A.A 2024/2025

RELAZIONE PROGETTO
METODOLOGIE DI PROGRAMMAZIONE

Matteo Pascuzzo
7072913

DESCRIZIONE DEL SISTEMA

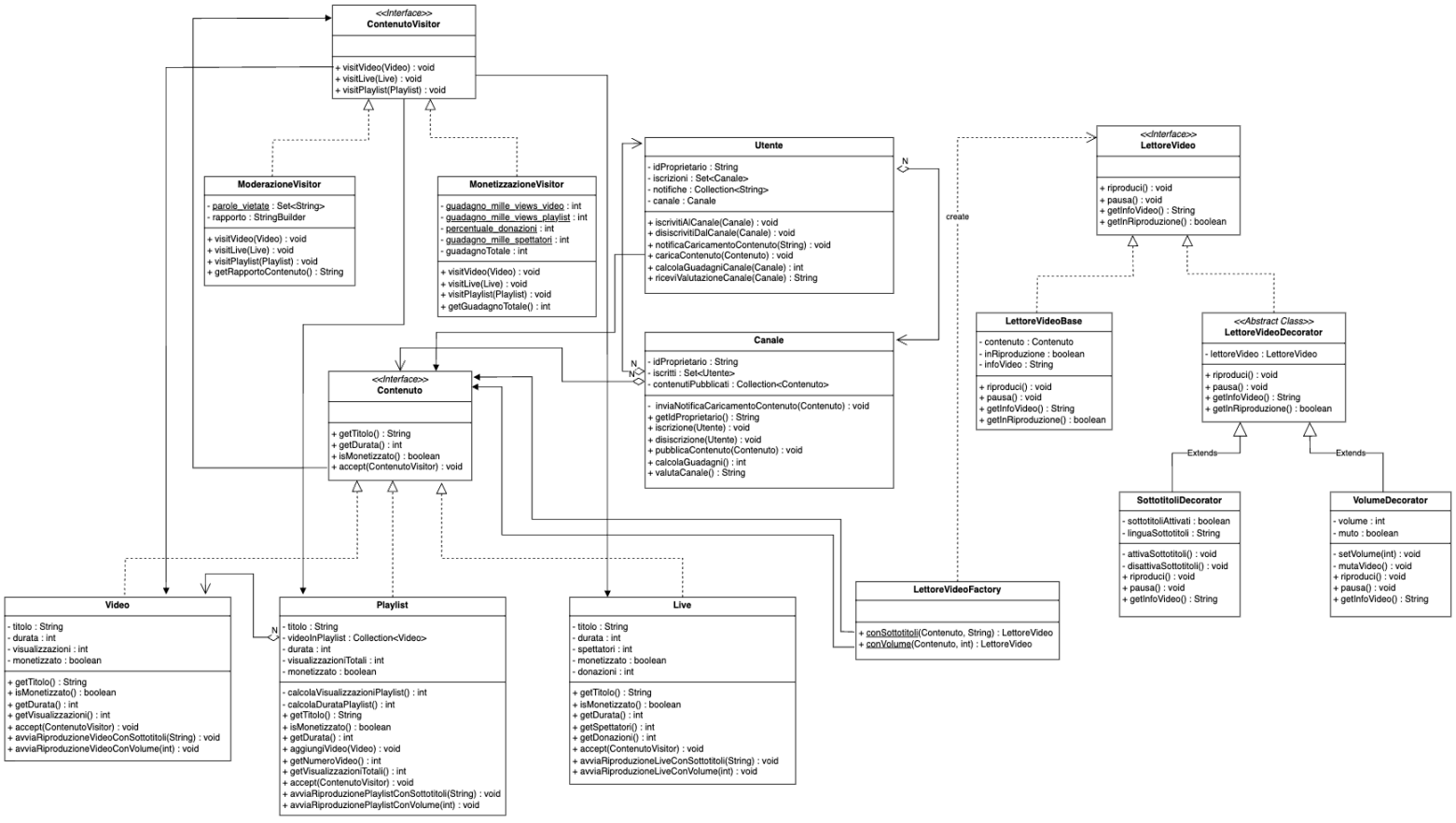
Il progetto rappresenta un prototipo di piattaforma per la condivisione di contenuti video che si ispira in forma molto basilare alla business logic della celeberrima piattaforma YouTube, implementando un piccolo sistema per il caricamento, la gestione, monetizzazione e moderazione di contenuti multimediali.
L'idea principale del progetto è quella di avere un sistema basato su canali, ovvero degli spazi personali dove ogni utente può pubblicare diversi tipi di contenuti. Infatti quando un utente si iscrive alla piattaforma avrà di default un canale nel quale potrà, se lo desidera, caricare dei contenuti.
La piattaforma supporta tre principali tipologie di contenuto multimediale, ciascuno con diverse caratteristiche: video, live (dirette streaming) e playlist, ovvero aggregati di video.
Il sistema offre due funzionalità di particolare interesse: controllare i guadagni ricevuti dal proprio canale e ricevere dei resoconti riguardanti la moderazione del proprio canale.
Il sistema di monetizzazione infatti implementa una logica che si adatta in base a ciascun tipo di contenuto mentre il sistema di moderazione consegna dei resoconti all'utente riguardanti la sicurezza e/o la violazione di politiche proprie della piattaforma.
È inoltre presente un sistema di notifiche che mantiene gli iscritti ad un determinato canale allineati con le pubblicazioni del canale di interesse, al quale si sono iscritti. Infatti quando un utente pubblica un contenuto tutti gli iscritti ricevono automaticamente una notifica.
Infine è stato implementato anche un lettore video, il quale permette di aggiungere sottotitoli e volume durante la riproduzione di contenuti. Tale lettore viene usato quando l'utente decide di avviare la riproduzione di un contenuto.

