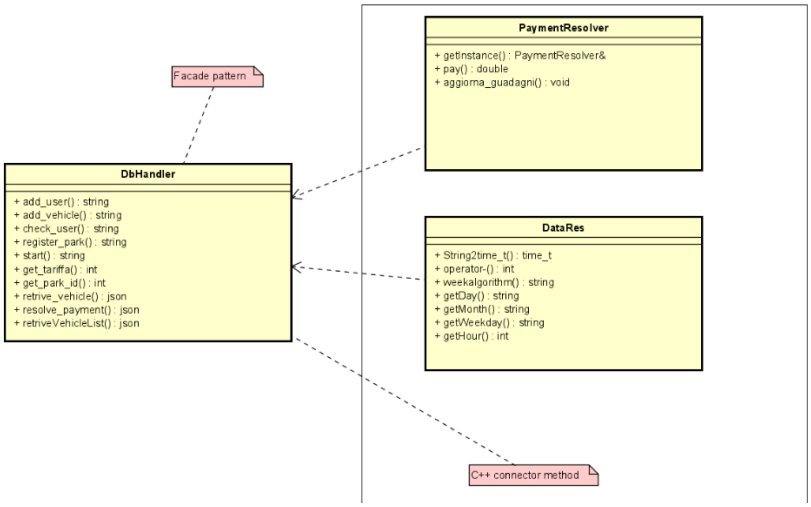


Figura 4 Pattern Facade

Pattern comportamentale: Template Method
Si è utilizzato il pattern GoF Template Method. La classe astratta AbstactGetGain definisce il metodo template get_incasso, mentre la classe ConcreteGetGain fornisce le implementazioni specifiche per le diverse opzioni di visualizzazione dell'incasso.

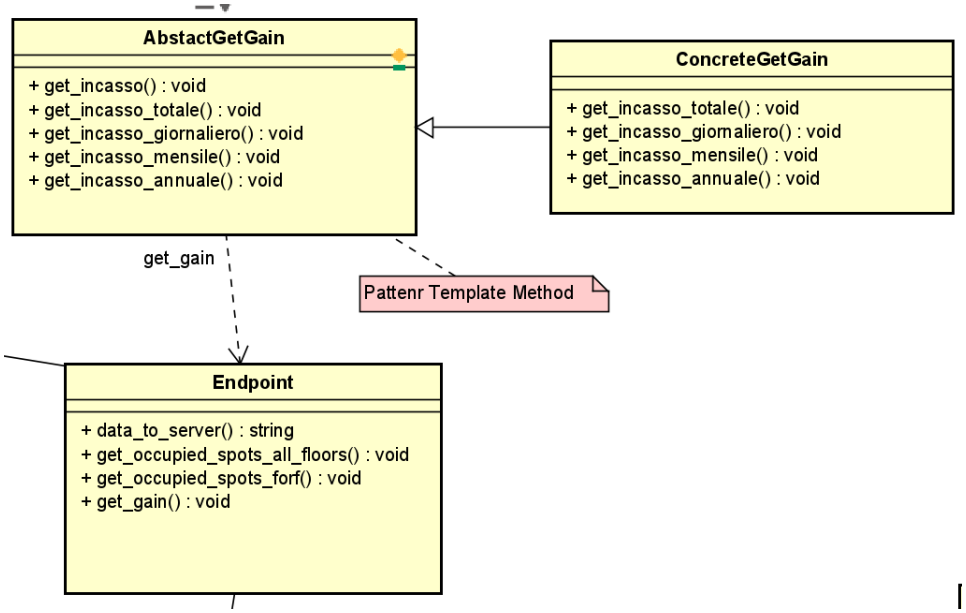


Figura 5 Pattern Template Method

Pattern creazionali: Singleton
Si è deciso di utilizzare il pattern Singleton ai fini di limitare la creazione ad una sola instanza della classe PaymentResolver, per aggiungere integrità al calcolo dei profitti, e su Handler per aggiungere sicurezza alle richieste del server.

