iSnapGaming

ODD

ODD

Introduction

L'Object Design Document (ODD) è un documento essenziale nel processo di sviluppo software. Tale documento si basa sui documenti RAD e SDD, che consolidano le informazioni raccolte durante l'analisi dei requisiti e la progettazione, l'ODD offre una guida dettagliata su come strutturare e implementare il sistema. Descrive le interfacce delle classi, le operazioni supportate, i tipi di dati utilizzati, i parametri delle procedure, i signature dei sottosistemi, trade-off . Questo documento include anche strategie per gestire compromessi di progettazione durante lo sviluppo, fungendo da "piano di costruzione" per il software.

Object design trade-offs

1.1 Robustezza vs Velocità di implementazione

Nel gestire i dati in ingresso, sappiamo quanto sia importante controllarli bene. Tuttavia, per rilasciare la prima versione del sistema il più velocemente possibile, abbiamo deciso di accettare che i controlli iniziali non siano perfetti. Ci impegniamo comunque a migliorare questi controlli nelle versioni future del sistema. Questa scelta ci permette di uscire rapidamente con la prima versione, sapendo che rafforzeremo la robustezza nei prossimi aggiornamenti.

1.2 Chiarezza vs Velocità di implementazione

Per semplificare il testing, sarebbe ideale scrivere codice molto chiaro. Tuttavia, con i tempi di sviluppo stretti per questa prima versione, non sempre potremo mantenere i nostri standard abituali. Nelle versioni future del sistema, potremo migliorare questo aspetto.

Interface documentation guidelines

1. Le classi hanno dei nomi comuni singolari.
2. I metodi sono denominati con frasi verbali.
3. Gli Error Status sono restituiti attraverso eccezioni.

References

- SDD: ci si riferisce al SDD quando si spiega l'organizzazione dei package, dato che quest' ultima è stata creata proprio a partire dalla suddivisione in subsystem.

Directories

main > java > com.isnapgaming > OrderManagement

- Cart.java
- CustomerOrder.java
- ItemCart.java
- OrderProduct.java
- OrderCreation.java

main > java > com.isnapgaming > PaymentManagement

- PaymentAuthorizationService.java
- PaymentAuthorizationServiceAdapter.java
- PaymentAuthServiceInterface.java

main > java > com.isnapgaming > ProductManagement

- Product.java

main > java > com.isnapgaming > StoreManagement.DAO

- AddressDAO.java
- CustomerDAO.jva
- CustomerOrderDAO.java
- ProductDAO.java
- UserDAO.java

main > java > com.isnapgaming > UserManagement

- Address.java
- Customer.java
- Manager.java
- OrderManager.java
- ProductManager.java
- User.java

main > java > com.isnapgaming > view

- AccessControlFilter.java
- AddProduct.java
- AddToCart.java
- GetOrderDetails
- GetOrdersList
- ImageServlet
- Login.java
- Logout.java
- PayOrder
- ProductDetails
- RoleSelection.java
- Signup
- UpdateCart
- UpdateStatus

main > java > testing

Sotto tale directory sono presenti tutti i file per poter inizializzare il database prima di ogni test case durante le attività di testing:

- RetrieveCredentials
- SQLScript

main > webapp > app_imgs > logo

- logo.png

main > webapp > app_imgs > products

Sotto tale directory sono presenti tutti i file riguardanti i prodotti:

- 331_1.jpg
- 652_1.jpg
- 751_1.jpg
- 111_1.jpg

main > webapp > fragments

- header.jsp
- footer.jsp

main > webapp > META-INF

- context.xml

main > webapp > scripts

- payment.js
- validate.js

main > webapp > styles

- access.css
- addProduct.css
- cart.css
- checkout.css
- confirmationPage.css
- errorPage.css
- footer.css
- header.css
- index.css
- orderDetails.css
- orderManagerDashboard.css
- product.css
- productManagerDashboard.css
- roleSelection.css
- updateStatus.css

main > webapp > WEB-INF

- web.xml

main > webapp

- addProduct.jsp
- cart.jsp
- checkout.jsp
- confirmationPage.jsp
- customerOrdersList.jsp
- errorPage.jsp
- index.jsp
- login.jsp
- orderDetails.jsp
- orderManagerDashboard.jsp
- product.jsp
- productManagerDashboard.jsp
- roleSelection.jsp
- signup.jsp
- updateStatus.jsp

Design Patterns

All’interno del nostro progetto sono stati utilizzati dei design pattern per riuscire a implementare alcune funzionalità del sistema.

