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	2023-05-13
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	2023-05-13	Edoardo Gasparini	Responsabile	Approvazione del documento
0.3.1	2023-05-09	Luca Polese *Edoardo Gasparini*	Progettista *Verificatore*	Aggiornamento §3.1
0.3.0	2023-05-07	Sebastien Biollo *Luca Annicchiarico*	Progettista *Verificatore*	Revisione complessiva di coerenza e coesione
0.2.2	2023-05-01	Edoardo Gasparini *Luca Polese*	Progettista *Verificatore*	Aggiornamento §3.4, §3.5 Aggiunto §3.3
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 §3.4, §3.5
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*$_G$ 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://web.archive.org/web/20110903163238/http://java.sun.com/blueprints/patterns/MVC-detailed.html
  https://web.archive.org/web/20190911171532/http://www.claudiodesio.com/ooa&d/mvc.htm
- **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/
- **Documentazione Unreal Engine**:
  - Releasing Your Project
    https://docs.unrealengine.com/5.0/en-US/preparing-unreal-engine-projects-for-release/
  - Project Launcher
    https://docs.unrealengine.com/5.0/en-US/using-the-project-launcher-in-unreal-engine/
  - Build Operations: Cook, Package, Deploy, and Run
    https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Deployment/BuildOperations/
  - Gameplay Framework
    https://docs.unrealengine.com/5.0/en-US/gameplay-framework-in-unreal-engine/
  - Gameplay Framework Quick Reference
    https://docs.unrealengine.com/5.0/en-US/gameplay-framework-quick-reference-in-unreal-engine/

Tutti i riferimenti a siti web sono verificati e aggiornati alla data 2023-05-13.

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*$_G$ 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*$_G$ 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*$_G$ 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*$_G$ 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*$_G$ 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*$_G$ offre molte possibilità di sviluppo a seconda delle esigenze del progetto.
- **Portabilità**: *Unreal Engine*$_G$ è disponibile su diverse piattaforme pertanto gli sviluppatori possono creare applicazioni *multi-piattaforma*$_G$ 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*$_G$ 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*$_G$, 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*$_G$ 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, ovvero 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 le schede tecniche dei prodotti esposti. Le principali caratteristiche di `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 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. A seconda delle proprie necessità, l'utente può aggiornare le caratteristiche dell'asset `Chair` quali il colore: al click dei bottoni nell'interfaccia l'utente può , 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*$_G$ 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*$_G$ si articola in:

- **Model** (che in seguito chiameremo anche modello): rappresenta i dati e la *Business Logic*$_G$ 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.

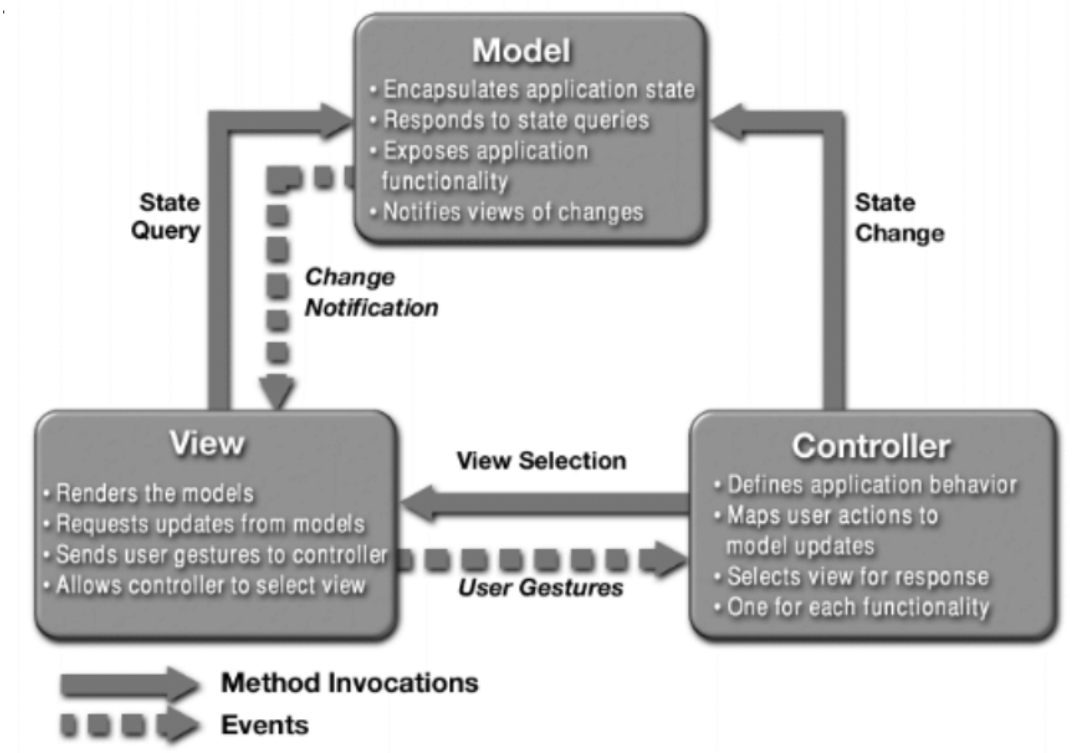

Figura 1: Pattern Model-View-Controller

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_G 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_G, 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 Observer in Unreal Engine

Per applicare il pattern architetturale MVC_G è necessario utilizzare il design pattern *Observer* che permette la comunicazione fra le componenti, limitando il grado di accoppiamento.