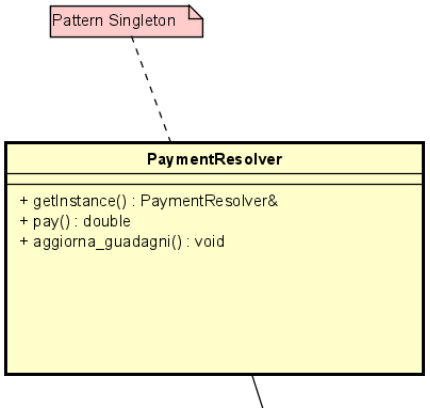


Figura 6 Pattern Singleton su PaymentResolver

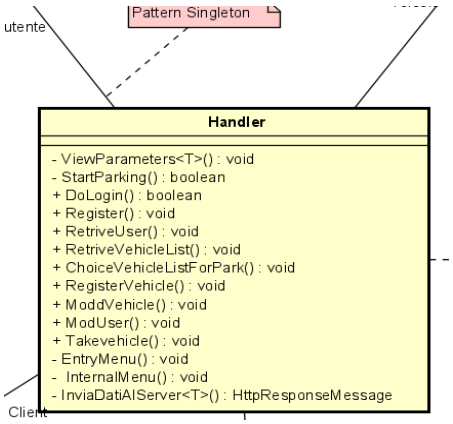


Figura 7 Pattern Singleton Handler

Pattern creazionali: Builder
È stato deciso di utilizzare il pattern Builder nella classe VehicleManager, la classe rappresenta un manager per la gestione dei veicoli. La sua funzione principale è quella di creare e gestire oggetti di tipo VehicleManager utilizzando il pattern Builder per distinguere tra diversi veicoli: autoveicoli, autovetture e motocicli. Nello specifico la classe in questione implementa un metodo EnterVehicle() viene utilizzato un oggetto VehicleBuilder per creare passo dopo passo un nuovo veicolo, impostando i suoi attributi come la targa, la marca, il modello, l'anno e il tipo. Infine, viene restituito l'oggetto VehicleManager creato.

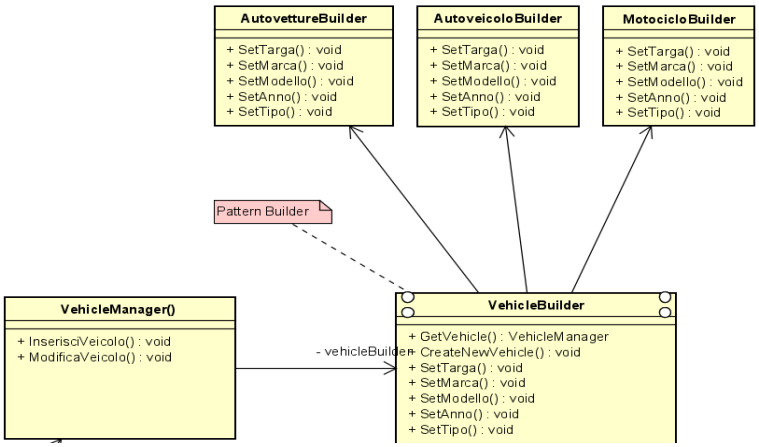


Figura 8 Pattern Builder

Note finali

Il progetto è stato svolto seguendo le linee guida dello sviluppo iterativo, ogni iterazione è stata suddivisa in una fase di analisi e una di progettazione. Nella prima fase venivano scelti i casi d'uso da trattare, approfonditi i requisiti a essi legati e veniva effettuata un'analisi orientata agli oggetti, individuando le entità del dominio e utilizzando gli strumenti forniti da UML. In seguito nella fase di progettazione avveniva il salto rappresentazionale con l'aiuto degli opportuni diagrammi, infine veniva implementata una parte di codice.
In tutte le iterazioni si è fatto uso del software Astah per la modellizzazione e per la stesura dei vari diagrammi.