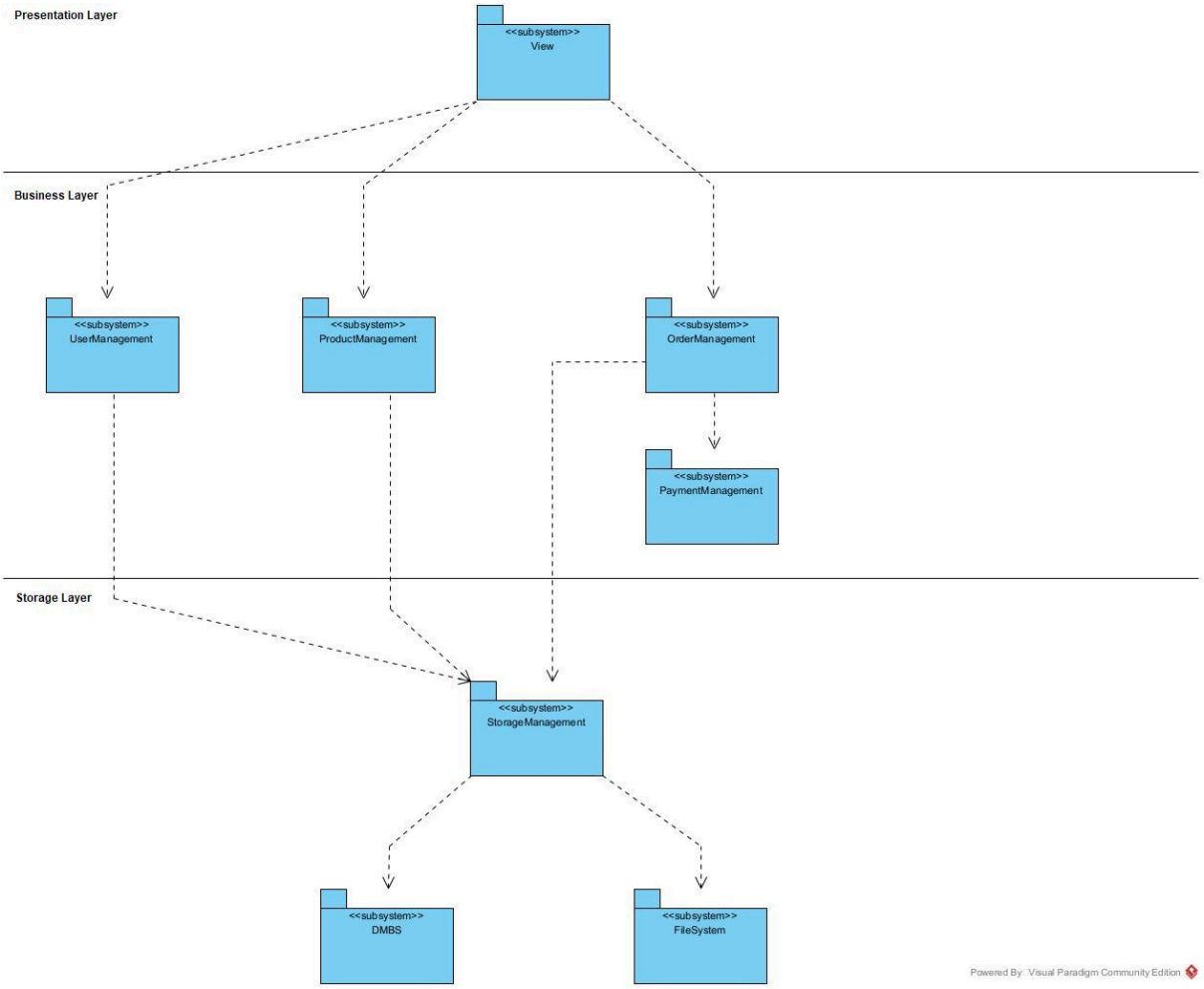
Ad esempio abbiamo ritenuto utile andare a utilizzare un pattern di tipo DAO che prevede la creazione di una classe DAO per ogni tipo di oggetto per il quale si desidera gestire la persistenza. Ogni classe DAO si occupa di gestire l’accesso ai dati per quel tipo di oggetto e fornisce metodi per eseguire le operazioni come il recupero, l’inserimento, l’aggiornamento e la cancellazione dei dati. L’obiettivo principale del pattern DAO è quello di separare l’accesso ai dati dall’applicazione stessa, in modo da poter modificare o sostituire facilmente la fonte di dati senza dover modificare il codice dell’applicazione.