Unreal Engine non permette l'implementazione canonica del pattern in quanto la creazione di classi astratte non rispetta la definizione classica. Lo specificatore `Abstract`, come espresso nella documentazione ufficiale, impedisce all'utente di aggiungere Attori di questa classe ai Livelli, non impedisce l'istanziazione della classe stessa.
Il pattern `Observer` necessitando di una classe `Subject` astratta non può dunque essere realizzato.

Unreal Engine fornisce gli `Event Dispatcher`, che permettono di mantenere lo stesso grado di disaccoppiamento. Il funzionamento è descritto nella sezione che segue.

3.3.1 Funzionamento degli Event Dispatcher

Un `Event Dispatcher` in *Unreal Engine*$_G$ è un meccanismo di comunicazione asincrono che consente agli oggetti di inviare e ricevere eventi. È una componente fondamentale del sistema di messaggistica di Unreal che viene utilizzata per implementare la comunicazione tra oggetti senza dover creare dipendenze dirette tra di loro.

Il funzionamento di un Event Dispatcher in Unreal Engine può essere descritto attraverso i seguenti passaggi:

- Definizione dell'evento: viene definito il nome del messaggio, e facoltativamente eventuali parametri;
- Registrazione dell'evento: le classi interessate ad ascoltare l'evento si registrano come ascoltatrici effettuando un binding;
- Invio dell'evento: Quando la classe desidera notificare gli ascoltatori su un evento, lo invoca chiamando il dispatcher stesso. Il dispatcher si preoccupa di notificare tutti gli ascoltatori registrati;
- Gestione dell'evento: Ogni componente registrata reagisce con una propria implementazione personalizzata.

3.3.2 Confronto Event Dispatcher con il pattern Observer

A differenza del design pattern Observer, l'Event Dispatcher in Unreal Engine è specifico per il framework di sviluppo. Tuttavia, ci sono alcune similitudini tra i due concetti. Segue un elenco di similitudini tra l'`Event Dispatcher` di Unreal Engine e il design pattern `Observer`:

- **Comunicazione asincrona**: Entrambi i concetti permettono la comunicazione asincrona tra oggetti. Gli eventi possono essere inviati e gestiti in momenti diversi

rispetto alla loro generazione, consentendo una comunicazione flessibile tra le parti interessate.

- **Separazione delle responsabilità**: Sia l'Event Dispatcher che l'Observer consentono la separazione delle responsabilità tra l'oggetto che genera l'evento (dispatcher o soggetto) e gli oggetti interessati all'evento (ascoltatori o osservatori). Ciò evita l'accoppiamento diretto tra gli oggetti e permette una maggiore modularità del codice.

- **Registrazione degli ascoltatori**: In entrambi i casi, gli ascoltatori devono registrarsi per ricevere gli eventi. L'Event Dispatcher di Unreal Engine offre un meccanismo di registrazione che associa una funzione o un metodo dell'oggetto ascoltatore all'evento specificato. Nell'Observer gli osservatori vengono registrati nel soggetto attraverso un metodo specifico (solitamente `attach()`).

- **Notifica degli eventi**: Sia l'Event Dispatcher che l'Observer notificano gli ascoltatori/osservatori quando si verifica un evento. L'Event Dispatcher di Unreal Engine invoca la funzione o il metodo associato all'evento, mentre nell'Observer pattern il soggetto chiama il metodo comune degli osservatori per notificarli.

- **Gestione personalizzata degli eventi**: Gli ascoltatori/osservatori in entrambi i casi possono gestire gli eventi in modo personalizzato. Possono definire le azioni da intraprendere quando un evento viene ricevuto, consentendo la flessibilità nell'implementazione delle risposte agli eventi.

- **Decoupling**: Entrambi i concetti promuovono un grado di decoupling tra i soggetti/dispatcher e gli ascoltatori/osservatori. Ciò significa che gli oggetti possono comunicare tra loro senza dover avere una **conoscenza diretta l'uno dell'altro**, migliorando la modularità e la manutenibilità del codice.

