

# SEVEN CLICKERS

## You Wish We Click

*7clickersgroup@gmail.com*

## Specifica Tecnica

<b>Versione</b>	0.0.8
<b>Stato</b>	In Sviluppo
<b>Uso</b>	Esterno
<b>Approvazione<sub>g</sub></b>	-
<b>Redazione</b>	Mirko Stella
<b>Verifica<sub>g</sub></b>	-
<b>Distribuzione</b>	<i>Seven Clickers</i> Prof. Vardanega Tullio Prof. Cardin Riccardo

### **Descrizione**

*Specifica Tecnica* dell'architettura di progetto

## Registro delle modifiche

Vers.	Data	Autore	Ruolo	Descrizione
0.0.8	15-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione StoreDiagram
0.0.7	13-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione InterfaceFeaturesDiagram
0.0.6	12-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione SidebarFeaturesDiagram
0.0.5	11-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione PlayerFeaturesDiagram
0.0.4	08-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione CartFeaturesDiagram
0.0.3	07-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Design pattern architetturale determinato dalle tecnologie adottate
0.0.2	05-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Elenco dei componenti
0.0.1	04-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Introduzione
0.0.0	03-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Creazione struttura documento

## Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Glossario . . . . .	3
1.2	Scopo del documento . . . . .	3
1.3	Suggerimenti per la comprensione del documento . . . . .	3
1.4	Riferimenti . . . . .	3
1.4.1	Riferimenti normativi . . . . .	3
1.4.2	Riferimenti informativi . . . . .	3
<b>2</b>	<b>Descrizione dell'architettura</b>	<b>3</b>
2.1	Elenco dei componenti . . . . .	3
2.2	Design pattern architetturale determinato dalle tecnologie adottate . . . . .	7
2.2.1	Redux-Toolkit . . . . .	7
2.2.2	React-three-fiber . . . . .	8
2.3	Architettura logica . . . . .	8
2.3.1	CartFeaturesDiagram . . . . .	9
2.3.2	PlayerFeaturesDiagram . . . . .	12
2.3.3	SidebarFeaturesDiagram . . . . .	14
2.3.4	InterfaceFeaturesDiagram . . . . .	15
2.3.5	StoreDiagram . . . . .	16
2.3.6	SkyLevelReactComponentsHierarchy . . . . .	17
2.3.7	UIReactComponentsHierarchy . . . . .	18
2.3.8	CanvasReactComponentsHierarchy . . . . .	19
2.4	Architettura di deployment . . . . .	20
2.5	Idiomi e pattern di livello più basso . . . . .	20
2.6	Altri aspetti di design . . . . .	20

## Elenco delle figure

1	UML delle classi CartFeaturesDiagram . . . . .	9
2	UML delle classi PlayerFeaturesDiagram . . . . .	12
3	UML delle classi SidebarFeaturesDiagram . . . . .	14
4	UML delle classi InterfaceFeaturesDiagram . . . . .	15
5	UML delle classi StoreDiagram . . . . .	16
6	UML delle classi SkyLevelReactComponentsHierarchy . . . . .	17
7	UML delle classi UIReactComponentsHierarchy . . . . .	18
8	UML delle classi CanvasReactComponentsHierarchy . . . . .	19

## Elenco delle tabelle

# 1 Introduzione

## 1.1 Glossario

In questo documento sono state segnate con il pedice "g" tutte le parole che, secondo noi, necessitano di una spiegazione ulteriore per evitare eventuali ambiguità o incomprensioni.

La spiegazione di questi termini la si può trovare nel documento di *Glossario*.

## 1.2 Scopo del documento

Il documento *Specifica Tecnica* ha lo scopo di descrivere i componenti utilizzati e le scelte progettuali fatte per la realizzazione del prodotto.

Dopo aver fornito un elenco descrittivo dei componenti verranno spiegati nel dettaglio, utilizzando l'ausilio degli schemi UML, i seguenti punti di interesse:

- Design pattern architetturale determinato dalle tecnologie adottate;
- Architettura logica (connessioni e interazioni tra componenti);
- Architettura di deployment (l'allocazione di componenti nel sistema in esecuzione);
- Idiomi, pattern di livello più basso che architetturale;
- Ogni altro aspetto progettuale che valorizza o caratterizza il design utilizzato.

## 1.3 Suggerimenti per la comprensione del documento

Per comprendere al meglio l'architettura utilizzata è importante comprendere in primo luogo il design pattern architetturale determinato dalle tecnologie adottate in quanto gran parte delle scelte fatte si basano su di esso.

Si suggerisce quindi di leggere la sezione 2.2 riguardante il design pattern architetturale determinato dalle tecnologie adottate prima di proseguire con le successive.

## 1.4 Riferimenti

### 1.4.1 Riferimenti normativi

- .

### 1.4.2 Riferimenti informativi

- .

# 2 Descrizione dell'architettura

## 2.1 Elenco dei componenti

- **Slices** Le Slice contengono una porzione dello stato globale dell'applicazione che è gestita da un reducer specifico.
  - **CartSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il carrello;
  - **ProductsSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti i prodotti presenti all'interno dell'ambiente 3D;
  - **PlayerSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il player;

- **RayCasterSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il prodotto con cui è possibile interagire nell'ambiente 3D;
  - **DecorationSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il caricamento delle decorazioni all'interno dell'ambiente 3D;
  - **SidebarSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti la side-bar;
  - **DecorationsSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti le decorazioni presenti all'interno dell'ambiente 3D.
- **Initial states** Gli stati iniziali contengono i dati di interesse comune tra componenti in modo da facilitarne il reperimento. I dati che non sono di interesse comune risiedono internamente ai componenti che li utilizzano.
    - **CartInitialState**: componente che contiene i dati relativi agli oggetti presenti all'interno del carrello;
    - **ProductsInitialState**: componente che contiene i dati relativi ai prodotti presenti all'interno dell'ambiente 3D;
    - **PlayerInitialState**: componente che contiene i dati relativi al player;
    - **RayCasterInitialState**: componente che contiene i dati utili ad interagire con un oggetto presente all'interno dell'ambiente 3D;
    - **DecorationInitialState**: componente che contiene la lista di tutte le decorazioni presenti all'interno dell'ambiente 3D;
    - **SidebarInitialState**: componente che contiene i dati utili a reperire informazioni sulla side-bar;
    - **DecorationInitialState**: componente che contiene i dati utili a reperire informazioni sulle decorazioni presenti all'interno dell'ambiente 3D.
  - **Actions** Un'action rappresenta un'operazione che viene eseguita nell'applicazione. Le azioni sono emesse dai componenti o da altre parti dell'applicazione e sono inviate ai reducers per aggiornare lo stato corrispondente. Per convenzione il nome delle azioni seguono il formato: [nome Slice dalla quale viene catturata].[nome azione].
    - **sidebar.toggleSidebarIsOpen**: azione emessa quando si apre o si chiude la sidebar relativa ad un prodotto presente all'interno della scena 3D.  
Payload: nessuno;
    - **products.setSelectedColor**: azione emessa quando si seleziona nella sidebar un nuovo colore per un prodotto.  
Payload: PayloadSetSelectedColor {id:number, selectedColor: String};  
id: ID del prodotto a cui cambiare il colore.  
selectedColor: colore selezionato;
    - **rayCaster.setLastProductPointed**: azione emessa per aggiornare l'ID che rappresenta l'ultimo oggetto puntato dal raycaster.  
Payload: {id:number}  
id: ID dell'ultimo prodotto puntato dal raycaster;
    - **rayCaster.toggleRayCasterEnabled**: azione emessa per abilitare o disabilitare il raycaster.  
Payload: nessuno;
    - **cart.addItem**: azione emessa per aggiungere uno o più prodotti al carrello.  
Payload: PayloadAddItems id: number, price: number, quantity: number, selectedColor: String  
id: ID del prodotto da aggiungere al carrello.  
price: prezzo del prodotto da aggiungere al carrello.  
quantity: quantità di prodotti da aggiungere al carrello.  
selectedColor: colore selezionato del prodotto da aggiungere al carrello.

