Progetto di una Web Application RESTful di un negozio di videogiochi "RiecoldddogGames"

Software Architectures and Methodologies

Marco Bardazzi, Mattia Baroncelli

Ingegneria Informatica

Anno Accademico 2025/2026

Indice

1 Introduzione

Il nostro programma permette ad un qualunque negozio di prodotti videoludici di gestire l'intera filiera di vendita, tra cui: la gestione dei prodotti che arrivano in negozio, l'organizzazione dei dipendenti, la gestione dei clienti registrati e la possibilità di avere un amministratore.

Il progetto è utile a singoli negozi che fanno parte di una catena, nel nostro caso la catena di appartenenza è di nostra invenzione e si chiama "EcologGames".

- Il cliente è colui che si reca nel punto vendita per compiere acquisti fisicamente. Per sfruttare tutti servizi del negozio (ad esempio prenotare un videogioco), è necessario che si registri con l'aiuto di un dipendente. In ogni caso, per la sola azione di compiere acquisti, non è necessaria alcuna registrazione.
- Il dipendente è colui che può svolgere semplici operazioni di gestione del magazzino interno, come ad esempio visionare prodotti disponibili o richiedere l'arrivo in lotto di prodotti. Inoltre, può occuparsi degli acquisti, delle prenotazioni e delle registrazioni dei clienti.
- L'amministratore è colui che può svolgere operazioni più complesse di gestione del magazzino. E' il ruolo che più si avvicina all'azienda madre, in particolare può aggiungere videogiochi nuovi al catalogo complessivo dei punti vendita, può modificare dati dei dipendenti e visionare liste di dipendenti e clienti, con i relativi dati personali.

2 Progettazione

2.1 Use Case Diagrams

Il software ha 3 diversi attori: il cliente (Customer), il dipendente (Employee) e l'amministratore. Il cliente compie acquisti e prenotazioni per il suo conto, il dipendente gestisce il magazzino interno e permette ai clienti di fare acquisti, l'amministratore si occupa della gestione della catena e compie azioni che possono influenzare tutti i punti vendita. Lo schema dei casi d'uso, è realizzato secondo lo standard UML mediante StarUML

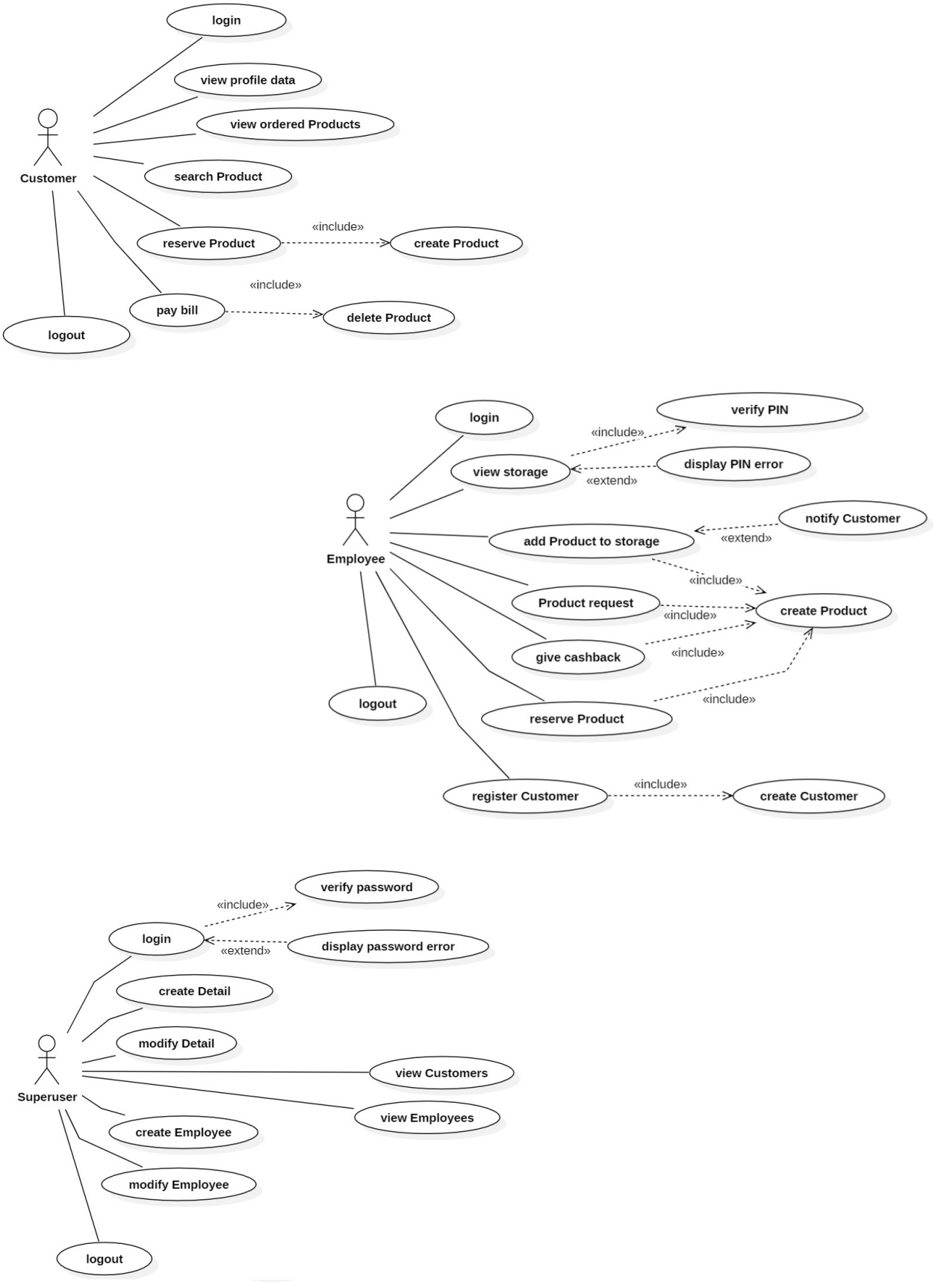

Figura 1: Use Case Diagram

2.2 Use Case Templates

Di seguito, illustriamo quelli che a nostro avviso sono i più importanti casi d'uso nel dettaglio.

2.2.1 Casi d'uso principali di un cliente

UC-1	Reserve Product
Descrizione	Il cliente prenota un videogioco per suo conto, in attesa di una notifica di quando sarà arrivato in negozio.
Livello	User goal
Attore principale	Customer
Azioni	1. Il cliente clicca "Prenota" dal menù a tendina della sua interfaccia. 2. Il sistema mostra una casella di testo e un pulsante "PRENOTA". 3. Il cliente inserisce il nome del videogioco nella casella. 4. Il cliente preme il pulsante "PRENOTA" 5. Il sistema notifica il cliente dell'avvenuta prenotazione.
Casi straordinari	5. Nel caso il gioco non esista o la casella di testo sia vuota, il sistema notifica l'errore al cliente.

