UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Laurea Triennale in Ingegneria Informatica

Tesina di Ingegneria del Software

Dichiaro che questo elaborato è frutto del mio personale lavoro, svolto sostanzialmente in maniera individuale e autonoma

Studente:

Francesco Della Casa

Anno Accademico 2023/2024

Indice

Capitolo 1

Introduzione

1.1 Origine del progetto e Ispirazione

QuickTalk nasce dall'esigenza di superare le barriere fisiche e geografiche che ostacolano una comunicazione vocale in tempo reale affidabile. Questa idea è emersa dall'osservazione di come altre piattaforme di collaborazione online, utilizzate nel lavoro remoto e nell'istruzione a distanza, abbiano trasformato le modalità di interazione. Da ciò si è manifestata l'opportunità di facilitare in modo affidabile questi nuovi approcci alla comunicazione.

1.2 Obiettivo

L'obiettivo di QuickTalk è superare le barriere comunicative e rispondere alle esigenze di vari settori per la comunicazione vocale. QuickTalk si propone di promuovere affidabilità, offrendo un'esperienza intuitiva e accessibile a utenti di ogni livello di competenza. Il software è progettato per essere flessibile e adattabile, soddisfacendo le preferenze individuali di ciascun utente. L'applicazione consente di avviare chiamate vocali in tempo reale, sfruttando tecnologie di streaming audio e strumenti di collaborazione online.

1.3 Visione a lungo termine del prodotto

Il futuro di QuickTalk è strettamente connesso all'innovazione tecnologica continua e alla crescente richiesta di applicazioni che offrano comunicazione affidabile a livello globale.

Capitolo 2

Specifica dei Requisiti

2.1 Requisiti funzionali

I requisiti funzionali definiscono cosa deve fare il sistema. Di seguito vengono riportati i requisiti funzionali non banali riguardanti QuickTalk, separando quelli del sistema da quelli dell'utente.

Per quanto riguarda il sistema:

- Memorizza e gestisce i dati dell'utente
- Notifica gli utenti
- Archivia le chiamate

Per quanto riguarda l'utente:

- Effettua accesso all'applicazione
- Inserisce e modifica dati Utente
- Invia e riceve inviti per Chiamate
- Comunica all'interno di una Chiamata
- Avvia la Chiamata
- Termina la Chiamata

2.2 Requisiti non funzionali

I requisiti non funzionali definiscono le modalità di sviluppo del software, specificando gli standard, le qualità e le caratteristiche necessarie affinché il sistema soddisfi le aspettative degli utenti in termini di affidabilità ed efficienza. L'applicazione deve

garantire una trasmissione audio in tempo reale con un ritardo inferiore ai 40 millisecondi, per rendere la chiamata il più simile possibile a una conversazione reale. Lo streaming audio viene elaborato e gestito tramite specifiche API, come OpenTok e Kurento. Il linguaggio utilizzato per lo sviluppo dell'applicazione è principalmente Java, che, grazie alle sue caratteristiche, consente una buona portabilità su diverse piattaforme.

2.3 Use Casa Diagram

Gli schemi UML Use Case sono una tipologia di diagrammi che consentono di rappresentare in modo chiaro le interazioni tra il sistema (QuickTalk) e gli attori coinvolti. Di seguito viene presentato il diagramma Use Case per QuickTalk:

