Specifica Tecnica

ShowRoom3D

SmokingFingertips | smoking.fingertips@gmail.com

Versione	1.0.0
Stato	approvato
Uso	esterno
Responsabile	Edoardo Gasparini
Redattori	Luca Polese
	Alberto Angeloni
	Luca Annicchiarico
	Davide Baggio
Verificatori	Alberto Angeloni
	Edoardo Gasparini
	Gabriele Saracco
	Sebastien Biollo
	Luca Annicchiarico
Destinatari	*Smoking Fingertips*
	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
Data di Approvazione	2023-04-14
Anno accademico:	2022/2023

Sommario:

Descrizione delle tecnologie e dell'architettura del prodotto *ShowRoom3D*$_G$.

Storico delle Modifiche

Versione	Data	Nominativo	Ruolo	Descrizione
1.0.0	2023-04-14	Edoardo Gasparini	Responsabile	Approvazione del documento
0.2.1	2023-04-01	Luca Annicchiarico *Sebastien Biollo*	Progettista *Verificatore*	Stesura §4,
0.2.0	2023-03-31	Alberto Angeloni *Luca Annicchiarico*	Progettista *Verificatore*	Revisione complessiva di coerenza e coesione
0.1.2	2023-03-23	Luca Polese *Luca Annicchiarico*	Progettista *Verificatore*	Stesura §3.3, §3.4
0.1.1	2023-03-18	Luca Polese *Sebastien Biollo*	Progettista *Verificatore*	Stesura §3.1 Aggiornamento §3.2
0.1.0	2023-03-15	Davide Baggio *Gabriele Saracco*	Progettista *Verificatore*	Revisione complessiva di coerenza e coesione
0.0.2	2023-03-10	Alberto Angeloni *Edoardo Gasparini*	Progettista *Verificatore*	Stesura introduzione alla sezione §3
0.0.1	2023-02-28	Luca Polese *Alberto Angeloni*	Progettista *Verificatore*	Stesura §1, §2

Indice

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Questo documento verrà utilizzato dal gruppo *Smoking Fingertips* allo scopo di fornire una panoramica dell'architettura del prodotto che verrà sviluppato, fornire delle informazioni per l'estensione del progetto e per descrivere le procedure per l'installazione e lo sviluppo in locale.
In particolare, il documento `Specifica Tecnica` illustra:

- L'architettura logica del sistema: una visione ad alto livello della struttura del sistema, che identifica i componenti principali e le relazioni tra di essi;
- Le tecnologie utilizzate: le tecnologie e le librerie di terze parti utilizzate nella realizzazione del sistema.
- I design pattern architetturali utilizzati, e quelli determinati dalle tecnologie adottate
- Gli idiomi, ovvero pattern di livello più basso che architetturale

1.2 Scopo del progetto

Il capitolato C6 *ShowRoom3D*$_G$ affidato al team si prefigge come scopo quello di realizzare uno showroom virtuale. L'utente accedendo all'applicazione sarà in grado di muoversi nello spazio visionando gli oggetti esposti. Ognuno degli elementi potrà essere configurato secondo le preferenze dell'utente. Una volta operata la scelta dei parametri, sarà altresì possibile aggiungere l'articolo modificato all'interno del carrello per eventuali acquisti.

1.3 Glossario

Per evitare possibili ambiguità che potrebbero sorgere durante la lettura dei documenti, alcuni termini utilizzati sono stati inseriti nel documento `Glossario` (che attualmente è nella sua versione *2.0.0*).
Il Glossario rappresenta una raccolta delle definizioni dei termini più rilevanti che hanno un significato particolare. Sarà possibile individuare il riferimento al Glossario per mezzo di una G a pedice del termine (esempio *way of working*$_G$).

1.4 Riferimenti

1.4.1 Riferimenti normativi

- `Norme di Progetto`
- **Capitolato d'appalto C6 - ShowRoom3D**: https://www.math.unipd.it/~tullio/IS-1/2022/Progetto/C6.pdf

1.4.2 Riferimenti informativi

- `Analisi dei Requisiti`
- **Beginning C++20 - From Novice to Professional**:
  https://galileodiscovery.unipd.it/permalink/39UPD_INST/prmo4k/alma9939902434906046
- **Beginning Unreal Engine 4 Blueprints Visual Scripting - Using C++: From Beginner to Pro**:
  https://galileodiscovery.unipd.it/permalink/39UPD_INST/prmo4k/alma9939903389606046
- **Model-View Patterns**:
  https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf
  https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
- **Design Pattern Comportamentali**:
  https://www.math.unipd.it/~rcardin/swea/2021/Design%20Pattern%20Comportamentali_4x4.pdf
- **Architettura di Unreal Engine**:
  https://docs.unrealengine.com/5.0/en-US/programming-in-the-unreal-engine-architecture/

Tutti i riferimenti a siti web sono verificati e aggiornati alla data 2023-04-15.

2 Tecnologie

In questa sezione vengono definiti gli strumenti e le tecnologie utilizzate per sviluppare e implementare il software del progetto *ShowRoom3D*$_G$. Si descriveranno pertanto le tecnologie e i linguaggi di programmazione utilizzati, le librerie e i framework necessari e le infrastrutture richieste. L'obiettivo principale è quello di garantire che il software sia realizzato con le tecnologie più appropriate e siano scelte le opzioni migliori in termini di efficienza, sicurezza e affidabilità.

2.1 Unreal Engine

Unreal Engine$_G$ è un motore di gioco *cross-platform*$_G$, che offre un insieme di strumenti e tecnologie per la creazione di videogiochi e applicazioni interattive in tempo reale. La tecnologia Unreal Engine è composta da un insieme di librerie di programmazione e strumenti di sviluppo, come l'editor grafico *Unreal Editor*$_G$, che consentono di creare scenari, personaggi, animazioni e di implementare la fisica del gioco e gli effetti speciali.
Versione utilizzata: 5.0.3
Link download: https://www.unrealengine.com/en-US/download
Vantaggi:

- **Prestazioni elevate**: Unreal Engine è stato progettato per offrire prestazioni elevate e sfruttare al massimo le risorse hardware disponibili. Ciò significa che il motore di gioco può gestire ambienti 3D complessi, effetti grafici avanzati e molti altri elementi interattivi, senza compromettere le prestazioni.
- **Facilità di utilizzo**: Unreal Engine è dotato di un'interfaccia utente intuitiva e di una documentazione completa, che lo rende facile da usare anche per sviluppatori meno esperti. Inoltre, Unreal Engine offre anche una vasta gamma di strumenti di sviluppo, come l'editor di livelli, il debugger e il profiler, che semplificano il processo di sviluppo e di debug.
- **Versatilità**: Unreal Engine è un motore di gioco molto versatile che può essere utilizzato per lo sviluppo di videogiochi, ma anche di altre applicazioni interattive come la realtà virtuale e aumentata. Ciò significa che Unreal Engine offre molte possibilità di sviluppo a seconda delle esigenze del progetto.
- **Portabilità**: Unreal Engine è disponibile su diverse piattaforme pertanto gli sviluppatori possono creare applicazioni multi-piattaforma utilizzando lo stesso motore di gioco. Ciò può ridurre notevolmente i costi di sviluppo e di manutenzione, e migliorare l'efficienza del processo di sviluppo.

