**WoodLot**

**Object Design Document**

**0.2**

****

Data: 27/12/2022

|  |  |
| --- | --- |
| **Coordinatore del progetto:** | |
| **Nome e Cognome:** | **Matricola:** |
| Alessia Ture | 0512107758 |

|  |  |
| --- | --- |
| **Partecipanti:** | |
| **Nome e Cognome:** | **Matricola:** |
| Raffaella Spagnuolo | 0512107794 |
| Salvatore Di Sanza | 0512106066 |

|  |  |
| --- | --- |
| **Scritto da:** | Alessia Ture |

|  |  |  |  |
| --- | --- | --- | --- |
| **Revisione History** | | | |
| **Data** | **Versione** | **Descrizione** | **Autore** |
| 23/12/22 | 0.1 | Stesura capitolo 1 | AT |
| 27/12/22 | 0.2 | Stesura capitolo 2 | AT |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Sommario

[1. Introduzione 4](#_Toc123017303)

[**1.1** **Object design goals** 4](#_Toc123017304)

[**1.2** **Object design trade-offs** 4](#_Toc123017305)

[**1.3** **Linee guida per la documentazione dell'interfaccia** 5](#_Toc123017306)

[**1.3.1** **Naming Convention** 5](#_Toc123017307)

[**1.3.2** **Nomenclatura delle componenti** 5](#_Toc123017308)

[**1.3.3** **Organizzazione dei componenti Angular** 5](#_Toc123017309)

[**1.3.4** **Variabili** 6](#_Toc123017310)

[**1.3.5** **Classi** 6](#_Toc123017311)

[**1.3.6** **Metodi** 6](#_Toc123017312)

[**1.3.7** **Organizzazione delle componenti** 6](#_Toc123017313)

[**1.3.8** **Organizzazione del codice** 6](#_Toc123017314)

[**1.4** **Design Pattern** 7](#_Toc123017315)

[**1.4.1** **Service Layer** 7](#_Toc123017316)

[**1.4.2** **Repository** 7](#_Toc123017317)

[**1.4.3** **Dependency injection** 7](#_Toc123017318)

[**1.5** **Componenti off-the shelf** 8](#_Toc123017319)

[**1.6** **Definizioni, acronimi e abbreviazioni** 8](#_Toc123017320)

[**1.6.1** **Definizioni** 8](#_Toc123017321)

[**1.6.2** **Acronimi** 9](#_Toc123017322)

[**1.6.3** **Abbreviazioni** 9](#_Toc123017323)

[**1.7** **Riferimenti** 9](#_Toc123017324)

[2. Packages 9](#_Toc123017325)

[**2.1** **Package WoodLot** 10](#_Toc123017326)

[3. Class Interfaces 11](#_Toc123017327)

[**3.1** **Gestione Ospite** 11](#_Toc123017328)

[**3.2** **Gestione Utente Registrato** 11](#_Toc123017329)

[**3.3** **Gestione Contadino** 12](#_Toc123017330)

[**3.4** **Gestione Catalogo** 12](#_Toc123017331)

[**3.5** **Gestione Ordine** 13](#_Toc123017332)

[**3.6** **Gestione Carrello** 13](#_Toc123017333)

[4. Glossario 15](#_Toc123017334)

# Introduzione

Dopo la realizzazione del Requirement Analysis Document e System Design Document, abbiamo descritto in linea di massima quello che sarà il nostro sistema, tralasciando gli aspetti implementativi.

Il seguente documento si pone come obiettivo quello di descrivere le linee guida in fase di implementazione del sistema WoodLot. In questo documento verranno descritti i trade-off generali e le convenzioni per la nomenclatura ed implementazione di classi ed interfacce.

## **Object design goals**

**Modularità**

Il sistema WoodLot deve basarsi su alta coesione e basso accoppiamento tra le sue componenti, inoltre si farà largo uso dei concetti di ereditarietà e i design pattern

**Robustezza**

Il sistema dovrà garantire robustezza reagendo a situazioni impreviste usufruendo di opportuni controlli degli errori e gestione delle eccezioni.

**Incapsulamento**

Il sistema garantisce l’occultamento dei dettagli implementativi delle classi grazie alle interfacce in modo fa rendere possibile l’utilizzo di funzionalità offerte da diversi componenti o layer sottoforma di blackbox.