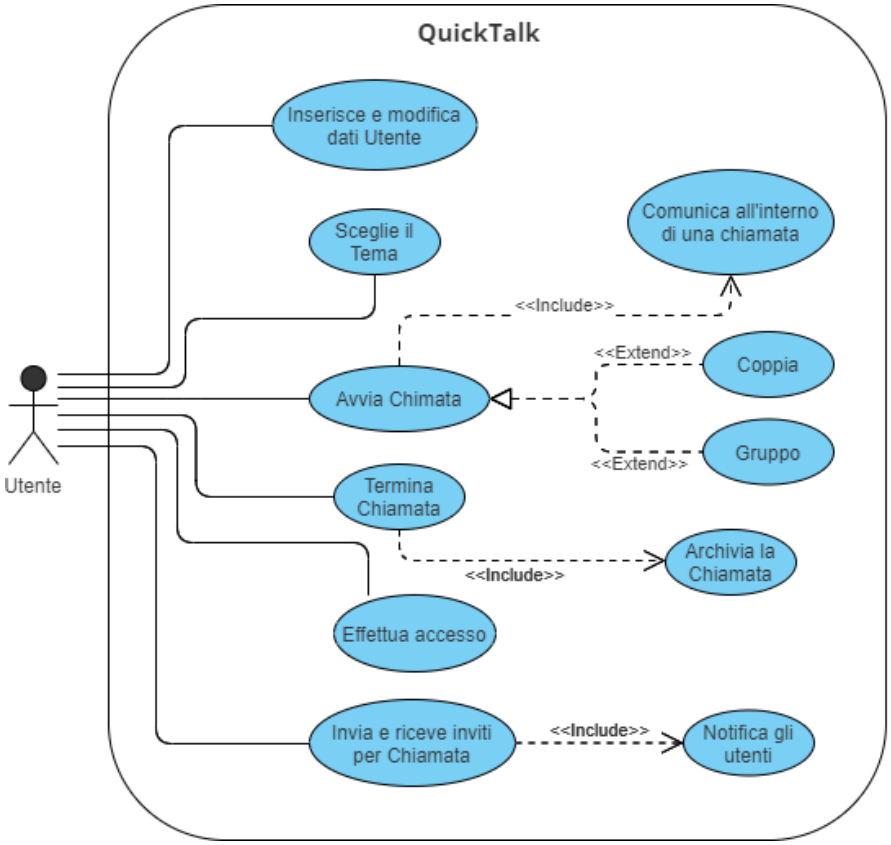

Figura 2.1: UML - Use case Diagram

Capitolo 3

Design del prodotto

QuickTalk adotta un'architettura con un design orientato agli oggetti e modulare, in grado di gestire la complessità del software e ottimizzare la struttura del progetto.

3.1 Class Diagram

Gli schemi UML Class Diagram sono una tipologia di diagrammi UML che rappresentano la struttura statica di un'applicazione sviluppata con un linguaggio orientato agli oggetti. Questi diagrammi descrivono le classi del sistema, inclusi i loro attributi, metodi e le interazioni statiche tra di esse.

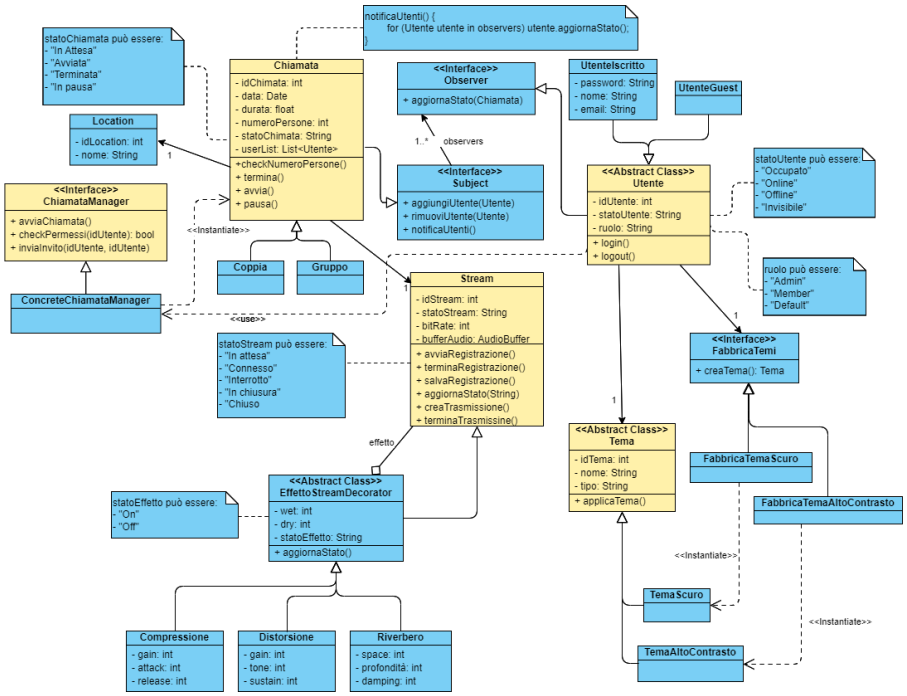

Figura 3.1: UML - Class Diagram

3.1.1 Descrizione Class Diagram

Il diagramma delle classi di QuickTalk offre una struttura statica e funzionale che, grazie all'impiego di design pattern e astrazioni, consente di soddisfare le esigenze degli utenti in modo flessibile, controllato e dinamico. In questo capitolo ci concentreremo esclusivamente sulle classi fondamentali, senza esaminare i design pattern, che saranno trattati in seguito. Non verranno considerati né il comportamento dell'applicazione relativo alla gestione e alla memorizzazione dei dati, né quello riguardante la gestione del canale di trasmissione multimediale.