Abbiamo anche pensato che fosse utile adottare un pattern di tipo interceptor per implementare il filtro che controlla i permessi necessari per eseguire una certa funzionalità. In tal modo, intercetteremo la richiesta e la rifiuteremo se il permesso richiesto non è presente.

Infine, abbiamo optato per un pattern adapter per incapsulare la logica di pagamento. Questa logica viene gestita tramite servizi forniti da terze parti, non sviluppati direttamente da noi. Utilizzare questo pattern è essenziale per evitare la dipendenza da quei servizi e dalle loro implementazioni, consentendoci di mantenere un disaccoppiamento efficace.

Packages

Il sistema è stato suddiviso nei seguenti subsystems:



A partire da questo subsystem sono stati quindi prodotti i seguenti packages:

1. ProductManagement
2. OrderManagement
3. PaymentManagement
4. UserManagement
5. StorageManagement
6. view

1. All'interno di ProductManagement sono presenti le classi per la gestione del prodotto. Quest'ultimo è mappabile direttamente al sottosistema ProductManagement**.**

2. All'interno di OrderManagement sono presenti le classi per la gestione dell'ordine. Quest'ultimo è mappabile direttamente al sottosistema OrderManagement**.**

3. All'interno di PaymentManagement sono presenti le classi per la gestione del pagamento. Quest'ultimo è mappabile direttamente al sottosistema PaymentManagement**.**

4. All’interno di UserManagement sono presenti le classi per la gestione dell’utente. Quest’ultimo è mappabile direttamente al sottosistema UserManagement**.**

5. All’interno di StorageManagement sono presenti tutte le classi necessarie per riuscire a garantire la persistenza dei dati. Quest’ultimo è direttamente mappabile al sottosistema StorageManagement. Il seguente package possiede delle dipendenze con i package UserManagement, OrderManagement, ProductManagement.

6. All’interno di view si trovano tutti i package che contengono le servlet, le quali gestiscono la logica di controllo dell’applicazione. Le servlet sono organizzate nei vari package sulla base del ruolo di cui ci permettono di realizzare le funzionalità. Questo package, ha una dipendenza con tutti gli altri perché realizzando la logica di controllo invoca i metodi delle classi che rappresentano i dati e permettono di effettuare la persistenza dei dati.

Class Interfaces

Nella specifica delle interfacce abbiamo deciso di non andare a specificare ogni getter e setter presenti nelle classi entity e, inoltre, di non andare a effettuare la specifica delle interfacce per tutti i metodi privati delle classi.
Non abbiamo effettuato la specifica delle interfacce per tutti i metodi privati delle classi.
Le classi DAO sono le uniche classi che accedono al database.

User Management

User	
Description	La classe User rappresenta un utente con ruolo generico registrato all’interno del sistema.
Invariant	**context** User **inv:** username <> null & password <> null & firstName <> null & lastName <> null & dateOfBirth <> null
makeUser	**context** User::makeUser(username: String, password: String, firstName: String, lastName: String, dateOfBirth: LocalDate) **pre:** username <> null & password <> null & firstName <> null & lastName <> null

	& dateOfBirth <> null **context** User::makeUser(username: String, password: String, firstName: String, lastName: String, dateOfBirth: LocalDate) **post:** result.username = username & result.password = password & result.firstName = firstName & result.lastName = lastName

Customer	
Description	Rappresenta un utente registrato con ruolo Customer.
Invariant	**context** Customer **inv:** **-**

Manager	
Description	Rappresenta un manager generico all'interno del sistema
Invariant	**context** Manager **inv**: -