2.2 C++

C++$_G$ è un linguaggio di programmazione multiparadigma, che supporta la programmazione ad oggetti, la programmazione generica, la programmazione funzionale e la programmazione ad eventi.
Versione utilizzata: 20
Vantaggi:

- **Prestazioni**: C++ è un linguaggio di basso livello che permette di scrivere codice ottimizzato e di sfruttare al massimo le risorse hardware disponibili. Questo rende C++ una scelta ideale per lo sviluppo di applicazioni a elevate prestazioni come i videogiochi.
- **Integrazione con Unreal Engine**: *Unreal Engine*$_G$ è stato sviluppato in C++ e quindi utilizzando lo stesso linguaggio di programmazione per scrivere il codice dello showroom. Sarà possibile sfruttare al meglio le funzionalità del motore di gioco, migliorando la velocità di sviluppo e la qualità del prodotto finale.
- **Portabilità**: C++ è un linguaggio di programmazione *cross-platform*$_G$, il che significa che il codice scritto in C++ può essere compilato e utilizzato su diverse piattaforme, tra cui *Windows*$_G$ e *Linux*$_G$. Ciò rende C++ un'ottima scelta per lo sviluppo di applicazioni *multi-piattaforma*$_G$ come i videogiochi.

2.3 Blueprint

Blueprint$_G$ è un linguaggio di programmazione visuale, che permette di creare logiche di gioco e di interagire con gli oggetti del gioco senza scrivere codice. Questo linguaggio è basato su un grafo a nodi che vengono collegati fra di loro da sinistra a destra.

- **Facilità di utilizzo**: i Blueprints sono un sistema di programmazione visuale che utilizza un'interfaccia drag-and-drop, rendendo il processo di sviluppo più intuitivo e accessibile anche ai non programmatori. Ciò significa che membri del team con competenze diverse possono partecipare al processo di sviluppo, permettendo di sviluppare rapidamente prototipi e iterare sulle funzionalità.
- **Velocità di sviluppo**: grazie all'interfaccia visuale dei Blueprints, il tempo necessario per creare funzionalità e logiche di gioco può essere ridotto notevolmente rispetto alla scrittura di codice tradizionale. Ciò consente di sviluppare rapidamente prototipi e di effettuare modifiche durante il processo di sviluppo, senza dover attendere tempi di compilazione e di esecuzione del codice.
- **Possibilità di creare logiche complesse**: pur essendo un sistema di programmazione visuale, i Blueprints consentono di creare logiche di gioco complesse grazie alla possibilità di combinare diversi nodi e funzioni. Ciò significa che è possibile creare una vasta gamma di funzionalità, anche per progetti di grandi dimensioni.

3 Architettura di sistema

Per descrivere l'architettura del sistema adotteremo un approccio top-down, partendo da una definizione più generale delle componenti per poi scendere nel dettaglio di ciascuna di esse. Il sistema permette di:

1. Navigare lo showroom;
2. Interagire con gli oggetti in esposizione;
3. Gestire i prodotti presenti nel carrello.

Con **navigazione** dello showroom, si intendono le operazioni che consentono all'utente di spostarsi nello spazio dello showroom utilizzando la tastiera e il mouse.
Con **interazione** con gli oggetti, si intendono le funzionalità di cui l'utente usufruisce per:

- visualizzare la scheda tecnica di un prodotto della collezione scelto e visualizzare i parametri modificabili;
- modificare le proprietà dell'oggetto visualizzato (nello specifico il colore);
- aggiungere il prodotto al carrello (o rimuovere gli ultimi elementi aggiunti);
- spostare l'oggetto per accostarlo a dei mobili disposti nello showroom come esempio.

Con **gestione** del carrello s'intendono tutte quelle operazioni che consentono all'utente di utilizzare il carrello per l'aggiunta, la rimozione e la visualizzazione dei prodotti di maggior interesse.

3.1 Attori presenti nel sistema

3.1.1 FirstPersonCharacter

Nel contesto del progetto didattico, l'attore `FirstPersonCharacter` rappresenta il personaggio controllato dall'utente stesso. L'utente è immerso nell'esperienza in prima persona e può muoversi all'interno dello showroom, interagire con gli oggetti presenti e visualizzare i dettagli dei prodotti esposti. Le principali caratteristiche del FirstPersonCharacter includono la possibilità di spostarsi avanti, indietro, a destra e a sinistra e di ruotare la visuale offrendo una vista panoramica a 360 gradi dell'ambiente circostante. Inoltre, l'attore è in grado di interagire con gli oggetti presenti nella showroom, ad esempio selezionando un prodotto per visualizzarne i dettagli premendo il pulsante `E` o visualizzare il carrello premendo il tasto `Q`. Grazie alle sue operazioni a disposizione dell'attore l'utente può fruire dello showroom in modo intuitivo e semplice.

3.1.2 Chair

L'attore `Chair` rappresenta genericamente le sedie esposte all'interno dello showroom. Le principali funzionalità di Chair includono la possibilità di ruotare e di posizionarsi nello spazio all'avvio dell'applicazione consentendo una visione completa del prodotto esposto. L'asset Chair può aggiornare le caratteristiche della sedia, come il colore, al click dell'utente dei bottoni nell'interfaccia, garantendo un'esperienza di interazione personalizzata e coinvolgente. Chair è in grado di attivare o nascondere il testo per favore l'interazione dell'utente, fornendo un feedback visivo immediato.

3.2 Modello architetturale

Il sistema è progettato seguendo il pattern architetturale `MVC`(*Model-View-Controller*$_G$) che si occupa di separare la gestione dei dati (modello) dalla loro visualizzazione (vista) e dal controllo dell'applicazione (controller). Le tre componenti in cui è suddivisa l'applicazione sono interconnesse e hanno l'obiettivo di migliorare l'organizzazione e la manutenibilità del codice, riducendone la complessità: ogni parte del sistema agisce indipendentemente dalle altre. È stato adottato questo pattern, in quanto permette di migliorare la separazione delle responsabilità nel design dell'applicazione.
La UI rappresenta l'elemento centrale del sistema, attraverso la quale, l'utente potrà usufruire di tutte le funzionalità e potrà visualizzare le informazioni messe a disposizione dal sistema. Il design pattern si articola in:

- **Model** (che in seguito chiameremo anche modello): rappresenta i dati e la logica di business dell'applicazione. Interagisce con i dati (memorizzati in locale) aggiornandoli. Il Modello è responsabile della gestione dei dati e delle regole che governano come si accede ai dati e come essi devono essere manipolati. È la base dell'applicazione ed è responsabile del mantenimento della coerenza e dell'integrità dei dati;

- **View** (che in seguito chiameremo anche vista): rappresenta l'interfaccia utente dell'applicazione. La Vista è responsabile della presentazione dei dati in un formato che l'utente può facilmente capire e con i quali può interagire;

- **Controller**: gestisce l'input dell'utente e aggiorna sia il modello che la vista. Fa da ponte tra il modello e la vista e gestisce il flusso di dati tra di loro. Il Controller è responsabile dell'elaborazione dell'input dell'utente, dell'aggiornamento del modello e della visualizzazione della vista appropriata.