L'applicazione include diverse classi fondamentali: Chiamata, ChiamataManager, Utente, Tema e Stream. La classe principale è Chiamata, che gestisce gli attributi e i comportamenti associati a una chiamata, supportata dalle classi Location, Utente e Stream. Una Chiamata può coinvolgere sia una coppia di utenti sia un gruppo. L'interfaccia ChiamataManager stabilisce le operazioni necessarie per avviare una chiamata. La classe Stream si occupa di gestire in modo semplificato il flusso del processo di streaming audio. La classe astratta Utente fornisce una struttura comune per le classi concrete UtenteGuest e UtenteIscritto, differendo in quanto un UtenteGuest non dispone di dati personali come password, nome o email, a differenza di un UtenteIscritto. Infine, la classe astratta Tema definisce le caratteristiche grafiche di base dell'applicazione, con esempi come un tema scuro o un tema ad alto contrasto.

3.1.2 Rispetto dei Principi SOLID

I principi SOLID sono un insieme di regole e linee guida per lo sviluppo di un software attraverso un linguaggio di programmazione orientato agli oggetti. L'obiettivo è quello di rendere il codice più comprensibile, flessibile e manutenibile. È fondamentale tenere in considerazione il contesto in cui questi principi di programmazione vengono applicati. Analizzando l'applicazione QuickTalk, si osserva che tali principi sono stati rispettati in modo corretto.

La classe Chiamata è dedicata esclusivamente alla gestione delle informazioni relative a una comunicazione, in linea con il principio di singola responsabilità (SRP). La classe Tema stabilisce le operazioni fondamentali che un tema può eseguire, rispettando il principio open-closed (OCP); ciò consente di creare nuovi temi senza dover modificare la classe stessa. L'uso di astrazioni, come la classe astratta Utente e l'interfaccia ChiamataManager, riduce la complessità delle dipendenze nel design dell'applicazione (DIP). In questo modo, i moduli di alto e basso livello si basano su astrazioni, promuovendo una struttura di dipendenze che favorisce il decoupling. Inoltre, la classe Utente aderisce al principio di sostituzione di Liskov (LSP), il quale afferma che gli oggetti di una superclasse dovrebbero poter essere sostituiti con oggetti di una sottoclasse senza compromettere il corretto funzionamento dell'applicazione. Infine, l'implementazione di astrazioni specifiche, come la classe

ChiamataManager, riflette una corretta applicazione del principio di segregazione delle interfacce (ISP), suggerendo che le sottoclassi che le implementano non saranno costrette a dipendere da metodi non necessari.

3.2 Activity Diagram

Gli Activity Diagram UML sono uno strumento utilizzato per rappresentare il flusso di lavoro o le attività di un sistema. Infatti aiutano a visualizzare il flusso delle operazioni e le dipendenze, facilitando l'analisi e la comunicazione tra sviluppatori e stakeholder. Di seguito, viene descritto il processo relativo alla gestione di una Chiamata da parte di un utente all'interno dell'applicazione QuickTalk.

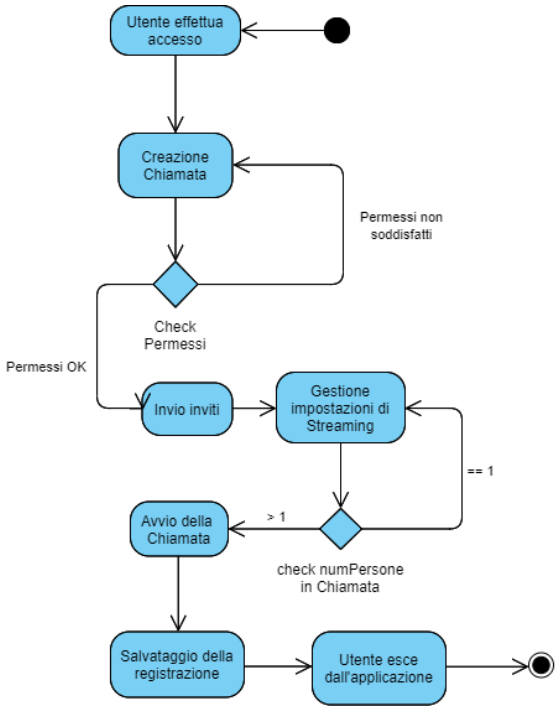

Figura 3.2: UML - Activity Diagram

3.3 Sequence Diagram

I Sequence Diagram sono una tipologia di schemi UML utilizzati per descrivere il flusso di interazioni tra oggetti di un sistema, offrendo una visione chiara dell'ordine temporale delle operazioni scambiate. Questi diagrammi sono utili per una migliore comprensione del comportamento dinamico di un software e per identificare le responsabilità dei vari oggetti. Di seguito viene analizzato il processo di creazione di una Chiamata all'interno dell'applicazione QuickTalk.