DESIGN PATTERNS APPLICATI

I design patterns applicati all’interno del progetto sono i seguenti:

- Observer
- Decorator
- Visitor
- Static Factory Methods

UML PROGETTO *



*è presente una copia all’interno del progetto

SCELTE DI DESIGN E IMPLEMENTATIVE

Observer

È stato deciso di usare il design pattern Observer per implementare la parte del sistema incaricata della gestione delle notifiche.
I partecipanti di questo pattern sono la classe “Canale” e la classe “Utente”. Si noti come non è richiesta la presenza di un’interfaccia comune né per i canali né per gli utenti poiché non è stata prevista la presenza di tipi specializzati per queste due classi.
I canali agiscono come Subject, ovvero oggetti da osservare e dei quali è interessante lo stato, mentre gli utenti iscritti agiscono da Observers, i quali ricevono una notifica quando qualcosa nello stato del Subject cambia e provvedono ad allinearsi. Attraverso i metodi “iscrivitiAlCanale()” e “disiscrivitiDalCanale()” il singolo utente può decidere di quali canali interessarsi e dai quali ricevere notifiche.
Quando un canale pubblica un contenuto il metodo “inviaNotificheCaricamentoContenuto(Contenuto contenuto)” itera su tutti gli iscritti al canale e li notifica della pubblicazione del contenuto. Conseguentemente tale notifica viene aggiunta alla lista di notifiche ricevute dall’utente in modo tale che essi possano visualizzarla.

Visitor

Il design pattern Visitor è stato applicato per gestire le operazioni su diversi tipi di contenuto multimediale. Infatti si è reputato necessario che il sistema dovesse supportare operazioni sulla gerarchia di oggetti di tipo “Contenuto” senza sporcare le classi “Video”, “Live” e “Playlist” aggiungendo logiche che non le riguardano.
Le funzionalità che si è deciso di implementare sono quelle di poter permettere ad un utente di vedere quanti guadagni sta ricevendo dal proprio canale e di ricevere una valutazione per quanto riguarda la sicurezza dei contenuti del proprio canale.
Invece di aggiungere tali metodi alle rispettive classi, si è deciso di applicare il design pattern Visitor così da rispettare il Single Responsibility Principle.

