Università Degli Studi Di Salerno

Progetto di Ingegneria del software 2018/2019

Object Design Document

Sommario

1. Introduzione

1.1 Object Design Trade-off

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto in linea di massima quello che sarà il nostro sistema e quindi i nostri obiettivi, tralasciando gli aspetti implementativi. Il seguente documento ha lo scopo di produrre un modello capace di integrare in modo coerente e preciso tutte le funzionalità individuate nelle fasi precedenti. In particolare, definisce le interfacce delle classi, le operazioni, i tipi, gli argomenti e la signature dei sottosistemi definiti nel System Design. Inoltre, sono specificati i trade-off e le linee guida.

Comprensibilità vs Tempo:
Il codice deve essere quanto più comprensibile possibile per facilitare la fase di testing ed eventuali future modifiche. Il codice sarà quindi accompagnato da commenti che ne semplifichino la comprensione. Ovviamente questa caratteristica aggiungerà un incremento di tempo allo sviluppo del nostro progetto.

Prestazioni vs Costi:
Essendo il progetto sprovvisto di budget, al fine di mantenere prestazioni accettabili, per alcune funzionalità verranno utilizzati dei template open source esterni in particolare Bootstrap in modo tale da evitare costi. In questo modo i costi saranno nulli e allo stesso tempo si avranno comunque delle prestazioni accettabili.

Interfaccia vs Usabilità:
L'interfaccia grafica è stata realizzata in modo da essere molto semplice, chiara e concisa, fa uso di form e pulsanti disposti in maniera da rendere semplice l'utilizzo del sistema da parte dell'utente finale.

Sicurezza vs Efficienza:
La sicurezza, come descritto nei requisiti non funzionali del RAD, rappresenta uno degli aspetti importanti del sistema. Tuttavia, dati i tempi di sviluppo molto limitati, ci limiteremo ad implementare sistemi di sicurezza basati su username e password degli utenti.

1.2 Linee Guida per la Documentazione delle Interfacce

Gli sviluppatori dovranno seguire alcune linee guida per la scrittura del codice:
Naming Convention
● È buona norma utilizzare nomi:
1. Descrittivi
2. Pronunciabili
3. Di uso comune
4. Lunghezza medio-corta
5. Non abbreviati
6. Evitando la notazione ungherese
7. Utilizzando solo caratteri consentiti (a-z, A-Z, 0-9)

Variabili:
● I nomi delle variabili devono cominciare con una lettera minuscola, e le parole seguenti con la lettera maiuscola. Quest'ultime devono essere dichiarate ad inizio blocco, solamente una per riga e devono essere tutte allineate per facilitare la leggibilità.

Esempio: elaboratoSessioneId
● È inoltre possibile, in alcuni casi, utilizzare il carattere **underscore** "**_**", ad esempio quando utilizziamo delle variabili costanti oppure quando vengono utilizzate delle proprietà statiche.

Esempio: CREATE_ARGOMENTO

Metodi:
● I nomi dei metodi devono cominciare con una lettera minuscola, e le parole seguenti con la lettera maiuscola. Il nome del metodo tipicamente consiste di un verbo che identifica una azione, seguito dal nome di un oggetto.

I nomi dei metodi per l'accesso e la modifica delle variabili dovranno essere del tipo getNomeVariabile() e setNomeVariabile(). Le variabili dei metodi devono essere dichiarate appena prima del loro utilizzo e devono servire per un solo scopo, per facilitare la leggibilità. Esempio: getId(), setId()

● I commenti dei metodi devono essere raggruppati in base alla loro funzionalità, la descrizione dei metodi deve apparire prima di ogni dichiarazione di metodo, e deve descriverne lo scopo. Deve includere anche informazioni sugli argomenti, sul valore di ritorno, e se applicabile, sulle eccezioni.

Classi e pagine :

- I nomi delle classi e delle pagine devono cominciare con una lettera maiuscola, e anche le parole seguenti all'interno del nome devono cominciare con una lettera maiuscola. I nomi di quest'ultime devono fornire informazioni sul loro scopo.

Ogni file sorgente java contiene una singola classe e dev'essere strutturato in un determinato modo:

● L’istruzione include che permette di importare all’interno della classe gli altri oggetti che la classe utilizza.

● La dichiarazione di classe caratterizzata da:

1. Dichiarazione della classe pubblica
2. Dichiarazioni di costanti
3. Dichiarazioni di variabili di classe
4. Dichiarazioni di variabili d’istanza
5. Costruttore
6. Commento e dichiarazione metodi.

```
/**
	 * Costruttore di Gioco
	 */
	public Gioco() {
		code = -1;
		name = "";
		description = "";
		quantity = 0;
	}

/**
	 * Costruttore di Gioco
	 */
	public Gioco(int code, int pegi, int quantity, int anno, String name,
String piattaforma, String description,
			String video, String genere, float price) {
		this.code = code;
		this.anno = anno;
		this.description = description;
		this.genere = genere;
		this.quantity = quantity;
		this.name = name;
		this.pegi = pegi;
		this.piattaforma = piattaforma;
		this.video = video;
		this.price = price;

	}

	/**
	 * @return int Il serial number del Gioco
	 */
	public int getCode() {
		return code;
	}

	/**
	 * Setta il serial number del Gioco
	 *
	 * @param int code Il serial number del Gioco
	 */

	public void setCode(int code) {
		this.code = code;
	}
```

```java
/**
 * @return string Il nome del Gioco
 */
public String getName() {
	return name;
}

/**
 * Setta il nome del Gioco
 *
 * @param string name Il nome del Gioco
 */

public void setName(String name) {
	this.name = name;
}

/**
 * @return string La descrizione del Gioco
 */
public String getDescription() {
	return description;
}

/**
 * Setta la descrizione del Gioco
 *
 * @param string description La descrizione del Gioco
 */
public void setDescription(String description) {
	this.description = description;
}

/**
 * @return float Il prezzo del Gioco
 */
public float getPrice() {
	return price;
}

/**
 * Setta il prezzo del Gioco
 *
 * @param float price Il prezzo del Gioco
 */
public void setPrice(float price) {
	this.price = price;
}

/**
 * @return int La quantità del Gioco
 */
public int getQuantity() {
	return quantity;
}

/**
 * Setta la quantità del Gioco
 *
 * @param int quantity La quantità del Gioco
 */

public void setQuantity(int quantity) {
```

```
		this.quantity = quantity;
	}

	/**
	 * @return String La piattaforma del Gioco
	 */
	public String getPiattaforma() {
		return piattaforma;
	}

	/**
	 * Setta la piattaforma del Gioco
	 *
	 * @param String piattaforma La piattaforma del Gioco
	 */
	public void setPiattaforma(String piattaforma) {
		this.piattaforma = piattaforma;
	}

	/**
	 * @return int L'anno del Gioco
	 */
	public int getAnno() {
		return anno;
	}

	/**
	 * Setta l'anno del Gioco
	 *
	 * @param int anno L'anno del Gioco
	 */
	public void setAnno(int anno) {
		this.anno = anno;
	}

	/**
	 * @return String Il link al video del Gioco
	 */
	public String getVideo() {
		return video;
	}

	/**
	 * Setta il link del video del Gioco
	 *
	 * @param String video Il link del video del Gioco
	 */
	public void setVideo(String video) {
		this.video = video;
	}

	/**
	 * @return String Il genere del Gioco
	 */
	public String getGenere() {
		return genere;
	}

	/**
	 * Setta il genere del Gioco
	 *
	 * @param String genere Il genere del Gioco
	 */
```

```java
public void setGenere(String genere) {
	this.genere = genere;
}

/**
 * @return int Il PEGI del Gioco
 */
public int getPegi() {
	return pegi;
}


/**
 * Setta il PEGI del Gioco
 *
 * @param int pegi Il PEGI del Gioco
 */
public void setPegi(int pegi) {
	this.pegi = pegi;
}
```

Script Javascript

Il codice Javascript deve seguire le stesse convenzioni per il layout e i nomi del codice Java.
I documenti Javascript devono essere iniziati da un commento analogo a quello presente nei file Java.
Le funzioni Javascript devono essere documentate in modo analogo ai metodi Java, scritte possibilmente un file esterno, per consentire maggiore leggibilità al codice.

Fogli di stile CSS

Tutti gli stili non inline devono essere collocati in fogli di stile separati.
Ogni regola CSS deve essere formattata come segue:

- I selettori della regola si trovano a livello 0 di indentazione, uno per riga;
- L'ultimo selettore della regola è seguito da parentesi graffa aperta ({);
- Le proprietà che costituiscono la regola sono listate una per riga e sono indentate rispetto ai selettori, e per identificare la fine di un'istruzione si usa il punto e virgola;
- La regola è terminata da una parentesi graffa chiusa (}), collocata da sola su una riga;

Le proprietà e le regole poco chiare dovrebbero essere precedute da un commento esplicativo.

1.3 Definizioni, acronimi e abbreviazioni

Acronimi:
- RAD: Requirements Analysis Document
- SDD: System Design Document
- ODD: Object Design Document

Abbreviazioni:
- DB: Database

1.4 Riferimenti
- B. Bruegge, A. H. Dutoit, Object Oriented Software Engineering - Using UML, Pattern and Java, Prentice Hall, 3rd edition, 2009
- Documento SDD del progetto GamesHub
- Documento RAD del progetto GamesHub

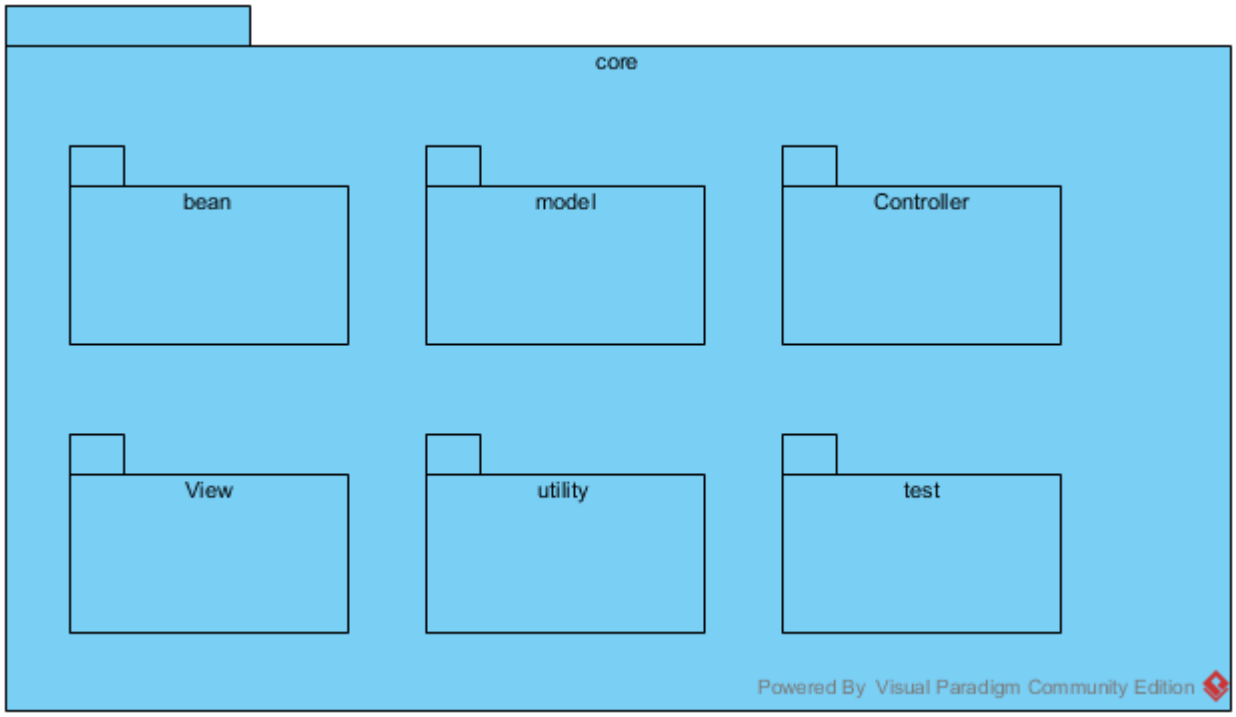
2. Packages

La gestione del nostro sistema è suddivisa in tre livelli (three-tier):

