Relazione Assignment #01

E-Bike Application

Software Architecture and Platforms

Ravaioli Michele - michele.ravaioli3@studio.unibo.it

23 ottobre 2024

Indice

1. Requisiti

Il progetto "E-Bike Application" prevede lo sviluppo di due prototipi basati su un'architettura client-server, con il backend implementato in due diverse versioni: la prima con uno stile architetturale a strati (layered architecture) e la seconda seguendo il modello architetturale clean (es. hexagonal o ports-and-adapters). I prototipi dovranno:

- Permettere l'interazione da remoto di utenti e amministratori con il sistema.
- Supportare la persistenza dei dati, garantendo la possibilità di cambiare il modello o la tecnologia di persistenza (es. MySQL, MongoDB, etc.) senza impattare sugli altri strati dell'applicazione.
- Consentire la modifica o l'aggiunta di front-end senza influenzare gli altri livelli, in particolare il livello di business logic.
- Consentire l'estensione dinamica delle funzionalità offerte agli utenti.

2. Quality Attributes

Nella progettazione del servizio di noleggio, entrambe le versioni dell'applicazione devono soddisfare specifici requisiti di qualità. In particolare, si deduce che il servizio debba garantire un'elevata disponibilità (*Availability*), supportare l'estensibilità delle funzionalità (*Extensibility*) e offrire un'interfaccia semplice e intuitiva per utenti e amministratori (*Simplicity*). Per verificare il rispetto di queste qualità, sono stati definiti i seguenti scenarios:

Scenario 1: Aggiunta bici al servizio (*Simplicity test*) **Goal:** Inserimento di una bicicletta nel magazzino del servizio **Environment:** Applicazione client collegata al server del servizio **Attori:** Amministratore
L'admin clicca il pulsante di inserimento bici L'admin specifica l'ID della bici La bici e' inserita nel magazzino
Response measure: L'intera operazione deve avvenire in un solo step

Scenario 2: Aggiunta nuova funzionalità al servizio (*Extensibility, Availability test*) **Goal:** Aggiunta nuova funzionalità al servizio **Environment:** Applicazione client collegata al server del servizio **Attori:** Amministratore
L'admin clicca il pulsante di inserimento nuova funzionalità L'admin specifica il nome della funzionalità La funzionalita' viene aggiunta sul server e mostrata nel client
Response measure: Il client e il server non devono mai interrompere l'esecuzione

3. Architettura

Per soddisfare questi quality attributes, si è scelto di adottare un'architettura Microkernel per entrambe le versioni dell'applicazione. Questo approccio consente di aggiungere nuove funzionalità al sistema in modo dinamico, senza interrompere il servizio.

Di seguito viene illustrato nel dettaglio come le due versioni del sistema sono state modellate e sviluppate.

3.1 Layered Architecture

L'architettura a strati è stata strutturata in tre livelli principali: **persistence layer**, **business logic layer** e **presentation layer**. Ciascuno di questi strati dipende solo da quello immediatamente sottostante: il business logic layer dipende dal persistence layer, e il presentation layer dipende dal business logic layer (Fig.1). Questo schema garantisce l'indipendenza e la modularità dei componenti, mantenendo una chiara separazione delle responsabilità.

Nel **persistence layer** sono presenti i componenti responsabili della gestione della memorizzazione dei dati. In questo caso, è implementata la classe *HashMapStorage*, che realizza l'interfaccia *DataStorage*. Questa interfaccia rappresenta una base di dati generica e non legata all'applicazione specifica, poiché deve essere il più possibile astratta, non avendo dipendenze dal business logic layer.

Il **business logic layer** contiene i componenti che modellano e gestiscono il servizio di noleggio e-bike. In questo strato si trovano sia le entità principali (come *User*, *EBike*, *Ride*), sia la logica che governa il sistema. L'interfaccia *RentalService* definisce il servizio, offrendo la possibilità di eseguire operazioni e aggiungere dinamicamente funzionalità tramite plugin (*RentalServicePlugin*). Questi plugin rappresentano delle funzioni che, prendendo in input lo stato corrente del sistema e dei parametri, elaborano un nuovo stato. Per questo anche le operazioni base del servizio (come inserimento bici, inserimento user) sono viste non come funzionalità built-in, bensì come estensioni del sistema. Per ciascuna operazione sono stati sviluppati plugin specifici: *AddUserPlugin*, *AddBikePlugin*, *StartRidePlugin* e *EndRidePlugin*. Per

rendere il sistema accessibile da remoto, il *RentalService* è stato implementato come servizio client-server, con RentalServiceServer e *RentalServiceClient*, che comunicano tramite HTTP (Fig.2).