Tabella 1: Prenotazione di un videogioco per un cliente - MOCKUP SOTTO

UC-2	Search Product
Descrizione	Il cliente fa una ricerca inserendo il nome di un videogioco, il sistema mostra le informazioni legate ad esso.
Livello	User goal
Attore principale	Customer
Azioni	1. Il cliente clicca "Cerca" dal menù a tendina della sua interfaccia. 2. Il sistema mostra una casella di testo e un pulsante "CERCA". 3. Il cliente inserisce il nome del videogioco nella casella. 4. Il cliente preme il pulsante "CERCA" 5. Il sistema mostra nome, console e prezzo del gioco.
Casi straordinari	5. Nel caso il gioco non esista o la casella di testo sia vuota, il sistema notifica l'errore al cliente.

Tabella 2: Ricerca della disponibilità di un videogioco per un cliente

2.2.2 Casi d'uso principali di un dipendente

UC-3	View Storage
Descrizione	Il dipendente riceverà le informazioni su tutti i prodotti in magazzino, dopo l'inserimento di un PIN univoco.
Livello	User goal
Attore principale	Employee
Azioni	1. Il dipendente clicca "Magazzino" dal menù a tendina della sua interfaccia. 2. Il sistema mostra una casella di testo, uno spazio per i risultati e un pulsante "INVIA/AGGIORNA". 3. Il dipendente inserisce il PIN nella casella. 4. Il dipendente preme il pulsante "INVIA/AGGIORNA" 5. Il sistema fornisce la lista di tutti i prodotti, comprensiva di ID, nome, console, prezzo e quantità disponibile.
Casi straordinari	5. Nel caso il PIN sia errato o la casella sia vuota, il sistema notifica l'errore al dipendente. 6. Nel caso in cui sia stato fatto un aggiornamento al database, il dipendente preme il pulsante "INVIA/AGGIORNA" per aggiornare la lista

Tabella 3: Consultazione della lista dei prodotti in magazzino

UC-4	Add Product to Storage
Descrizione	Il dipendente registra l'arrivo di un videogioco in una determinata quantità.
Livello	User goal
Attore principale	Employee
Azioni	1. Il dipendente clicca "Aggiungi al magazzino" dal menù a tendina della sua interfaccia. 2. Il sistema mostra due caselle di testo e un pulsante "AGGIUNGI AL MAGAZZINO". 3. Il dipendente inserisce l'ID del videogioco arrivato nella prima casella e la quantità nella seconda. 4. Il dipendente preme il pulsante "AGGIUNGI AL MAGAZZINO" 5. Il sistema notifica il dipendente dell'avvenuta aggiunta dei prodotti..
Casi straordinari	5. Nel caso in cui l'ID del videogioco non esista o e/o non è stato riempito il campo della quantità, il sistema notifica l'errore al dipendente

Tabella 4: Aggiunta di prodotti in magazzino - MOCKUP SOTTO

UC-5	Request Product
Descrizione	Il dipendente richiede l'arrivo di un videogioco in una determinata quantità
Livello	User goal
Attore principale	Employee
Azioni	1. Il dipendente clicca "Richiedi" dal menù a tendina della sua interfaccia. 2. Il sistema mostra due caselle di testo e due pulsanti. 3. Il dipendente inserisce l'ID o il nome del videogioco da richiedere nella prima casella e la quantità nella seconda. 4. Il dipendente preme il pulsante "RICHIEDI PER GAMEID" se ha inserito l'ID del videogioco, altrimenti preme il pulsante "RICHIEDI PER NOME" se ha inserito il nome. 5. Il sistema notifica il dipendente dell'avvenuta richiesta.
Casi straordinari	5. Nel caso in cui l'ID o il nome del videogioco non esista o e/o non è stato riempito il campo della quantità, il sistema notifica l'errore al dipendente

Tabella 5: Richiesta di arrivo di un videogioco

UC-6	Notify Customer
Descrizione	Se un dipendente registra l'arrivo di un videogioco e quest'ultimo era prenotato da uno o più clienti, il sistema si occuperà di riservare il prodotto a loro e notificarli.
Livello	Function
Attore principale	Employee
Azioni	1. Il sistema notifica un cliente tramite la sua email dell'arrivo del gioco da lui prenotato.
Condizioni preliminari	Il dipendente deve registrare l'arrivo di un videogioco prenotato.

Tabella 6: Notifica di un cliente per email

2.2.3 Casi d'uso principali dell'amministratore

UC-7	Create Detail
Descrizione	L'amministratore inserisce nel database una nuova istanza nella tabella Detail, creando di fatto un nuovo videogioco nell'archivio.
Livello	User goal
Attore principale	Superuser
Azioni	1. L'amministratore clicca "Aggiungi gioco" dal menù a tendina della sua interfaccia. 2. Il sistema mostra tre caselle di testo e un pulsante. 3. L'amministratore inserisce il nome del videogioco, la console e il prezzo. 4. L'amministratore preme il pulsante "AGGIUNGI AL DATABASE" 5. Il sistema notifica l'amministratore dell'avvenuta aggiunta.
Casi straordinari	5. Se uno dei campi viene lasciato vuoto, il sistema notifica l'amministratore di un errore.

Tabella 7: Aggiunta di un videogioco al database - MOCKUP SOTTO

2.3 Mockups

Ecco alcuni Mockups che sono veri e propri screenshot della GUI realizzata a questo proposito.