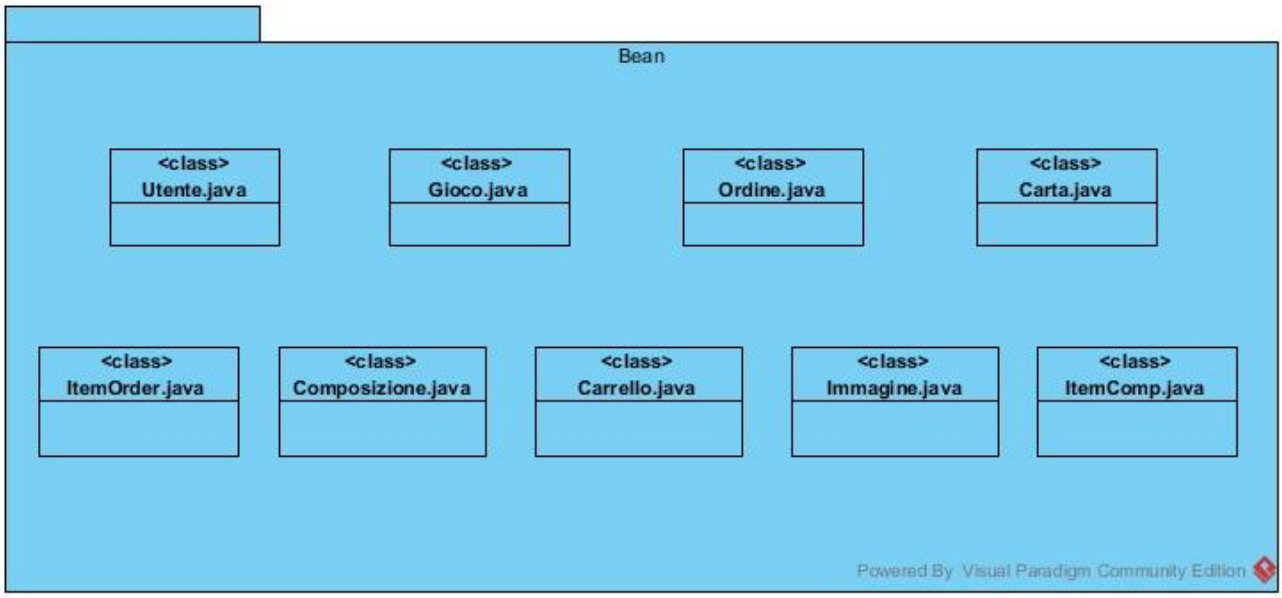
- Interface layer
- Application Logic layer
- Storage layer

Il package GamesHub contiene sotto package che a loro volta inglobano classi atte alla gestione delle richieste utente. Le classi contenute nel package svolgono il ruolo di gestore logico del sistema.

Interface layer	Rappresenta l'interfaccia del sistema, ed offre la possibilità all'utente di interagire con quest'ultimo, offrendo sia la possibilità di inviare, in input, che di visualizzare, in output, dati.
Application Logic layer	Ha il compito di elaborare i dati da inviare al client, e spesso grazie a delle richieste fatte al database, tramite lo Storage Layer, accede ai dati persistenti. Si occupa di varie gestioni quali: **1. Gestione Utente** **2. Gestione Carrello** **3. Gestione Ordini** **4. Gestione Catalogo**
Storage layer	Ha il compito di memorizzare i dati sensibili del sistema, utilizzando un DBMS, inoltre riceve le varie richieste dall' Application Logic layer inoltrandole al DBMS e restituendo i dati richiesti.



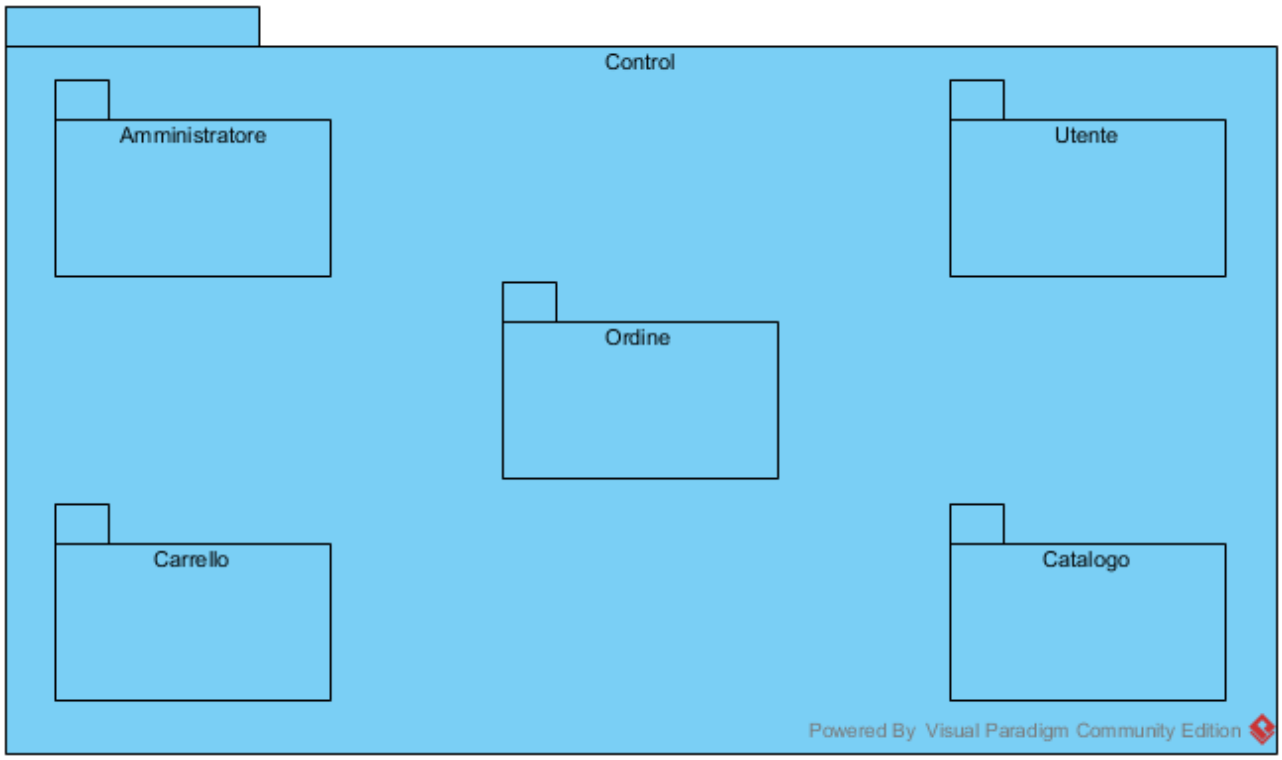
2.1 Package Bean



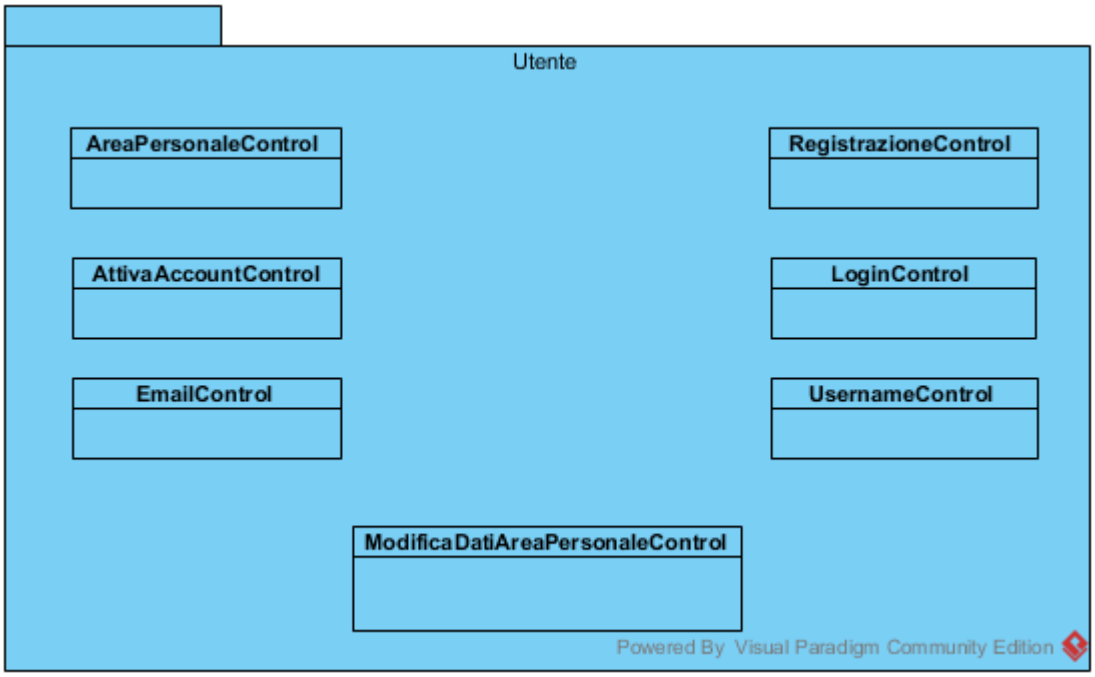
Classe:	Descrizione:
Utente.java	Descrive un utente registrato al sistema
Gioco.java	Descrive un gioco memorizzato nel sistema
Ordine.java	Descrive un ordine effettuato

Immagine.java	Descrive un immagine relativa ad un gioco
Composizione.java	Descrive la composizione di un ordine effettuato
Carta.java	Descrive una carta utilizzata per gli acquisti
Carrello.java	Descrive un carrello di un utente
ItemOrder.java	Descrive un ordine temporaneo
ItemComp.java	Descrive la composizione di un ordine temporaneo.