**Manutenibilità**

Il sistema dovrà garantire un elevato grado di leggibilità e comprensibilità in modo da garantire interventi di manutenzione semplificati, tramite lo standard Sun.

## **Object design trade-offs**

**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 di sviluppo.

**Performance VS Usabilità**

L’interfaccia permette un uso facilitato del sistema per tutti i tipi di utenti della piattaforma. È stato preferito l’usabilità per rendere l’esperienza il più agevole possibile rendendo l’interfaccia chiara e pulita. Il sistema deve garantire tempo di riposta brevi per ogni funzionalità

**Sicurezza VS Efficienza**

Dati i tempi di sviluppo piuttosto ristretti per lo sviluppo, ci limiteremo ad implementare sistemi di sicurezza basati su username e password crittografata, garantendo un controllo sugli accessi basato su determinati ruoli.

## **Linee guida per la documentazione dell'interfaccia**

È richiesto agli sviluppatori di seguire le seguenti linee guida al fine di essere consistenti all’interno del progetto e facilitare la comprensione delle funzionalità di ogni componente.

### **Naming Convention**

È buona norma utilizzare nomi

* Descrittivi
* Di uso comune
* Di lunghezza medio-corta
* Non abbreviati

### **Nomenclatura delle componenti**

* Ogni nome di un file/componente dovrà avere un nome significativo
* Ogni componente dovrà terminare con il nome.“estensione” che rappresenta il tipo del file
* Ogni file dovrà avere un nome auto esplicativo in modo da consentire una facile individuazione della componente anche se generale
* Ogni metodo e ogni file devono essere preceduti da un commento che riporti l’obiettivo, il nome dell’autore/i e una giustificazione delle decisioni particolari o dei calcoli

### **Organizzazione dei componenti Angular**

* Utilizzare kebab-case per nominare i selettori dei componenti,
* I componenti devono avere un nome composto formato da una parte descrittiva e dal suffisso: .component.ts
* Le direttive devono avere un nome composto formato da una parte descrittiva e dal suffisso: directive.ts
* Le pipe devono avere un nome composto formato da una parte descrittiva e dal suffisso: pipe.ts
* Gli interceptors devono avere un nome composto formato da una parte descrittiva e dal suffisso: intercpetor.ts
* I moduli devono avere un nome composto formato da una parte descrittiva e dal suffisso: .modules.ts
* I moduli di routing devono avere un nome composto formato da una parte descrittiva e dal suffisso: -routing.module.ts;
* La parte del nome file che precede il suffisso dovrebbe essere il più descrittiva possibile per facilitare la rapida ricerca dei nomi dei file.
* In genere i componenti hanno 3 file: file .ts, .scss/css e .html, tali file devono essere messi in una cartella che corrisponda al nome del componente
* Spostare la logica riutilizzabile nei servizi e mantenere i componenti semplici e concentrati sullo scopo previsto.
* Situare le proprietà in alto seguite da metodi, porre i membri privati dopo i membri pubblici, in ordine alfabetico.

### **Variabili**

* Ogni variabile deve avere nome in loverCamelCase
* Inizializzare le variabili locali nel punto in cui sono state dichiarate a meno che il suo valore iniziale non dipenda da un calcolo che occorre eseguire prima.
* In determinati casi, è possibile utilizzare il carattere underscore “\_” ad esempio quando si fa uso di variabili costanti
* I nomi di costanti dovranno essere definiti in maiuscolo esempio: MAX\_NUMBER

### **Classi**

* Ogni classe deve avere un nome in CamelCase
* Ogni classe deve avere un nome singolare
* Ogni classe che modella un’entità deve avere per nome un sostantivo che possa associarla alla corrispondente entità del dominio
* Ogni classe che modella la logica di business per una determinata entità deve avere un nome composto da quello dell’entità su cui opera seguito da “Service”
* Ogni classe che modella una collezione in memoria per una determinata classe di entità deve avere nome composto da quello dell’entità su cui opera seguito da “Repository”
* Ogni classe che realizza un form deve avere nome composto dal sostantivo che descrive il form seguito dal suffisso “Form”
* Per avere una documentazione chiara e completa, ogni classe deve avere un commento Javadoc in cui viene data una breve descrizione della classe e viene definito l’autore della classe.