- **cart.removeItem**: azione emessa per rimuovere un prodotto dal carrello.  
Payload: {id:number}  
id: ID del prodotto da rimuovere al carrello.
- **cart.removeAll**: azione emessa per rimuovere tutti i prodotti presenti nel carrello.  
Payload: nessuno;

## • Model components

Classi offerte dalle librerie utilizzate:

- **Camera**: classe offerta dalla libreria three.js. Rappresenta l'utente all'interno dell'ambiente 3D. Modificando gli attributi di questa classe l'utente compie movimenti spaziali e può esplorare l'ambiente che lo circonda da prospettive diverse;
- **Octree**: classe offerta dalla libreria three.js. L'Octree è una struttura dati ad albero utilizzata per la rappresentazione e la gestione di dati spaziali tridimensionali. In particolare, viene utilizzata per dividere lo spazio tridimensionale in regioni più piccole, suddividendo ciascuna regione in otto sotto-regioni. Utile per determinare con quale prodotto si sta interagendo o per rilevare le collisioni;
- **Vector3**: classe offerta dalla libreria three.js. Rappresenta un vettore tridimensionale, ovvero una grandezza fisica caratterizzata da una direzione e da una lunghezza. Il vettore tridimensionale viene comunemente utilizzato per definire la posizione, la rotazione, la scala e la direzione degli oggetti all'interno di una scena 3D;
- **Capsule**: classe offerta dalla libreria three.js. La classe Capsule offre la possibilità di impostare una forma di collisione personalizzata per l'oggetto rappresentato dalla capsula;
- **Store**: Lo store in Redux è l'oggetto che tiene traccia dello stato dell'applicazione. Contiene il reducer che specifica come le azioni influenzano lo stato e offre metodi per accedere allo stato corrente, inviare azioni e registrare funzioni di callback per essere avvisati quando lo stato cambia;
- **RootReducer**: componente Redux che combina tutti i reducers della tua applicazione in uno stato globale. Questa funzione viene passata allo store per gestire lo stato complessivo dell'applicazione.

Classi definite dall'utente:

- **CartItem**: classe che rappresenta un item all'interno del carrello. Quando viene aggiunto un prodotto al carrello viene creato un item che ne raccoglie le informazioni utili alla sua visualizzazione all'interno del carrello e all'acquisto;
- **Product**: classe che rappresenta un prodotto acquistabile dall'utente;
- **PayloadAddItems**: classe che rappresenta il payload da passare ad una AddItems action;
- **Decoration**: classe che rappresenta una decorazione all'interno della scena;
- **DecorationModelGLTF**: classe che rappresenta il modello 3D di una decorazione da caricare all'interno della scena;
- **PayloadSetSelectedColor**: classe che rappresenta il payload da passare ad una SetSelectedColor action;
- **ModelGLTF**: classe che rappresenta il modello 3D di un prodotto acquistabile da caricare all'interno della scena;
- **Color**: classe che rappresenta un colore che può essere associato ad un ModelGLTF;
- **Coordinate**: classe utilizzata per indicare un punto all'interno dell'ambiente 3D;
- **Player**: classe con il compito di tenere traccia ed aggiornare le informazioni riguardanti lo stato dell'utente all'interno dell'ambiente 3D. Le informazioni principali di cui si occupa la classe sono la posizione spaziale, la posizione della camera e la velocità con cui si muove l'utente.

## • UI React components

I seguenti componenti hanno il compito di visualizzare i dati del modello e costituiscono l'interfaccia utente.

- **UI**: componente React che contiene i componenti grafici che vanno a costituire l'interfaccia utente;
- **Crosshair**: componente react-three-fiber che viene utilizzato per aiutare l'utente a puntare un determinato punto nello spazio 3D. Solitamente viene indicato con un punto o una croce al centro dello schermo ed usato come puntatore dall'utente;
- **PlayerPosition**: componente React con il compito di visualizzare a schermo la posizione in cui si trova l'utente all'interno della scena 3D;
- **Cart**: componente React che rappresenta il carrello;
- **CartItem**: componente React che rappresenta un prodotto all'interno del carrello. Contiene le informazioni basilari necessarie alla visualizzazione e all'acquisto di un prodotto;
- **ProductUI**: componente React che contiene i componenti dell'interfaccia utente per la visualizzazione del ProductInteractionPrompt e della sidebar;
- **ProductInteractionPrompt**: componente React che contiene i componenti grafici che vanno a costituire la parte dell'interfaccia utente relativa alla visualizzazione del nome del prodotto con cui si sta interagendo;
- **Sidebar**: componente React che contiene i componenti necessari a visualizzare i dettagli del prodotto con cui si sta interagendo e i componenti con cui è possibile modificare le caratteristiche del prodotto e aggiungerne la quantità specificata al carrello;
- **ColorSelector**: componente React che contiene i componenti necessari per compiere un'azione di selezione colore;
- **SelectColorItem**: componente React con cui è possibile compiere un'azione di selezione colore;
- **ProductDetails**: componente React per la visualizzazione dei dettagli di un prodotto;

## • 3D React components I seguenti componenti hanno il compito di visualizzare i dati del modello costituiscono il canvas sul quale viene renderizzata la scena 3D.