I partecipanti del pattern sono l'interfaccia "ContenutoVisitor" e le due classi concrete che la implementano: "MonetizzazioneVisitor" e "ModerazioneVisitor". È stato ritenuto più appropriato implementare il Visitor nella sua versione con i metodi che hanno void come tipo di ritorno e quindi i visitor concreti mantengono uno stato che poi provvederanno a restituire tramite metodi getter. Altro motivo per il quale si è deciso di usare questa soluzione è il fatto che essa garantisce flessibilità, ovvero se un giorno si decidesse di aggiungere un'altra funzionalità ciò non richiederebbe la modifica delle classi concrete di Contenuto, il che permette il rispetto dell'Open-Closed Principle.

Decorator

Quando si è pensato di implementare un prototipo di sistema per la riproduzione dei video è stato naturale pensare di utilizzare il design pattern Decorator. La scelta di questo pattern deriva dalla necessità di avere un lettore video di base (che banalmente riproduce un contenuto) che possa essere arricchito con funzionalità opzionali proprio come quando aggiungiamo i sottotitoli ad un video o ne gestiamo il volume. Se non usassimo questo pattern avremmo un'esplosione del numero di sottoclassi che rappresentano tutte le combinazioni delle varie funzionalità.
I partecipanti del pattern sono: "LettoreVideoDecorator" che mantiene un riferimento al "LettoreVideo" che sta decorando, la classe concreta "LettoreVideoDiBase" e i due decoratori concreti "SottotitoliDecorator" e "VolumeDecorator". Tramite il double-dispatch ogni decorator può aggiungere del comportamento prima o dopo aver delegato l'operazione al componente decorato.

Static Factory Methods

L'idea dietro l'implementazione del lettore video è quella di usare i lettori video nel momento in cui l'utente decide di riprodurre un contenuto. La responsabilità di riprodurre i contenuti è stata lasciata ai singoli contenuti concreti i quali hanno però bisogno di creare un'istanza di un lettore video per poi iniziare la riproduzione del contenuto tramite il relativo metodo riproduci() presente nel decoratore. È stato quindi deciso di usare Static Factory Methods per gestire la creazione di oggetti di tipo

LettoreVideo ed evitare l'uso di istruzioni "new" all'interno dei contenuti concreti, cosa che avrebbe accoppiato i contenuti concreti con il lettore video rendendo le classi tightly coupled.
I metodi sono stati implementati in una classe apposita per evitare di "inquinare" le classi che si occupano della creazione dei decoratori. Infatti se i metodi fossero stati inseriti nella classe "LettoreVideoBase" allora tale classe avrebbe avuto due responsabilità diverse: implementare la logica propria dei lettori oltre che conoscere tutti i decoratori concreti (quindi dipendere da essi) e questo avrebbe violato il Dependency Inversion Principle.
Problemi simili sarebbero stati riscontrati anche se i metodi statici fossero stati divisi tra la classe che implementa il lettore video base e le classi che implementano i decoratori concreti, infatti così facendo i decoratori concreti avrebbero dovuto sapere come creare un lettore video e crearne un'istanza e quindi i decoratori avrebbero dovuto conoscere tutti i decorati, non solo, ogni decoratore concreto dovrebbe conoscere tutti gli altri nel caso in cui si voglia esporre un modo per creare lettori video con più decorazioni. Per mantenere il rispetto del Single Responsibility Principle si è deciso di evitare anche questa implementazione.
La conseguenza di questa scelta è che il costruttore della classe "LettoreVideo" non può essere reso privato cioè non se ne può impedire l'uso per creare delle istanze della classe. Questo difetto viene però mitigato dalla flessibilità offerta dal pattern nel caso di estensioni future.