2.3.1 Reserve Product

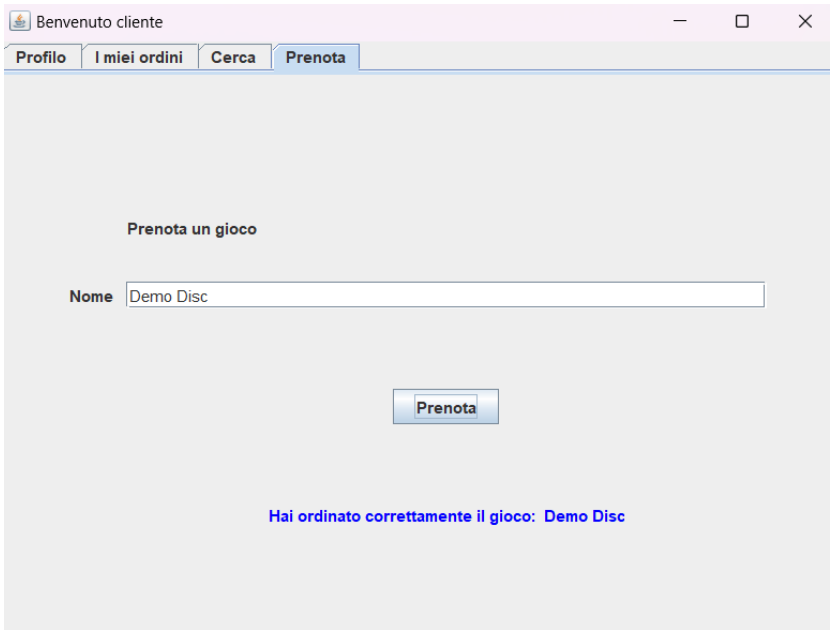

Figura 2: Screenshot di Reserve Product presente nella GUI cliente

2.3.2 Add To Storage

Figura 3: Screenshot di Add To Storage presente nella GUI dipendente

2.3.3 Request Product

Figura 4: Screenshot di Request Product presente nella GUI dipendente

2.3.4 Create Detail

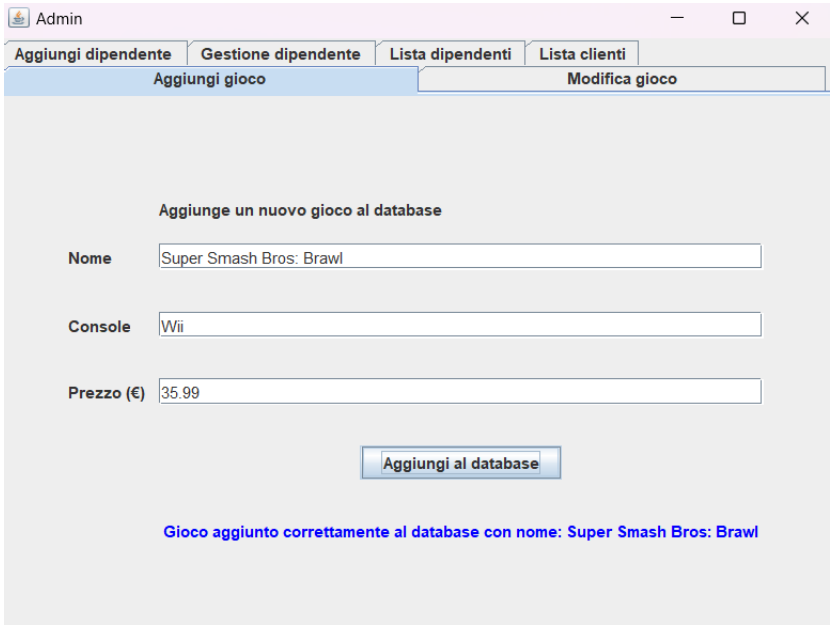

Figura 5: Screenshot di Create Detail presente nella GUI amministratore

2.4 Class Diagram

Di seguito (Figura 2), è riportato il diagramma delle classi del software e come queste sono legate e interagiscono tra loro.

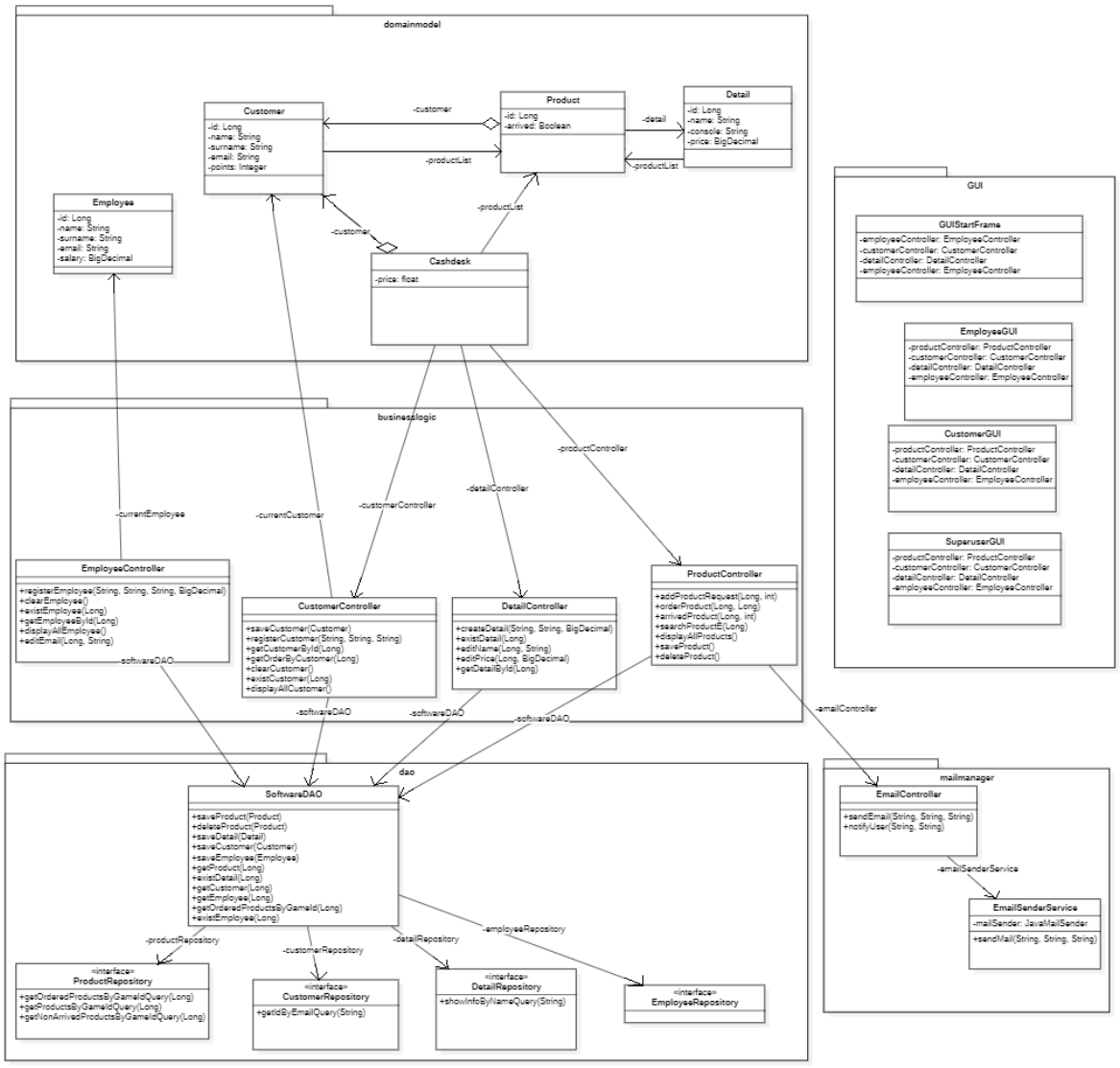

Figura 6: Diagramma UML

I package contenuti all'interno del software sono i seguenti:

- domainModel: rappresenta il modello dei dati. Contiene tutte le classi tramite le quali sono stati rappresentati gli oggetti nel software. Qua sono definiti gli attributi di tutti i tipi presenti.
- businessLogic: rappresenta la logica di controllo del sistema. Contiene le classi che si occupano di modificare i dati, comunicando esclusivamente con dei servizi per poter accedere al database senza farlo direttamente. In questo package sono presenti, dunque, tutti i controller per i modelli mappati nel Database.
- dao: contiene tutte le classi necessarie all'interazione con il database.
- GUI: contiene tutte le classi che implementano e mostrano le interfacce di login e dei tre diversi Use Case.
- mailManager: contiene tutte le classi che servono a gestire l'invio automatico delle e-mail da parte del negozio.

2.5 DAO Pattern

Per la comunicazione con il Database, è necessario che il client non possa interagire direttamente con lo stesso. Pertanto, abbiamo deciso di implementare il Design Pattern DAO.