- **Riusabilità del codice**: Sia l'Event Dispatcher che il design pattern Observer consentono di scrivere codice riusabile. Gli oggetti possono essere facilmente riutilizzati come dispatcher/soggetti o ascoltatori/osservatori in diversi contesti senza dover apportare modifiche significative al codice sorgente.

- **Facilità di manutenzione**: Entrambi i concetti semplificano la manutenzione del codice, poiché le modifiche apportate al dispatcher/soggetto o agli ascoltatori/osservatori possono essere isolate e gestite in modo indipendente, riducendo l'impatto sul resto del sistema.

3.4 Diagrammi delle classi

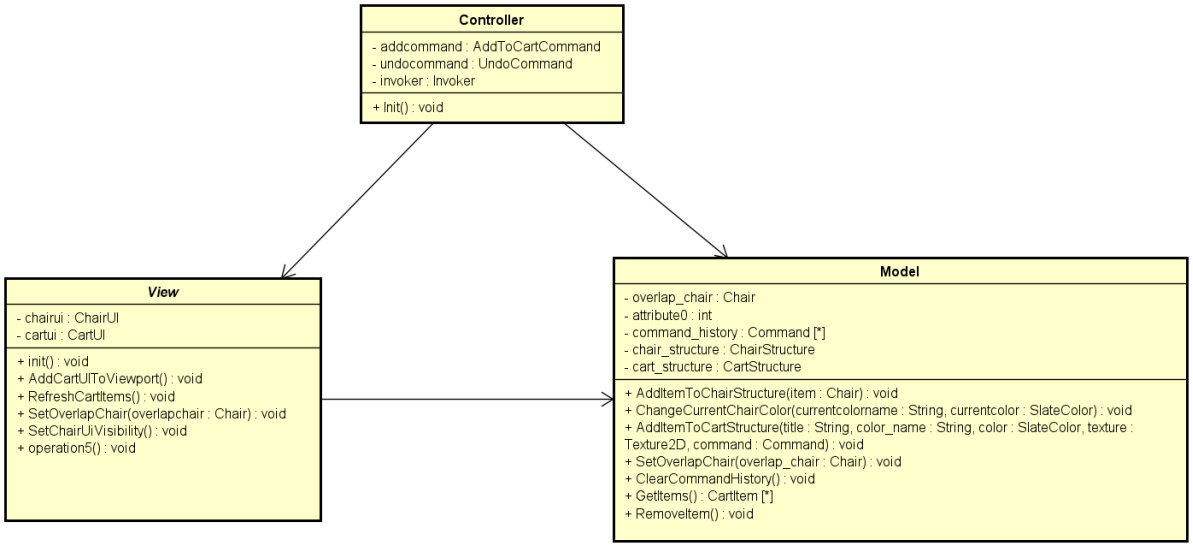

Figura 2: Diagramma delle classi del sistema

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.

3.4.1 Model

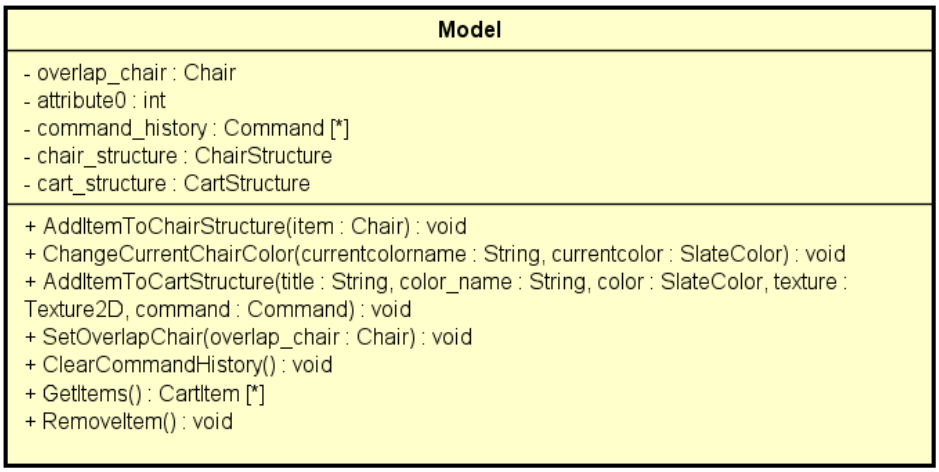

Figura 3: Diagramma delle classi relative al Model

Il diagramma delle classi del **Model** è costituito dall'omonima classe `Model`, e dalle classi `CartStructure` e `ChairStructure` che figurano come campi del modello per rappresentare lo stato del sistema.

`Model` è la classe che permette di:

- gestire il carrello e i prodotti in esso contenuti;
- gestire le sedie nella scena principale del progetto;
- rispondere a richieste di interazione, cambiando stato in base alla sedia di interesse dell'utente.