Nel **presentation layer** sono contenute le classi necessarie per la creazione di un'interfaccia grafica semplice, basata su Swing. Questa interfaccia permette di eseguire operazioni sul sistema di noleggio e-bike in modo intuitivo, anche da remoto, utilizzando il *RentalService*. Tramite un pulsante, è possibile aggiungere dinamicamente dei plugin a servizio, senza che questo si interrompa.

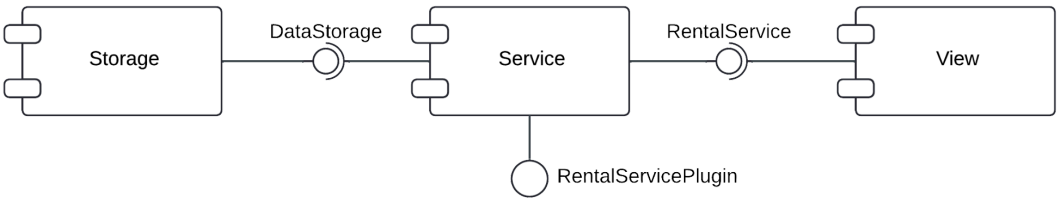

Fig.1: Diagramma dei componenti per la layered architecture.

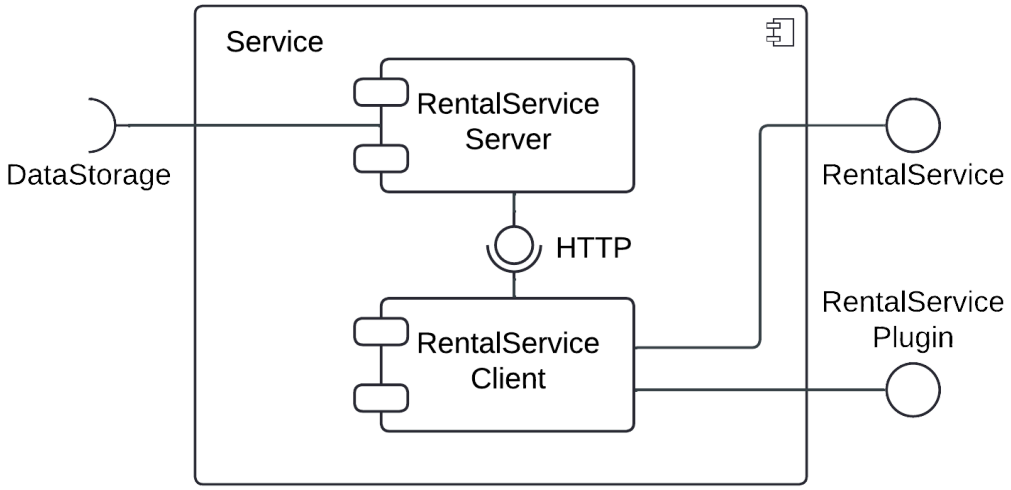

Fig.2: Schema della business logic nella layered architecture.

Per garantire il rispetto dei vincoli architetturali, sono state definite tre **fitness function** utilizzando *ArchUnit*. Questi metodi verificano che gli strati siano correttamente separati e che le dipendenze tra di essi seguano le regole stabilite. In particolare:

- **presentationLayerRule** verifica che il presentation layer dipenda esclusivamente dal business logic layer.
- **businessLayerRule** controlla che il business logic layer dipenda solo dal persistence layer.
- **persistenceLayerRule** assicura che il persistence layer non abbia dipendenze verso alcun altro strato.

Queste regole garantiscono l'indipendenza tra i vari livelli dell'architettura, preservando la modularità e il rispetto dei principi di progettazione.

3.2 Clean Architecture

Il codice per questa versione del software è stato preso e riadattato da quello della layered architecture, riutilizzando la parte di presentation realizzata in Swing e gran parte della business logic. Tuttavia, sono state introdotte delle differenze architetturali significative, poiché in questa seconda versione il problema viene modellato seguendo i principi della **clean architecture**, che adotta una visione domain-centered.