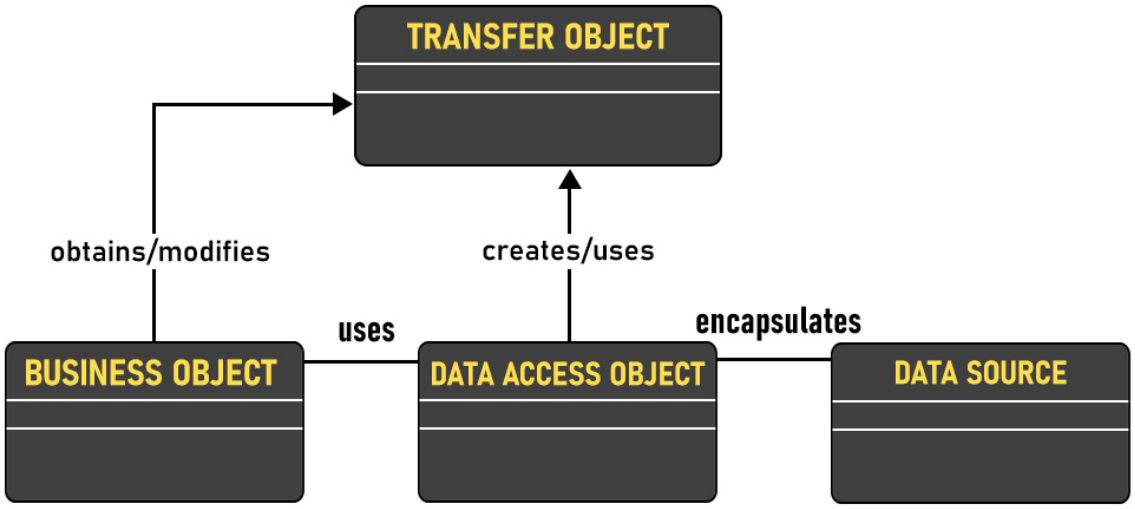

Figura 7: Rappresentazione del Pattern DAO

DAO (Data Access Object) è un design pattern architetturale per mantenere la persistenza dei dati che sono registrati in un database. Con persistenza si intende la capacità di un qualsiasi dato di "sopravvivere" all'intera esecuzione di un programma, cioè di continuare a esistere ed essere coerente anche dopo la terminazione. Questo pattern serve inoltre a separare la logica di accesso ai dati dal resto dell'applicazione. E' progettato per isolare il codice che gestisce l'accesso e la manipolazione dei dati da quello che implementa la businessLogic del nostro software. La classe SoftwareDAO definisce e mette a disposizione metodi CRUD (Create, Read, Update, Delete) per i dati nel Database.

Le classi che sfruttano SoftwareDAO sono:

- Product
- Detail
- Customer
- Employee

Da notare che queste sono le 4 entità presenti nel Database. Inoltre, la classe Cashdesk non necessita dell'uso della classe SoftwareDAO per essere modificata, in quanto non è un modello registrato nel Database e non ha bisogno di persistenza.

2.6 Entity Relationship Diagram

Di seguito è riportato il diagramma ER del Database che abbiamo progettato. La cardinalità di entrambe le relazioni presenti sono (1;N).

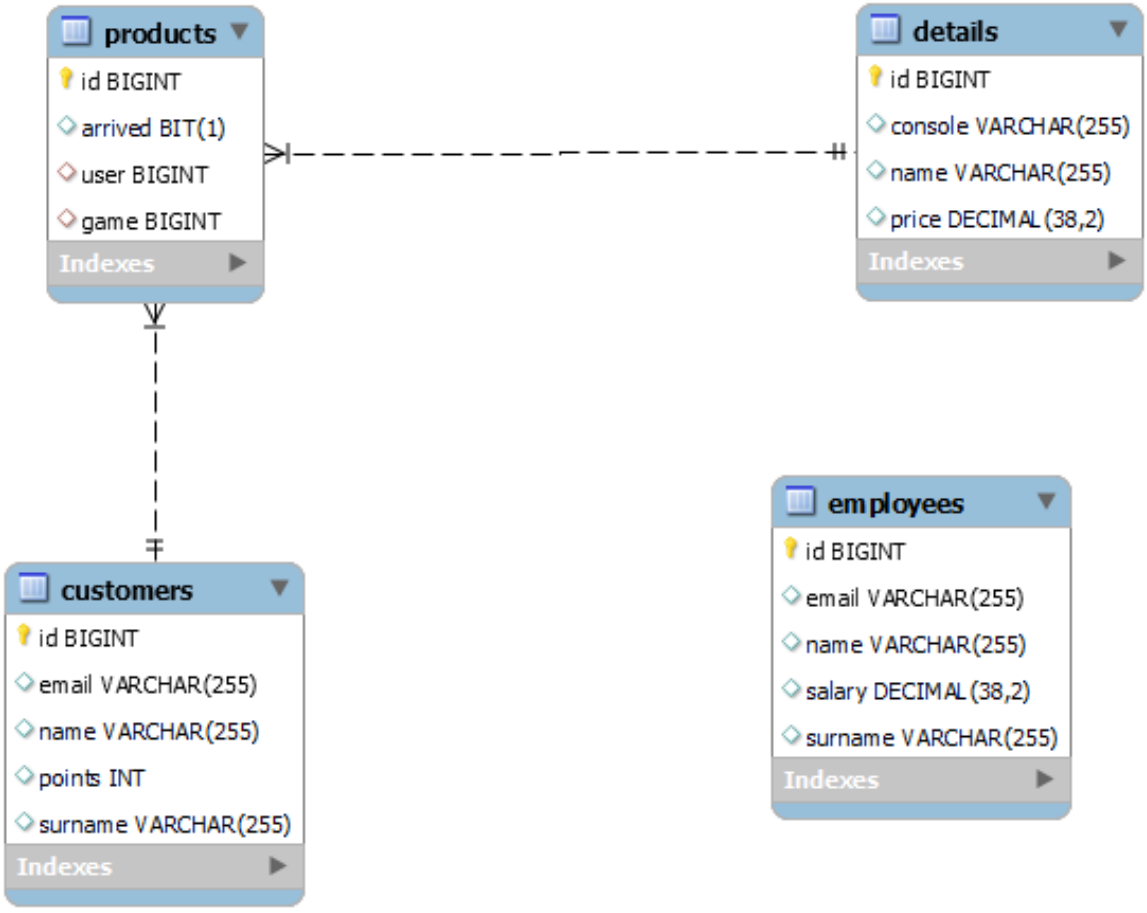

Figura 8: Diagramma ER

La relazione tra products e detail sta a significare il legame tra il prodotto (nonché un'istanza di videogioco) e il videogioco stesso. La relazione tra products e customers sta a significare che un prodotto può appartenere (cioè essere stato prenotato) da un cliente.