STRATEGIE PER TESTARE

In seguito si mostrano le strategie usate per effettuare i test del sistema.

package model:

- Classe CanaleTest
    - testInviaNotificaCaricamentoVideo() → viene creato un utente con la propria lista di iscrizioni e notifiche, viene creato un canale con la sua lista di iscritti e con il rispettivo possessore creator il quale pubblicherà

due video. Il metodo viene testato iscrivendo l'utente al canale, facendo pubblicare un
video dal creator e verificando che l'utente riceva la corretta notifica. Viene inoltre testato il comportamento nel caso in cui l'utente si disiscrive dal canale di interesse, il creator pubblica un video e l'utente giustamente non riceve la notifica.
    - testCalcolaGuadagniTuttoMonetizzato() → viene testato il comportamento del MonetizzazioneVisitor nel caso tutti i contenuti siano monetizzati. Vengono quindi creati gli oggetti da testare, aggiunti ad un canale e si verifica che il valore numerico restituito dal metodo sia quello aspettato.
    - testValutaCanale() → viene testato il comportamento del ModerazioneVisitor nel caso in cui il canale contenga tutti i tipi di oggetto. Vengono creati gli oggetti da testare, aggiunti ad un canale verificando che la stringa restituita sia quella aspettata.

- Classe UtenteTest

    - testIscrivitiAlCanaleNonAncoraIscritto() → viene creato un utente con la sua lista di iscrizioni e viene creato un canale con la sua lista di iscritti e di contenuti. Viene chiamato il metodo per iscrivere l'utente al canale e si verifica che esso sia presente nella lista degli utenti iscritti al canale.
    - testIscrivitiAlCanaleGiaIscritto() → come il test precedente ma ci si assicura che l'utente non venga iscritto due volte allo stesso canale.
    - testDisiscrivitiDalCanaleIscritto() → si controlla che iscrivendo un utente ad un canale e poi disiscrivendolo esso sia effettivamente rimosso dalla lista degli iscritti al canale.
    - testDisiscrivitiDalCanaleNonIscritto() → come il test precedente ma si controlla che disiscrivendo un utente da un canale al quale non è iscritto non si manifestino comportamenti scorretti.
    - testCaricaContenuto() → ci si assicura che quando viene caricato un contenuto sul proprio canale esso sia aggiunto alla lista dei contenuti del canale. Se l'id dell'utente è diverso dall'id del canale allora l'utente riceve un messaggio nelle notifiche.

package lettori:

- Classe LettoreVideoBaseTest
    - testRiproduci() → vengono creati un video ed un lettoreVideo e ci si assicura che chiamando il metodo riproduci() il valore del campo “inRiproduzione” sia true.
    - testPausa() → vengono creati un video ed un lettoreVideo, ci si assicura che chiamando il metodo riproduci() il valore del campo “inRiproduzione” sia true e poi che chiamando il metodo pausa() il valore del campo sia false.
    - testGetInfoVideo() → vengono creati un video ed un lettoreVideo e ci si assicura che chiamando il metodo getInfoVideo() si ottenga la stringa desiderata.

- Classe SottotitoliDecorator
    - testGetInfoSottotitoliAttivati() → si creano gli stessi oggetti dei test sul LettoreVideoBase ma si aggiunge il decoratore responsabile dell’aggiunta dei sottotitoli e si controlla che il metodo getInfoVideo() restituisca la stringa corretta.
    - testGetInfoSottotitoliDisattivati() → come il test precedente ma si controlla che questa volta i sottotitoli siano disattivati.

- Classe VolumeDecorator
    - testGetInfoNonMuto() → si creano gli stessi oggetti dei test sul LettoreVideoBase ma si aggiunge il decoratore responsabile dell’aggiunta del volume e si controlla che il metodo getInfoVideo() restituisca la stringa corretta.
    - testGetInfoMuto() → come il test precedente ma si controlla che il video abbia l’audio disattivato.