- **Provider**: Il Provider in Redux è un componente React che consente di rendere lo store accessibile a tutti i componenti dell'applicazione. Viene avvolto intorno al componente principale dell'applicazione e accetta lo store come prop;
- **App**: componente radice dell'applicazione che contiene tutti gli altri componenti. È avvolto dal Provider di Redux per fornire accesso allo store a tutti i componenti figli;
- **Canvas**: componente react-three-fiber che comprende gli elementi grafici che vanno a costituire la scena 3D. Il componente Canvas si occupa di creare un nuovo canvas HTML e di associarlo a una nuova istanza di THREE.WebGLRenderer, che viene utilizzata per renderizzare la scena Three.js all'interno del componente React. Fornisce una camera e una serie di props opzionali utili alla configurazione dell'ambiente 3D;
- **Scene**: componente react-three-fiber che rappresenta l'ambiente 3D. Si tratta di uno spazio virtuale 3D in cui gli oggetti vengono posizionati, orientati e illuminati;
- **PointerLock**: componente react-three-fiber che consente di "bloccare" il puntatore del mouse all'interno di un elemento specifico della pagina web. Permette quindi di controllare la camera e di disattivare i controlli della camera quando necessario (ad esempio quando si apre un menu di interazione con un prodotto);
- **Environment**: componente react-three-fiber utilizzato per creare una scena base configurata secondo impostazioni standard settabili;

- **Map:** componente React che contiene i componenti che costituiscono la scena. Tra essi possiamo trovare: `player`, `lights`, `models`, `decorations`, `raycaster`;
- **FlashLight:** componente che rappresenta una luce direzionale utilizzata dall'utente per illuminare l'ambiente 3D;
- **Player:** componente React che contiene il player;
- **Lights:** componente React che rappresenta le luci all'interno della scena;
- **Models:** componente React che rappresenta un prodotto all'interno dell'ambiente 3D; I modelli vengono caricati all'interno della scena specificandone il path del modello `.gltf`, la posizione in cui si vogliono collocare e le caratteristiche del prodotto legato al modello da caricare;
- **Decorations:** una decorazione è un modello `.gltf` che si differisce dal modello di un prodotto per il fatto che non è modificabile e non può essere aggiunto al carrello;
- **RayCaster:** componente `three.js` che rappresenta un oggetto utilizzato per effettuare collisioni e interazioni tra gli oggetti nella scena e il puntatore del mouse. Il raycaster genera un raggio virtuale che parte dalla posizione del puntatore del mouse e attraversa lo schermo fino a raggiungere un oggetto nella scena, se presente. È possibile quindi utilizzare questa informazione per eseguire azioni in risposta all'interazione dell'utente.

## 2.2 Design pattern architetturale determinato dalle tecnologie adottate

### 2.2.1 Redux-Toolkit

I componenti che costituiscono l'architettura utilizzata seguono il pattern offerto dalla libreria Redux-Toolkit.

Redux-Toolkit è pensato per integrarsi con React e il principale vantaggio che offre è quello di poter gestire i dati condivisi tra i componenti React in modo centralizzato semplificando la gestione dello stato globale dell'applicazione (in alternativa ogni componente React dovrebbe passare il proprio stato tramite props ai suoi diretti discendenti).

I componenti che formano l'architettura di Redux-Toolkit sono:

- **Store:** componente che contiene lo stato globale dell'applicazione.

All'avvio dell'applicazione viene configurato utilizzando `RootReducer` e i componenti che utilizzano lo stato globale fanno il `subscribe` allo *store* in modo da venire renderizzati ogni volta che un dato di interesse cambia valore. Questo modo di operare può essere visto come un pattern *Observer* in cui lo *store* è il *Subject* e gli *Observers* sono i componenti React che hanno fatto il `subscribe` allo *store*;

- **RootReducer:** componente utilizzato per configurare lo store combinando più slice;
- **Slice:** componente che contiene un proprio stato che rappresenta una porzione dello stato globale dell'applicazione, i *reducer* che operano sullo stato e i *selector* per consentire ai suoi client il reperimento dei dati.

Per definire una *slice* è buona norma raggruppare i dati in modo che siano legati da un sottoinsieme di funzionalità offerte dal sistema che lavorano su dati comuni;

- **Reducer:** componente che riceve come parametri uno stato iniziale (*InitialState*) e una *action* (composta da un *type* e un *payload*) e restituisce lo stato dopo aver operato sui dati.

React-Toolkit gestisce le chiamate ai *reducer* quindi i `dispatch` delle *action* avvengono specificando solamente l'oggetto che rappresenta il *payload*;

- **Actions:** oggetto composto da un *type* e da un *payload* di cui viene effettuato il `dispatch` quando opportuno.

Il *payload* è un oggetto che contiene i dati da passare al *reducer* che catturerà l'*action*;



- **InitialState:** componente che contiene i dati di una *slice* su cui essa opera.

Importante precisare che Redux-Toolkit utilizzando la libreria immer gestisce anche l'immutabilità dei dati in modo che i reducer restituiscano delle copie dello stato in modo che esso non possa venire modificato dall'esterno e utilizzato in modo improprio.

L'unico modo per modificare i dati dello stato globale è quindi con il dispatch di un'*action*;

- **Selector:** funzione che prende lo stato corrente di una *slice* come argomento e ritorna un sottoinsieme specifico del suo stato. In altre parole, un *selector* consente di 'selezionare' una parte specifica dello stato in modo da poterla utilizzare in modo isolato all'interno di un componente React.

### 2.2.2 React-three-fiber

Questa libreria fornisce un 'punto d'incontro' tra React (libreria javascript per la creazione di interfacce utente) e Three.js (libreria usata per la modellazione dell'ambiente 3D) semplificando la creazione dei componenti da inserire all'interno dell'ambiente 3D.

React-three-fiber rende la scrittura del codice dichiarativa creando dei componenti React 'preconfezionati' che rappresentano i componenti 3D. Questi componenti sono personalizzabili modificandone le caratteristiche tramite le props di React.

Un esempio è il componente *Canvas* che fornisce con un'unica dichiarazione la *scene* e la *camera* con una configurazione standard adatta alla maggior parte dei casi di utilizzo. Per inserire i componenti all'interno dell'ambiente è sufficiente dichiararli come figli del componente *scene*.

## 2.3 Architettura logica

Per facilitare la lettura dei diagrammi delle classi è stato scelto di organizzarli per feature in modo che ogni diagramma rappresenti i componenti che permettono l'implementazione di funzionalità specifiche. Sono presenti dei diagrammi che non seguono questa convenzione che sono utili per avere una visione generale sulle dipendenze di alcuni componenti.