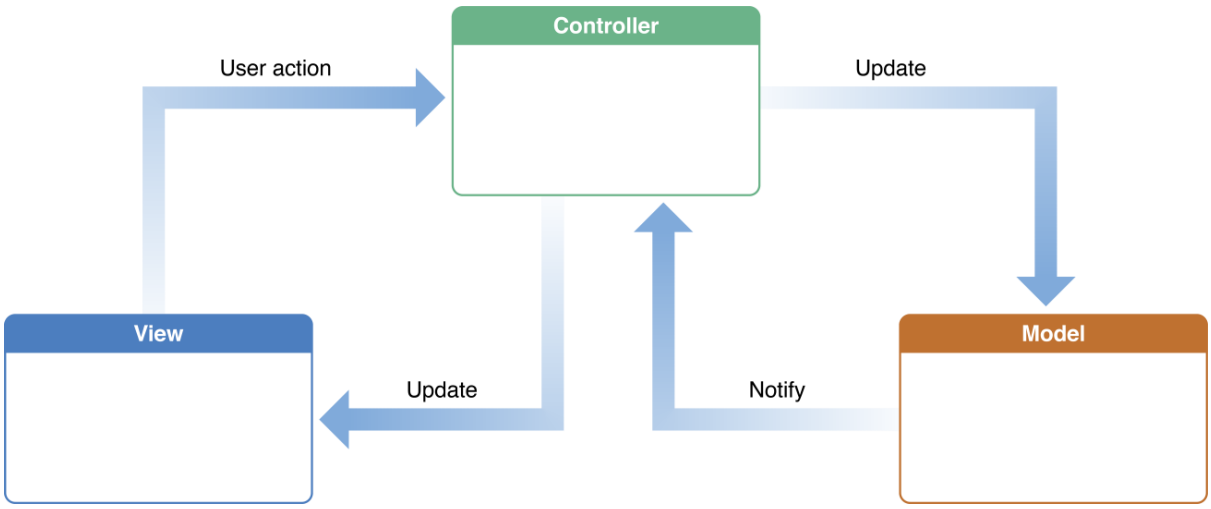

Figura 1: Pattern Model-View-Controller

La scelta di questo pattern architetturale è dettata dalle seguenti ragioni:

1. **Separazione dei compiti**: Il modello, la vista e il controller sono separati l'uno dall'altro, aiutando a mantenere il codice organizzato e ad evitare eccessiva complessità. *Business Logic*$_G$ e *Presentation Logic*$_G$ pertanto sono separate e comunicano solamente mediante il controller. Ciò semplifica la manutenzione del software e la scalabilità nel tempo;

2. **Flessibilità**: Il pattern MVC consente di apportare modifiche con maggior facilità in quanto ogni strato è separato da un'altro, permettendo di modificare ogni componente limitando la necessità di dover agire anche sugli altri;

3. **Pulizia del codice**: Con MVC, il codice è organizzato in tre componenti distinte, ciascuno con una responsabilità specifica. Ciò rende più facile per gli sviluppatori capire come funziona l'applicazione e apportare modifiche senza influire su altre parti del codice;

4. **Testing facilitato**: Le componenti sono indipendenti fra loro, pertanto è più facile testarle separatamente. Ciò consente agli sviluppatori di identificare e correggere gli errori più rapidamente ed efficientemente.

L'utilizzo di MVC in questo contesto consente di separare la logica applicativa dal livello di presentazione. In tal modo, la vista si limita a mostrare i dati, permettendo al controller di gestire le relazioni tra i componenti e il modello di dati.

3.3 Diagrammi delle classi

La suddivisione delle parti del progetto segue quella del pattern architetturale MVC_G come descritto nella sezione precedente.
In questa sezione ne verranno approfonditi i comportamenti e la logica.

3.3.1 Model

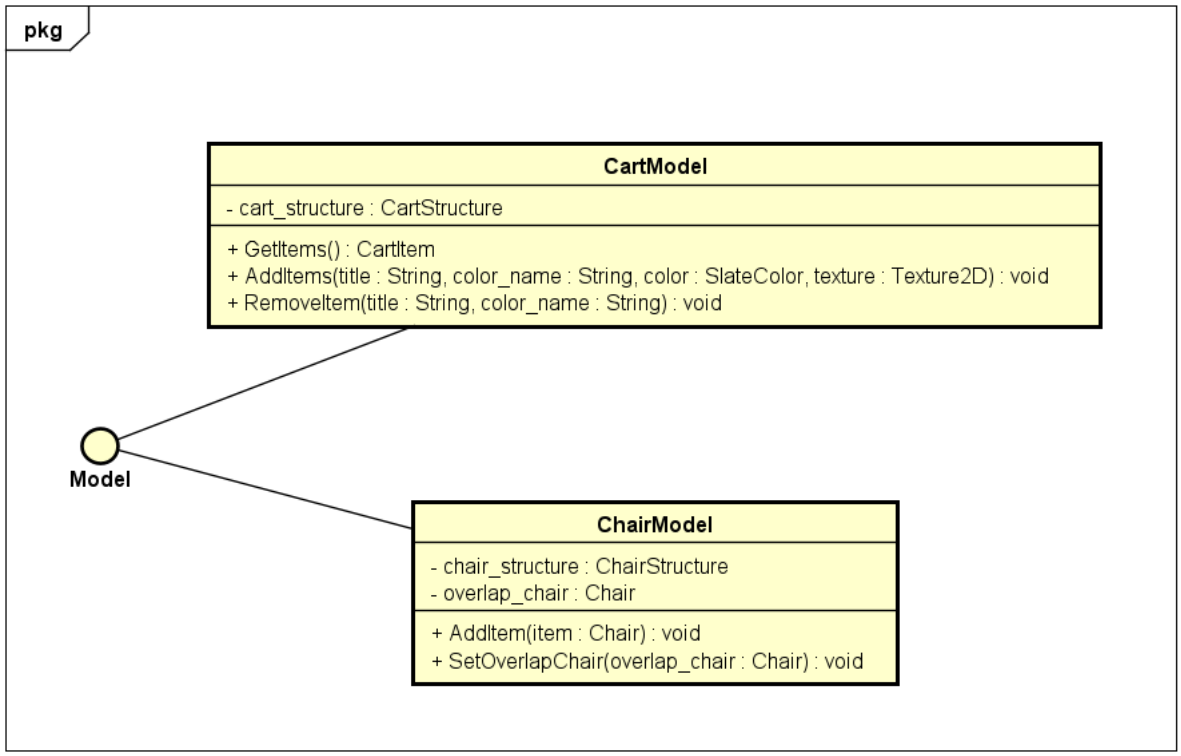

Figura 2: Diagramma delle classi relative al Model

Il diagramma delle classi del **Model** è costituito dall'interfaccia `Model` e dalle classi concrete `CartModel` e `ChairModel`.