2.2 Package Control

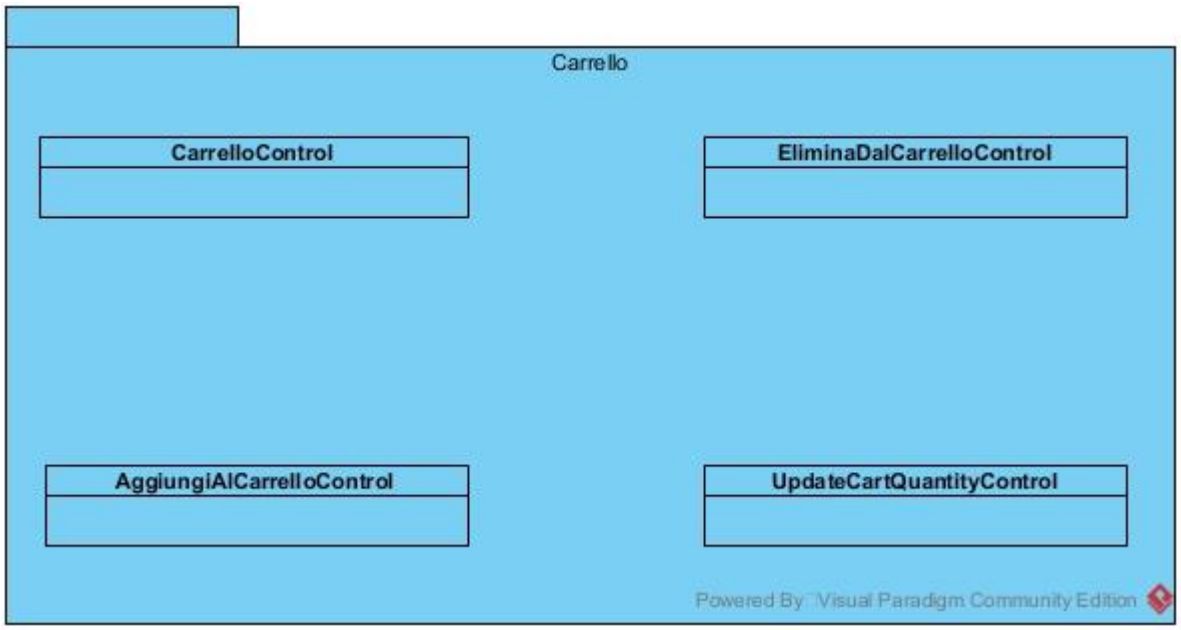


2.2.1 Package Utente



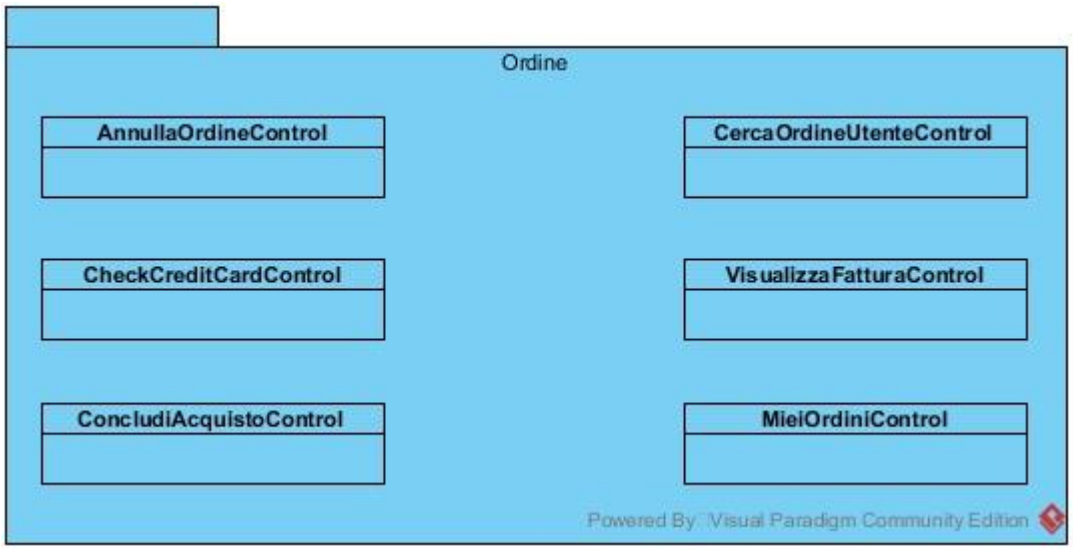
Classe:	Descrizione:
AreaPersonaleControl.java	Controller che gestisce l'area personale degli utenti
ModificaDatiAreaPersonaleControl.java	Controller che gestisce la modifica dell'area personale degli utenti
AttivaAccountControl.java	Controller che gestisce il conferma registrazione
RegistrazioneControl.java	Controller che gestisce la registrazione
LoginControl.java	Controller che gestisce il login
EmailControl.java	Controller che si occupa di verificare che l'e-mail scelta dall'utente non sia stata già utilizzata.
UsernameControl.java	Controller che si occupa di verificare che l'username scelto dall'utente non sia stato già utilizzato.

2.2.2 Package Carrello



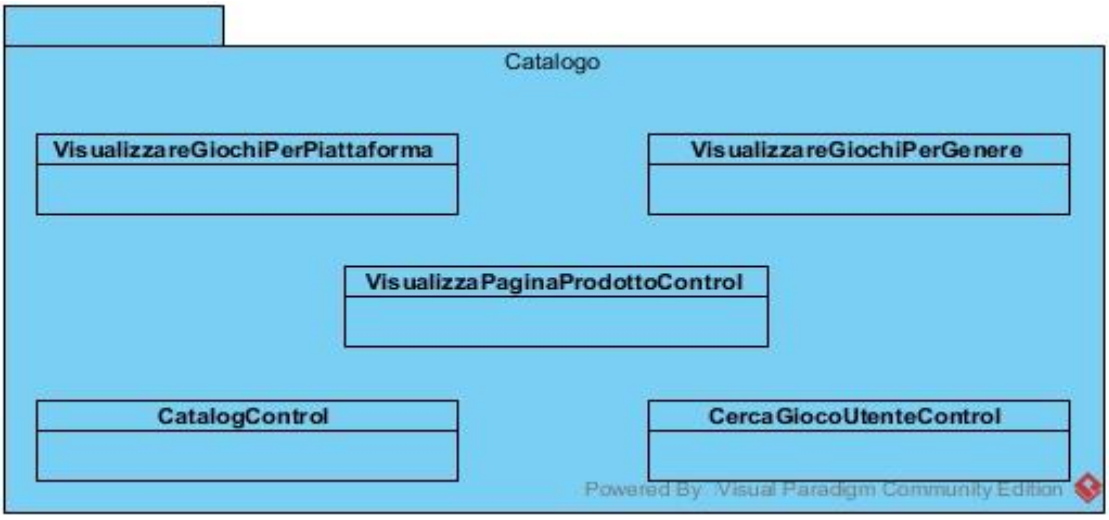
Classe:	Descrizione:
CarrelloControl.java	Controller che gestisce l'accesso al carrello
AggiungiAlCarrelloControl.java	Controller che gestisce l'inserimento di un prodotto nel carrello
EliminaDalCarrelloControl.java	Controller che gestisce l'eliminazione di un prodotto dal carrello
UpdateCartQuantityControl.java	Controller che gestisce la quantità dei prodotti nel carrello

2.2.3 Package Ordini



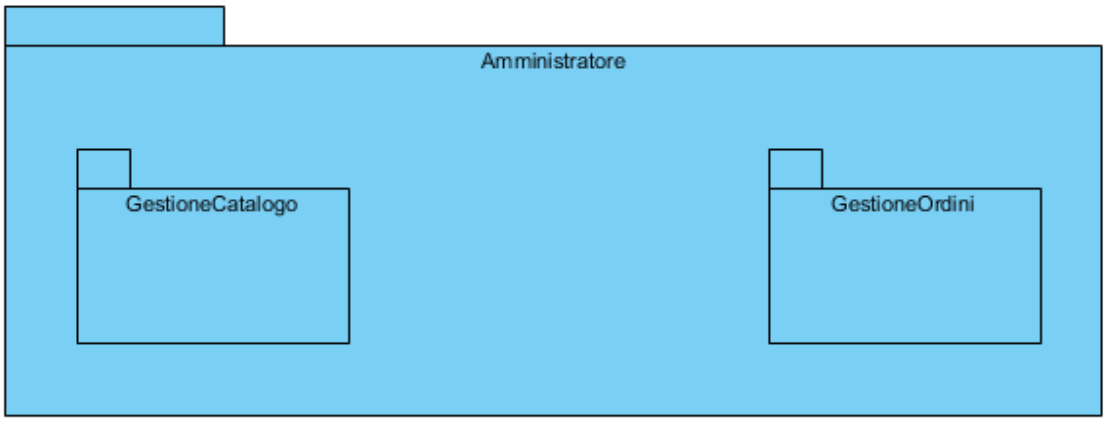
Classe:	Descrizione:
AnnullaOrdineControl.java	Controller che gestisce l'annullamento di un ordine
ConcludiAcquistoControl.java	Controller che gestisce la realizzazione di un ordine
CercaOrdineUtenteControl.java	Controller che gestisce la ricerca di ordini
MieiOrdiniControl.java	Controller che gestisce la lista degli ordini effettuati da un utente
CheckCreditCardControl	Controller che gestisce la validazione dei dati della carta di credito di un utente
VisualizzaFatturaControl	Controller che gestisce la visualizzazione della fattura per un determinato ordine

2.2.4 Package Catalogo

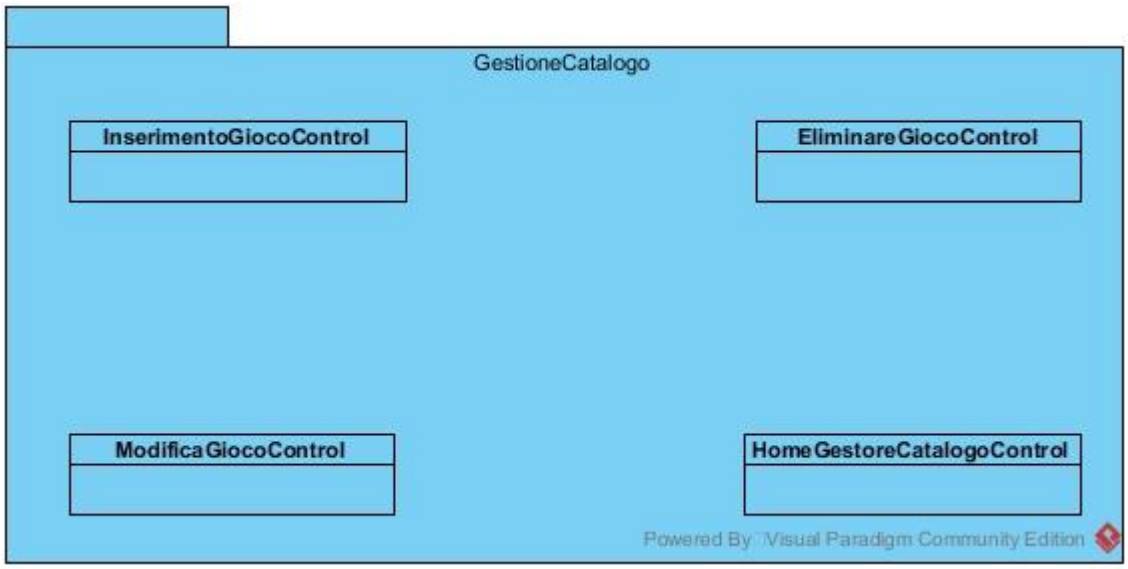


Classe:	Descrizione:
VisualizzareGiochiPerPiattaforma.java	Controller che gestisce la visualizzazione dei giochi per piattaforma
VisualizzareGiochiPerGenere.java	Controller che gestisce la visualizzazione dei giochi per genere
CatalogControl.java	Controller che gestisce la visualizzazione del catalogo
CercaGiocoUtenteControl.java	Controller che gestisce la ricerca dei giochi
VisualizzaPaginaProdottoControl.java	Controller che gestisce la visualizzazione della pagina del prodotto selezionato dall'utente