Presenta un metodo "`get`" per il campo `cart_structure`, un metodo "`set`" per modificare il campo `overlap_chair` e dei metodi per l'aggiunta e la rimozione di sedie dal carrello:

- `AddItemToChairStructure(item : Chair)` metodo che viene chiamato nel programma per tenere traccia delle sedie presenti all'interno della scena creando così un riferimento all'interno del modello. Questo consentirà di manipolare le sedie all'interno del modello stesso;
- `SetOverlapChair(overlap_chair: Chair)` mantiene un riferimento alla sedia con cui l'utente fa overlap. Viene aggiornato ogni volta che l'utente interagisce con una sedia in esposizione;
- `GetItems`: ritorna la lista di tutti gli elementi di tipo `Chair` presenti nella struttura `cart_structure`;
- `AddItemToCartStructure(title : String, color_name : String, color: SlateColor, texture : Texture2D, command : Command)`: 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 e 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;
- `ChangeCurrentChairColor(currentcolorname : String, currentcolor : SlateColor)` funzione che permette di aggiornare il colore della sedia su cui l'utente sta facendo overlap. Tale informazione viene aggiornata in seguito al click dell'utente della `ListView` contenente tutti i colori disponibili per la sedia selezionata;
- `ClearCommandHistory()` operazione che viene chiamata al termine della visualizzazione della scheda tecnica da parte dell'utente. La variabile locale `command_history` tiene traccia di tutte le ultime operazioni effettuate dall'utente sulla sedia corrente per il periodo in cui visualizza l'elemento della collezione.

Come anticipato in precedenza, gestisce la comunicazione con la classe `View` attraverso il bind di eventi chiamati `Notify*()`. Ogni Notify viene chiamato quando l'aggiornamento del modello comporta un aggiornamento anche della vista. Segue una descrizione dell'uso specifico delle chiamate:

- `NotifyOverlapChair()`: notifica il cambiamento di stato della `overlap_chair` (vedi UpdateFromModel());
- `NotifyAddCommand()`: notifica l'avvenuta aggiunta di una sedia al carrello (vedi AddCommandModel());
- `NotifyUndoCommand()`: notifica l'avvenuto annullamento dell'ultima aggiunta di una sedia al carrello (vedi UndoCommandModel());
- `NotifyColorChanged()`: notifica l'avvenuto aggiornamento del colore di `overlap_chair` (vedi UpdateTitleColor());
- `NotifyHistoryCleared()`: notifica di avvenuto svuotamento della lista di comandi (vedi HistoryCleared());
- `NotifyEmptyCommandStack()`: notifica che non ci sono più comandi reversibili (vedi EmptyStack()).

3.4.1.1 CartStructure

`CartStructure` rappresenta la struttura dati ideata per tracciare gli elementi presenti all'interno del carrello virtuale. In particolare, il modello 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:

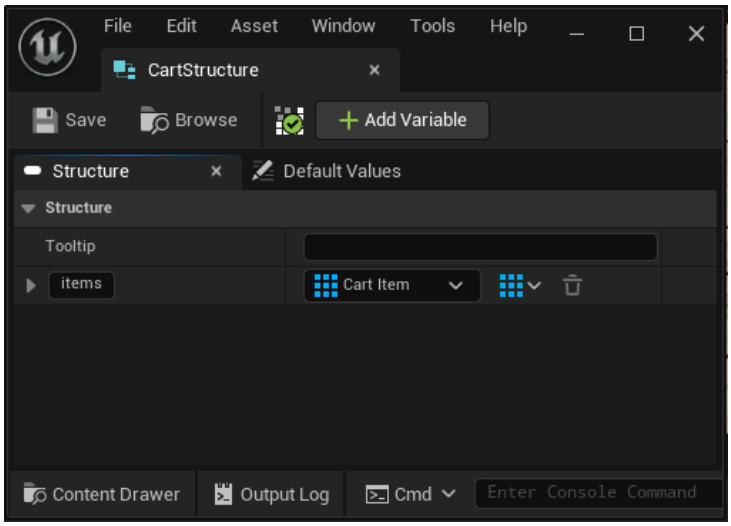

Figura 4: Definizione della struttura CartStructure

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.