`CartModel` è la classe che permette di gestire il carrello e i prodotti in esso contenuti. Presenta un metodo "`get`" per il campo `cart_structure` e dei metodi per l'aggiunta e la rimozione di sedie dal carrello:

- `GetItems`: ritorna la lista di tutti gli elementi;
- `AddItems`: aggiunge nuovi elementi al carrello (attualmente in quantità 1, poi verrà determinato un numero differente di oggetti). Si verifica se esiste già una sedia nel carrello che abbia le stesse caratteristiche di quella che si desidera inserire. Nel caso in cui l'oggetto desiderato abbia lo stesso nome e colore di un oggetto già presente, verrà incrementata la quantità di tale prodotto all'interno del carrello. In caso contrario, verrà aggiunto al carrello come nuova entità;

- `RemoveItem`: permette di rimuovere un oggetto dal carrello.

`ChairModel` è la classe che permette di gestire le sedie nella scena principale del progetto e di controllare se l'utente si trova in un posizione tale da sovrapporsi ad un elemento della collezione. Presenta un metodo "`set`" per il campo `overlap_chair`:

- `AddItem`: aggiunge tutte le sedie che si trovano nella scena creando così un riferimento all'interno di `ChairModel`. Quest'ultimo passo consentirà di manipolazione le sedie all'interno del modello stesso;

- `SetOverlapChair`: mantiene un riferimento alla sedia con cui l'utente fa overlap. Questo viene aggiornato automaticamente ogni volta che l'utente interagisce con una sedia in esposizione.

3.3.1.1 CartStructure

`CartStructure` rappresenta la struttura dati ideata per tracciare gli elementi presenti all'interno del carrello virtuale. In particolare, `CartModel` si avvale di questa struttura per creare, gestire e mantenere l'elenco degli oggetti selezionati dall'utente durante la navigazione all'interno dell'ambiente 3D. La struttura dati è stata progettata in modo da garantire una gestione ottimale degli oggetti presenti nel carrello, assicurando al contempo la massima flessibilità e adattabilità del sistema ai bisogni dell'utente. Segue una rappresentazione della struttura:

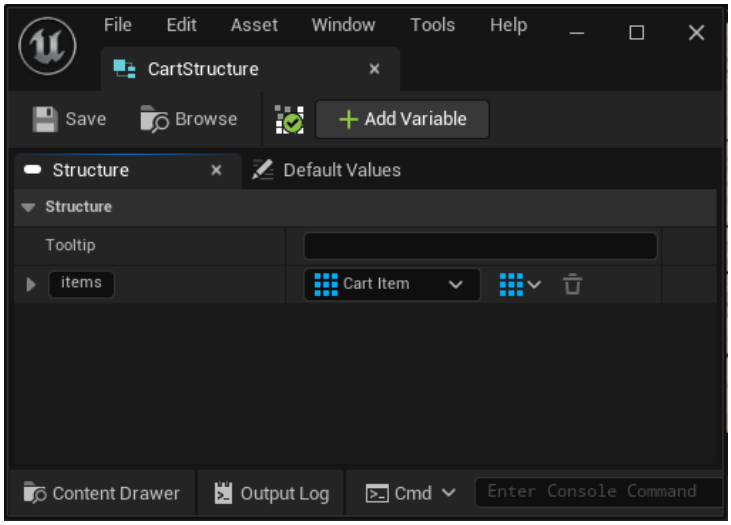

Figura 3: Definizione della struttura CartStructure

CartItemStructure

`CartItemStructure` rappresenta la struttura dati utilizzata da `CartItem` per descrivere in modo univoco le caratteristiche di ogni oggetto che verrà aggiunto al carrello. Questa struttura descrive per ogni oggetto:

- Titolo;

- Texture adottata;
- Colore;
- Quantità.

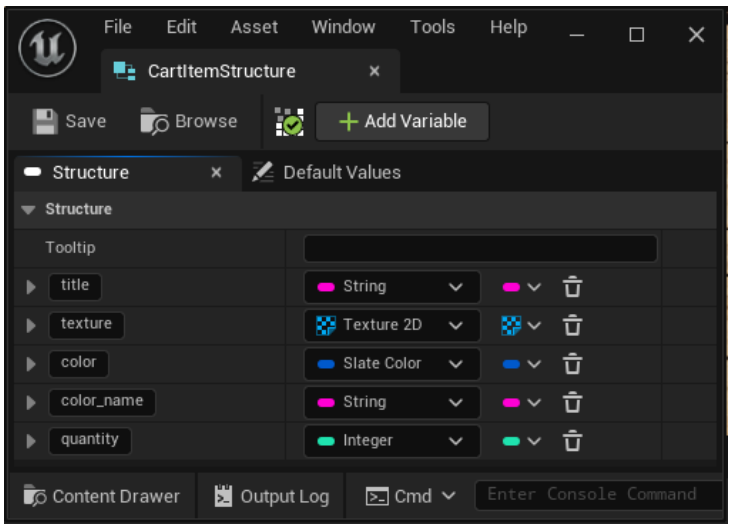

Figura 4: Definizione della struttura CartItemStructure

3.3.1.2 ChairStructure

`ChairStructure` rappresenta la struttura dati ideata per tracciare le sedie presenti all'interno dello showroom. In particolare, `ChairModel` si avvale di questa struttura per gestire e mantenere l'elenco degli oggetti all'interno dell'ambiente 3D. La struttura dati è stata progettata in modo da facilitare l'aggiornamento dei prodotti della collezione, adattandosi ai bisogni del venditore. Segue una rappresentazione della struttura:

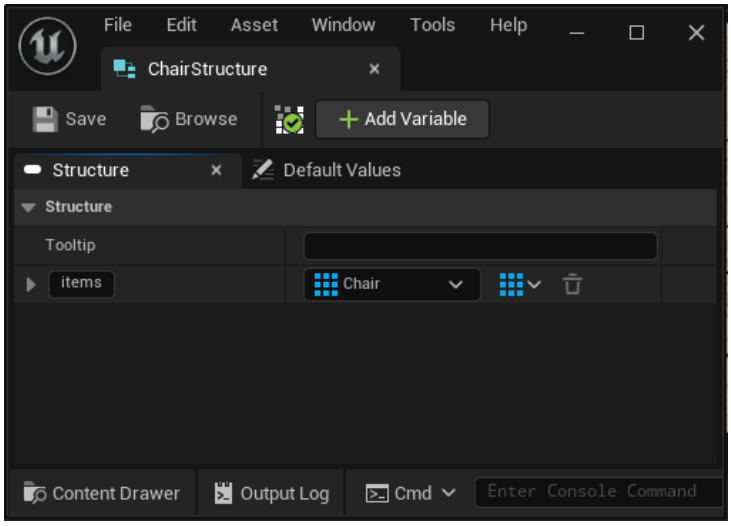

Figura 5: Definizione della struttura ChairStructure

3.3.2 View

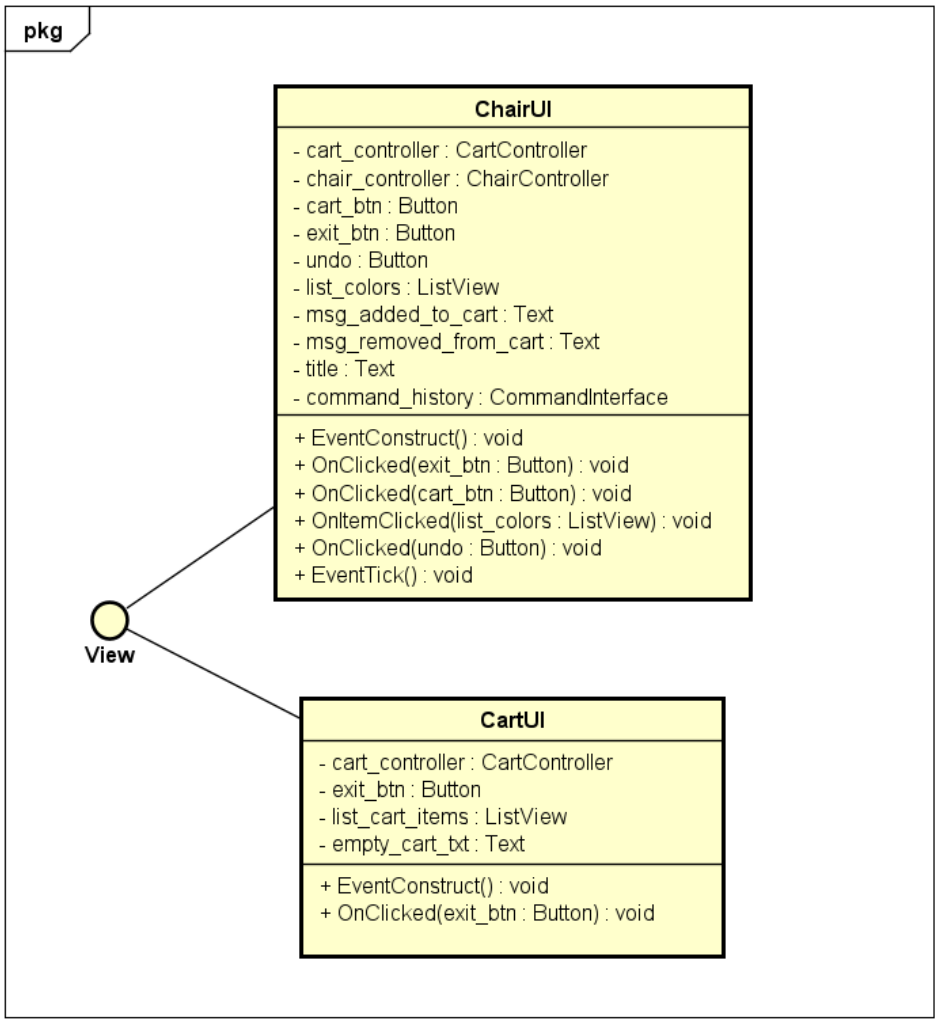

Figura 6: Diagramma delle classi relative alla View

Il diagramma delle classi della **View** è costituito dall'interfaccia `View` e dalle classi `ChairUI` e `CartUI`.

`ChairUI` si occupa di presentare tutti gli elementi che compongono le rispettive viste per la visualizzazione della scheda tecnica della sedia che l'utente sceglie durante la sua navigazione dello showroom.

`CartUI` si occupa di presentare tutti gli elementi che compongono le viste per la visualizzazione degli oggetti presenti all'interno del carrello.

Le classi `Button`, `Image`, `ListView` e `Text` appartengono alla libreria fornita da Unreal Engine, pertanto l'interazione con tali oggetti è gestito interamente dal motore grafico. Gli eventi scatenati dal click dei pulsanti, invece, è gestita tramite le funzioni `OnClicked()`

le quali conterranno un Event Listener ridefinito per ogni bottone a disposizione nella vista (a seconda dello scopo).

3.3.3 Controller

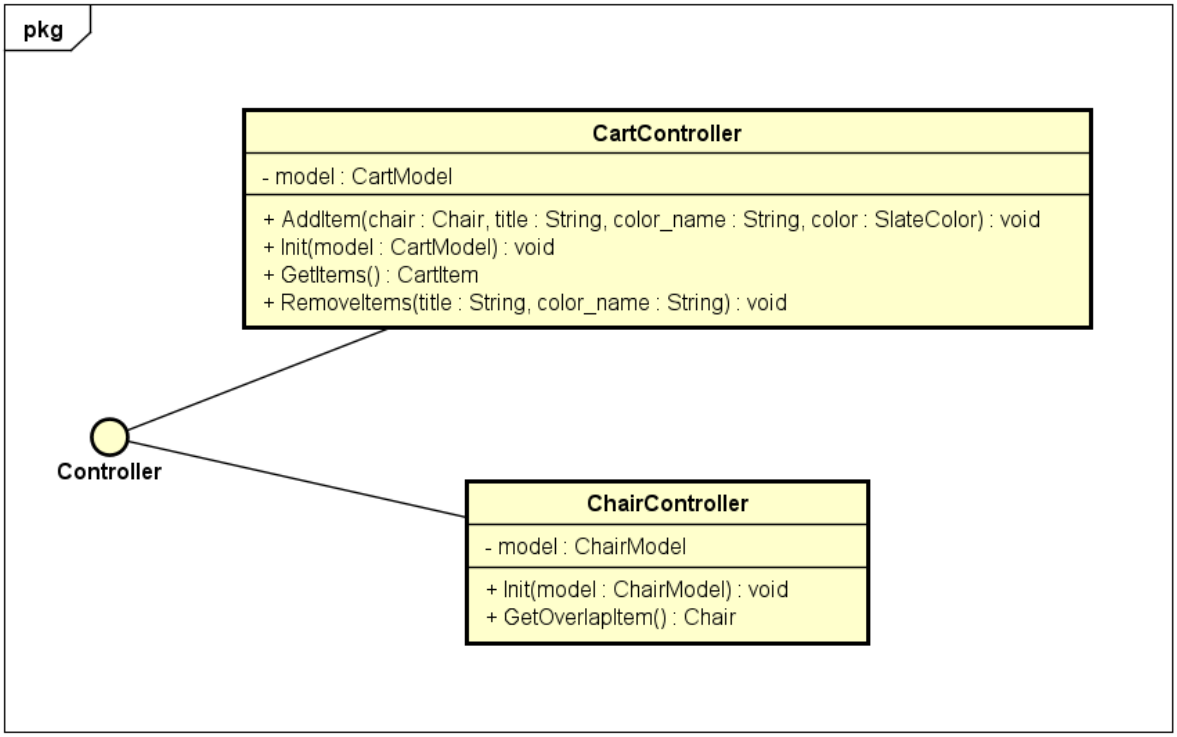

Figura 7: Diagramma delle classi relative al Controller

Il diagramma delle classi del **Controller** è costituito dall'interfaccia `Controller` e dalle classi concrete `CartController` e `ChairController`.
Il controller funge da tramite tra la vista e il modello e ha il compito di gestire le richieste dell'utente, apportando eventuali modifiche alle altre due componenti del sistema.
Nell'architettura dell'applicazione si è optato per l'utilizzo di due controller per gestire le diverse viste, le quali hanno funzionalità differenti. Questa scelta consente di lavorare in modo modulare favorendo l'estendibilità e la manutenibilità del sistema.
Ogni controller ha un riferimento implicito alla vista che viene gestito automaticamente dal motore grafico *Unreal Engine*$_G$. Questo permette al controller di accedere a tutti gli elementi della view e modificarli di conseguenza. Tuttavia, è importante sottolineare che la view non comunica direttamente con il model. Il controller fungendo da intermediario tra la view e il model, gestisce tutte le comunicazioni tra i due e garantisce che la logica di business sia separata dalla presentazione grafica.