2.2.5 Package Amministratore



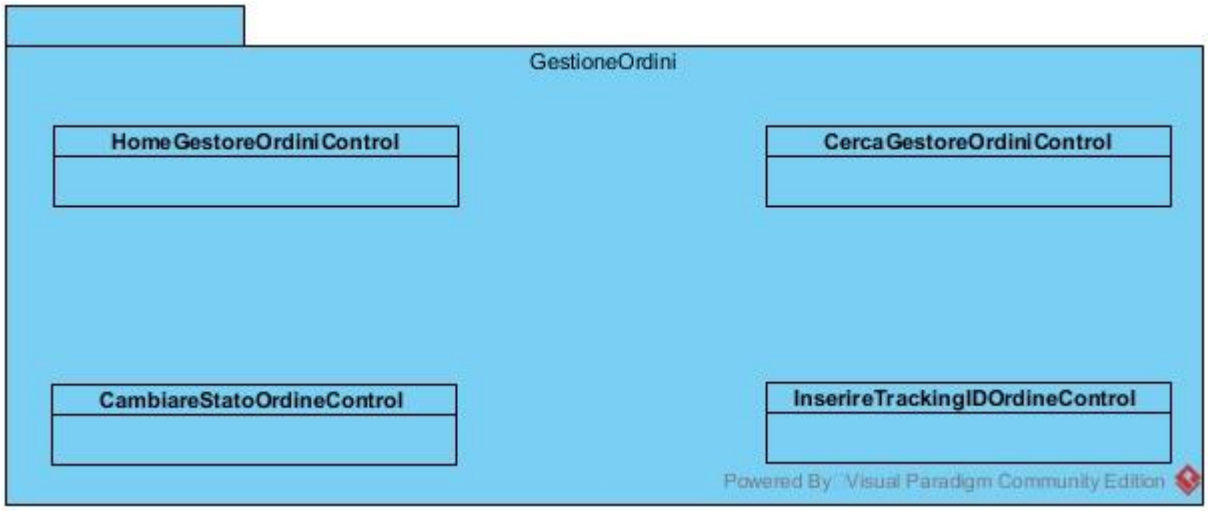
2.2.5.1 Package GestioneCatalogo



Classe:	Descrizione:
InserimentoGiocoControl.java	Controller che gestisce l'inserimento di un gioco nel catalogo
EliminareGiocoControl.java	Controller che gestisce l'eliminazione di un gioco dal catalogo
ModificaGiocoControl.java	Controller che gestisce la modifica delle informazioni di un gioco

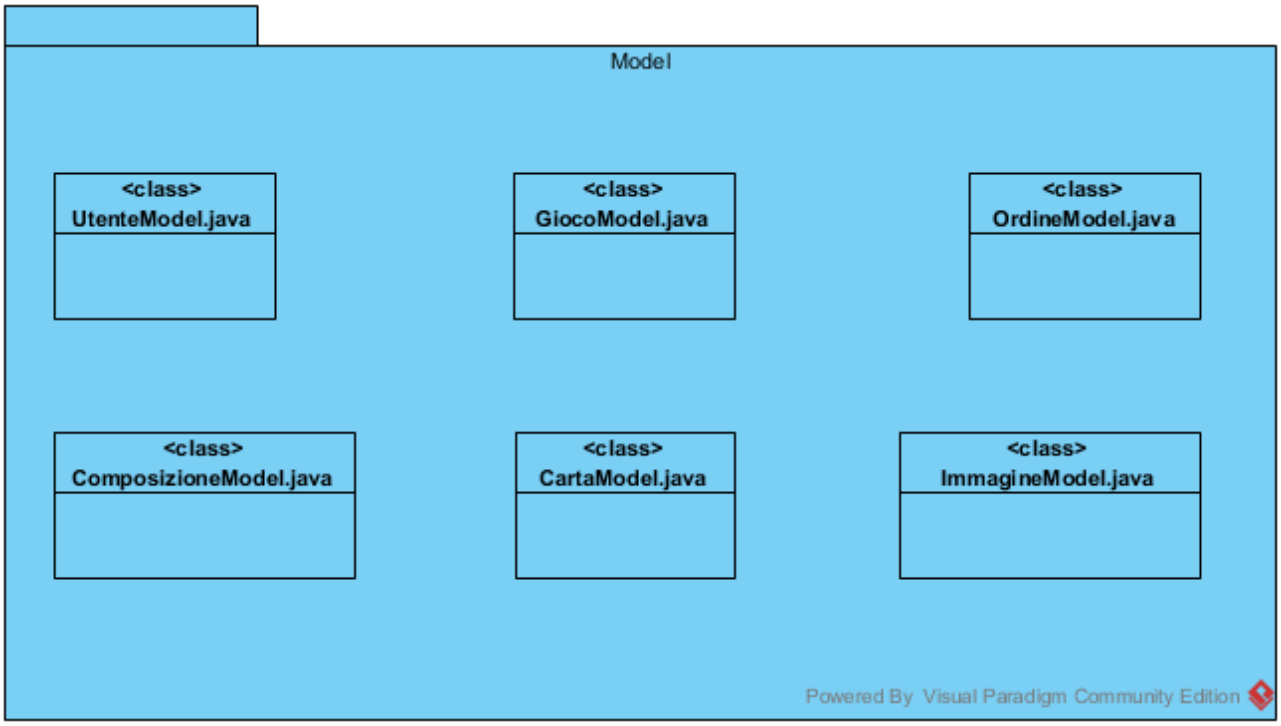
HomeGestoreCatalogoControl.java	Controller che gestisce l'accesso alla gestione del catalogo

2.2.5.2 Package GestioneOrdini



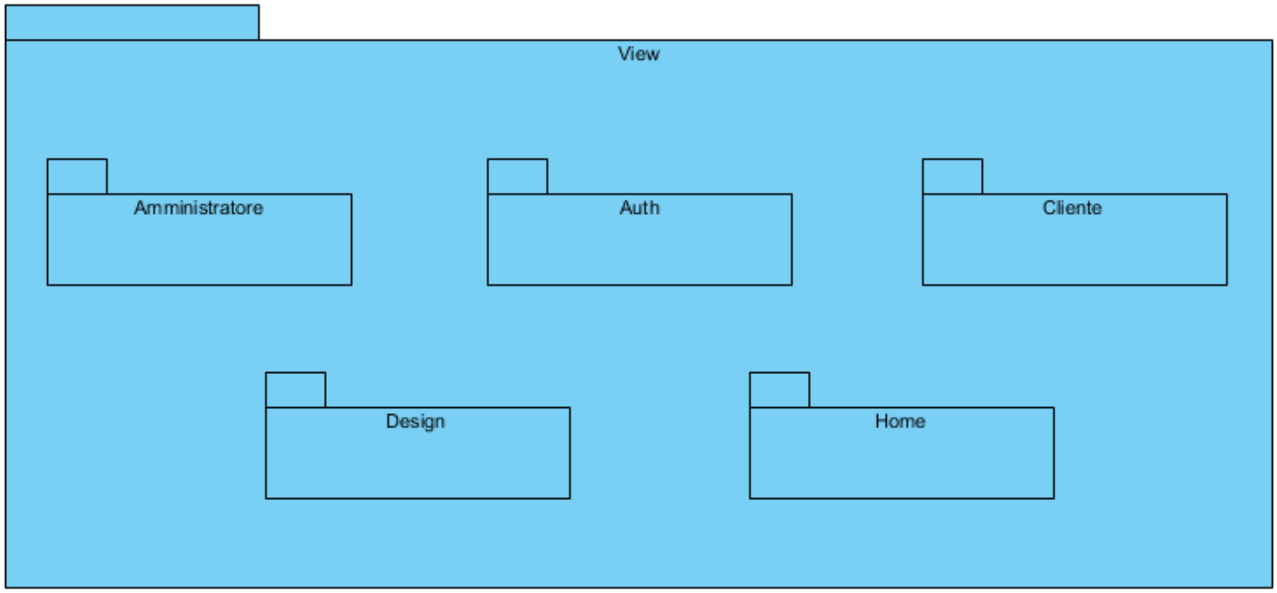
Classe:	Descrizione:
HomeGestoreOrdiniControl.java	Controller che gestisce la visualizzazione della lista degli ordini
CambiareStatoOrdineControl.java	Controller che gestisce il cambiamento di un ordine
InserireTrackingIDOrdineControl.java	Controller che gestisce l'inserimento del tracking id di un ordine
CercaGestoreOrdiniControl.java	Controller che gestisce la ricerca degli ordini effettuati dagli utenti

2.3 Package Model

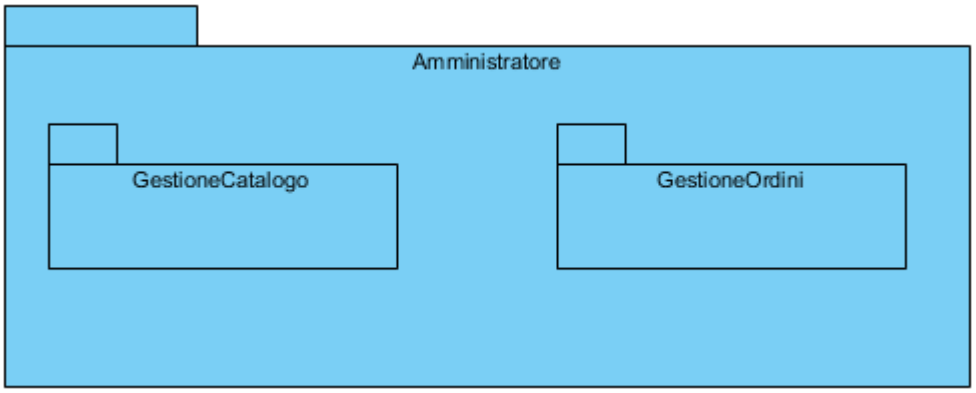


Classe:	Descrizione:
UtenteModel.java	Il model che effettua tutte le query riguardanti le funzionalità degli account, interfacciandosi al db al quale è connesso
GiocoModel.java	Il model che effettua tutte le query riguardanti le funzionalità dei giochi, interfacciandosi al db al quale è connesso
OrdineModel.java	Il model che effettua tutte le query riguardanti le funzionalità degli ordini, interfacciandosi al db al quale è connesso
ImmagineModel.java	Il model che effettua tutte le query riguardanti le funzionalità delle immagini di un gioco, interfacciandosi al db al quale è connesso
ComposizioneModel.java	Il model che effettua tutte le query riguardanti le funzionalità delle composizioni degli ordini, interfacciandosi al db al quale è connesso
CartaModel.java	Il model che effettua tutte le query riguardanti le funzionalità delle carte utilizzate per i pagamenti, interfacciandosi al db al quale è connesso