In questo approccio, è la **business logic** a definire le interfacce (*port*), che vengono implementate da componenti esterni al dominio (*adapter*). Per questo caso specifico, la business logic continua a definire le interfacce *RentalService* e *RentalServicePlugin* come nel precedente progetto, ma introduce anche nuove interfacce come *RentalServiceStorage*, per modellare la persistenza dei dati, e *RideSimulator*, per fornire diverse simulazioni delle ride (Fig.3).

Anche in questa versione è stata adottata un'architettura **client-server**, utilizzando le classi *RentalServiceServer* e *RentalServiceClient* (Fig.4). Nel layer degli adapter si trovano le implementazioni delle interfacce, come ad esempio *HashMapStorage* per la persistenza e *RoadSimulationImpl* per la simulazione delle ride.

Questa organizzazione permette di mantenere una netta separazione tra il dominio centrale e i dettagli implementativi.

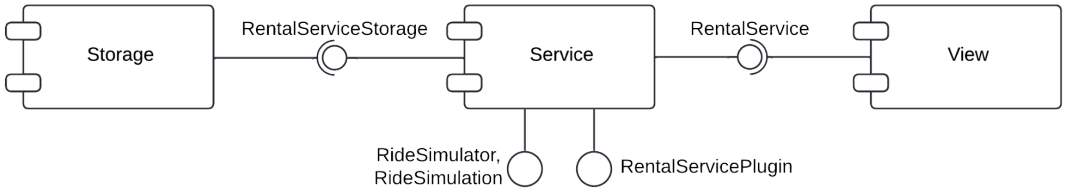

Fig.3: Diagramma dei componenti per la clean architecture.

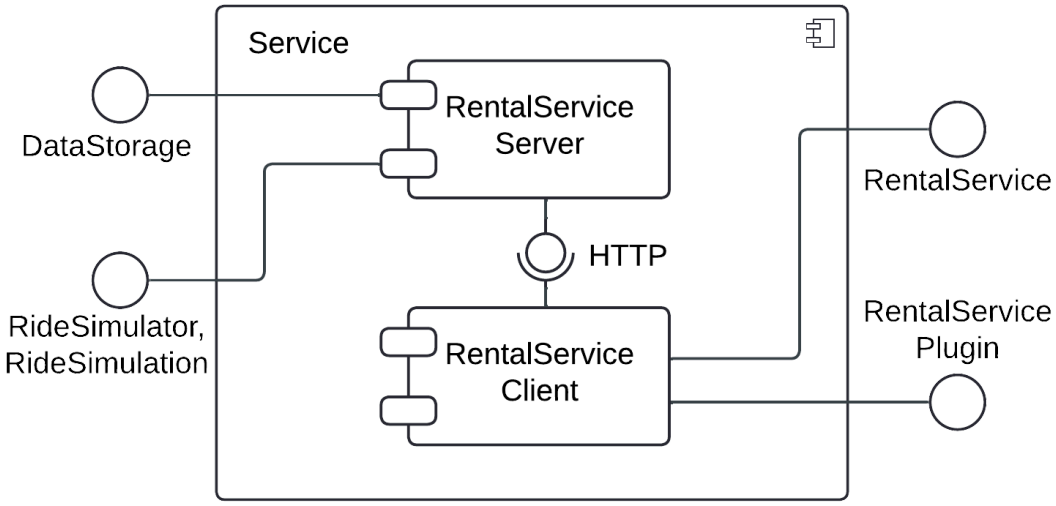

Fig.4: Schema della business logic nella clean architecture.

Sebbene il software sia molto simile a quello del problema precedente, l'architettura è stata modificata, e di conseguenza sono state formulate nuove **fitness function** per garantire il rispetto dei nuovi vincoli architetturali. In particolare, sono state definite due regole:

- **businessDependencyRule**: verifica che la business logic non dipenda in alcun modo dalle implementazioni degli adapter, assicurando così l'indipendenza del dominio centrale rispetto ai dettagli esterni.

- **portsRule**: controlla che le **ports** siano esclusivamente interfacce definite dalla business logic e che vengano implementate solo dagli adapter esterni.

Queste regole garantiscono che la separazione tra il dominio e gli elementi esterni, caratteristica fondamentale della clean architecture, venga rispettata.