3 Implementazioni delle classi

Il progetto prevede la divisione in package Java: il modello dati nel package domainModel, la logica di controllo nel package businessLogic ed infine la gestione del Database nel package DAO
Il codice sorgente è articolato nel seguente modo:

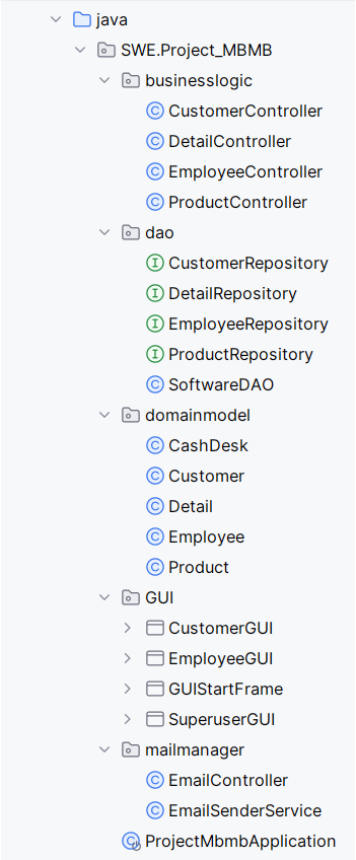

Figura 9: Suddivisione dei package all'interno dell'IDE

3.1 domainModel

Il domainModel è il package che si occupa di definire un modello di composizione di classi su cui è possibile eseguire i casi d'uso espressi nello Use Case Diagram. Questo package si articola nelle seguenti classi:

3.1.1 Detail

Rappresenta un oggetto appartenente al "catalogo" di videogiochi potenzialmente disponibili in negozio. Gli attributi sono:

- id: un identificativo univoco del videogioco (da ora in poi ci riferiremo a lui come GameID)
- name: una stringa contenente il nome del videogioco
- console: una stringa contenente la console su cui è stato pubblicato quel videogioco
- price: contiene il prezzo attuale del videogioco
- products: lista delle istanze (Product) di quel videogioco presenti in magazzino o che devono arrivare

3.1.2 Product

Rappresenta una singola istanza di un videogioco che è presente nel catalogo. NB: un oggetto di tipo Product non esiste senza un riferimento a Detail, come nella realtà un prodotto deve essere "qualcosa" del catalogo. Gli attributi sono:

- id: un identificativo univoco del prodotto
- arrived: un booleano che specifica se il prodotto è arrivato o meno in negozio
- detail: un riferimento (sempre non nullo) ad un Detail che specifica di che videogioco si tratta
- customer: un riferimento ad un Customer che specifica di chi è l'eventuale prenotazione del prodotto stesso. Nel caso in cui il campo sia nullo/vuoto, il prodotto non è stato prenotato da nessuno ed è in vendita libera in negozio

3.1.3 Customer

Rappresenta un cliente registrato che può accedere alle funzionalità riservate a lui. Da notare che non solo un Customer può compiere acquisti: lo possono fare anche clienti non registrati, senza però sfruttare gli altri servizi. Gli attributi sono:

- id: un identificativo univoco del cliente (da ora in poi ci riferiremo a lui come UserID)
- name: una stringa contenente il nome del cliente
- surname: una stringa contenente il cognome del cliente
- email: una stringa contenente l'e-mail del cliente
- points: un contatore di punti che vengono incrementati (sotto opportune condizioni) con gli acquisti del cliente
- products: lista dei prodotti prenotati dal cliente, sia arrivati in negozio che non

3.1.4 Employee

Rappresenta un dipendente assunto che lavora all'interno del negozio. Può compiere operazioni di gestione limitate all'interazione con il magazzino e alla registrazione di nuovi clienti. Gli attributi sono:

- id: un identificativo univoco del dipendente
- name: una stringa contenente il nome del dipendente
- surname: una stringa contenente il cognome del dipendente
- email: una stringa contenente l'e-mail del dipendente
- salary: contiene lo stipendio attuale del dipendente

3.1.5 Cashdesk

Rappresenta in maniera astratta il registratore di cassa con cui il dipendente effettua pagamenti e resi all'interno del software. La classe si occupa inoltre di aggiungere punti al cliente, qualora fosse uno registrato. Gli attributi sono:

- productList: la lista temporanea dei prodotti finora scannerizzati
- customer: un riferimento ad un Customer che, pagando, riceverà i punti. Se il campo è nullo, non verranno assegnati punti.
- price: rappresenta il subtotale attuale.
- productController

- detailController
- customerController

Gli ultimi 3 attributi non sono stati descritti perché saranno approfonditi successivamente.
I metodi di Cashdesk sono elencati nel seguente snippet:

```
1 usage  Mattia Baroncelli
public void addProduct(Long productID){...}
1 usage  Mattia Baroncelli
public void givePoints(){...}
1 usage  Mattia Baroncelli
public void pay(){...}
1 usage  Mattia Baroncelli
public void cashBack(Long gameId){...}
1 usage  Mattia Baroncelli
public String displayProducts(){...}
2 usages  Mattia Baroncelli
public float getPrice() { return price; }
Mattia Baroncelli
public void setCustomer(Long customerid) { this.customer=customerController.getCustomerById(customerid); }
```

Figura 10: Metodi di Cashdesk

3.2 businessLogic

businessLogic è il package che si occupa della gestione dei dati. Espone i metodi per creare, modificare e eliminare gli elementi del domainModel, mai accedendo direttamente al Database.

3.2.1 DetailController

La suddetta classe espone i metodi per creare e modificare gli oggetti di tipo Detail. I principali metodi implementati nella classe sono i seguenti:

- createDetail: dati nome, console e prezzo, crea un nuovo Detail e salva il risultato nel Database
- searchProduct: dato il nome di un Detail, restituisce nome, console e prezzo del Detail stesso
- existDetail: dato un plausibile GameID, restituisce un booleano che indica se esiste un Detail con quell'ID
- editPrice: dato il GameID e un nuovo prezzo, aggiorna il prezzo del Detail corrispondente e salva il risultato nel Database

3.2.2 ProductController

La suddetta classe espone i metodi per creare, eliminare e modificare gli oggetti di tipo Product. La classe, inoltre, contiene un riferimento a EmailController, in quanto la classe si occupa di inviare l'e-mail una volta arrivato un particolare Product. I principali metodi implementati nella classe sono i seguenti:

- addProductRequest: dati GameID e una quantità, vengono create istanze di Product con il campo arrived inizializzato a "falso" e salva i risultati nel Database. Il metodo, come controllo, verifica se il GameID è effettivamente corrispondente ad un Detail
- orderProduct: dati GameID e UserID, permette ad un dipendente di ordinare un gioco per conto di un cliente e salva il risultato nel Dataase. Il metodo, come controllo, verifica se il GameID è effettivamente corrispondente ad un Detail e se l'UserID è effettivamente corrispondente ad un Customer

- arrivedProduct: dati GameID e una quantità, il sistema registra l'arrivo dei videogiochi in questione. Se una o più istanze di quel videogioco sono state prenotate da uno o più clienti, questi ultimi vengono notificati e i prodotti interessati non sono mai vendibili liberamente. I restanti vengono aggiunti regolarmente al magazzino, impostando a "true" l'attributo arrived. Se il prodotto non è stato ordinato né da un cliente, né dal negozio (eventualità di un rifornimento non chiesto), viene creata sul momento un'istanza di Product con quel GameID
- searchProduct: dato GameID, restituisce nome, console, prezzo e quantità nel Database

3.2.3 CustomerController

La suddetta classe espone i metodi per creare e modificare gli oggetti di tipo Customer. La classe, inoltre, contiene un attributo identificativo currentCustomer: utile per la GUI (lo vedremo in seguito). I principali metodi implementati nella classe sono i seguenti:

- registerCustomer: dati nome, cognome e e-mail, registra un nuovo cliente inizializzando i punti a 0 e salva il risultato nel Database
- getOrderByCustomer: dato UserID, restituisce tutti i videogiochi che il cliente ha prenotato, indicando se questi sono o non sono arrivati in negozio
- displayAllCustomer: mostra l'elenco di tutti i customer, con le relative informazioni

3.2.4 EmployeeController

La suddetta classe espone i metodi per creare e modificare gli oggetti di tipo Employee. La classe, inoltre, contiene un attributo identificativo currentEmployee: utile per la GUI (lo vedremo in seguito). I principali metodi implementati nella classe sono i seguenti:

- registerEmployee: dati nome, cognome, e-mail e stipendio registra un nuovo dipendente e salva il risultato nel Database
- editSalary: dato ID e un valore decimale, modifica lo stipendio del dipendente con quell'id
- displayAllEmployee: mostra l'elenco di tutti i dipendenti, con le relative informazioni

3.3 DAO

Questo package si occupa di gestire le connessioni e le operazioni sul database, attraverso il pattern DAO, tramite l'utilizzo di un database MySQL e l'utilizzo del framework Spring Boot JPA (Java Persistence API), di cui parleremo più tardi.