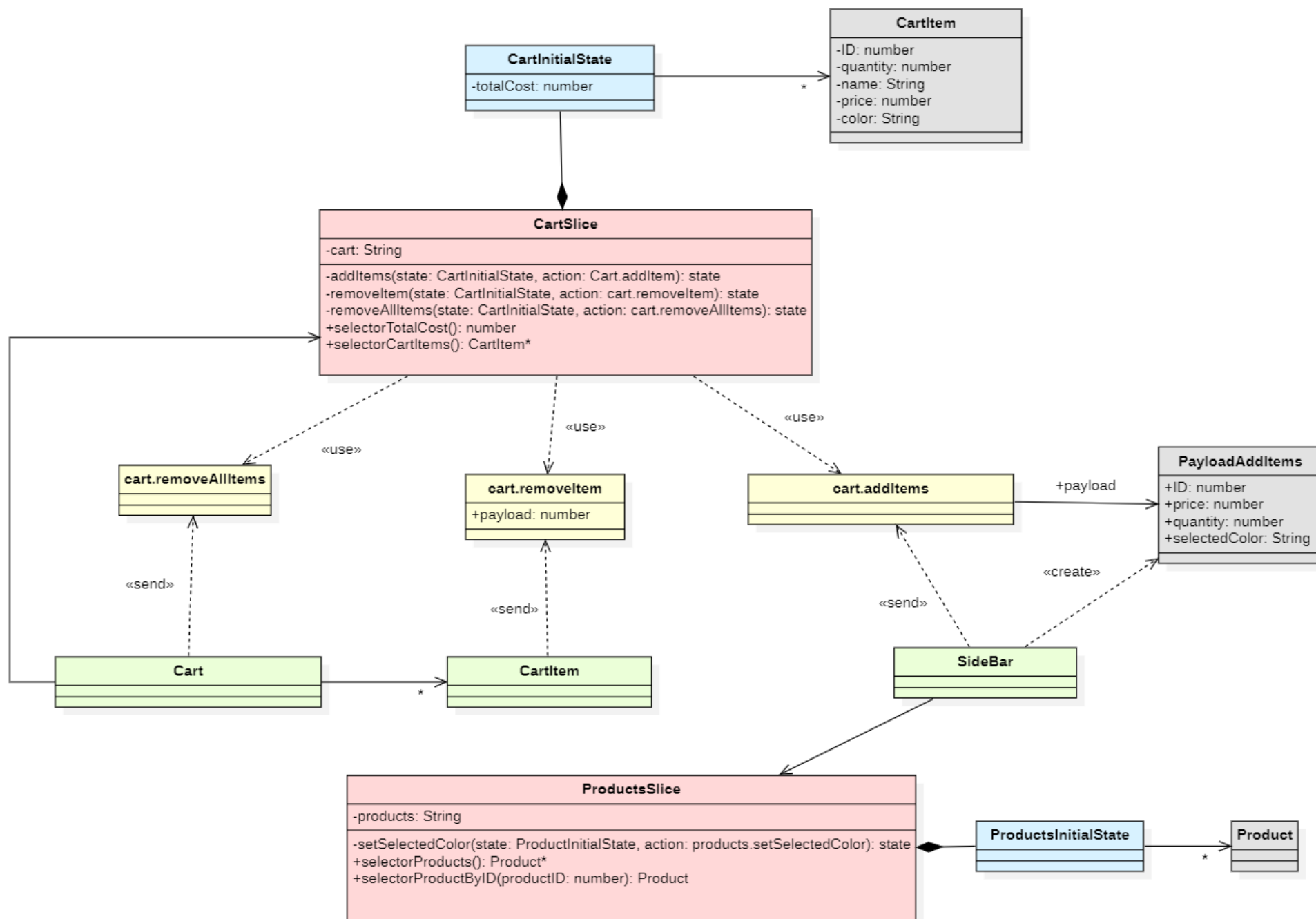
I diagrammi prodotti che rappresentano funzionalità specifiche sono:

- **CartFeaturesDiagram:** include i componenti che svolgono le funzioni riguardanti il carrello;
- **PlayerFeaturesDiagram:** include i componenti che svolgono le funzioni riguardanti le interazioni dell'utente con l'ambiente 3D;
- **SidebarFeaturesDiagram:** include i componenti che svolgono le funzioni riguardanti la side-bar;
- **InterfaceFeaturesDiagram:** include i componenti necessari per il corretto aggiornamento dell'interfaccia utente.

I diagrammi prodotti che forniscono una visione generale delle dipendenze tra componenti sono:

- **StoreDiagram:** include lo *store*;
- **SkyLevelReactComponentsHierarchy:** contiene i macrocomponenti React che costituiscono l'applicazione;
- **UIReactComponentsHierarchy:** contiene i componenti che costituiscono l'interfaccia utente;
- **CanvasReactComponentsHierarchy:** contiene i componenti che costituiscono gli elementi dell'ambiente 3D.

### 2.3.1 CartFeaturesDiagram



**Figura 1:** UML delle classi CartFeaturesDiagram.

**Legenda:** [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

**Descrizione del diagramma:** `CartFeaturesDiagram` include i componenti che svolgono le funzioni riguardanti il carrello.

- **CartSlice**

**Dipendenze:**

- *CartInitialState* (composizione): si occupa della costruzione e distruzione dell'istanza di *CartInitialState* che non viene condivisa con altri componenti in quanto *CartSlice* si occupa in modo esclusivo della sua gestione.
- *CartItem* (dipendenza semplice <create>): si occupa della costruzione dei *CartItem* da inserire all'interno della lista *items* di *CartInitialState*.
- *cart.addItem* (Dipendenza semplice <use>): cattura un'istanza di *cart.addItem* e il reducer ne utilizza il payload per aggiungere un item ad *items* in *CartInitialState*. Se le caratteristiche specificate nel payload corrispondono a quelle di un item presente ne incrementa la quantità del valore specificato altrimenti viene aggiunto un nuovo item con le caratteristiche e quantità specificate.
- *cart.removeItem* (Dipendenza semplice <use>): cattura un'istanza di *cart.removeItem* e il reducer ne utilizza il payload per decrementare la quantità dell'item specificato. Se la quantità dopo il decremento risulta essere uguale a 0 l'item viene rimosso da *items* in *CartInitialState*.
- *cart.removeAllItems* (Dipendenza semplice <use>): cattura un'istanza di *cart.removeAllItems* e chiama il reducer che svuota la lista *items* presente in *CartInitialState* ed imposta il *totalCost* a 0.

**Interazioni:**

- *CartInitialState*: viene modificato in base alle actions catturate dal reducer della slice.

**Actions catturate:**

- *cart.addItem*: utilizzata dal reducer per chiamare *addItem*.
- *cart.removeItem*: utilizzata dal reducer per chiamare *removeItem*.
- *cart.removeAllItems*: utilizzata dal reducer per chiamare *removeAllItems*.

- **CartInitialState**

**Dipendenze:**

- *CartItem* (Associazione): contiene la lista degli item che rappresentano i prodotti presenti nel carrello.

- **Cart**

**Dipendenze:**

- *CartItem* (Composizione): crea le istanze di *CartItem* durante il rendering. *Cart* gestisce interamente il ciclo di vita delle istanze di un *CartItem* che quindi per composizione non può essere condiviso con altri componenti.
- *cart.removeAllItems* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione *cart.removeAllItems*.
- *CartSlice* (Associazione): possiede come attributo implicito (dovuto a Redux) un'istanza di *CartSlice*.

**Interazioni:**

- *CartSlice*: chiama i selettori *selectorCartItems* e *selectorTotalCost* per reperire gli items e il costo totale dei prodotti allo scopo di compiere il rendering dei suoi componenti figli.

**Action emesse:**

- *cart.removeAllItems*: emessa quando l'utente rimuove tutti i prodotti dal carrello.

**• CartItemReact****Dipendenze:**

- *cart.removeItem* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione *cart.removeItem*.

**Action emesse:**

- *cart.removeItem*: emessa quando l'utente rimuove il prodotto corrispondente all'item dal carrello.