### **Metodi**

* Ogni metodo deve avere nome in loweCamelCase
* Il nome del metodo è costituito da un verbo che identifica un’azione, seguito dal nome di un oggetto. I nomi dei metodi per l’accesso e la modifica delle variabili devono essere del tipo getNomeVariabile() e setNomeVariabile()
* Ai metodi va aggiunta una descrizione tramite JavaDoc, la quale deve essere posizionata prima della dichiarazione del metodo e deve descriverne lo scopo. La descrizione del metodo deve includere anche informazioni riguardanti gli argomenti, il valore di ritorno e le eccezioni.

### **Organizzazione delle componenti**

* Tutte le classi che realizzano un sottosistema devono essere racchiuse nello stesso pacchetto Java

### **Organizzazione del codice**

* Il codice Java deve essere indentato in maniera adeguata (tramite tabulazione corrispondente a due spazi) e l’apertura del blocco di codice deve avvenire nella stessa riga in cui è specificato il nome della classe o la firma di quel metodo che quel blocco definisce.
* Ogni classe dovrà avere una documentazione che ne espliciti le funzionalità
* Il codice HTML deve essere indentato in maniera adeguata (tramite tabulazione corrispondente a due spazi) e gli attributi devono essere indicati in minuscolo
* Il codice deve rispettare la formattazione standard di scrittura: spazio dopo la virgola, spazio prima e dopo ogni carattere matematico o di confronto, lasciare una linea vuota dopo la dichiarazione di una condizione, di ciclo o di una funzione.
* Quando un’espressione supera la lunghezza della linea, occorre spezzarla secondo i seguenti principi: interrompere la linea dopo una virgola, interrompere la linea prima di un operatore, allineare la nuova linea con l’inizio dell’espressione nella linea precedente.
* Se le regole precedenti rendono il codice più confuso o il codice è troppo spostato verso il margine destro, utilizzare solo otto spazi di indentazione.
* I valori immutabili definiti in classi Java devono essere definiti come “static final”
* Ogni messaggio d’errore che verrà stampato a video dovrà riportare un messaggio che ne identifichi con chiarezza il tipo di errore e la provenienza.
* Le annotazioni previste per una classe o per un metodo dovranno apparire una per riga, subito dopo il blocco di documentazione.
* Ogni eccezione deve avere nome esplicativo del problema segnalato

## **Design Pattern**

### **Service Layer**

Tipicamente le applicazioni richiedono differenti tipi di interfacce ai dati che salvano e la loro logica che implementano. Anche se lavorano su dati differenti le interfacce spesso richiedono interazioni comuni con l’applicazione per accedere, manipolare i dati che invocano e la logica di business. Il service layer è un’astrazione sopra il dominio logico. Definisce i confini di un’applicazione con livello di servizi che stabilisce un insieme di operazioni e coordina le risposte di ogni operazione. In Spring Boot il service Layer definisce quali funzionalità vengono fornite, come si accede, cosa passare e cosa ottenere in cambio indipendentemente da qualsiasi porta e dal funzionamento interno. È molto utile per i servizi web, code di messaggi ed eventi programmati. L’uso di questo pattern ha come conseguenza: maggior riusabilità del codice, minore accoppiamento tra i livelli del controller e il dominio degli oggetti.

### **Repository**

Repository ci permette di creare un’interfaccia con le firme di metodi per la richiesta di dati persistenti che risiedono o su un database o su un file statico.

Oltre all’interfaccia ci permette di creare una classe che implementa i metodi definiti che si occupano della gestione vera e propria. In SpringBoot, il pattern Repository può essere utilizzato per la definizione di interfacce, semplificando il lavoro nella creazione delle query.

Grazie all’applicazione di questo Design Pattern: riduzione della duplicazione delle query, riduzione accoppiamento tra le classi di business logic e la base dati e aumento della modularità del sistema per la persistenza.

### **Dependency injection**

Il design pattern dependency injection è utilizzato nel framework Angular per gestire le dipendenze fra componenti. Lo scopo di questo pattern è fare in modo che una classe riceva le dipendenze di cui ha bisogno da una sorgente esterna invece di istanziarle autonomamente.