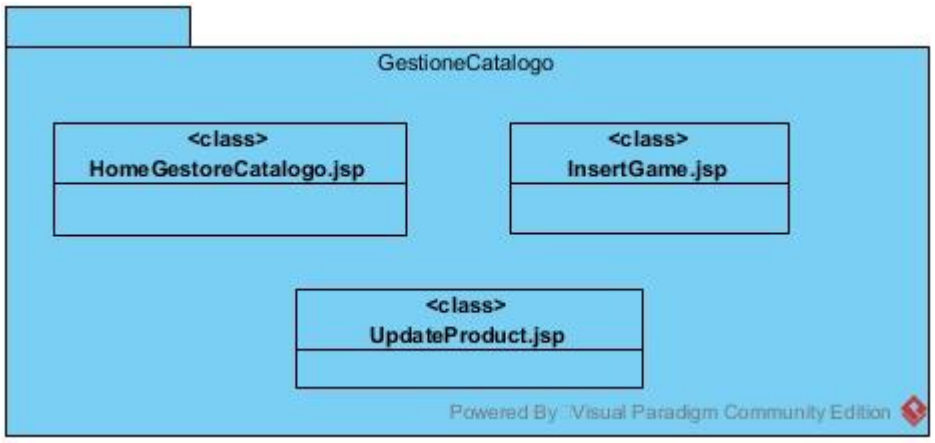
2.4 Package View



2.4.1 Amministratore



4.1.1 GestioneCatalogo

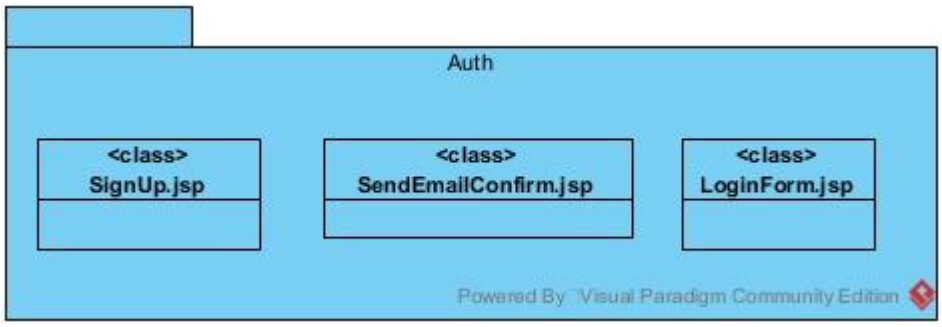


Classe:	Descrizione:
HomeGestoreCatalogo.jsp	La view che consente al gestore del catalogo di visualizzare l'intero catalogo dei giochi presenti
InsertGame.jsp	La view che consente al gestore del catalogo di inserire un gioco nel catalogo
UpdateProduct.jsp	La view che consente al gestore del catalogo di modificare le informazioni di un gioco

2.4.1.2 GestioneOrdini

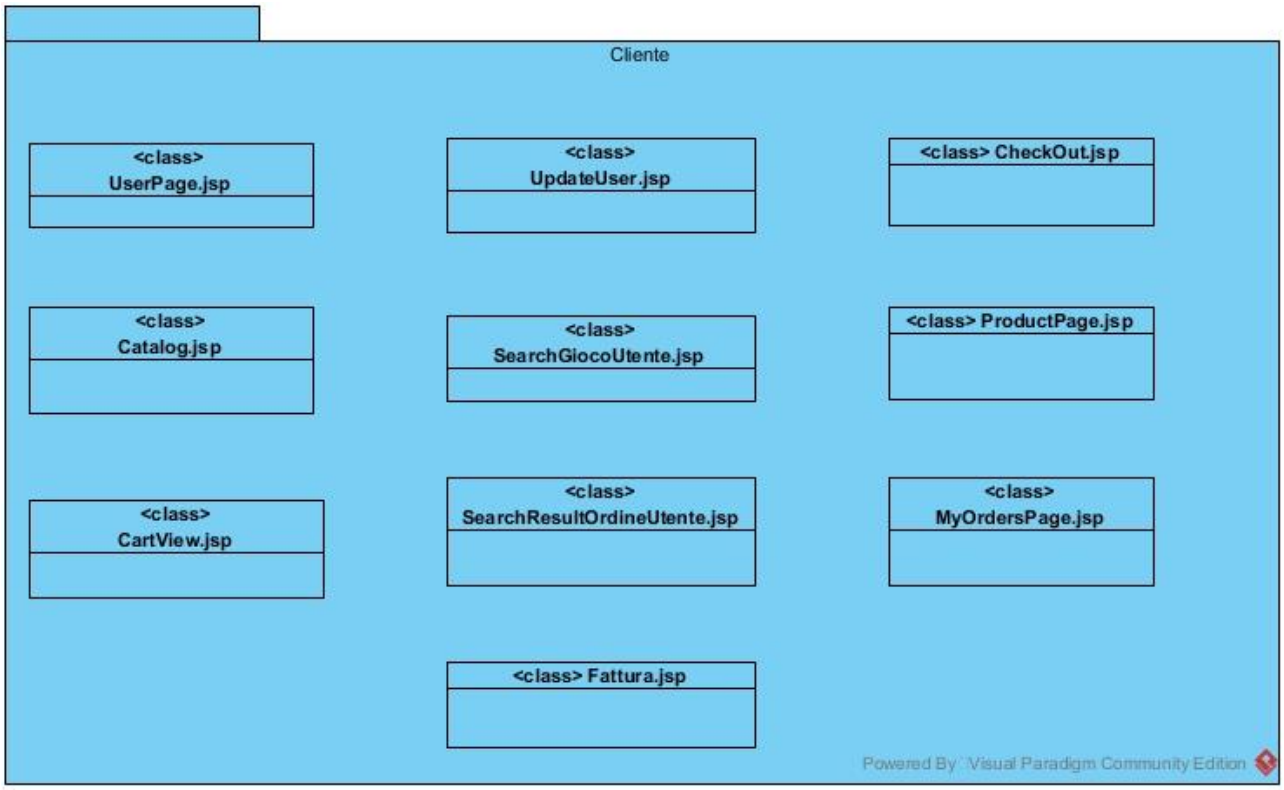
Classe:	Descrizione:
HomeGestoreOrdini.jsp	La view che consente al gestore degli ordini di visualizzare la lista di tutti gli ordini effettuati dai clienti

2.4.2 Auth



Classe:	Descrizione:
SignUp.jsp	La view che consente ad un utente di registrarsi al sito GamesHub
SendEmailConfirm.jsp	La view che informa l'utente che è stata inviata un e-mail di conferma.
LoginForm.jsp	La view che consente ad un utente di accedere al sito

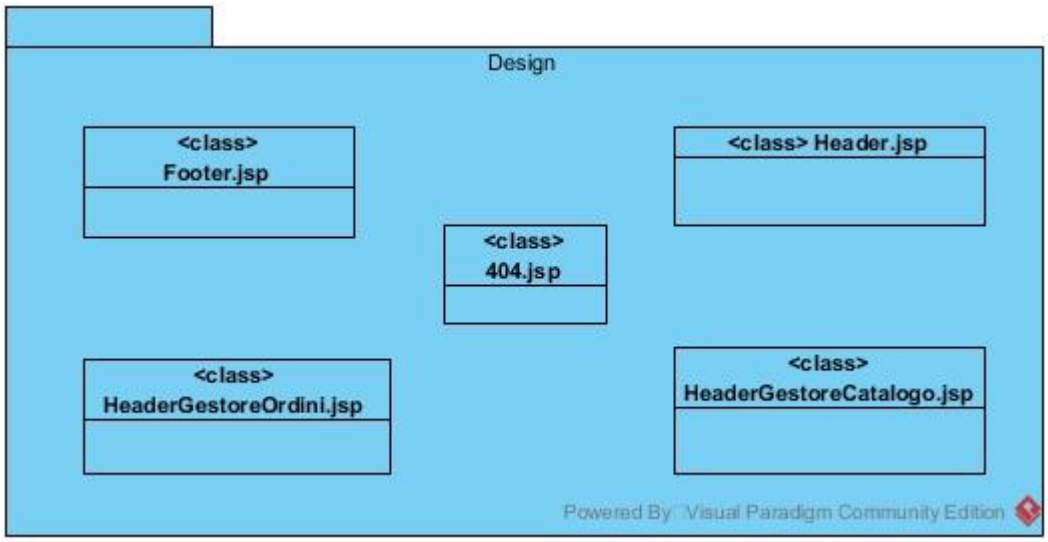
2.4.3 Cliente



Classe:	Descrizione:
UserPage.jsp	La view che consente ad un cliente di visualizzare la propria area personale
MyOrderPage.jsp	La view che consente ad un cliente di visualizzare la lista dei suoi ordini
SearchResultOrdineUtente.jsp	La view che consente ad un cliente di ricercare un ordine tra quelli effettuati
Catalog.jsp	La view che consente ad un cliente di visualizzare il catalogo dei giochi
ProductPage.jsp	La view che consente ad un cliente di visualizzare la pagina di un particolare gioco
Fattura.jsp	La view che consente ad un cliente di visualizzare la fattura di un ordine
SearchGiocoUtente.jsp	La view che consente ad un cliente di visualizzare la lista di tutti i giochi che soddisfano un particolare criterio specificato dal cliente
CartView.jsp	La view che consente ad un cliente di visualizzare il proprio carrello
UpdateUser.jsp	La view che consente ad un cliente di modificare i propri dati personali

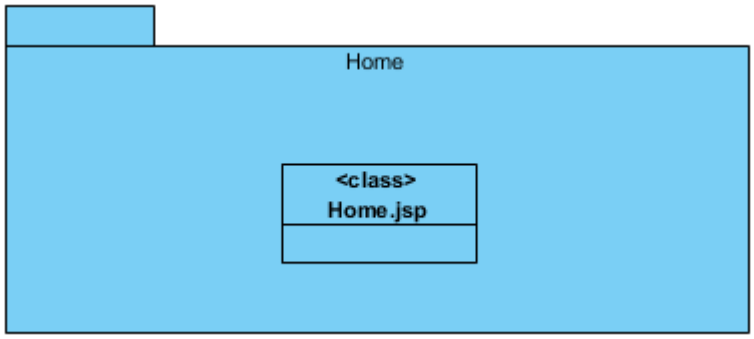
CheckOut.jsp	La view che consente di visualizzare le informazioni necessarie per concludere l'acquisto.