**• Sidebar****Dipendenze:**

- *ProductsSlice* (Associazione): possiede come attributo implicito (dovuto a Redux) un'istanza di *ProductsSlice*.
- *PayloadAddItems* (Dipendenza semplice <create>): crea il payload da fornire alla action *cart.addItem*.
- *cart.addItem* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione *cart.addItem*.

**Interazioni:**

- *ProductsSlice*: chiama il selettore *selectorProductByID* per reperire le informazioni del prodotto allo scopo di compiere il rendering dei suoi componenti figli.

2.3.2 PlayerFeaturesDiagram

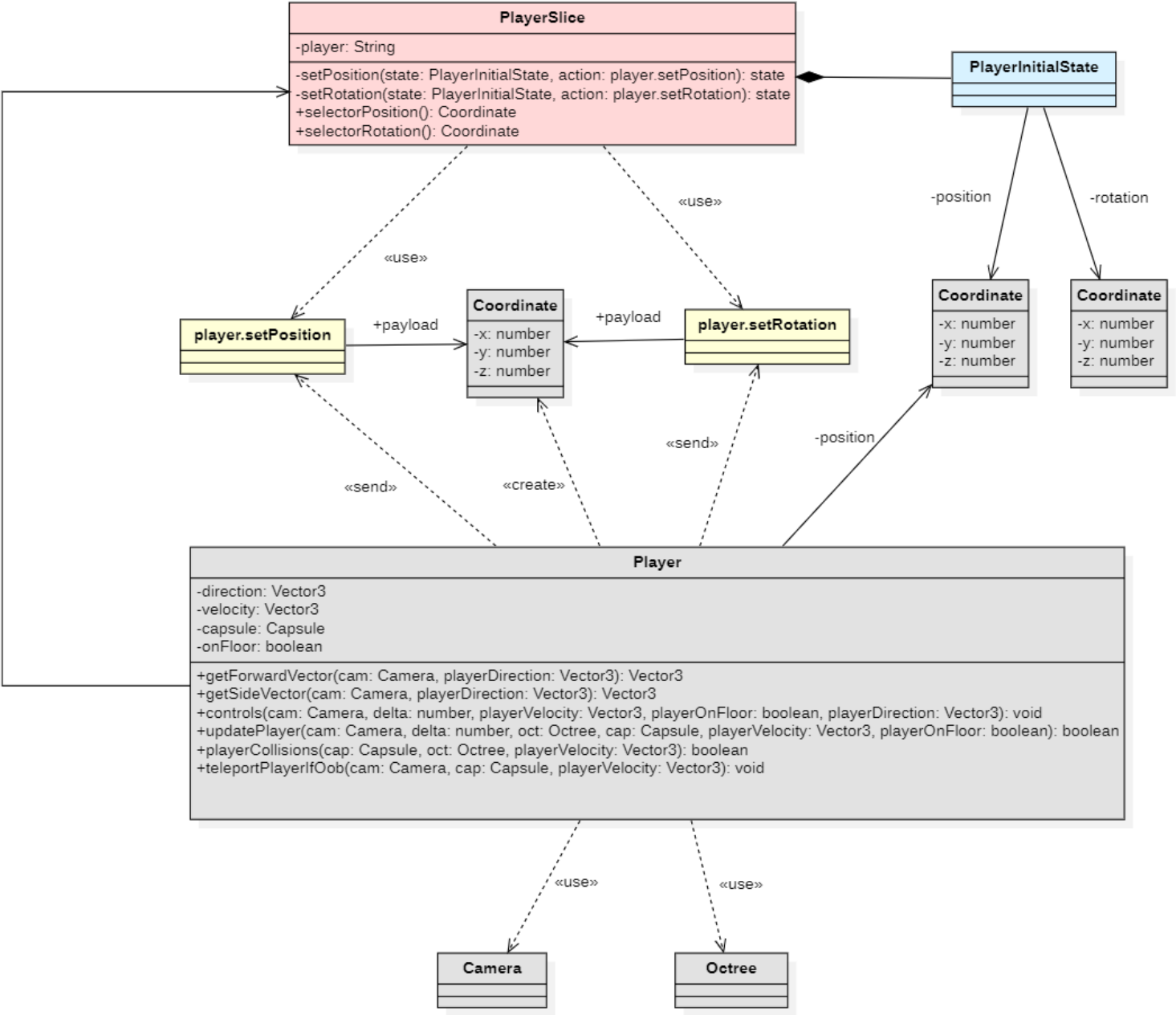


Figura 2: UML delle classi PlayerFeaturesDiagram.

Legenda: [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

**Descrizione del diagramma:** PlayerFeaturesDiagram include i componenti che svolgono le funzioni riguardanti le interazioni dell'utente con l'ambiente 3D.

- **PlayerSlice**

**Dipendenze:**

- *PlayerInitialState* (composizione): si occupa della costruzione e distruzione dell'istanza di *PlayerInitialState* che non viene condivisa con altri componenti in quanto *PlayerSlice* si occupa in modo esclusivo della sua gestione.
- *player.setPosition* (Dipendenza semplice <use>): cattura un'istanza di *player.setPosition* e il reducer ne utilizza il payload impostare l'attributo position in *PlayerInitialState*.
- *player.setRotation* (Dipendenza semplice <use>): cattura un'istanza di *player.setPosition* e il reducer ne utilizza il payload impostare l'attributo rotation in *PlayerInitialState*.

**Interazioni:**

- *PlayerInitialState*: viene modificato in base alle actions catturate dal reducer della slice.

**Actions catturate:**

- *player.setPosition*: utilizzata dal reducer per chiamare *setPosition*.
- *player.setRotation*: utilizzata dal reducer per chiamare *setRotation*.

- **PlayerInitialState**

**Dipendenze:**

- *Coordinate* (Associazione): ha un attributo di tipo *Coordinate* per position e rotation che indicano la posizione del player e la posizione della camera rispettivamente.

- **Player**

**Dipendenze:**

- *PlayerSlice* (Associazione):
- *Coordinate* (Associazione):
- *Coordinate* (Dipendenza semplice <create>):
- *Camera* (Dipendenza semplice <use>):
- *Octree* (Dipendenza semplice <use>):
- *player.setPosition* (Dipendenza semplice <send>):
- *player.setRotation* (Dipendenza semplice <send>):

**Interazioni:**

- *CartInitialState*:

**Action emesse:**

- *cart.removeItem*: emessa quando l'utente rimuove il prodotto corrispondente all'item dal carrello.