`CartController` gestisce l'interazione fra `CartModel` e `CartUI`.

- `AddItem`: funzione che richiama il metodo `CartModel::AddItems`;

- `Init`: funzione che inizializza il controller;
- `GetItems`: funzione che richiama il metodo `CartModel::GetItems`;
- `RemoveItems`: funzione che richiama il metodo `CartModel::RemoveItem`.

`ChairController` gestisce l'interazione fra `ChairModel` e `ChairUI`.

- `Init`: funzione che inizializza il controller;
- `GetOverlapItem`: ritorna l'oggetto `overlap_chair` da `ChairModel`.

3.4 Design Pattern utilizzati

Nel corso della progettazione del prodotto *ShowRoom3D*$_G$, sono stati applicati due design pattern per garantire la modularità, l'estensibilità e la manutenibilità del codice. In particolare, si è fatto uso del pattern `Observer` che permette gestire gli eventi generati dalle interazioni degli utenti con l'interfaccia grafica, e il pattern `Command` per implementare un sistema di aggiunta/undo.
L'uso combinato di questi pattern ci ha permesso di gestire efficacemente le interazioni degli utenti e le conseguenze delle loro azioni, fornendo una soluzione scalabile. In questa sezione, verranno esplorati nel dettaglio questi pattern in dettaglio.
Nell'implementazione dei pattern si sono utilizzate prevalente la composizione e l'ereditarietà:

- La **composizione** ha permesso al team di creare oggetti più complessi a partire da oggetti più semplici, che possono essere facilmente gestiti e modificati;
- L'**ereditarietà** viene invece utilizzata per creare nuove classi a partire da quelle esistenti, ereditandone le proprietà e i metodi. Questo permette di evitare la duplicazione del codice e di semplificare la gestione delle classi.

3.4.1 Observer

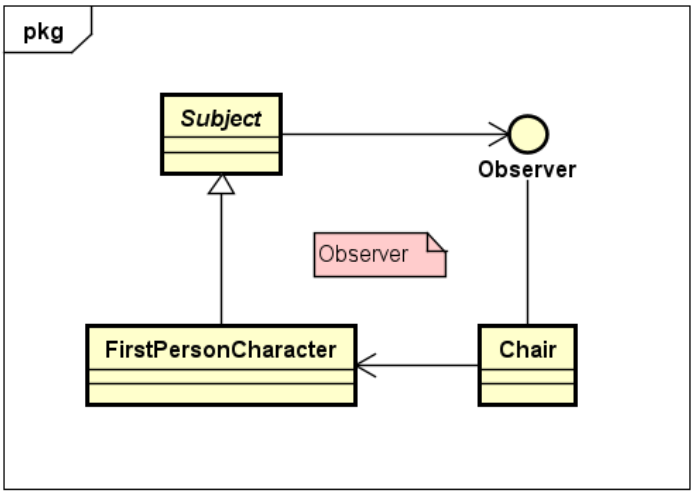

Figura 8: Diagramma delle classi relative al design pattern Observer

3.4.1.1 Utilizzo

Il gruppo *Smoking Fingertips* ha deciso di utilizzare il pattern `Observer` per facilitare la comprensione dell'utente riguardo all'interazione con gli oggetti presenti nella showroom. In particolare, attraverso l'utilizzo di questo *design pattern*$_G$ è stato possibile mostrare a video una scritta che si attiva ogni volta che l'utente si avvicina ad una specifica sedia, fornendo informazioni utili sull'azione da compiere per poter visualizzare la scheda tecnica del prodotto in esposizione.

3.4.1.2 Funzionamento

Il pattern Observer è una soluzione di design pattern che è stata utilizzata per mantenere una relazione uno-a-molti (utente-sedie disposte), in modo che quando l'utente (`Subject` nell'immagine) si sposta all'interno dello spazio della showroom (cambiando il suo stato), tutti gli oggetti dipendenti (`Observer` nell'immagine) vengano notificati automaticamente e aggiornati di conseguenza.
Nel caso specifico d questo progetto è stata creata una lista di subscribers (le sedie esposte) all'interno del Subject. Ogni volta che il personaggio si muove, viene inviata una notifica di aggiornamento con la posizione dell'utente a tutti gli observer iscritti.
Gli Observer, le sedie, a seconda della posizione ricevuta:

- mostreranno la scritta per `Premere E per interagire` se l'utente è nell'area della sedia;
- rimuoveranno la scritta, qualora l'utente si stia allontanando dall'oggetto.

In questo modo, quando l'utente si avvicina a una sedia, la sedia è in grado di reagire al cambiamento di stato del personaggio. Questo pattern è risultato utile per gestire le interazioni tra l'attore principale e gli oggetti all'interno di un'applicazione e per aumentare l'efficienza. Attualmente il pattern **Observer** viene utilizzato esclusivamente per gestire le interazioni tra l'utente del gioco e le sedie presenti. Tuttavia questo pattern può essere esteso e utilizzato anche per gestire le interazioni tra il personaggio e altri oggetti, come ad esempio un tavolo.