3.3.1 JPA Repositories

Queste repository sono fornite dall'API appena menzionata. Sono delle interfacce che vengono implementate dal momento in cui vengono specificati la classe su cui operare e il tipo dell'id (chiave primaria) degli oggetti di quella classe. I principali metodi che vengono forniti dalle JPA Repository sono i seguenti:

- save: Dato un oggetto, lo salva/lo aggiorna nel Database
- delete: Dato un oggetto, lo cancella nel Database
- findById: Dato l'ID, restituisce l'oggetto corrispondente (se esiste)
- existsByID: Dato l'ID, restituisce un booleano che comunica se l'oggetto esiste all'interno del Database

Inoltre, le Repository permettono di scrivere query (di sola lettura) nel linguaggio SQL relative alle entità della classe a cui si riferiscono, tramite l'annotazione Spring Boot @Query

Essendo 4 le entità rappresentate nel nostro database, le JPA Repository sono 4:

- ProductRepository
- DetailRepository
- CustomerRepository
- EmployeeRepository

3.3.2 SoftwareDAO

Si interfaccia con il Database per la gestione delle varie tabelle, occupandosi di separare un "Oggetto Java" dall'entità corrispondente salvata nel Database. Dentro la classe sono contenute le 4 JPA Repository illustrate prima. Inoltre, SoftwareDAO si occupa di inizializzarle. Questa classe è quella che avvia il collegamento che mette in comunicazione programma e Database, attraverso il passaggio del paramentro ApplicationContext ottenuto una volta avviata l'applicazione. I principali metodi che vengono forniti da SoftwareDAO sono i seguenti:

- save[Nome Classe]: salva, attraverso la corrispondente Repository, l'oggetto nel Database e notifica il successo dell'operazione
- delete[Nome Classe]: cancella, attraverso la corrispondente Repository, l'oggetto nel Database e notifica il successo dell'operazione
- getAll[Nome Classe]: fornisce una lista contenenti tutti gli oggetti attualmente esistenti di quella classe
- get[Nome Classe]: restituisce un'occorrenza di un oggetto data la sua chiave primaria
- getDetailByName: restituisce un'occorrenza di un oggetto Detail dato il suo nome.
- exist[Nome Classe]: restituisce un booleano che, dato un ID, indica se l'oggetto è presente nel Database
- getOrderedProductsByGameId: mostra la lista dei prodotti corrispondenti a un preciso videogioco, ordinati da uno o più clienti
- getNonArrivedProductsByGameId: mostra la lista dei prodotti corrispondenti a un preciso videogioco, ma che non sono ancora arrivati in negozio

3.4 mailManager

Questo package si occupa di incapsulare le 2 classi adibite al servizio di invio di email ai clienti registrati. Il servizio è usato quando un videogioco ordinato da un preciso utente, arriva in negozio.

3.4.1 EmailSenderService

Questa classe è un @Service offerto dal framework Spring Boot Java Mail Sender. Si occupa di stabilire una connessione SMTP e inviare una email. Il metodo contenuto in essa è sendMail, che prende in ingresso 3 stringhe contenenti destinatario, oggetto e contenuto, inviando correttamente l'email e notificando il terminale del successo dell'operazione.

3.4.2 EmailController

Questa classe contiene un'occorrenza di EmailSenderService e si occupa di inizializzarla. Si tratta di un intermediario tra l'utente (chi usa il software) e Service. Controlla gli invii di email tramite due metodi:

- notifyUser: compila un messaggio preimpostato con i dati del cliente e del videogioco arrivato. Il risultato sarà presente nel corpo del messaggio inviato
- sendEmail: chiama il metodo sendMail del @Service passandogli i parametri di cui ha bisogno


Tom and Jerry: Guerra all'ultimo baffo è arrivato in negozio

E ecologgamesstore@gmail.com
a alattucci

Ciao Alessandro,
Il tuo gioco: Tom and Jerry: Guerra all'ultimo baffo è arrivato in negozio. Vieni a ritirarlo quando desideri.

+++Se ricevi questa mail e non hai ordinato tu il gioco informaci immediatamente dell'accaduto rispondendo a questa mail.+++

Il Team di EcologGames
Via Sette Marzo 151, Prato (PO) 59100
+39 0574442633
ecologgamesstore@gmail.com

Figura 11: Email inviata al cliente che ha prenotato il videogioco

3.5 GUI

Questo package contiene tutte le classi riguardanti l'interfaccia grafica del programma. E' stata realizzata con Java Swing e con l'editor fornito dall'IDE. Sono presenti 4 frame, di cui 3 riguardanti i 3 diversi casi d'uso, e il primo per effettuare il login. Di seguito la schermata di login (GUIStartFrame):

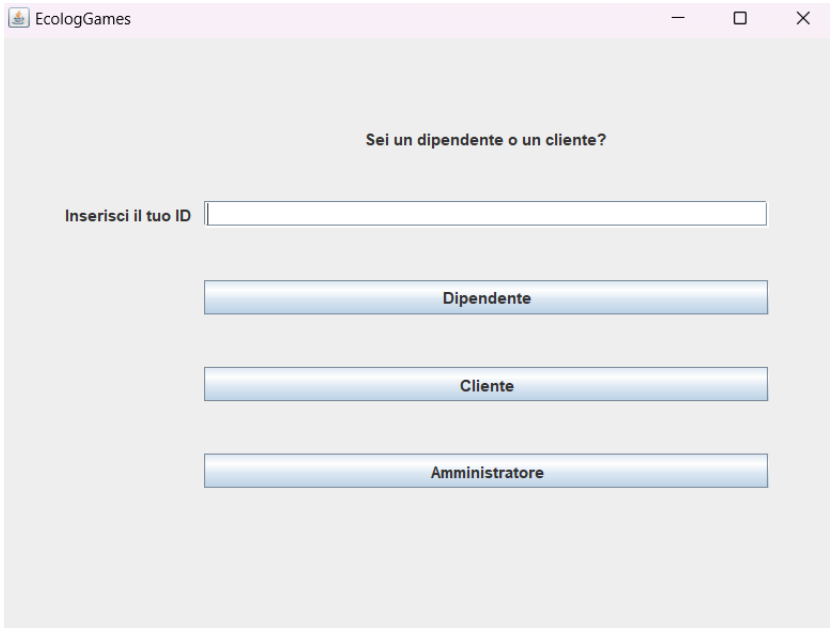

Figura 12: Schermata di login con la scelta tra 3 possibili attori

Le 4 interfacce grafiche sono rappresentate dalle seguenti 4 classi:

- GUIStartFrame (raffigurata sopra)
- EmployeeGUI
- CustomerGUI
- SuperuserGUI

4 Test (JUnit)

I test sono stati realizzati con l'obiettivo di testare tutte le funzionalità dell'applicativo, ponendo particolare attenzione all'esecuzione corretta di tutti i metodi. Per realizzare i test è stato utilizzato JUnit. Di seguito si riporta l'insieme dei test effettuati, e l'esito positivo degli stessi.