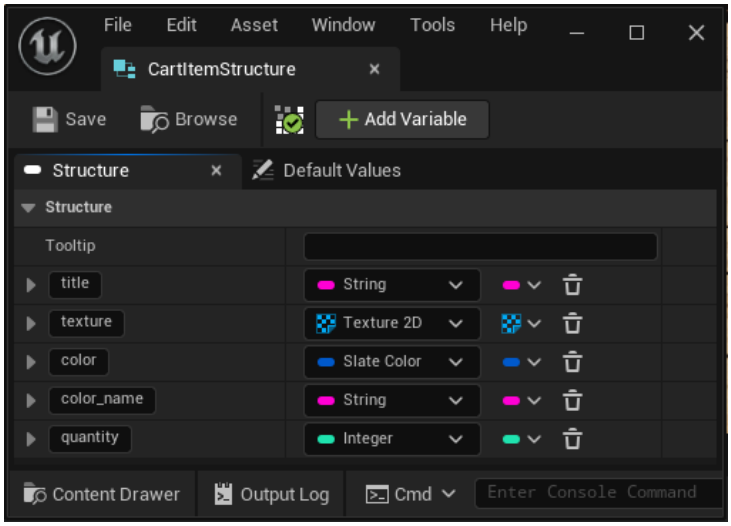

Figura 5: Definizione della struttura CartItemStructure

3.4.1.2 ChairStructure

`ChairStructure` rappresenta la struttura dati ideata per tracciare le sedie presenti all'interno dello showroom. In particolare il modello 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:

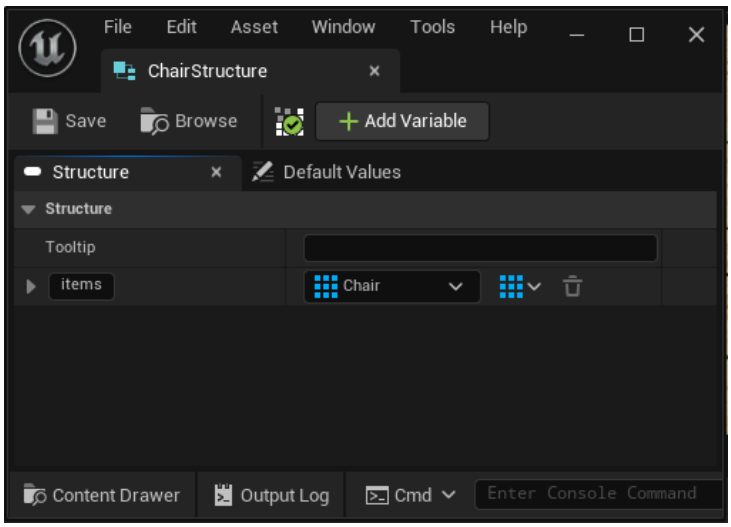

Figura 6: Definizione della struttura ChairStructure

3.4.2 View

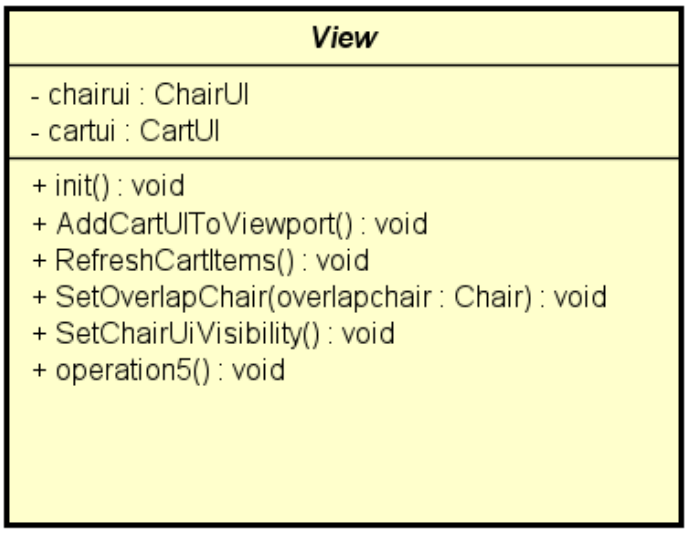

Figura 7: Diagramma delle classi relative alla View

Il diagramma delle classi **View** è costituito dall'omonima classe `View`, che possiede due campi dati di tipo `UserWidget` (fornito da *Unreal Engine*$_G$) chiamati `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 gestita interamente dal motore grafico.
Gli eventi scatenati dal click dei pulsanti sono gestiti tramite le funzioni `OnClicked()` le quali contengono un Event Listener ridefinito per ogni bottone a disposizione della vista (a seconda dello scopo).
Similmente alla classe `Model`, la classe `View` gestisce la comunicazione con la classe `Controller` attraverso il bind di eventi chiamati `Notify*()`. Ogni Notify viene chiamato quando l'utente esegue un'azione che dev'essere gestita dal `Controller`.
Segue una descrizione dell'uso specifico delle chiamate:

- `NotifyKeyboardE()`: notifica il click del tasto `E` della tastiera (vedi EPressed());
- `NotifyKeyboardQ()`: notifica il click del tasto `Q` della tastiera (vedi QPressed());
- `NotifyCartBtn()`: notifica il click del pulsante `cart_btn` della `UserWidget` `ChairUI` (vedi AddItemToCart());

- `NotifyExitButton()` notifica il click del pulsante `exit_btn` della `UserWidget ChairUI` (vedi ExitFromChairUI());
- `NotifyUndo()`: notifica il click del pulsante `undo` della `UserWidget ChairUI` (vedi Undo());
- `NotifyOnItemClicked()`: notifica il click di uno dei colori disponibili nella sedia con cui si sta interagendo (vedi OnItemClicked()).