3.4.2 Command

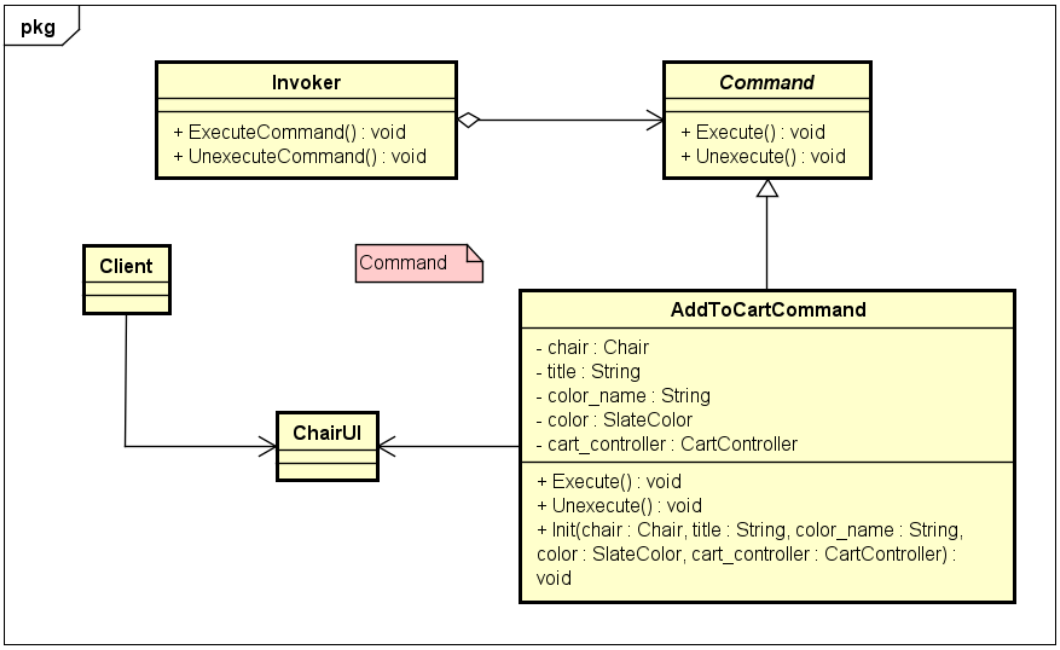

Figura 9: Diagramma delle classi relative al design pattern Command

3.4.2.1 Utilizzo

Il gruppo *Smoking Fingertips* ha scelto di utilizzare il pattern `Command` per gestire l'aggiunta delle sedie al carrello e la rimozione degli ultimi elementi aggiunti. L'utilizzo del pattern Command consente di separare l'azione dell'utente dall'oggetto che la esegue, garantendo maggiore flessibilità e facilità di gestione delle operazioni.
Questa scelta è stata motivata dalla necessità di consentire all'utente di effettuare l'azione di **undo** dell'aggiunta al carrello, ripristinando lo stato precedente della transazione.
Il pattern Command è permette di incapsulare le richieste di azione in oggetti separati, rendendoli indipendenti dal contesto in cui sono stati creati. In questo modo, è possibile creare un'istanza dell'oggetto Command per ogni operazione di aggiunta o rimozione dal carrello, e successivamente aggiungerli o rimuoverli a seconda delle esigenze dell'utente. La gestione delle azioni di undo è possibile grazie alla presenza di una coda di comandi, che consente di ripercorrere i passi in avanti o indietro nel tempo, fino al momento desiderato.

3.4.2.2 Funzionamento

L'implementazione pratica del pattern `Command` prevede la creazione di un'interfaccia `Command`. Le classi derivate concrete incapsulano le azioni. Nel caso della classe derivata concreta `AddToCartCommand` le operazioni disponibili sono quelle di aggiunta al carrello e di annullamento dell'ultima operazione nel carrello.

Per consentire l' **aggiunta** di un oggetto al carrello, la classe `AddToCartCommand` implementa il metodo `Execute()` che richiama il metodo `AddItems()` del Controller `cart_controller`.

Per consentire l'**undo** dell'azione di aggiunta, la classe `AddToCartCommand` implementa il metodo `Unexecute()`, che esegue l'operazione inversa alla precedente, cioè la rimozione degli ultimi prodotti aggiunti al carrello. Per fare ciò richiama il metodo `RemoveItem()` del Controller `cart_controller`.

Inoltre, la classe Command deve memorizzare lo stato dell'azione di aggiunta, in modo da permettere il ripristino dello stato precedente l'undo.
L'utente potrà effettuare l'operazione di undo un numero di volte corrispondente alla quantità di operazioni di aggiunta che sono state eseguite.

Allo stato attuale l'unico `Command` implementato è quello per l'aggiunta di un oggetto nel carrello. Seguono spiegazioni pratiche di come avvengono le chiamate.

Aggiunta di un oggetto al carrello

Per aggiungere un oggetto al carrello si utilizza il comando `AddToCartCommand`. Dopo aver creato un'istanza del comando, esso viene passato come parametro all'`Invoker` per l'esecuzione tramite il metodo `execute()`. Questo metodo sfrutta il pattern Command per incapsulare la richiesta dell'utente. Successivamente, il comando viene aggiunto alla cronologia dei comandi eseguiti (`commandHistory`) tramite il metodo `push()`, in modo da poterlo ripetere o annullare in seguito. Le istruzioni che vengono eseguite sono:

```
Command command = new AddToCartCommand();
Invoker invoker = new Invoker(command);
invoker.execute();
commandHistory.push(command);
```

Annullamento dell'ultima operazione

Per annullare l'ultima aggiunta al carrello, è necessario recuperare l'operazione svolta eseguendo il metodo `pop()` dallo stack `commandHistory`. Successivamente, si istanzia l'`Invoker` passando come parametro il comando ottenuto dall'istruzione precedente. Quando l'utente richiede l'annullamento di un'azione, viene chiamato il metodo `unexecute()` dell'`Invoker`, il quale a sua volta richiama il metodo `unexecute()` della classe `Command`. In questo modo, l'utente può facilmente annullare le azioni eseguite in precedenza senza dover ripercorrere manualmente tutte le operazioni effettuate. Le istruzioni che vengono eseguite sono:

```
command = commandHistory.pop();
invoker = new Invoker(command);
invoker.unexecute();
```

3.5 Architettura di Unreal Engine

Il *Gameplay Framework* di Unreal Engine è composto da classi che includono le funzionalità per la rappresentazione dei giocatori e degli oggetti, il loro controllo tramite input o logica AI, la creazione di heads-up display (HUD) e telecamere per i giocatori.

La rappresentazione dei giocatori e degli NPC è relegata ai *Pawn* e ai *Character*. I Pawn sono attori che possono essere posseduti da un Controller e sono predisposti ad una gestione facile dell'input e di azioni tipiche dei giocatori, essi possono essere non umanoidi. I Character invece sono dei Pawn umanoidi, contengono *CapsuleComponent* per la gestione delle collisioni e *CharacterMovementComponent* per replicare movimenti "human-like" e contenete funzionalità per l'implementazione di animazioni. Per controllare dei Pawn tramite input si fa uso dell'attore *Controller* e dell'interfaccia *PlayerController*.

L'*HUD*, o heads-up display, è una componente 2D che permette la visualizzazione di informazioni nella stessa "visuale" dell'utente durante il gioco, contenuto in PlayerController. La classe *PlayerCameraManager* gestisce la o le camere disponibili per un giocatore, PlayerController specifica una classe relativa alla camera da usare e istanza un attore per essa.

Il concetto di gioco è suddiviso in *GameMode* e *GameState*; esse definisce le regole del gioco e tengono traccia del progresso del gioco e dei giocatori. La classe *PlayerState* contiene tutte le informazioni relative a giocatori e bot come nome, punteggio, livello etc..

Il seguente diagramma mostra come le classi di gameplay interagiscono tra loro. Un gioco è formato da un'istanza di *GameMode* e un'istanza di *GameState*, i giocatori umani sono associati a *PlayerController*. La classe *PlayerController* permette al giocatore di impossessarsi di un *Pawn*, cosicché esso possa avere una rappresentazione fisica nel mondo del gioco. La classe *PlayerController* è anche responsabile per la gestione degli input del giocatore, fornisce un HUD e un oggetto *PlayerCameraManager* per la gestione della telecamera.