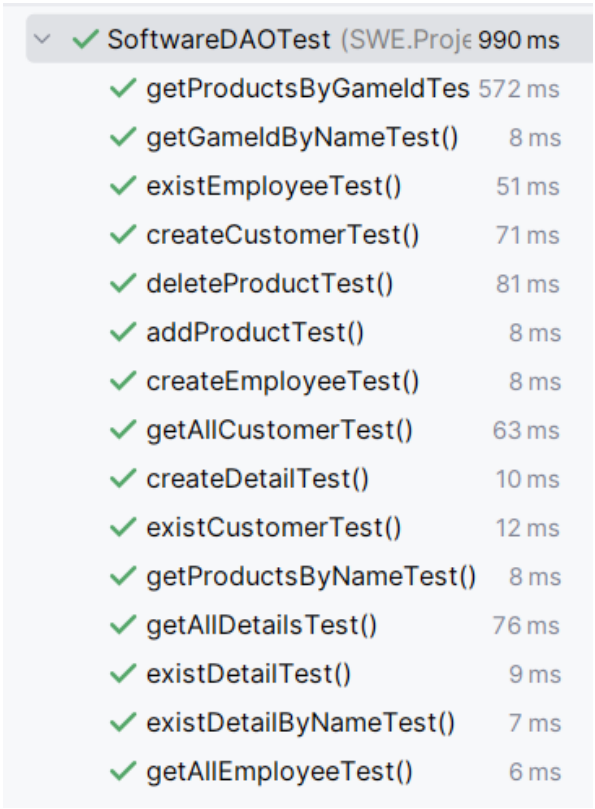

Figura 13: Test eseguiti con esito positivo

Di seguito, saranno mostrati e commentati gli snippet di codice di alcuni dei test eseguiti.

4.1 getProductsByGameIdTest

```
@Test
void getProductsByGameIdTest(){

    softwareDAO= new SoftwareDAO(productRepository,detailRepository,customerRepository,employeeRepository);
    List<Long> list= softwareDAO.getProductsByGameId(40L);
    Assertions.assertEquals( expected: 2,list.size());

}
```

Figura 14: Snippet di getProductsByGameIdTest

Questo test serve a verficare la correttezza della funzione getProductsByGameId, la quale restituisce una lista di tutti i prodotti esistenti che sono appartenenti ad un particolare videogioco. Contando manualmente le occorrenze grazie alla tabella sul Database, in quel momento i Product con GameID uguale a 40 erano 2. E' stato nostro compito verificare se il metodo in questione restituisse una lista di dimensione 2.

4.2 existDetailByNameTest

```
@Test
void existDetailByNameTest(){
    softwareDAO= new SoftwareDAO(productRepository,detailRepository,customerRepository,employeeRepository);
    Assertions.assertEquals( expected: true,softwareDAO.existDetailByName("Crash Bandicoot"));
    Assertions.assertEquals( expected: false,softwareDAO.existDetailByName("8F"));

}
```

Figura 15: Snippet di existDetailByNameTest

Questo test serve a verficare la correttezza della funzione existDetailByName, la quale restituisce un booleano sull'esistenza di un Detail dato un ID. E' stato nostro compito verificare se il metodo in questione restituisse vero o falso in modo corretto.

4.3 addProductTest

```
@Test
void addProductTest() {
    Assertions.assertEquals( expected: 63,productRepository.findAll().size());
    Product product=new Product();
    productRepository.save(product);
    Assertions.assertEquals( expected: 64,productRepository.findAll().size());
}
```

Figura 16: Snippet di addProductTest

Questo test serve a verficare la correttezza della funzione addProduct, la quale crea un prodotto e lo salva sul Database. E' stato nostro compito verificare se la lista complessiva dei prodotti aumentasse di 1 appena creato un prodotto. Nello specifico, prima di creare un nuovo Product, abbiamo contato manualmente i Product presenti nel Database (nel nostro caso 63). Successivamente, abbiamo creato un Product e verificato che la quantità fosse aumentata a 64.
Da notare come questo test verifichi anche la correttezza della funzione findAll.

5 Note sulla connessione al Database

In questo capitolo troveremo un approfondimento su come è stato realizzato il collegamento al Database.

5.1 DBMS: MySQL

Il DBMS utilizzato è MySQL, sviluppato da Oracle Corporation. Si tratta di un RDBMS (DBMS relazionale), per cui i dati sono organizzati in tabelle messe in relazione tra loro. In questo caso, le classi Java sono le entità (gli attributi sono le colonne) e gli oggetti sono le occorrenze. Abbiamo inoltre fatto uso di MySQL Workbench, un interfaccia grafica fornita in aggiunta al software che ci è stata di aiuto.