Sviluppando un MVC_G, la classe **View** deve rispondere alle chiamate `Notify*()` lanciate dal modello, pertanto sono stati implementati i seguenti eventi custom di `update`:

- `UpdateFromModel()`: aggiorna il riferimento della sedia con cui sta interagendo l'utente;
- `AddCommandModel()`: aggiorna lo `UserWidget` così che notifichi l'utente della corretta aggiunta della sedia al carrello;
- `UndoCommandModel()`: aggiorna lo `UserWidget` così che notifichi l'utente del corretto annullamento dell'aggiunta della sedia al carrello;
- `EmptyStack()`: disattiva il bottone `undo`;
- `UpdateTitleColor()`: aggiorna il colore del titolo della sedia, con il colore che è stato selezionato dall'utente;
- `HistoryCleared()`: chiude la finestra di visualizzazione della specifica tecnica della sedia con cui si sta interagendo.

3.4.3 Controller

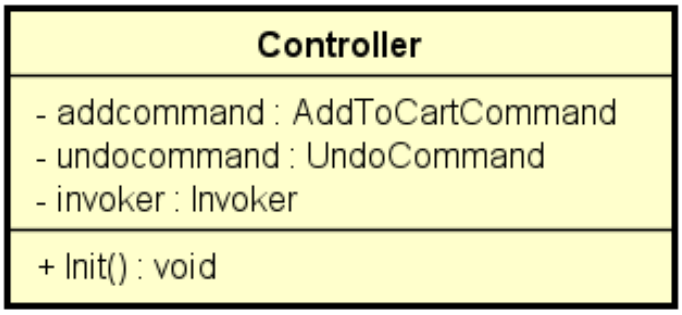

Figura 8: Diagramma delle classi relative al Controller

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. Queste connessioni permettono al sistema di garantire che la logica di business sia separata dalla presentazione grafica.
Il controller ha un riferimento alla vista ed uno al modello.
L'unica funzione messa a disposizione dal Controller è `Init`, ossia il metodo che inizializza il controller.
La classe **Controller** deve rispondere alle chiamate `Notify*()` lanciate dalla vista, pertanto sono stati implementati i seguenti eventi custom di `update`:

- `EPressed()`: gestisce l'evento generato dalla pressione del tasto `E`;
- `QPressed()`: gestisce l'evento generato dalla pressione del tasto `Q`;
- `AddItemToCart()`: gestisce l'evento generato dal click del bottone `cart_btn`;
- `Undo()`: gestisce l'evento generato dal click del bottone `undo`;
- `OnItemClicked()`: gestisce l'evento generato dal click di uno dei bottoni che determinano i colori di una sedia;
- `ExitFromChairUI()`: gestisce l'evento generato dal click del bottone `exit_btn`.

3.5 Design Pattern

Nel corso della progettazione del prodotto *ShowRoom3D*$_G$, è stato utilizzato un design pattern per garantire la modularità, l'estensibilità e la manutenibilità del codice: il pattern `Command`. Con questo pattern, è stato implementato un sistema di aggiunta/undo.
L'uso di questo pattern ci ha permesso di gestire efficacemente le interazioni degli utenti e le conseguenze delle loro azioni, fornendo una soluzione scalabile. In questa sezione, verrà esplorato nel dettaglio questo pattern.
Nell'implementazione del pattern sono state utilizzate prevalentemente la composizione e l'ereditarietà:

- La ***Composizione***$_G$ ha permesso al team di creare oggetti più complessi a partire da oggetti più semplici, che possono essere facilmente gestiti e modificati;
- L'***Ereditarietà***$_G$ 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.