Supponiamo di avere una certa classe A che vuole usare le funzionalità di un’altra classe B. Si dice che in questi casi che la classe A ha come dipendenza la classe B. La classe A potrebbe creare un’istanza della classe B all’interno del suo costruttore, ma così facendo un oggetto di tipo A sarebbe poco configurabile e flessibile. Per risolvere tale problema quando creiamo un’istanza della classe A, possiamo passare come argomento al suo costruttore un oggetto della classe B. Tale operazione può essere effettuata ogni volta manualmente passando l’istanza come argomento, oppure è possibile incaricare un terzo oggetto che prende il nome di Injector che si incarichi di risolvere le dipendenze.

Grazie a tale pattern è possibile semplificare la fase di sviluppo dell’applicazione rendendo più flessibili e facilmente testabili i diversi componenti.

## **Componenti off-the shelf**

Nella realizzazione della nostra piattaforma andremo ad utilizzare componenti off-the-shelf già disponibili per facilitare lo sviluppo del progetto.

* Per la progettazione del lato Backend verrà utilizzato Spring Boot e l’interazione con esso avverrà tramite linguaggio di programmazione Java.
* Per la progettazione del lato Frontend verrà utilizzato Angular, una libreria JavaScript. Inoltre, per la modellazione dello stile grafico si utilizzerà la libreria CoreUI (<https://coreui.io/angular/docs/getting-started/introduction>) e la libreria MDB Bootstrap (<https://mdbootstrap.com/docs/angular/>)
* I dati verranno memorizzati all’interno di un DMBS MySQL mediante JPA, Spring Boot utilizza di default la tecnologia Hibernate. La connessione al DBMS è garantita dal driver JDBC.

## **Definizioni, acronimi e abbreviazioni**

### **Definizioni**

* **Package**: raggruppamento di classi, interfacce o file correlati;
* **Design pattern**: template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e
* flessibilità;
* **Interfaccia**: insieme di signature delle operazioni offerte dalla classe;
* **Cammel case**: “notazione a cammello”, consiste nello scrivere parole composte unendo tutte le lettere ma lasciando le iniziali maiuscole.
* **LowerCammelCase**: convenzione per scrivere i nomi unendo tutte le lettere dove però la prima lettera inizia con una minuscola.
* **Kebab case**: è la pratica di scrivere gli identificatori separando le parole che li compongono tramite trattini (-)
* **Bootstrap**: è un framework open source che contiene una raccolta di strumenti liberi per la creazione di siti web ed applicazioni web
* **Javadoc**: è un applicativo utilizzato per generare automaticamente documentazione al fine di renderla facilmente accessibile e leggibile

### **Acronimi**

* **RAD:** Requirements Analysis Document
* **SDD:** System Design Document
* **ODD:** Object Design Document

### **Abbreviazioni**

* **DB**: Database

## **Riferimenti**

* Benrd Bruegge – Allen H. Dutoit, Object-Oriented Software Enginering: Using UML
* RAD WoodLot
* SDD WoodLot

# Packages

In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese e ricalca la struttura Client Server.

Il lato server ha una struttura di directory standard definite direttamente da Maven contenente:

* **.idea,** intelliJ root folder, contenete file di configurazione del progetto
* **.mvn**, contiene tutti i file di configurazione per Maven
* **src**,contiene tutti i file sorgente
  + **main** 
    - **java** contiene le classi Java relative alle componenti Control, Service e Entity
    - **resources** contiene i file relativi alla componente relativa all’agente intelligente
  + **test**,contiene tutto il necessario per il testing
    - java: contiene le classi Java per l’implementazione del testing
* **target** contiene tutti i file prodotti dal sistema di build di Maven

Il lato client ha una struttura di tipo standard Angualr

* **.angular**
* **.idea** intelliJ root folder, contenete file di configurazione del progetto
* **.vscode**
* **src** 
  + **app**, contenente tutti i file sorgenti
    - **componenti**, contiene tutte i componenti Angular comuni a più pagine
    - **entita** contenente tutte le entità
    - **modelli**
    - **pagine** contenente tutti i componenti Angular che realizzano le pagine
    - **servizi** contenente tutti i servizi dell’applicazione
  + **assets**
    - **img** contenente tutte le immagini, suddivise per cartelle in base alla pagina in cui dovranno essere inserite
  + **envorinmetns**

## **Package WoodLot**

Nella presente sezione di mostra la struttura del package principale di WoodLot

La struttura generale è stata ottenuta a partire da tre principali scelte:

1. Creare un package per la logica di applicazione chiamato “Application Logic. All’interno di quest’ultimo creare dei package per ogni sottosistema. I quali sono:
   1. Gestione Ospite: pacchetto che contiene classi service per la gestione dell’ospite
   2. Gestione Utente Registrato: pacchetto che contiene classi service per la gestione dell’utente registrato
   3. Gestione Utente: pacchetto che contiene classi service per la gestione dell’utente
   4. Gestione Contadino: pacchetto che contiene classi service per la gestione del contadino
   5. Gestione Catalogo: pacchetto che contiene classi service per la gestione del catalogo
   6. Gestione Carrello: pacchetto che contiene classi service per la gestione del carrello
2. Creare un package “Web” in cui inserire tutte le classi controller
3. Creare un package separato per le classi del model, contente le classi entity e i DAO per l’accesso al DB
4. Creare un package chiamato “Utils” in cui inserire eventuali classi di utilità per il sistema e usabili da più sottosistemi

Il livello della logica applicativa interagisce con:

* Il package model
* Il package Web
* Il package utils

Per ciò che concerne la dipendenza tra i packages, la suddivisione precedentemente illustrata ha portato alla creazione di una relazione tra il package model e tutti gli altri package del sistema.

Immagine che contiene screenshot

Descrizione generata automaticamente

# Class Interfaces

## **Gestione Ospite**

|  |  |
| --- | --- |
| Nome classe | OspiteService |
| Descrizione | Questa classe permette di gestite le operazioni inerenti ad un ospite |
| Metodi | +registrazioneUtente (Utente utente):Utente |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | +registrazioneUtente (utente: Utente): Utente |
| Descrizione | Questo metodo consente di registrare un utente alla piattaforma |
| Pre-condizione | **context** OspiteSercice :: registrazioneUtente (utente: Utente)  **pre:** utente != null |
| Post-condizione | **context** OspiteSercice :: registrazioneUtente (utente: Utente)  **post:** utente != null |
| Eccezioni |  |

## **Gestione Utente Registrato**

|  |  |
| --- | --- |
| Nome classe | UtenteRegistratoService |
| Descrizione | Questa classe permette di gestite le operazioni inerenti ad un utente registrato |
| Metodi | +login (email: String, password: String): Utente  +logout (utente: Utente)  + reimpostarePassword (email: String, password: String) |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | +login (email: String, password: String) : Utente |
| Descrizione | Questo metodo consente di autenticare un utente registrato |
| Pre-condizione | **context** UtenteRegistratoService::login(email: String, password: String)  **pre**: email != null && password != null |
| Post-condizione | **context** UtenteRegistratoService::login(email: String, password: String)  **post**: utente != null && utente instaceof Utente || utente instaceof Contadino || utente instaceof ResponsabileCatalogo |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | +logout (utente: Utente ) |
| Descrizione | Questo metodo consente di effettuare il logout dalla piattaforma |
| Pre-condizione | **context** UtenteRegistratoSercice :: logout (utente: Utente)  **pre:** utente != null |
| Post-condizione | / |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | +reimpostarePassword (email: String, password: String) |
| Descrizione | Permette di reimpostare la password |
| Pre-condizione | **context** UtenteRegistatoService :: reimpostarePassword (email: String, password: String)  **pre:** email != null && password != null |
| Post-condizione | **context** UtenteRegistatoService :: reimpostarePassword (email: String, password: String)  **post:** |
| Eccezioni |  |

## **Gestione Contadino**

|  |  |
| --- | --- |
| Nome classe | ContadinoService |
| Descrizione | Questa classe permette di gestite le operazioni inerenti al contadino |
| Metodi | +aggiornareStato() |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | +aggiornareStato() |
| Descrizione | Permette di aggiornare lo stato di un albero. |
| Pre-condizione |  |
| Post-condizione |  |
| Eccezioni |  |