2.4.4 Design



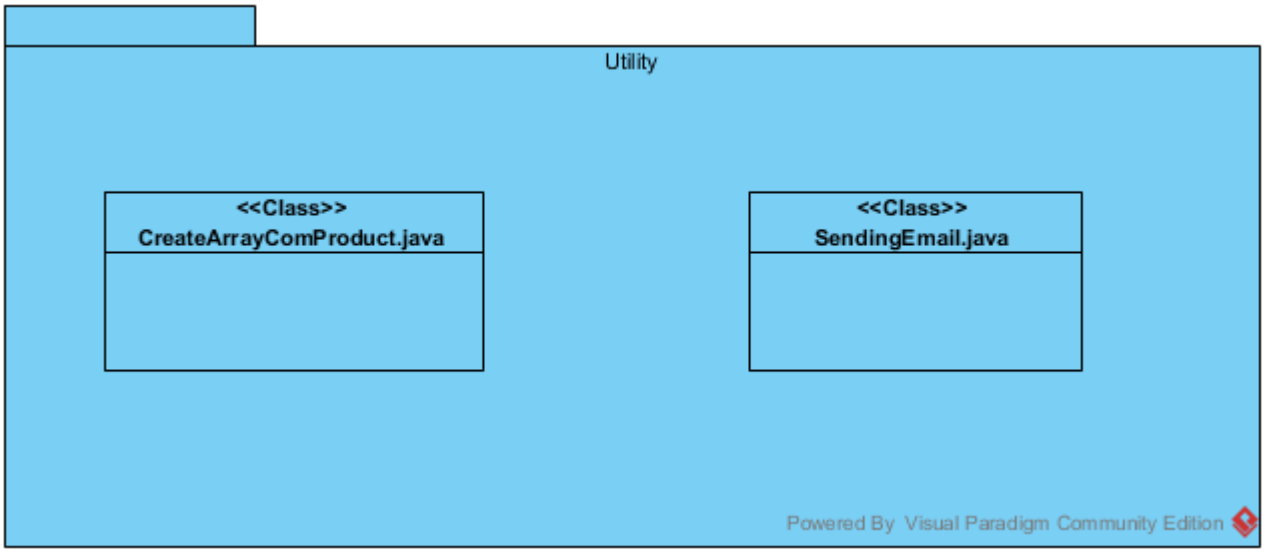
Classe:	Descrizione:
Footer.jsp	Contiene il footer comune a tutte le pagine
Header.jsp	Contiene l'header comune a tutte le pagine
404.jsp	View che visualizza l'errore 404 pagina non trovata
HeaderGestoreOrdini.jsp	Contiene l'header per le pagine per gestore degli ordini
HeaderGestoreCatalogo.jsp	Contiene l'header per le pagine per gestore del catalogo

2.4.5 Home



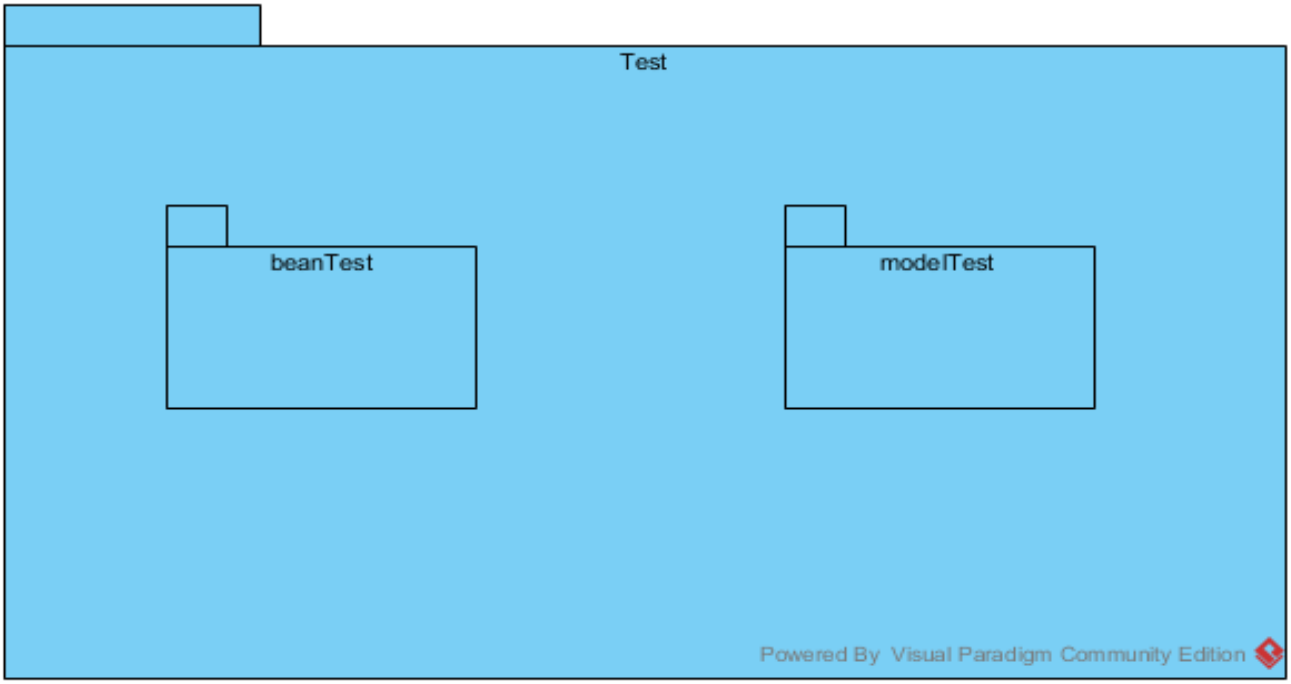
Classe:	Descrizione:
Home.jsp	La Homepage di GamesHub

2.5 Package Utility

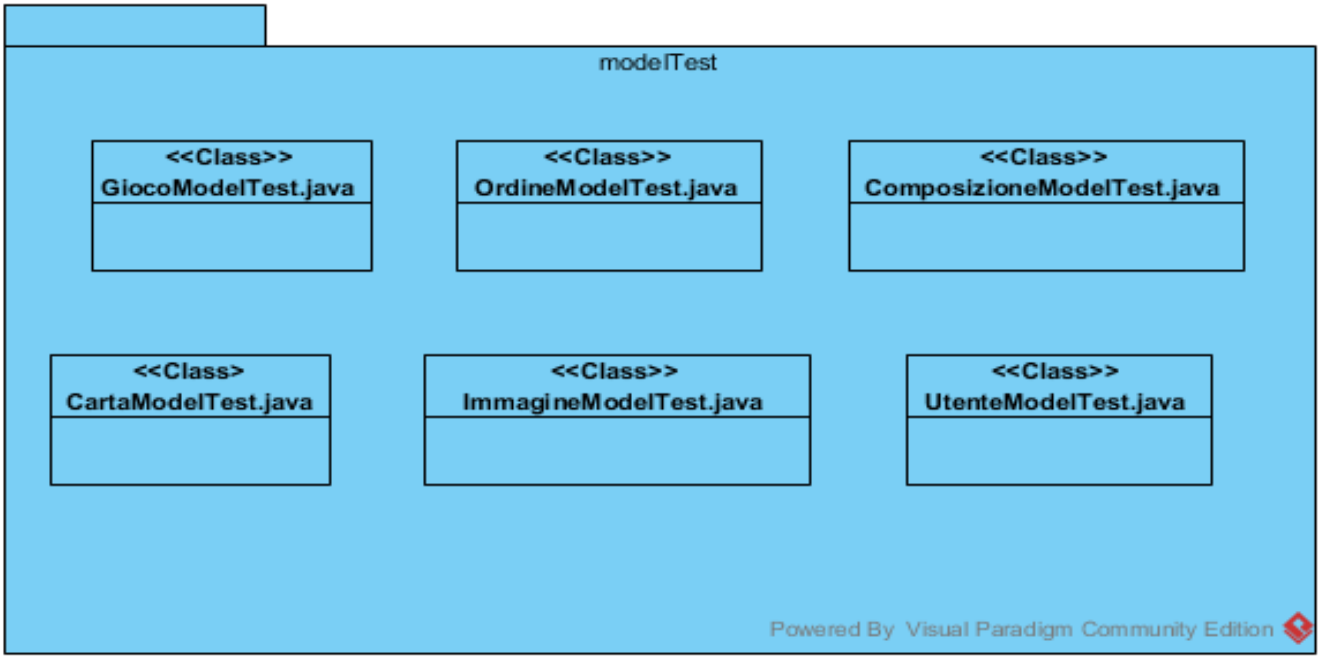


Classe:	Descrizione:
CreateArrayComProduct.java	Classe utilizzata per creare un array di ItemOrder.
SendingEmail.java	Classe utilizzata per inviare l'email di conferma quando un utente effettua la registrazione a GamesHub.

2.6 Package Test

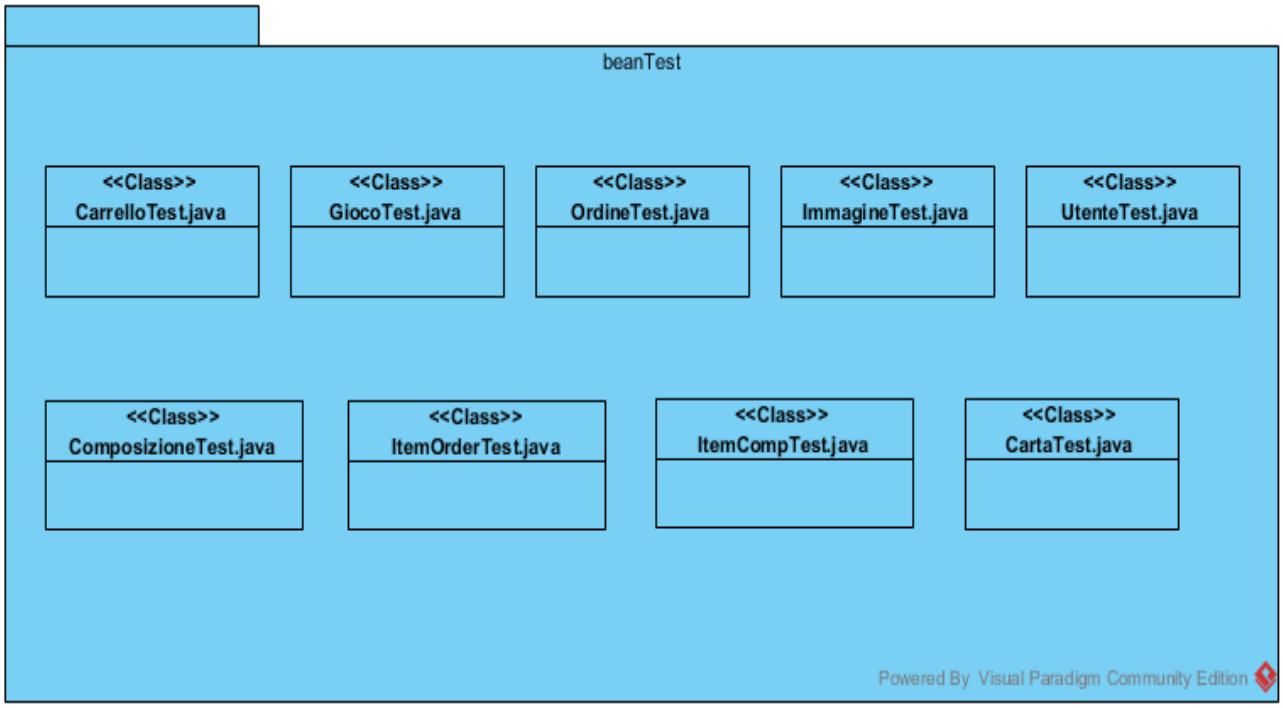


2.6.1 Package modelTest



Classe:	Descrizione:
GiocoModelTest.java	La classe che consente di testare i metodi del model GiocoModel.
OrdineModelTest.java	La classe che consente di testare i metodi del model OrdineModel.
ComposizioneModelTest.java	La classe che consente di testare i metodi del model ComposizioneModel.
CartaModelTest.java	La classe che consente di testare i metodi del model CartaModel.
UtenteModelTest.java	La classe che consente di testare i metodi del model UtenteModel.
ImmagineModelTest.java	La classe che consente di testare i metodi del model ImmagineModel.