OrderManager	
Description	Classe che rappresenta un OrderManager.
Invariant	**context** OrderManager **inv:** **-**
getAllCustomerOrders	Metodo che restituisce tutti gli ordini presenti all'interno del database. **context** OrderManager::getAllCustomerOrders(dataSource: DataSource) **pre**: - **context** OrderManager::getAllCustomerOrders(dataSource: DataSource) **post**: -

checkProduct	Metodo che permette di aggiornare lo stato dell'ordine in *UnderPreparation* **context** OrderManager::checkProduct(order: CustomerOrder, dataSource: DataSource) **pre**: order <> null & dataSource <> null, & order.getStatus() = "TO_BE_MANAGED" **context** OrderManager::checkProduct(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = "UNDER_PREPARATION"
packProduct	Metodo che permette di aggiornare lo stato dell'ordine in *ReadyForSending.* **context** OrderManager::packProduct(order: CustomerOrder, dataSource: DataSource) **pre**: order <> null & dataSource <> null & order.getStatus() = "UNDER_PREPARATION" **context** OrderManager::packProduct(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = "READY_FOR_SENDING"
replaceProduct	Metodo che permette di aggiornare lo stato dell'ordine in *UnderPreparation.* **context** OrderManager::replaceProduct(order: CustomerOrder, dataSource: DataSource) **pre**: & order <> null & dataSource <> null order.getStatus() = "READY_FOR_SENDING" **context** OrderManager::replaceProduct(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = "UNDER_PREPARATION"
contactCourier	Metodo che permette di aggiornare lo stato dell'ordine in *Shipped.* **context** OrderManager::contactCourier(order: CustomerOrder, dataSource: DataSource) **pre**: order <> null & dataSource <> null & order.getStatus() = "READY_FOR_SENDING" **context** OrderManager::contactCourier(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = "SHIPPED"
restoreOrder	Metodo che permette di aggiornare lo stato dell'ordine in *UnderPreparation.*

	context OrderManager::restoreOrder(order: CustomerOrder, dataSource: DataSource) **pre**: order <> null & dataSource <> null & order.getStatus() = “SHIPPED” **context** OrderManager::restoreOrder(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = “UNDER_PREPARATION”
confirmDelivery	**context** OrderManager::confirmDelivery(order: CustomerOrder, dataSource: DataSource) **pre**: order <> null & dataSource <> null & order.getStatus() = “SHIPPED” **context** OrderManager::confirmDelivery(order: CustomerOrder, dataSource: DataSource) **post**: order.getStatus() = “DELIVERED”

ProductManager	
Description	Rappresenta un Product Manager che gestisce i prodotti all’interno del sistema
Invariant	**context** ProductManager **inv:** **-**
addProduct	Metodo che permette di inserire un nuovo prodotto nel sistema. **context** ProductManager::addProduct(product: Product, dataSource: DataSource) **pre:** product <> null & dataSource <> null **context** ProductManager::addProduct(product: Product, dataSource: DataSource) **post:** this.getProductByProdCode(product.getProdCode(), DataSource) = product
updateProduct	Metodo che permette di aggiornare un prodotto già presente nel sistema. **context** ProductManager::updateProduct(product: Product, dataSource: DataSource) **pre:** product <> null & dataSource <> null

	context ProductManager::updateProduct(product: Product, dataSource: DataSource) **post:** this.getProductByProdCode(product.getProdCode(), DataSource) = product
removeProduct	Metodo che permette di rimuovere un prodotto dal catalogo rendendolo, cioè, non disponibile alla vendita. **context** ProductManager::removeProduct(product: Product, dataSource: DataSource) **pre:** product <> null & dataSource <> null **context** ProductManager::removeProduct(product: Product, dataSource: DataSource) **post**: this.getProductByProdCode(product.getProdCode(), DataSource).isAvailable = false
makeProductAvailable	Metodo che permette di rendere un prodotto disponibile alla vendita. **context** ProductManager::makeProductAvailable(product: Product, dataSource: DataSource) **pre**: product <> null & dataSource <> null **context** ProductManager::makeProductAvailable(product: Product, dataSource: DataSource) **post**: this.getProductByProdCode(product.getProdCode(), DataSource).isAvailable = true
getProductByProdCode	Metodo che restituisce il prodotto che ha come prodCode quello specificato. **context** ProductManager::getProductByProdCode(prodCode: Integer, dataSource: DataSource) **pre:** prodCode > 0 & dataSource <> null **context** ProductManager::getProductByProdCode(prodCode: Integer, dataSource: DataSource) **post:** result <> null & result.prodCode = prodCode

getAllProducts	Metodo che restituisce tutti i prodotti presenti nel sistema. **context** ProductManager::getAllProducts(dataSource: DataSource) **pre:** dataSource <> null **context** ProductManager::getAllProducts(dataSource: DataSource) **post:** -