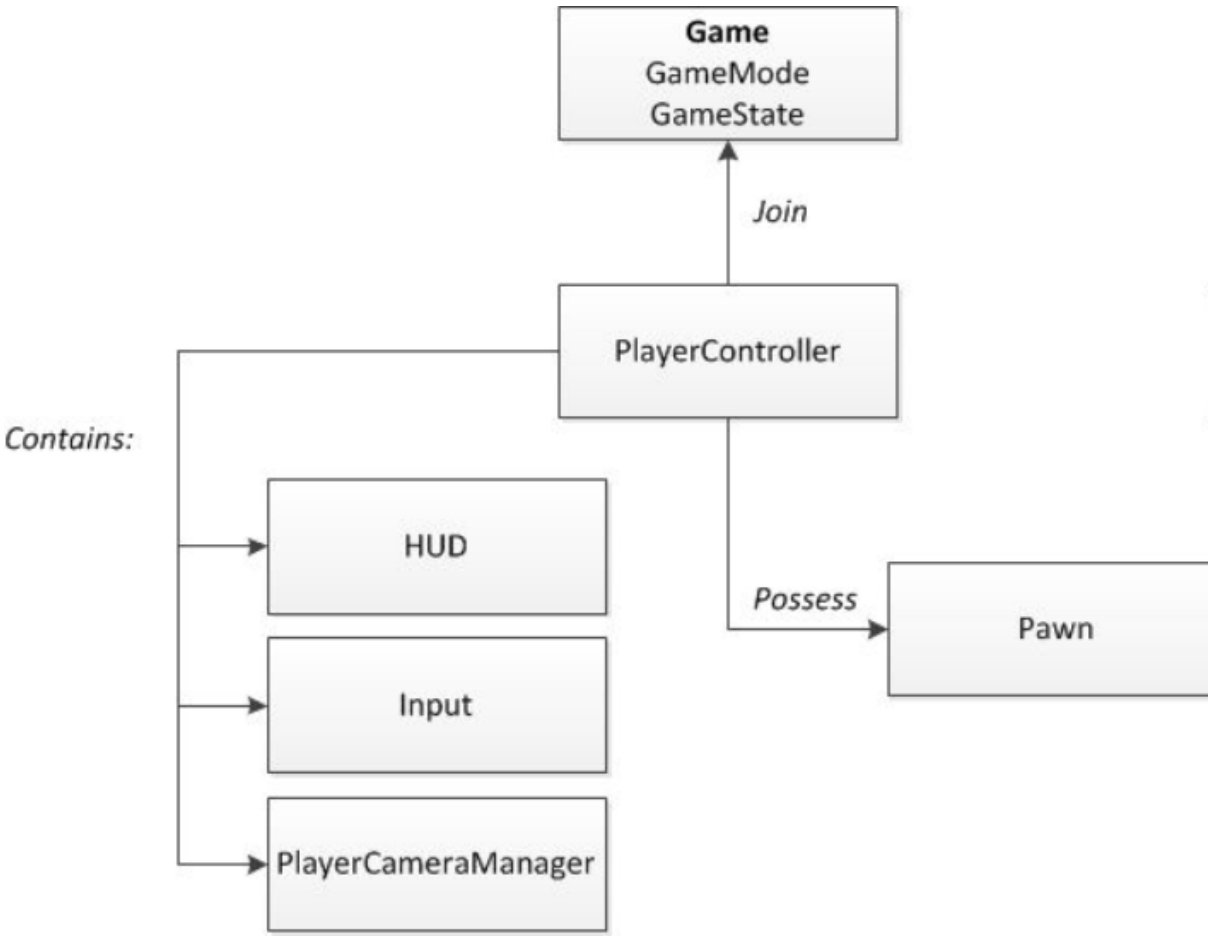

Figura 10: Gameplay Framework di Unreal Engine

4 Architettura di deployment

Per effettuare un rilascio del progetto è necessario:

1. Compilare il progetto tramite *Unreal Editor*$_G$;
2. Pubblicare una release nell'apposita repository *Github*$_G$ contente i package generati da Unreal Editor.

4.1 Creazione della release tramite Unreal Editor

Se non presente sarà necessario creare un nuovo **Launch Profile** custom il quale conterrà tutte le impostazioni per il processo di rilascio. Di seguito si descrive come creare e configurare il suddetto profilo:

Tramite Unreal Editor:

1. Aprire il **Project Launcher**;
2. Creare un nuovo **Launch Profile** custom tramite il pulsante *Add (+)*;

3. Impostare nome e descrizione del profilo e configurare le impostazioni per il processo di rilascio.

Di seguito le impostazioni per ogni sezione del processo di creazione:

Project

Selezionare il progetto *ShowRoom3D*.

Build

Impostare **Build configuration** a **Shipping**.
La configurazione Shipping permette un'esecuzione con prestazioni ottimali eliminando comandi della console, statistiche e strumenti di profiling.

Cook

1. Selezionare il metodo **By the Book**;
2. Selezionare tutte le piattaforme su cui distribuire:
    - Windows;
    - Linux.
3. Selezionare le localizzazioni:
    - it-IT.
4. Selezionare la mappa:
    - FirstPersonMap;
5. Nel menù **Release/DLC/Patching Settings**:
    - Selezionare **Create a release version of the game for distribution**;
    - Inserire il numero della versione della release.
6. Nel menù **Advanced Settings** impostare **Cooker Build Configuration** a **Shipping** e spuntare le opzioni:
    - Compress content;
    - Save packages without versions;
    - Store all content in a single file (UnrealPak).

- **Release/DLC/Patching Settings**: premette la creazione di una versione di release usata per la distribuzione.
- **Compress Content**: comprime i files generati; permette una dimensione minore del progetto ma tempi di caricamento maggiori.

- **Save Packages Without Versions**: assume che la versione inserita sia la corrente, permette patch di dimensioni più piccole.
- **Store all content in a single file (UnrealPak)**: genera un singolo file UnrealPak al posto di molti files.

Package

1. Impostare la build a **Package & Store Locally**;
2. Impostare il path locale per il salvataggio della release;
3. Selezionare **Is this build for distribution to the public**.

Deploy

1. Selezionare **Do Not Deploy** per non effettuare deploy su dispositivi al termine del processi cook e package.
2. Tornare al profilo principale tramite il pulsante **back** in altro a destra.
3. Cliccare il pulsante **Launch**.

Il Project Launcher genererà la release.

4.2 Distribuzione tramite Github releases

L'ultimo passaggio riguarda la distribuzione del software. Per fare ciò sarà necessario:

1. Aprire la repository usando un account abilitato;
2. Aprire la sezione **Releases** e cliccare **Draft a new release**;
3. Inserire il titolo: "Release vX.Y.Z" inserendo la versione corretta;
4. Inserire una descrizione *(opzionale)*;
5. Inserire il progetto in formato .rar e il codice sorgente nei formati .zip e .tar.gz;
6. Cliccare **Publish release**.

Riferimenti bibliografici e sitografici

- Documentazione Unreal Engine | Releasing Your Project https://docs.unrealengine.com/5.0/en-US/preparing-unreal-engine-projects-for-release/
- Documentazione Unreal Engine | Project Launcher https://docs.unrealengine.com/5.0/en-US/using-the-project-launcher-in-unreal-engine/
- Documentazione Unreal Engine | Build Operations: Cook, Package, Deploy, and Run https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Deployment/BuildOperations/
- Documentazione Unreal Engine | Gameplay Framework https://docs.unrealengine.com/5.0/en-US/gameplay-framework-in-unreal-engine/
- Documentazione Unreal Engine | Gameplay Framework Quick Reference https://docs.unrealengine.com/5.0/en-US/gameplay-framework-quick-reference-in-unreal-engine/