2.6.2 Package beanTest



Classe:	Descrizione:
GiocoTest.java	La classe che consente di testare i metodi del bean Gioco.
OrdineTest.java	La classe che consente di testare i metodi del bean Ordine.
ComposizioneTest.java	La classe che consente di testare i metodi del bean Composizione.
CartaTest.java	La classe che consente di testare i metodi del bean Carta.
UtenteTest.java	La classe che consente di testare i metodi del bean Utente.
ImmagineTest.java	La classe che consente di testare i metodi del bean Immagine.
ItemOrderTest.java	La classe che consente di testare i metodi del bean ItemOrder.
ItemCompTest.java	La classe che consente di testare i metodi del bean ItemComp.
CarrelloTest.java	La classe che consente di testare i metodi del bean Carrello.

3. Class interfaces

Nome classe	UtenteModel
Descrizione	Descrive un utente registrato nel sistema.
Pre-condizioni	**Context** UtenteModel :: saveUser(user : Utente) **pre:** user!=null AND user non è presente nel database. **Context** UtenteModel :: deleteUser(username : String) **pre:** username!=null AND username!= “ ” AND esiste un Utente nel database con username=”username”. **Context** UtenteModel :: getUser(username : String) **pre:** username!=null AND username!= “ ” AND esiste un Utente nel database con username=”username”. **Context** UtenteModel :: getAllUser(order: String) **pre:** order!=null AND order!= “ ”.

	Context UtenteModel :: isAnUser(username : String) **pre:** username!=null AND username!= “ ”. **Context** UtenteModel :: activateUser(email: String, hash: String) **pre:** email!=null AND email!= “ ” AND hash!=null AND hash!=” “. **Context** UtenteModel :: doUpdate(username: String, password: String, telefono: String, indirizzo: String) **pre:** username !=null AND username != “ ” AND password !=null AND telefono != null AND indirizzo != null AND esiste un utente nel database con username=”username”
Post-condizioni	**Context** UtenteModel :: saveUser(user : Utente) **post:** user è un nuovo utente salvato nel database. **Context** UtenteModel :: deleteUser(username : String) **post:** L’utente con l’username=”username” è stato eliminato dal database. **Context** UtenteModel :: getUser(username : String) **post:** viene restituito l’utente che ha l’username uguale alla stringa passata come parametro. **Context** UtenteModel :: getAllUser(order: String) **post:** vengono restituiti tutti gli utenti presenti nel database ordinati in base al valore della stringa “order”. **Context** UtenteModel :: isAnUser(username : String) **post:** viene restituito TRUE se esiste un utente con l’username uguale alla stringa passata come parametro FALSE se non esiste un utente con l’username uguale alla stringa passata come parametro **Context** UtenteModel :: activateUser(email: String, hash: String) **post:** viene restituito TRUE: l’utente con email e hash uguali alle stringhe passate come parametro viene attivato (utente.verificato=1) FALSE altrimenti **Context** UtenteModel :: doUpdate(username: String, password: String, telefono: String, indirizzo: String) **post**: le informazioni sull’utente che ha l’username uguale a quello passato come parametro sono state aggiornate con i parametri passati al metodo.

Nome classe:	OrdineModel
Descrizione:	Descrive un ordine registrato nel sistema.
Pre-condizioni:	**Contex** OrdineModel::addOrdine(ordine: Ordine) **pre**: ordine !=null AND ordine non è presente nel database. **Contex** OrdineModel::getListaOrdiniUtente(username: String**) pre**: username!=null AND username!= " " **Contex** OrdineModel::getListaOrdini () **pre**: null. **Contex** OrdineModel::getListaOrdini (idOrdine: String, annoOrdine: String, statoOrdine: String) **pre**: idOrdine>=0 AND annoOrdine != null AND annoOrdine != " " AND statoOrdine != null AND statoOrdine != " ". **Contex** OrdineModel::deleteOrdine (idOrdine: Integer) **pre**: idOrdine>=0 AND esiste un Ordine nel database con id="idOrdine". **Contex** OrdineModel::getOrder (idOrdine : Integer) **pre**: idOrdine>=0 AND esiste un Ordine nel database con id="idOrdine". **Contex** OrdineModel::changeOrderState (idOrdine: Integer, stato: String) **pre**: idOrdine>=0 AND stato != null AND stato!= " " AND esiste un Ordine nel database con id="idOrdine". **Contex** OrdineModel::addTrackingID (idOrdine: Integer, trackingID: String) **pre**: idOrdine>=0 AND trackingID != null AND trackingID != " "AND esiste un Ordine nel database con id="idOrdine". **Contex** OrdineModel::doMaxIdOrder () **pre**: null. **Contex** OrdineModel::isAnOrder (idOrdine : Integer) **pre**: idOrdine>=0.
Post- condizioni:	**Contex** OrdineModel::addOrdine(ordine: Ordine) **post**: ordine è un nuovo Ordine salvato nel database. **Contex** OrdineModel::getListaOrdiniUtente(username: String) **post**: vengono restituiti tutti gli ordini che hanno ordine.utente="username". **Contex** OrdineModel::getListaOrdini () **post**: vengono restituiti tutti gli ordini presenti nel database. **Contex** OrdineModel::getListaOrdini (idOrdine: String, annoOrdine: String, statoOrdine: String) **post**: vengono restituiti tutti gli ordini che hanno IdOrdine="idOrdine" OR DataOrdine = "annoOrdine" OR Stato = "statoOrdine"

	Contex OrdineModel::deleteOrdine (idOrdine: Integer) **post**: L' ordine con id ="idOrdine" è stato eliminato dal database. **Contex** OrdineModel::getOrder (idOrdine : Integer) **post**: viene restituito l'ordine che ha l'id uguale all'intero passato come parametro. **Contex** OrdineModel::changeOrderState (idOrdine: Integer, stato: String) **post**: Lo stato dell'ordine che ha id = "idOrdine" è stato aggiornato con il nuovo stato passato come parametro al metodo. **Contex** OrdineModel::addTrackingID (idOrdine: Integer, trackingID: String) **post**: E' stato aggiunto il tracking id all'ordine che ha id = "idOrdine". **Contex** OrdineModel::doMaxIdOrder () **post**: viene restituito l'id dell'ultimo ordine effettuato. **Contex** OrdineModel::isAnOrder (idOrdine : Integer) **post**: viene restituito TRUE se esiste un ordine con l'id uguale all'intero passato come parametro FALSE se non esiste un ordine con l'id uguale all'intero passato come parametro.

Nome classe:	GiocoModel
Descrizione:	Descrive un gioco registrato nel sistema.
Pre-condizioni:	**Contex** GiocoModel::saveGame(gioco : Gioco) **pre**: gioco!=null AND gioco non è presente nel database. **Contex** GiocoModel::getGioco(code: int) **pre**: code >0 AND esiste un Gioco nel database con serialNumber = "code". **Contex** GiocoModel::deleteGame (code: int) **pre**: code>0 AND esiste un Gioco nel database con serialNumber = "code". **Contex** GiocoModel::doRetriveAll () **pre**: null. **Contex** GiocoModel::updateQuantity (code: int, quantità :int) **pre**: code>=0 AND quantità>=0 AND esiste un Gioco nel database con serialNumber = "code". **Contex** GiocoModel::updateGame (code: int, video: String, descrizione: String, prezzo: float , qty : int) **pre**: (video!=null AND video != " ") AND

	(descrizione!= null AND descrizione != “ “) AND esiste un Gioco nel database con serialNumber = ”code”.
Post- condizioni:	**Contex** GiocoModel::saveGame(gioco : Gioco) **post**: gioco è un nuovo Gioco salvato nel database. **Contex** GiocoModel::getGioco(code: int) **post**: viene restituito il gioco che ha il serialNumber uguale all’intero passato come parametro. **Contex** GiocoModel::deleteGame (code: int) **post**: Il Gioco con serialNumber = ”code” è stato eliminato dal database. **Contex** GiocoModel::doRetriveAll () **post**: vengono restituiti tutti i Giochi presenti nel database. **Contex** GiocoModel::updateGame (code: int, video: String, descrizione: String, prezzo: float , qty : int) **post:** le informazioni sul gioco che ha il serialNumber uguale a quello passato come parametro sono state aggiornate con i parametri passati al metodo.

Nome classe:	ComposizioneModel
Descrizione:	Descrive una composizione registrata nel sistema.
Pre-condizioni:	**Contex** ComposizioneModel::addComposizione(composizione : Composizione) **pre**: composizione != null AND composizione non è presente nel database. **Contex** ComposizioneModel::searchComposizione(idOrdine : Integer, idGioco: Integer) **pre**: idOrdine>=0 AND idGioco>0 AND esiste una Composizione nel database con composizione.Ordine = ”idOrdine” AND composizione.Gioco = “idGioco”. **Contex** ComposzioneModel::deleteComposizone (idOrdine: Integer) **pre**: idOrdine>=0 AND esiste una Composizione nel database con composizione.Ordine = ”idOrdine”.
Post- condizioni:	**Contex** ComposizioneModel::addComposizione(composizione : Composizione) **post**: composizione è una nuova Composizione salvata nel database.

	Contex ComposizioneModel::searchComposizione(idOrdine : Integer, idGioco: Integer) **post**: vengono restituite tutte le composizioni che hanno composizione.Ordine =”idOrdine”. **Contex** ComposzioneModel::deleteComposizone (idOrdine: Integer) **post**: La composizione con composizione.Ordine = “idOrdine” è stata eliminata dal database.

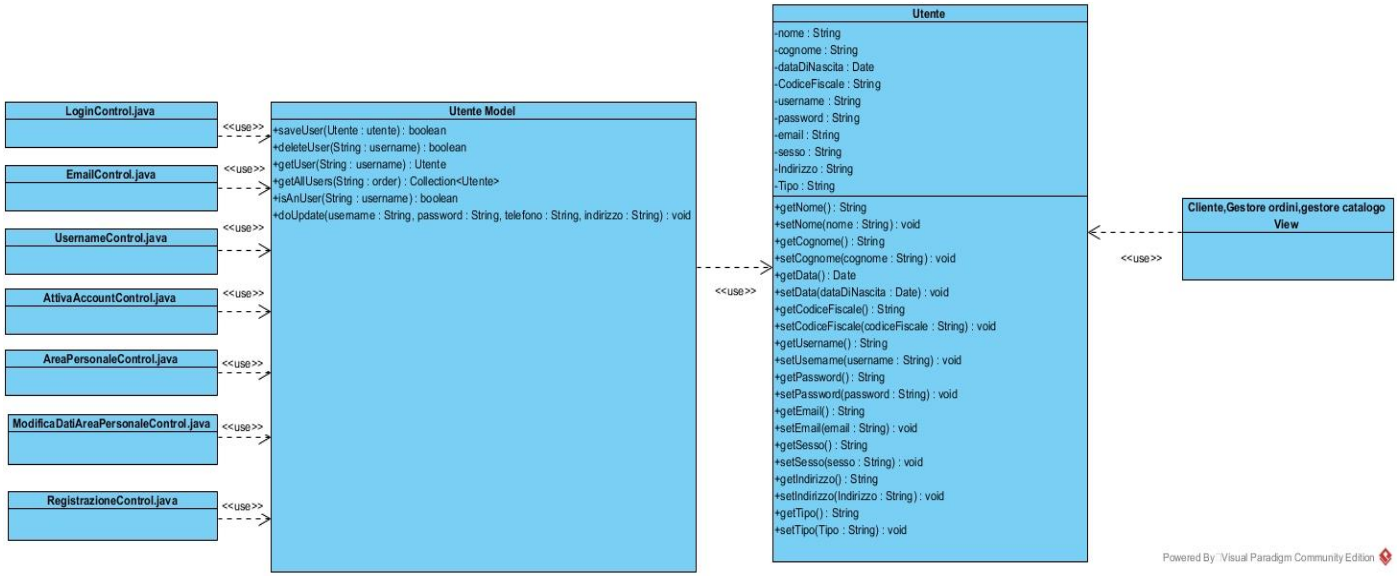
Nome classe:	CartaModel
Descrizione:	Descrive carte di credito registrata nel sistema
Pre-condizioni:	**Contex** CartaModel::addCarta(carta : Carta) **pre**: carta!= null. **Contex** CartaModel::setCarta (carta : Carta) **pre**: carta != null. **Contex** CartaModel::deleteCarta (user : String) **pre**: user != null AND user != “ “. **Contex** CartaModel::existCarta (username: String) **pre**: username!= null AND username!= “ “ **Contex** CartaModel::doUpdate (saldo: float, numeroCarta: String) **pre**: username!= null AND username!= “ “ and saldo>=0. **Contex** CartaModel::getCarta (username: String) **pre**: username!= null AND username!= “ “