Address	
Description	Rappresenta un indirizzo associato ad un Customer.
Invariant	**context** Address **inv**: customerId > 0 & street <> null & city <> null & postalCode > 0
makeAddress	**context** Address::makeAddress(customerId: Integer, street: String, city: String, postalCode: Integer) **pre**: customerId > 0 & street <> null & city <> null & postalCode > 0 **context** Address::makeAddress(customerId: Integer, street: String, city: String, postalCode: Integer) **post**: result.customerId = customerId & result.street = street & result.city = city & result.postalCode = postalCode

Product Management

Product	
Description	Rappresenta un generico prodotto inserito nel sistema.
Invariant	**context** Product **inv**: -
makeProduct	**context** Product::makeProduct(prodCode: Integer,

	name: String, softwareHouse: String, platform: Product.Platform, price: Integer, quantity: Integer, category: Product. Category, pegi: Product.Pegi, releaseYear: Integer, imagePath: String) **pre:** prodCode > 0 & name <> null & softwareHouse <> null & platform <> null & price >= 0 & quantity >= 0 & category <> null & pegi <> null & releaseYear <> null & imagePath <> null **context** Product::makeProduct(prodCode: Integer, name: String, softwareHouse: String, platform: Product.Platform, price: Integer, quantity: Integer, category: Product. Category, pegi: Product.Pegi, releaseYear: Integer, imagePath: String) **post:** result.prodCode = prodCode & result.name = name & result.softwareHouse = softwareHouse & result.platform = platform & result.price = price & result.quantity = quantity & result.category = category & result.pegi = pegi & resut.releaseYear = releaseYear & result.imagePath = imagePath

Order Management

Cart	
Description	Rappresenta il carrello associato ad un Customer.
Invariant	**context** Cart **inv:** items <> null & totalPrice >= 0
addToCart	Metodo che consente di aggiungere un prodotto al carrello. Se il prodotto è già presente, allora ne viene semplicemente aggiornata la quantità. **context** Cart::addToCart(product: Product, quantity: Integer) **pre**: product <> null & quantity > 0

	context Cart::addToCart(product: Product, quantity: Integer) **post**: this.getItems().includes(product) = True
removeFromCart	Metodo che consente di rimuovere un prodotto dal carrello. **context** Cart::removeFromCart(product: Product) **pre**: product <> null & this.getItems().includes(product) = True **context** Cart::removeFromCart(product: Product) **post**: this.getItems().includes(product) = False
decreaseQuantityCart	Metodo che consente di decrementare la quantità di un prodotto nel carrello. **context** Cart:decreaseQuantityCart(product: Product, quantity: Integer) **pre**: product <> null & this.getItems().includes(product) = True **context** Cart:decreaseQuantityCart(product: Product, quantity: Integer) **post**: this.getItems().includes(product) = false || quantità del carrello decrementata di quantity

CustomerOrder	
Description	Rappresenta un ordine associato ad un Customer.
Invariant	**context** CustomerOrder **inv**: id > 0 & customerId <> null & (status = “TO_BE_MANAGED” || status = “UNDER_PREPARATION” || status = “READY_FOR_SENDING” || status = “SHIPPED”, || status = “DELIVERED”) & address <> null & orderDate <> null & totalAmount > 0 & products <> null
makeCustomerOrder	Metodo che consente di creare un oggetto CustomerOrder con i campi indicati. **context** CustomerOrder::makeCustomerOrder(customerId: Integer, address: String, orderDate: LocalDate, products: List<OrderProduct>) **pre**: customerId <> null & customerId <> 0 & address <> null

	& address <> “” & orderDate <> null & products<> null **context** CustomerOrder::makeCustomerOrder(customerId: Integer, address: String, orderDate: LocalDate, products: List<OrderProduct>) **post**: this.getCustomerId() = customerId & this.getStatus() = “TO_BE_MANAGED” & this.getAddress() = address & this.getOrderDate() = orderDate, & this.getProducts() = products
addProduct	Metodo che consente di aggiungere un prodotto alla lista dei prodotti associati all’ordine. **context** CustomerOrder::addProduct(product: Product) **pre**: product <> null **context** CustomerOrder::addProduct(product: Product) **post**: this.getProducts().includes(product) = True
calculateTotalAmount	Metodo che calcola il prezzo totale da pagare. **context** CustomerOrder::calculateTotalAmount() **pre**: - **context** CustomerOrder::calculateTotalAmount() **post**: totalAmount > 0