2.3.3 SidebarFeaturesDiagram

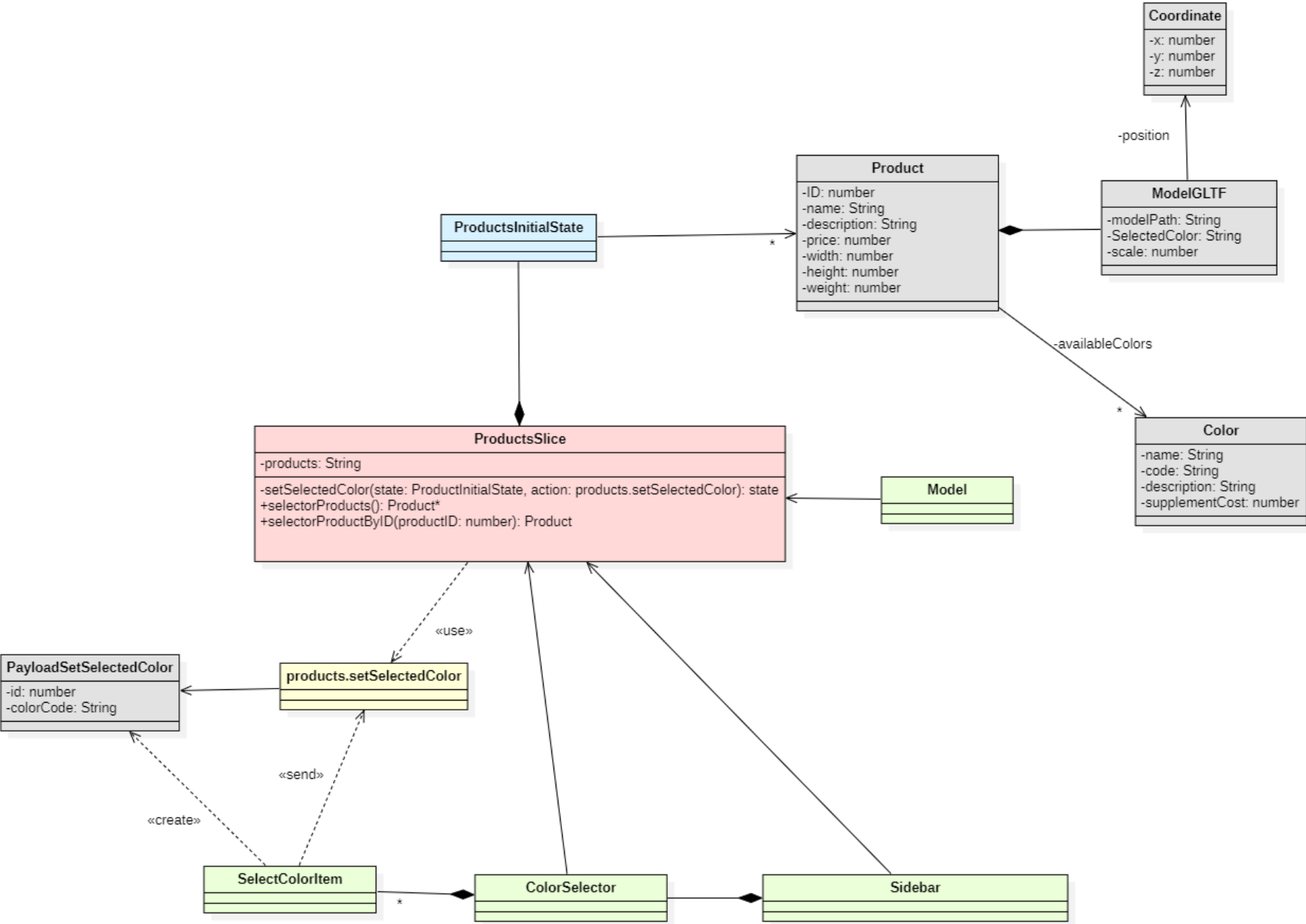


Figura 3: UML delle classi SidebarFeaturesDiagram.

Legenda: [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

2.3.4 InterfaceFeaturesDiagram

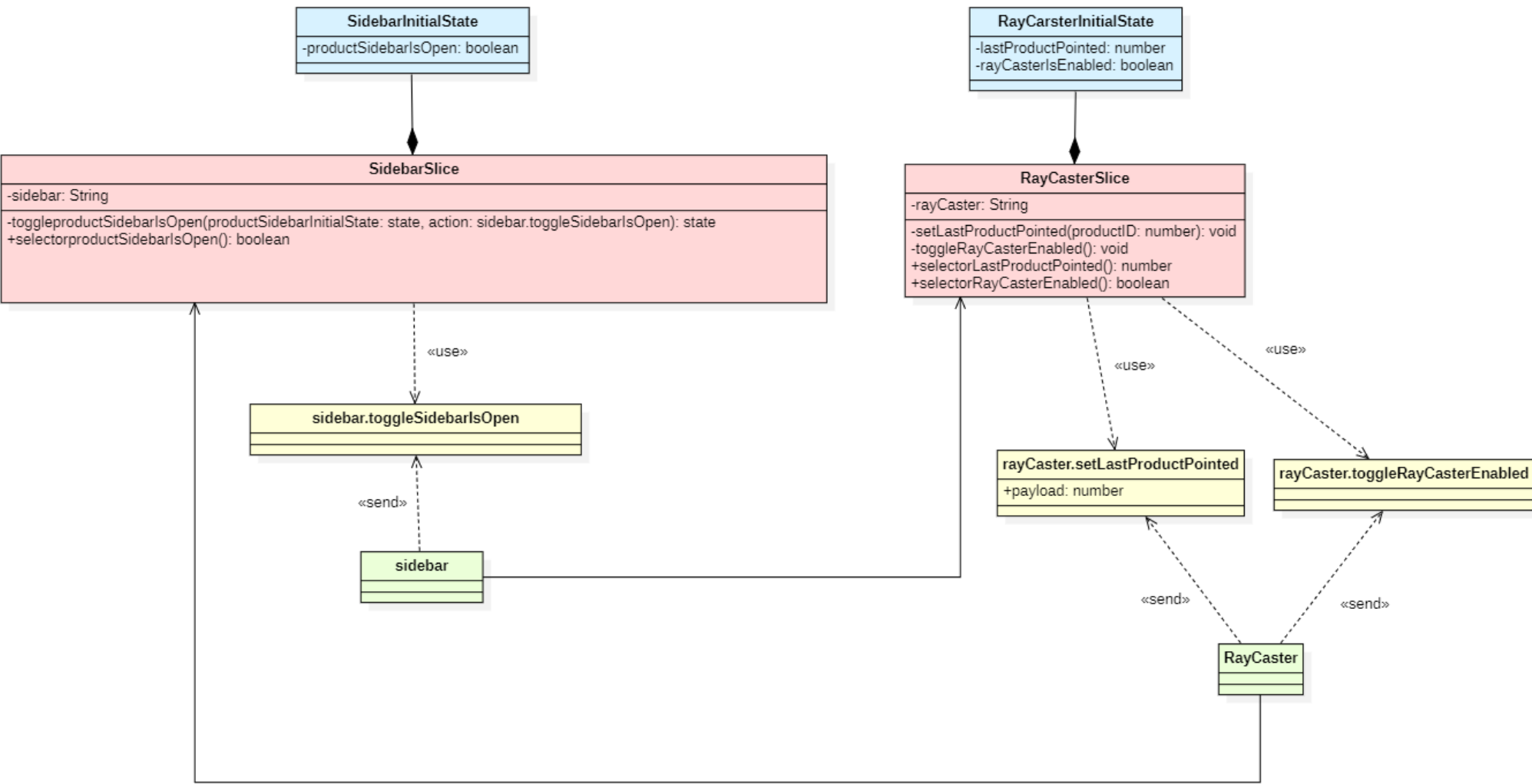


Figura 4: UML delle classi InterfaceFeaturesDiagram.