Post- condizioni:	**Contex** CartaModel::addCartaCredito(cartaCredito : CartaCredito) **post**: cartaCredito è una nuova carta di credito salvata nel database. **Contex** CartaModel::setCartaCredito(cartaCredito : CartaCredito) **post**: le informazioni sulla carta di credito che ha il numero carta uguale a quella passata come parametro sono state aggiornate con quelle di cartaCredito. **Contex** CartaModel::deleteCarta (user : String) **post**: se carta di credito che ha l'user="user" è stata eliminata dal database, restituisce true, altrimenti restituisce false. **Contex** CartaModel::existCarta (username: String) **post**: se nel database è presente una carta di credito con user="username", restituisce true, altrimenti restituisce false. **Contex** CartaModel::doUpdate (saldo: float, numeroCarta: String) **post**: il saldo della carta di credito che ha il numero carta = "numeroCarta" è stato aggiornato con il saldo passato come parametro. **Contex** CartaModel::getCartaCredito(username: String) **post**: viene restituita la carta di credito associata all'utente con username="username".

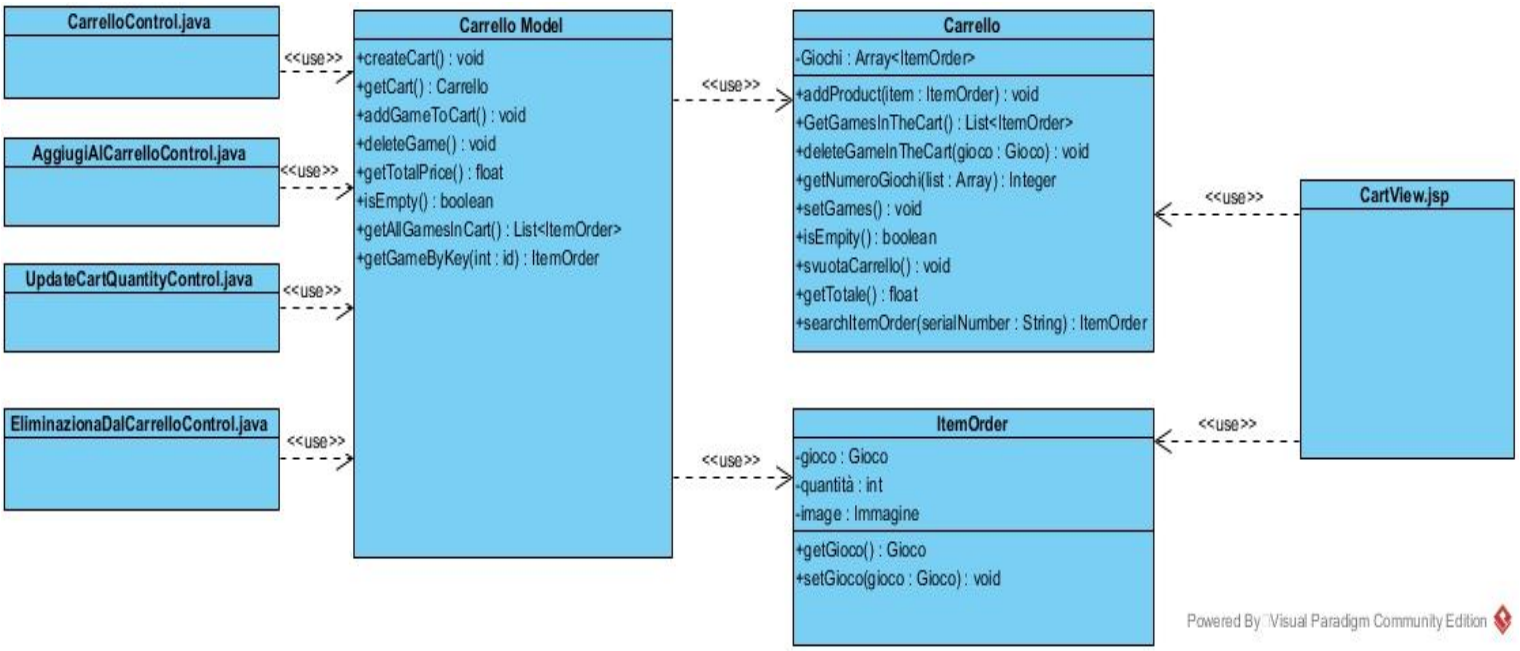
Nome classe:	ImmagineModel
Descrizione:	Descrive un immagine registrata nel sistema
Pre-condizioni:	**Contex** ImmagineModel::saveImage(image: Immagine) **pre**: image!= null. **Contex** ImmagineModel::deleteImage (path: String) **pre**: path!= null AND path != " ". **Contex** ImmagineModel::doRetriveAll () **pre**: null. **Contex** ImmagineModel::getImage (path: String) **pre**: path != null AND path != " "

Post- condizioni:	**Contex** ImmagineModel::saveImage(image: Immagine) **post:** image è una nuova immagine salvata nel database. **Contex** ImmagineModel::deleteImage (path: String) **post**: l'immagine con il percorso path è stata eliminata. **Contex** ImmagineModel::doRetriveAll() **post**: vengono restituite tutte le immagini presenti nel database. **Contex** ImmagineModel::getImage (path: String) **post**: viene restituita l'immagine che ha il percorso="path".

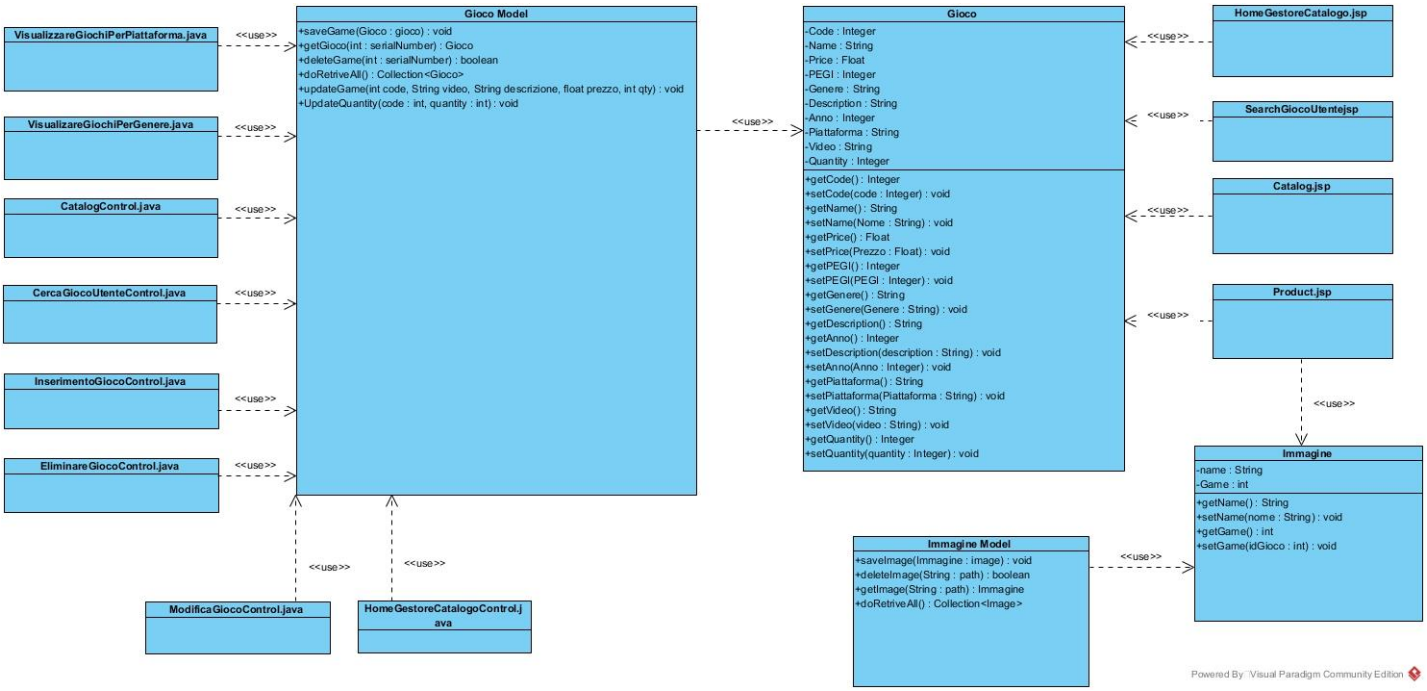
3.1 Gestione Utente



3.2 Gestione carrello


CarrelloControl.java
<<use>>
Carrello Model
+createCart() : void
+getCart() : Carrello
+addGameToCart() : void
+deleteGame() : void
+getTotalPrice() : float
+isEmpty() : boolean
+getAllGamesInCart() : List<ItemOrder>
+getGameByKey(int : id) : ItemOrder
<<use>>
Carrello
-Giochi : Array<ItemOrder>
+addProduct(item : ItemOrder) : void
+GetGamesInTheCart() : List<ItemOrder>
+deleteGameInTheCart(gioco : Gioco) : void
+getNumeroGiochi(list : Array) : Integer
+setGames() : void
+isEmpity() : boolean
+svuotaCarrello() : void
+getTotale() : float
+searchItemOrder(serialNumber : String) : ItemOrder
<<use>>
CartView.jsp
AggiugiAlCarrelloControl.java
<<use>>
UpdateCartQuantityControl.java
<<use>>
EliminazionaDalCarrelloControl.java
<<use>>
<<use>>
ItemOrder
-gioco : Gioco
-quantità : int
-image : Immagine
+getGioco() : Gioco
+setGioco(gioco : Gioco) : void
<<use>>
Powered By Visual Paradigm Community Edition

3.3 Gestione catalogo



3.4 Gestore ordini