ItemCart	
Description	Classe che rappresenta un particolare prodotto nel carrello di un customer.
Invariant	**context** ItemCart **inv**: cart<> null & product <> null & quantity > 0
getTotalPrice	Metodo che calcola il costo totale di un prodotto in relazione alla quantità inserita. **context** ItemCart::getTotalPrice() **pre**: - **context** ItemCart::getTotalPrice() **post**: getTotalPrice() = this.price * this.quantity

OrderProduct	
Description	Classe che rappresenta un particolare prodotto presente in un CustomerOrder.
Invariant	**context** OrderProduct **inv**: orderId > 0 & productId > 0 & quantity > 0
getTotalPrice	Metodo che calcola il costo totale di un prodotto in relazione alla quantità presente nell'ordine. **context** OrderProduct::getTotalPrice() **pre**: - **context** OrderProduct::getTotalPrice() **post**: getTotalPrice() = this.price * this.quantity

OrderCreation	
Description	Classe che contiene la logica di business relativa alla creazione di un ordine e al pagamento di quest'ulitmo, utilizzata nelle fasi finali del processo di acquisto di un prodotto.
Invariant	**context** OrderCreation **inv**: cart <> null & customer <> null & address <> null
makeOrder	Metodo che crea un ordine al momento del checkout. **context** makeOrder::makeOrder(cart: Cart, customer: User, address: String) **pre**: cart <> null & customer <> null & address <> null **context** OrderProduct::getTotalPrice() **post**: result.customer<>null &result.address<>null &result.orderDate<>null &result.products<>null
pay	Metodo che permette di finalizzare il pagamento dell'ordine. **context** OrderCreation::pay(order: CustomerOrder, address: String, cardNumber: String, expirationDate: LocalDate, cvv: String) **pre**: -

	context OrderCreation::pay(order: CustomerOrder, address: String, cardNumber: String, expirationDate: LocalDate, cvv: String) **post**: -

Storage Management

UserDAO	
Description	Classe DAO relativa alla classe User che consente di gestirne la persistenza.
doSave	Metodo che permette di rendere persistente un User e ne restituisce il relativo id assegnato nel database. **context** UserDAO::doSave(user: User) **pre:** user <> null **context** UserDAO::doSave(user: User) **post:** this.findByKey(user..getId()) = user
getUserByUsernameAndPassword	Metodo che restituisce, se esiste, l'utente che ha username e password specificati. **context** UserDAO::getUserByUsernameAndPassword(username: String, password: String) **pre:** username <> null & username <> "" **context** UserDAO::getUserByUsernameAndPassword(username: String, password: String) **post:** Se username e password corrispondono a un utente presente nel database allora: result = user altrimenti result = null
getUserRoles	Metodo che restituisce i ruoli associati ad un utente. **context** UserDAO::getUserRoles(userId: Integer) **pre:** userId > 0 **context** UserDAO::getUserRoles(userId: Integer) **post:** result <> null
isCustomer	Metodo che restituisce true se un utente è un Customer, false altrimenti.

	context UserDAO::isCustomer(userId: Integer) **pre:** userId > 0 **context** UserDAO::isCustomer(userId: Integer) **post:** -
isOrderManager	Metodo che restituisce true se un utente è un OrderManager, false altrimenti. **context** UserDAO::isOrderManager(userId: Integer) **pre:** userId > 0 **context** UserDAO::isOrderManager(userId: Integer) **post:** -
isProductManager	Metodo che restituisce true se un utente è un ProductManager, false altrimenti. **context** UserDAO::isOrderManager(userId: Integer) **pre:** userId > 0 **context** UserDAO::isOrderManager(userId: Integer) **post:** -
findByKey	Metodo che restituisce l'utente che ha come id quello specificato. **context** UserDAO::findByKey(userId: Integer) **pre:** userId > 0 **context** UserDAO::findByKey(userId: Integer) **post:** result <> null
findByUsername	Metodo che restituisce l'utente che ha come username quello specificato. **context** UserDAO::findByUsername(username: String) **pre:** username <> 0 **context** UserDAO::findByUsername(userId: Integer) **post:**