Legenda: [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]



2.3.5 StoreDiagram

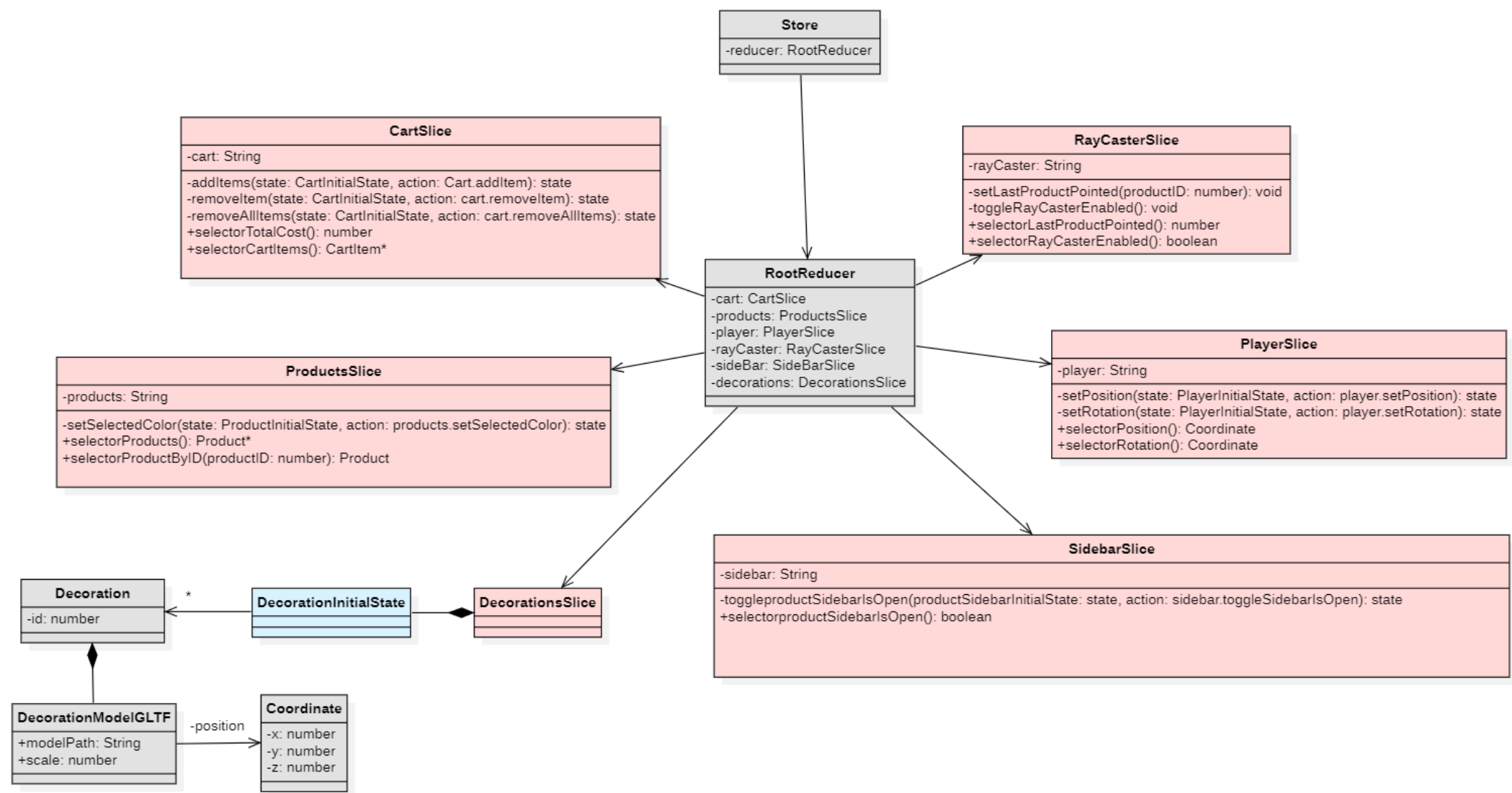
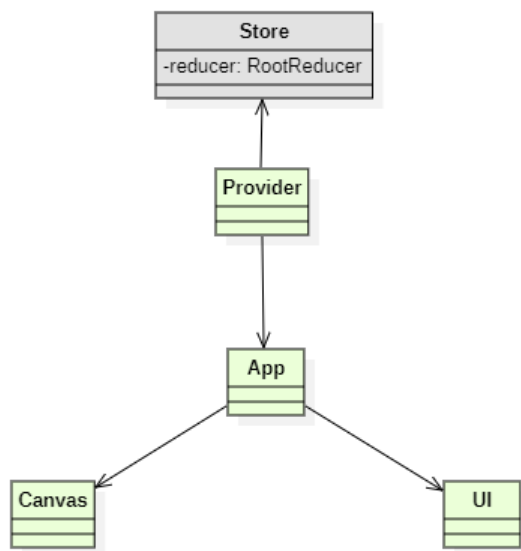


Figura 5: UML delle classi StoreDiagram.

Legenda: [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

### 2.3.6 SkyLevelReactComponentsHierarchy



**Figura 6:** UML delle classi SkyLevelReactComponentsHierarchy.

**Legenda:** [*Slices*: rosso] - [*Actions*: giallo] - [*Model classes*: grigio] - [*Initial states*: azzurro] - [*UI React components*: verde]