3.5.1 Command

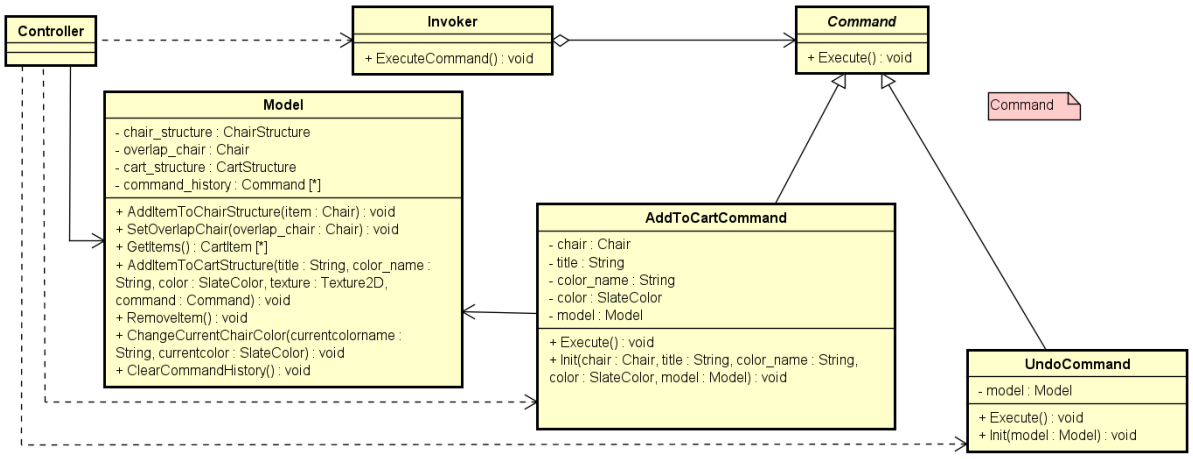

Figura 9: Diagramma delle classi relative al design pattern Command

3.5.1.1 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 la transazione.
Il pattern Command 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 di interfaccia Command per ogni operazione di aggiunta o rimozione dal carrello (che rispettivamente permetteranno di aggiungere o rimuovere oggetti dal carrello a seconda delle richieste 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 indietro nel tempo, fino al momento desiderato.

3.5.1.2 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` l'operazione disponibile è quella di aggiunta al carrello, mentre la classe derivata concreta `UndoCommand` permetterà l'annullamento della precedente operazione.

Per consentire l'**aggiunta** di un oggetto al carrello, il Controller sarà responsabile di gestire l'evento di click del pulsante specifico presente nell'interfaccia grafica `ChairUI`. Una volta che l'evento viene catturato, il Controller chiamerà l'`Invoker` che eseguirà la funzione `Execute()` e da quest'ultima verrà richiamata l'omologa funzione nella classe concreta `AddToCartCommand`. Da questa classe, il metodo `AddItemToCartStructure()` del `Receiver` (individuato nella classe `Model`) sarà chiamato per aggiungere l'oggetto specifico al carrello.

Per consentire l'**undo** di un oggetto al carrello, il Controller sarà responsabile di gestire l'evento di click del pulsante specifico presente nell'interfaccia grafica `ChairUI`. Una volta che l'evento viene catturato, il Controller chiamerà l'`Invoker` che eseguirà la funzione `Execute()` e da quest'ultima verrà richiamata l'omologa funzione nella classe concreta `UndoCommand`. Da questa classe, il metodo `RemoveItem()` del `Receiver` (individuato nella classe `Model`) sarà chiamato per eseguire l'operazione inversa alla precedente, cioè la rimozione dell'ultimo prodotto aggiunto al carrello.

Inoltre, il Receiver memorizza tutte le ultime n operazioni di aggiunta effettuate sulla sedia che l'utente sta visualizzando. Questo permette di ripristinare lo stato precedente l'aggiunta.
L'utente potrà pertanto effettuare l'operazione di **undo** un numero di volte corrispondente alla quantità di operazioni di aggiunta che sono state eseguite.

Seguono spiegazioni pratiche di come avvengono le chiamate.

Aggiunta di un oggetto al carrello

Per aggiungere un oggetto al carrello si utilizza il command `AddToCartCommand`. Dopo aver creato un'istanza del comando nel Controller, 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` presente nel `Model`) tramite il metodo `AddToCartCommand()`, in modo da poterlo ripetere o annullare in seguito. La gestione dell'aggiunta dell'oggetto è affidata al `Model` che al termine dell'operazione richiamerà il `Notify()` verso la vista per segnalare all'utente il completamento della richiesta.
Le istruzioni che vengono eseguite sono:

```
Command command = new AddToCartCommand();
Invoker invoker = new Invoker(command);
invoker.Execute();

//L'invoker chiama l'esecuzione di AddToCartCommand.Execute() da cui parte
```

```
//la chiamata:
model.AddItemToCartStructure();

//All'interno del metodo precedente verrà fatto un push nello storico
//delle chiamate dei comandi tramite la funzione:
commandHistory.Add(command);
```

Annullamento dell'ultima operazione

Per annullare l'ultima aggiunta al carrello si utilizza il command `UndoCommand`. Dopo aver creato un'istanza del comando nel Controller, 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. È necessario recuperare l'operazione svolta eseguendo il metodo `RemoveItem()` del `Model` che recupera l'ultimo oggetto aggiunto il carrello a partire dal comando che viene rimosso dallo stack `commandHistory`. La gestione della rimozione dell'oggetto è affidata al `Model` che al termine dell'operazione richiamerà il `Notify()` verso la vista per segnalare all'utente il completamento della richiesta.
Tramite il comando di undo, 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 command = new UndoCommand();
Invoker invoker = new Invoker(command);
invoker.Execute();