CustomerDAO	
Description	Classe DAO relativa alla classe Customer che consente di gestirne la persistenza.

doSave	Metodo che permette di rendere persistente un Customer e ne restituisce il relativo id assegnato nel database. **context** CustomerDAO::doSave(customer: Customer) **pre:** customer <> null **context** CustomerDAO::doSave(customer: Customer) **post:** this.findByKey(customer..getId()) = customer
findByKey	Metodo che restituisce il Customer che ha come id quello specificato. **context** CustomerDAO::findByKey(customerId: Integer) **pre:** customerId > 0 **context** CustomerDAO::doSave(customerId: Integer) **post:** result <> null
findAddressesByCustomerId	Metodo che restituisce gli addresses associati al customer che ha come id quello specificato. **context** CustomerDAO::findAddressesByCustomerId(customerId: Integer) **pre:** customerId > 0 **context** CustomerDAO::findAddressesByCustomerId(customerId: Integer) **post:** result <> null

ProductDAO	
Description	Classe DAO relativa alla classe Product che consente di gestirne la persistenza.
doSave	Metodo che permette di rendere persistente un Product e ne restituisce il relativo id assegnato nel database. **context** ProductDAO::doSave(product: Product) **pre:** product <> null **context** ProductDAO::doSave(product: Product) **post:** this.findByKey(product..getId()) = product

doUpdate	Metodo che permette di aggiornare un prodotto nel database. **context** ProductDAO::doUpdate(product: Product) **pre:** product <> null **context** ProductDAO::doUpdate(product: Product) **post:** this.findByKey(product..getId()) = product
findByKey	Metodo che restituisce il Prodotto che ha come id quello specificato. **context** ProductDAO::findByKey(productId: Integer) **pre:** productId > 0 **context** ProductDAO::findByKey(product: Integer) **post:** result <> null
findByProdCode	Metodo che restituisce il prodotto che ha prodCode uguale a quello specificato. **context** ProductDAO::findByProdCode(productId: Integer) **pre:** productId <> null **context** ProductDAO::findByProdCode(productId: Integer) **post:** result <> null
findByCategory	Metodo che restituisce tutti i prodotti appartenenti alla categoria specificata. **context** ProductDAO::findByCategory(category: Product.Category) **pre:** category <> null **context** ProductDAO::findByCategory(category: Product.Category) **post:** result <> null
doRetrieveAll	Metodo che restituisce tutti i prodotti presenti nel database. **context context** ProductDAO::doRetriveAll() **pre:** - **context context** ProductDAO::doRetriveAll()**post:** -

AddressDAO	
Description	Classe DAO relativa alla classe Address che consente di gestirne la persistenza.
doSave	Metodo che permette di rendere persistente un Address e ne restituisce il relativo id assegnato nel database. **context** AddressDAO::doSave(address: Address) **pre:** address <> null **context** AddressDAO::doSave(address: Address) **post:** this.findByKey(address..getId()) = address
findByKey	Metodo che restituisce l'Address che ha come id quello specificato. **context** AddressDAO::findByKey(addressId: Integer) **pre:** addressId > 0 **context** AddressDAO::findByKey(addressId: Integer) **post:** result.getId() = addressId
findByCustomerId	Metodo che restituisce l'Address che ha come customerId quello specificato. **context** AddressDAO::findByCustomerId(customerId: Integer) **pre:** customerId > 0 **context** AddressDAO::findByCustomerId(customerId: Integer) **post:** result.getCustomerId() = customerId

CustomerOrderDAO	
Description	Classe DAO relativa alla classe CustomerOrder che consente di gestirne la persistenza.
doSave	Metodo che permette di rendere persistente un CustomerOrder e ne restituisce il relativo id assegnato nel database. Inoltre, crea la relazione tra CustomerOrder e OrderProduct nel database. **context** CustomerOrderDAO::doSave(order: CustomerOrder) **pre**: order <> null