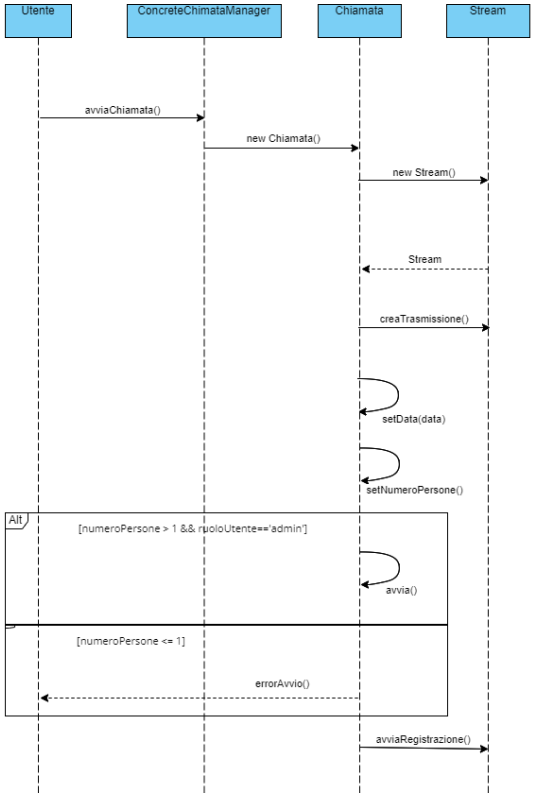

Figura 3.3: UML - Sequence Diagram

Capitolo 4

Design Pattern

I design pattern sono soluzioni generali, affidabili e riutilizzabili per problemi comuni che si manifestano durante la progettazione di software. Offrono linee guida su come affrontare specifici problemi di progettazione in modo efficace. Di seguito, viene presentata un'analisi dei design pattern utilizzati nella progettazione dell'applicazione QuickTalk, partendo dal diagramma delle classi.

4.1 Observer Pattern

L'Observer Pattern è un design pattern comportamentale che consente di mantenere aggiornati gli oggetti quando si verifica un evento che potrebbe interessarli. Nel nostro caso, il soggetto osservabile è la Chiamata. Quando la Chiamata cambia stato, ad esempio durante un aggiornamento, notificherà automaticamente tutti gli utenti (osservatori) "connessi". Gli osservatori, a loro volta, potranno aggiornare il proprio stato interno in risposta alla notifica ricevuta. Il disaccoppiamento tra soggetti e osservatori permette di apportare modifiche senza influire sugli altri componenti, facilitando anche la scalabilità e consentendo l'aggiunta di nuovi osservatori senza alterare la logica esistente.

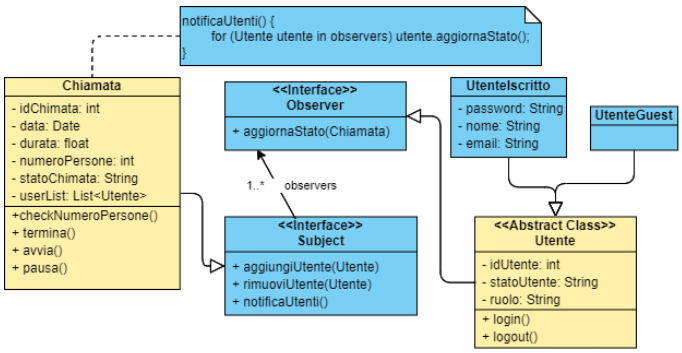

Figura 4.1: DP - Observer

4.2 Strategy Pattern

Lo Strategy Pattern è un design pattern comportamentale che consente di separare l'algoritmo dalla sua implementazione, permettendo di cambiare facilmente la strategia scelta senza dover modificare il codice che la utilizza. Considerando un oggetto di tipo Tema, questo pattern favorisce il disaccoppiamento dei componenti, promuove il principio open-closed e migliora la riusabilità del codice. Ad esempio, per aggiungere un nuovo tema, è sufficiente fornire una nuova implementazione della classe astratta, senza dover alterare le classi esistenti. Il principale vantaggio di questo pattern è che consente all'Utente di scegliere il tipo di tema da utilizzare.