//L'invoker chiama l'esecuzione di UndoCommand.Execute() da cui parte
//la chiamata:
model.RemoveItem();

//All'interno del metodo precedente verrà fatto un pop nello storico
//delle chiamate dei comandi tramite la funzione:
commandHistory.Remove(command);
```

3.6 Architettura di Unreal Engine

Il *Gameplay Framework* di *Unreal Engine*$_G$ 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 della *HUD*$_G$ e telecamere per i giocatori.

La rappresentazione dei giocatori e degli *NPC*$_G$ 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_G, 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_G e un oggetto *PlayerCameraManager* per la gestione della telecamera.

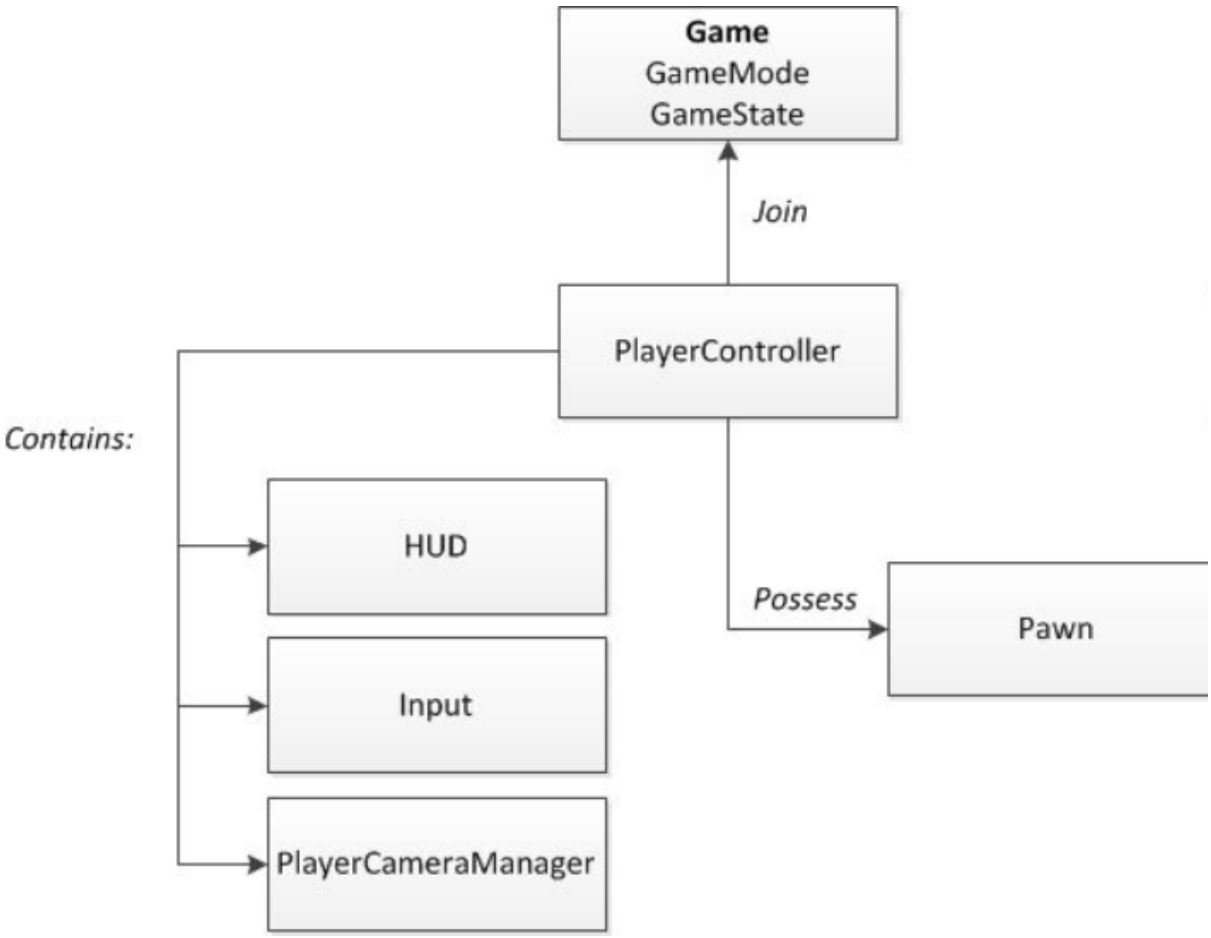

Figura 10: Gameplay Framework di Unreal Engine

4 Architettura di deployment

Per effettuare un rilascio del progetto è necessario:

1. Compilare il progetto tramite *Unreal Editor*$_G$;
2. Pubblicare una *Release*$_G$ 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*$_G$ version of the game for distribution**;
    - Inserire il numero della versione della *Release*$_G$.
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*$_G$ 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 $Release_G$;
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 $Release_G$.

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**.