	context CustomerOrderDAO::doSave(order: CustomerOrder) **post**: this.findByKey(order.getId()) = order & resi persistenti gli OrderProduct associati nel database
updateStatus	Metodo che permette di aggiornare lo stato di un ordine all'interno del database. **context** CustomerOrderDAO::updateStatus(orderId: int, status: CustomerOrder.Status) **pre**: orderId > 0 & this.findByKey(orderId) <> null **context** CustomerOrderDAO::updateStatus(orderId: int, status: CustomerOrder.Status) **post**: this.findByKey(orderId).getStatus() = status
findByKey	Metodo che restituisce il CustomerOrder che ha come id quello specificato. **context** CustomerOrderDAO::findByKey(id: int) **pre**: id > 0 & CustomerOrder presente nel database **context** CustomerOrderDAO::findByKey(id: int) **post:** this.findByKey(id) <> null
findByStatus	Metodo che restituisce tutti i CustomerOrder che hanno lo stato specificato. **context** CustomerOrderDAO::findByStatus(status: CustomerOrder.Status) **pre**: - **context** CustomerOrderDAO::findByStatus(status: CustomerOrder.Status) **post**: result->forAll(co:CustomerOrder | co.getStatus() = status)
doRetrieveAll	Metodo che restituisce tutti i CustomerOrder salvati nel database. **context** CustomerOrderDAO::doRetrieveAll() **pre**: - **context** CustomerOrdeDAO::doRetrieveAll() **post**: -

PaymentManagement

PaymentAuthorizationServiceAdapter

Description	Classe utilizzata per effettuare pagamenti al servizio esterno di pagamento.
checkPaymentData	Metodo che controlla la validità dei dati di pagamento inseriti. In tal caso restituisce true, altrimenti false. **context** PaymentAuthorizationServiceAdapter::checkPaymentData(cardNumber: String, expiration: LocalDate, cvv: String, price: Integer) **pre**: - **context** PaymentAuthorizationServiceAdapter::checkPaymentData(cardNumber: String, expiration: LocalDate, cvv: String, price: Integer) **post**: cardNumber rispetta formato di soli numeri **** **** **** **** & expiration <> null & expiration valida & cvv <> null & cvv.length() = 3 & cvv formato soli numeri & price >= 0
executePayment	**context** PaymentAuthorizationServiceAdapter::executePayment(cardNumber: String, expiration: LocalDate, cvv: String, price: Integer) **pre**: this.checkPaymentData(cardNumber, expiration, cvv, price) = true **context** PaymentAuthorizationServiceAdapter::executePayment(cardNumber: String, expiration: LocalDate, cvv: String, price: Integer) **post**: result = esito pagamento

Object Design Model Optimization

Di seguito si riportano le modifiche effettuate al Class Diagram iniziale:

- La classe Order è stata rinominata in CustomerOrder per evitare ambiguità in termini di denominazione con altre classi di librerire Java. Inoltre vi è stato aggiunto l'attributo *id* per tenere traccia dell'identificativo univoco assegnato alla particolare istanza all'interno del database.
- Per una corretta e completa gestione dei prodotti, è stato aggiunto l'attributo *prodCode* alla classe Product. Tale attributo, permette di identificare un prodotto in base al suo codice univoco.

- Per semplificare la rimozione o l'aggiunta (se già presente nel sistema) di un prodotto al catalogo dei prodotti disponibili alla vendita è stato aggiunto l'attributo *disponibile*.
- Per i motivi riportati nei due punti precedenti sono stati aggiunti i metodi *makeProductAvailable* e *getProductByProdCode* nella classe ProductManager.
- E' stato aggiunto il metodo *getAllProducts* nella classe ProductManager per recuperare tutti i prodotti presenti nel sistema in ottica dello svolgimento del ruolo del ProductManager. Tale metodo si interfaccia con la classe ProductDAO.
- Nella classe Cart è stato rimosso il metodo *updateQuantity*. E' stato aggiunto, di conseguenza, il metodo *decreaseQuantity* che consente di diminuire la quantità di un prodotto nel carrello. Delll'incremento della quantità, invece, se ne occupa il metodo *addToCart* il quale, oltre all'aggiunta di un prodotto (se non presente), si occupa di incrementarne la quantità qualora esso fosse già presente nel carrello.
- Per facilitare la creazione di un'istanza di un CustomerOrder è stato implementato il metodo *makeCustomerOrder*.
- E' stata trasformata la classe associativa *ItemCart* della relazione *molti a molti* tra Cart e Product in un oggetto che ha due relazioni binarie: una con Cart e una con Product.
- E' stata trasformata la classe associativa *OrderLine* della relazione molti a molti tra Order (si fa riferimento al nome del Class Diagram non ristrutturato) e Product in un oggetto che ha due relazioni binarie: una con CustomerOrder e una con Product.