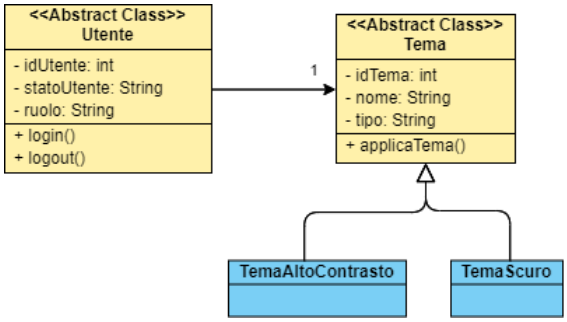

Figura 4.2: DP - Strategy

4.3 Factory Pattern

Il Factory Pattern è un design pattern creazionale che fornisce un'interfaccia per creare oggetti nella superclasse, delegando alle sottoclassi la decisione su quale tipo di oggetto concreto istanziare. Nel nostro contesto, un utente deve avere un tema, che verrà creato tramite il metodo factory creaTema(), definito nell'interfaccia FabbricaTemi e implementato dalle classi concrete FabbricaTemaScuro e FabbricaTemaAltoContrasto. Il Factory Pattern segue l'inversione delle dipendenze e il principio open-closed, permettendo di aggiungere nuovi temi, come FabbricaTemaChiaro, senza modificare il codice esistente, rendendo il sistema più flessibile.

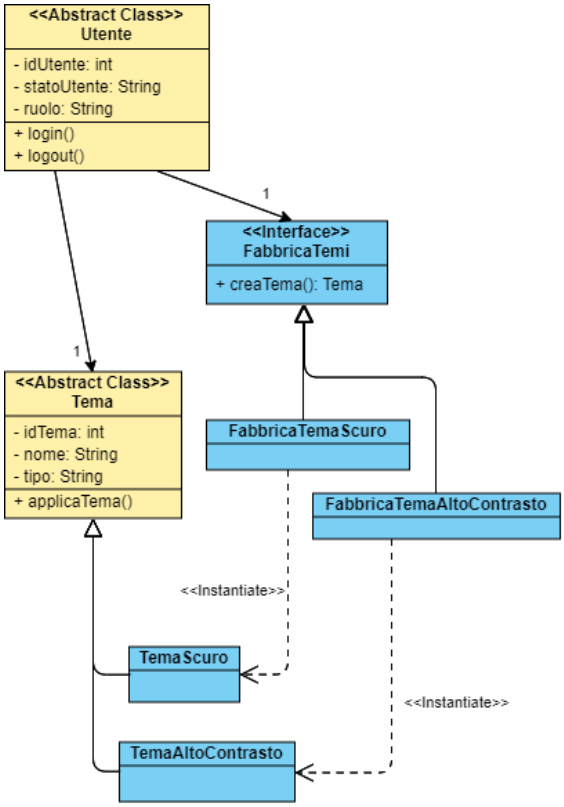

Figura 4.3: DP - Factory

4.4 Decorator Pattern

Il Decorator Pattern è un design pattern strutturale utilizzato per aggiungere dinamicamente nuove responsabilità agli oggetti. I decoratori offrono un'alternativa flessibile alle sottoclassi per estendere funzionalità e comportamenti. Nel caso dello Stream, è possibile applicare "plug-in" che aggiungono effetti al suono trasmesso, migliorando la qualità dell'audio in situazioni di connessione di scarsa qualità.

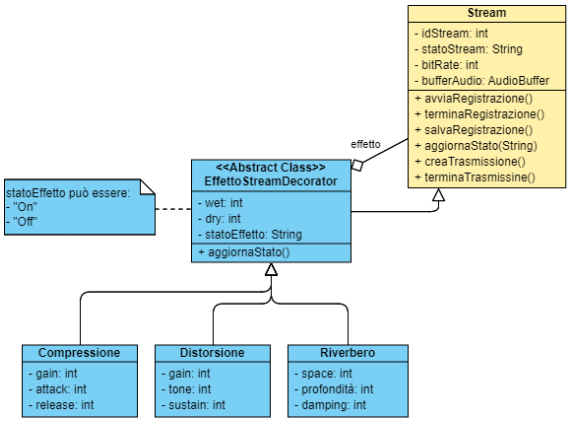

Figura 4.4: Enter Caption