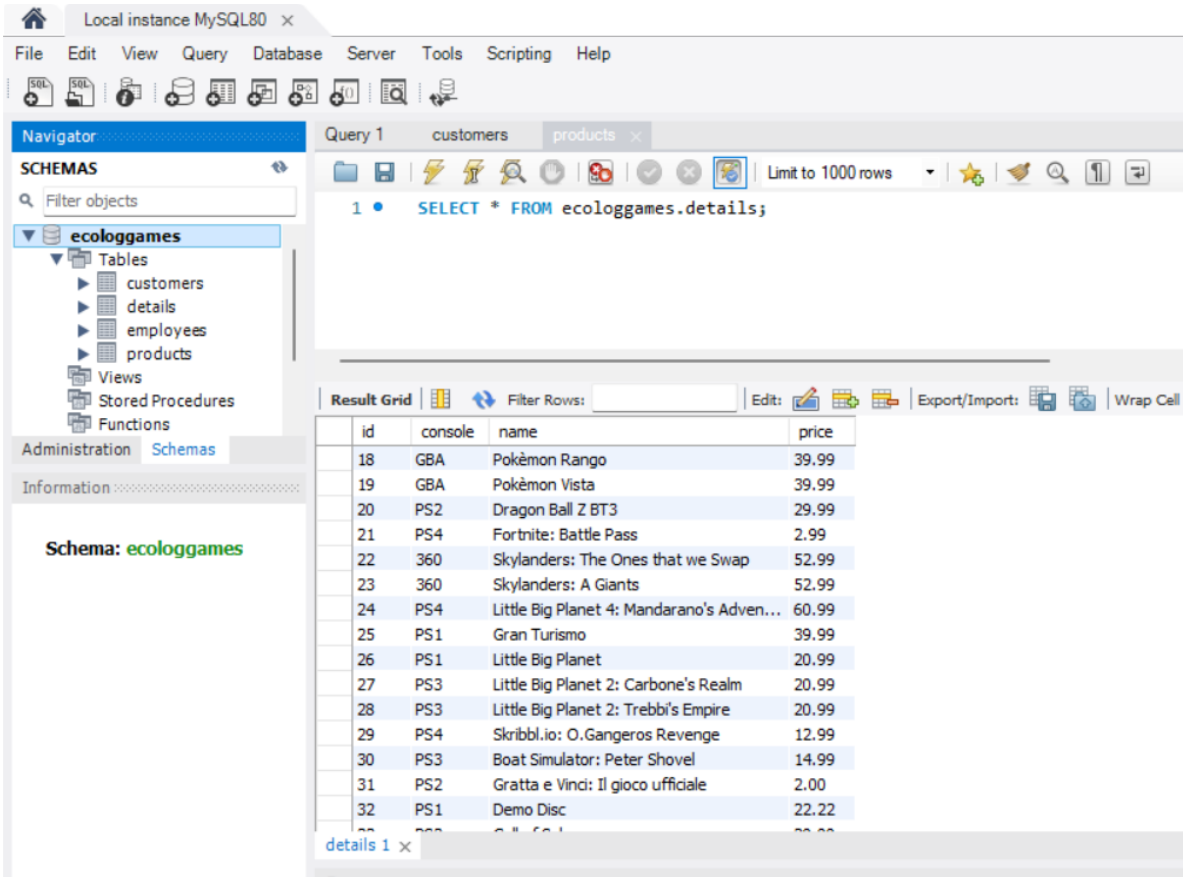

Figura 17: MySQL Workbench 8.0

5.2 Inizializzazione della connessione

Come detto in precedenza, abbiamo fatto uso del Framework Spring Boot, in particolare di uno dei suoi servizi chiamato Spring Data JPA, che permette di mappare facilmente classi Java all'interno di un Database, e di garantirne la loro persistenza. Questo servizio, inoltre, provvede alla connessione tra applicativo Java e Database. Lo fa attraverso il comando SpringApplication.run, che inoltre restituisce un oggetto di tipo ApplicationContext. Quest'ultimo è fondamentale perché serve a inizializzare qualsiasi classe che fa uso delle Repository che comunicano con il Database.

Nel file application.properties, vengono scritti DBMS, indirizzo IP, porta e credenziali di accesso al Database.

```
spring.datasource.url=jdbc:mysql://25.12.212.238:3306/ecologgames?useSSL=false
spring.datasource.username=mattia
spring.datasource.password=mattia
```

Figura 18: Snippet di application.properties, nella parte che riguarda il Database

5.3 Mappatura delle classi Java nel Database

Spring Data JPA mette a disposizione delle annotazioni (nella forma "@Annotazione") che forniscono delle funzionalità per le classi che devono far parte del Database: nello specifico, le classi Product, Detail, Customer, Employee. Le annotazioni utilizzate sono:

- @Entity: da posizionare prima della dichiarazione della classe. Specifica che questa classe diventerà un'entità (tabella) nel Database.
- @Table: specifica che nome deve avere la tabella corrispondente (se si vuole con nome diverso dalla classe) e lo schema relazionale a cui fa riferimento
- @Id: da posizionare prima dell'attributo che diventerà la chiave primaria, solitamente un Longint
- @GeneratedValue: opzionale, indica che la chiave primaria è generata automaticamente. Specifica anche la strategia di generazione (nel nostro caso la strategia adottata è IDENTITY, ovvero le chiavi primarie vengono generate con valori contigui)
- @ManyToOne / @OneToMany: indica che l'attributo in questione ha un vincolo di integrità referenziale con uno di un'altra tabella (indicata dopo l'annotazione @JoinColumn)

```
@Entity
@Table
        (
        name="products",
        schema="ecologgames"
)
public class Product {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;
    5 usages
    private Boolean arrived;

    4 usages
    @ManyToOne
    @JoinColumn(name="game")
    private Detail detail;

    5 usages
    @ManyToOne
    @JoinColumn(name="user")
    private Customer customer;
```

Figura 19: Snippet della classe Product, mappata nel Database

La prima volta in cui viene inserita l'annotazione @Entity in una classe ed eseguito il programma, il DBMS procederà a effettuare automaticamente una query di creazione (CREATE TABLE) e inserirà come colonne tutti gli attributi.

5.4 Implementazione delle query

Spring Data JPA, inoltre, fornisce un'annotazione compatibile solamente con le interfacce che diventano JPA Repository (@Repository). Tramite queste interfacce, è possibile utilizzare i metodi CRUD, quali aggiunta, cancellazione e modifica di occorrenze. Spring Data JPA mette a disposizione queste funzioni senza il bisogno di riscriverle esplicitamente. Oltre a questo, all'interno della JPA Repository, è possibile scrivere query in linguaggio SQL, con l'uso dell'annotazione @Query e con la possibilità di avere parametri in ingresso e decidere il tipo di ritorno.

```
@Repository
public interface ProductRepository extends JpaRepository<Product,Long> {
    1 usage  Mattia Baroncelli
    @Query  ("SELECT id FROM Product WHERE customer != null AND arrived=false  AND detail.id = ?1 ")
    List<Long> getOrderedProductsByGameIdQuery(Long gameid);
    1 usage  Mattia Baroncelli
    @Query  ("SELECT id FROM Product WHERE customer = null AND arrived=false  AND detail = ?1 ")
    List<Long> getNonArrivedProductsByGameIdQuery(Detail gameid);
    1 usage  Mattia Baroncelli
    @Query("SELECT  id  FROM Product   WHERE detail.id=?1 AND customer=null AND arrived=true")
    List<Long>  getProductsByGameIdQuery(Long gameid);
    2 usages  Mattia Baroncelli
    @Query("SELECT  id FROM Product WHERE detail.name=?1 AND customer=null AND arrived=true")
    List<Long> getProductsByNameQuery(String name);
}
```

Figura 20: Snippet di ProductRepository con le relative query

Da notare che i metodi contenuti all'interno delle JPA Repository rispettano la sintassi richiesta dalle interfacce: infatti è presente solo la dichiarazione dei metodi e non l'implementazione, come erroneamente si potrebbe pensare vedendo scritta la query SQL sopra il metodo. Java, infatti, non la esegue, bensì è il DBMS a riceverla, eseguirla e implementare di fatto il metodo.

5.5 Rete privata: Hamachi

Per l'intera realizzazione a 4 mani del progetto, c'era la necessità di avere a disposizione il Database sincronizzato su entrambi i nostri computer. La soluzione più pratica è stata quella di creare una rete privata in cui sono presenti solo i computer che possono accedere al Database, e impostare su MySQL l'indirizzo IPv4 fornito all'host dopo la creazione della rete privata (porta 3306). L'applicazione usata è LogMeIn Hamachi.

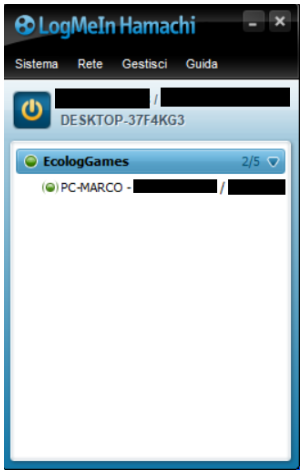

Figura 21: Hamachi