2.3.7 UIReactComponentsHierarchy

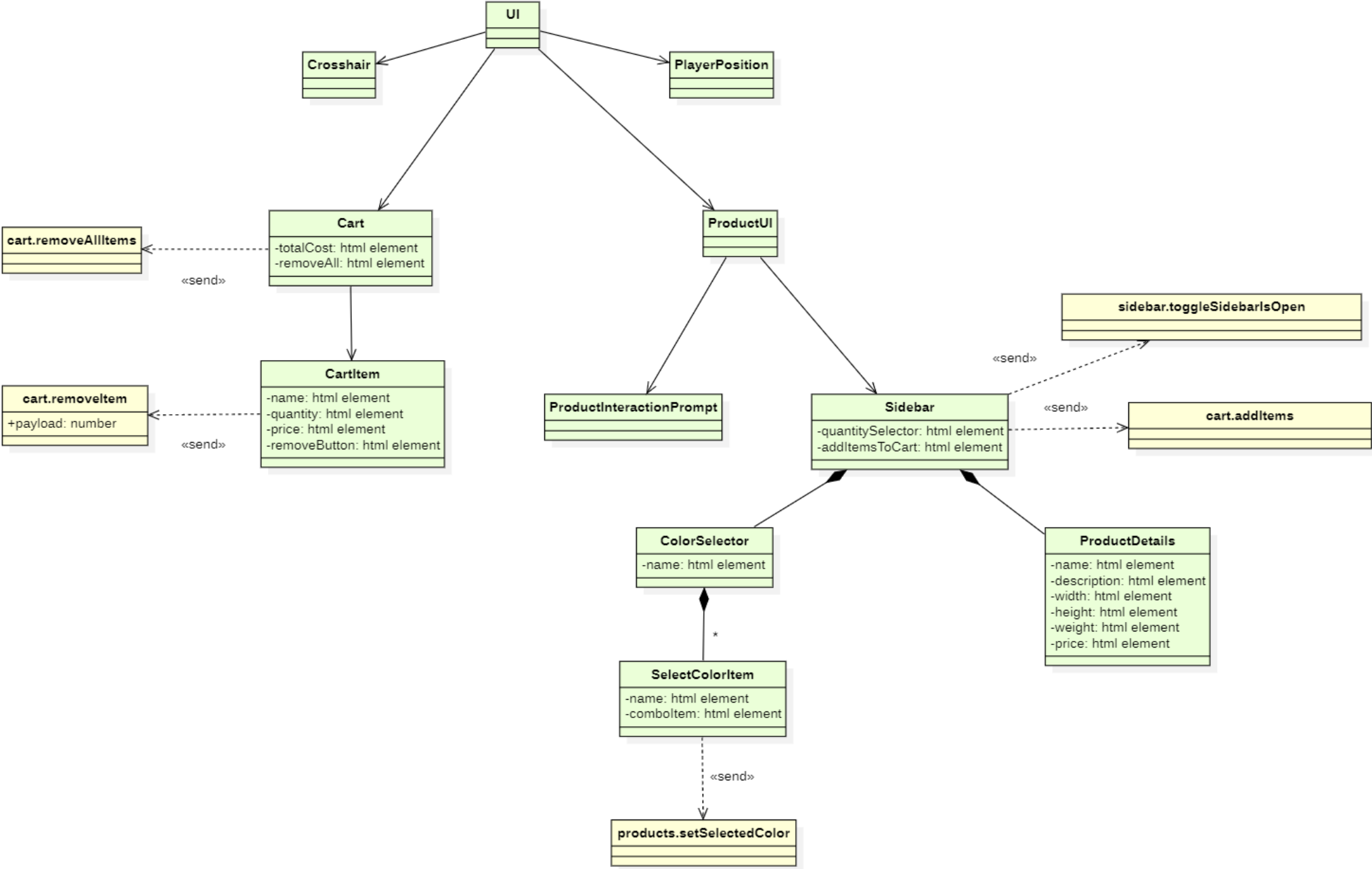
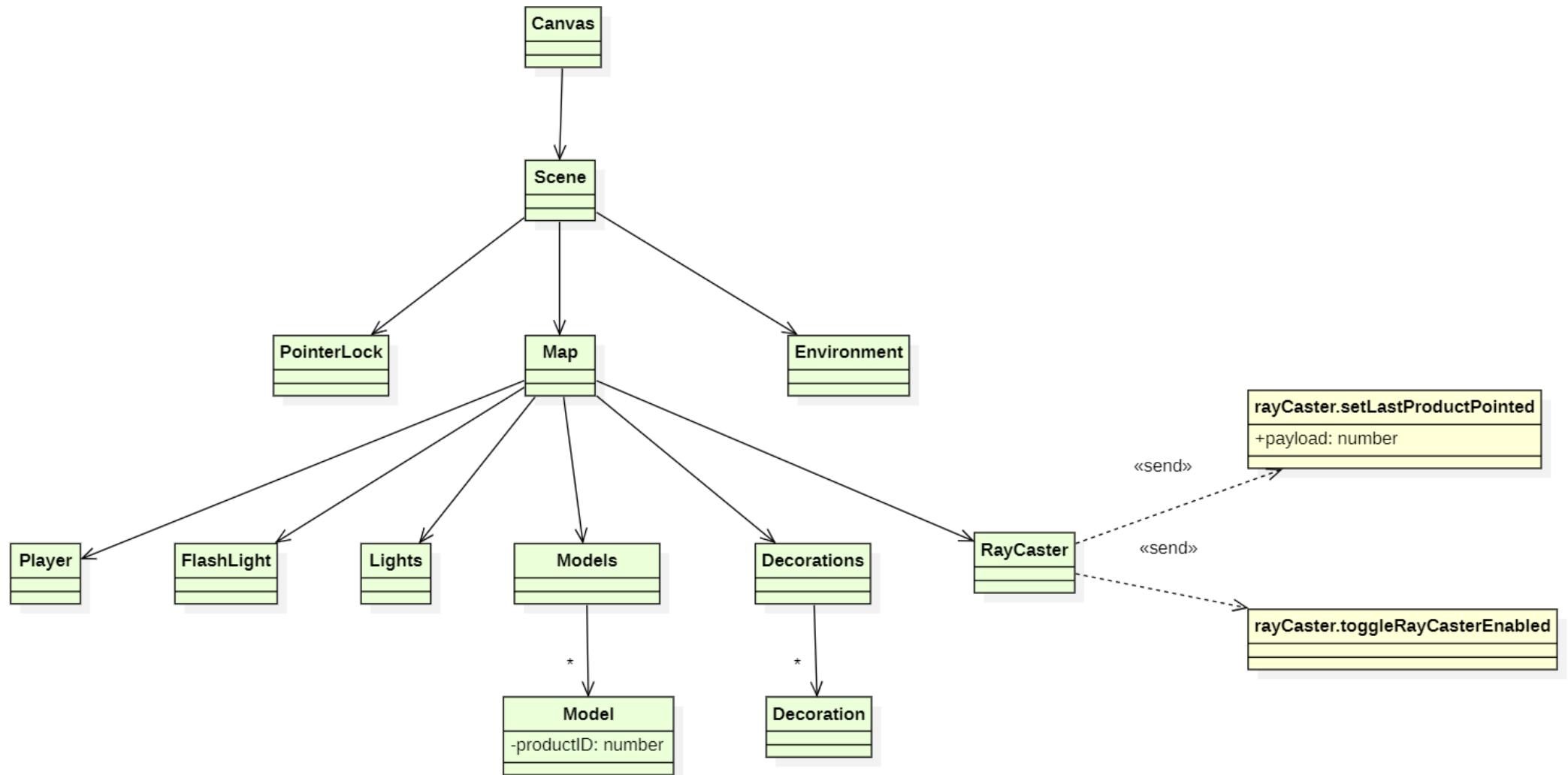


Figura 7: UML delle classi UIReactComponentsHierarchy.

Legenda: [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

### 2.3.8 CanvasReactComponentsHierarchy



**Figura 8:** UML delle classi CanvasReactComponentsHierarchy.

**Legenda:** [Slices: rosso] - [Actions: giallo] - [Model classes: grigio] - [Initial states: azzurro] - [UI React components: verde]

2.4 Architettura di deployment

2.5 Idiomi e pattern di livello più basso

2.6 Altri aspetti di design