## **Gestione Catalogo**

|  |  |
| --- | --- |
| Nome classe | Catalogo Service |
| Descrizione | Questa classe permette di gestite le operazioni inerenti al catalogo |
| Metodi | +aggiungereProdotto(prodotto: Prodotto)  +eliminareProdotto(prodotto: Prodotto)  +filtrareCatalogo (parametro: String) |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | + aggiungereProdotto(prodotto: Prodotto) |
| Descrizione | Permette di aggiungere un prodotto al catalogo |
| Pre-condizione | **context** CatalogoService :: aggiungereProdotto(prodotto: Prodotto)  **pre:** prodotto != null |
| Post-condizione | **context** CatalogoService :: aggiungereProdotto(prodotto: Prodotto)  **post:** |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + eliminareProdotto(prodotto: Prodotto) |
| Descrizione | Permette di eliminare un prodotto dal catalogo |
| Pre-condizione | **context** CatalogoService :: eliminareProdotto(prodotto: Prodotto)  **pre:** prodotto != null |
| Post-condizione | **context** CatalogoService :: eliminareProdotto(prodotto: Prodotto) |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + filtrareCatalogo (parametro: String) |
| Descrizione |  |
| Pre-condizione | **context** CatalogoService :: filtrareCatalogo (parametro: String)  **pre:** parametro != null |
| Post-condizione | **context** CatalogoService :: filtrareCatalogo (parametro: String)  **post:** |
| Eccezioni |  |

## **Gestione Ordine**

|  |  |
| --- | --- |
| Nome classe | OrdineService |
| Descrizione | Questa classe permette di gestite le operazioni inerenti all’ordine |
| Metodi | +effettuareOrdine(ordine: Ordine) |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | + effettuareOrdine(ordine: Ordine) |
| Descrizione | Permette di creare un nuovo ordine |
| Pre-condizione |  |
| Post-condizione |  |
| Eccezioni |  |

## **Gestione Carrello**

|  |  |
| --- | --- |
| Nome classe | CarrelloService |
| Descrizione | Questa classe permette di gestite le operazioni inerenti al carrello |
| Metodi | +aggiungereProdotto(prodotto:Prodotto)  +modificareQuantitaProdotto(prdotto:Prodotto, int quantita)  +eliminareProdotto(prodotto: Prodotto)  +eVuoto() : boolean  +controlloPresenza(prodotto: Prodotto) : boolean  +svuota() |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome Metodo | + aggiungereProdotto(prodotto:Prodotto) |
| Descrizione | Permette di aggiungere un prodotto al carrello |
| Pre-condizione | **context** CarrelloService :: aggiungereProdotto(prodotto: Prodotto )  **pre:** prodotto != null |
| Post-condizione | **context** CarrelloService :: aggiungereProdotto(prodotto: Prodotto )  **post:** self.prodotti -> include (p| p.getCodice() = prodotto.getCodice()) |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + modificareQuantitaProdotto(prodotto: Prodotto, quantita : int) |
| Descrizione | Permette di modificare la quantità di un prodotto presente nel carrello |
| Pre-condizione | **context** CarrelloService :: modificareQuantitàProdotto(prodotto: Prodotto, quantita : int)  **pre**: prodotto != null && quantita >0 && (prodotti -> include p| p.getCodice() = prod.getCodice()) |
| Post-condizione | **context** CarrelloService :: modificareQuantitàProdotto(prodotto: Prodotto, int quantita)  **post:** self.prodotti-> include(p|p.getCodice() = prodotto.getCodice() && p.getQuantita() = quantita) |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + eliminareProdotto(prodotto:Prodotto) |
| Descrizione | Permette di eliminare un prodotto dal carrello |
| Pre-condizione | **context** CarrelloService :: eliminareProdotto(prodotto:Prodotto )  **pre:** prodotto != null && prodotti -> include(p|p.getCodice() = prodotto.getCodice()) |
| Post-condizione | **context** CarrelloService :: eliminareProdotto(prodotto:Prodotto)  **post:** prodotti -> !include (p| p.getCodice() = prod. getCodice()) |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + eVuoto() |
| Descrizione | Permette di valutare se il carrello è vuoto |
| Pre-condizione | **/** |
| Post-condizione | **context** CarrelloService :: eVuoto()  **post:** true if prodotto -> size() = 0 else false |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + controlloPresenzaProdotto(prodotto : Prodotto) |
| Descrizione | Permette di valutare se un prodotto è presente nel carrello |
| Pre-condizione | **context**  CarrelloService :: controlloPresenzaProdotto(prodotto:Prodotto)  **pre:** prodotto != null |
| Post-condizione | **context** CarrelloService :: controlloPresenzaProdotto(prodotto:Prodotto)  **post:**  true if prodotti -> include (p| p.getCodice() = prod. getCodice()) else false |
| Eccezioni |  |

|  |  |
| --- | --- |
| Nome Metodo | + svuota() |
| Descrizione | Permette di eliminare tutti i prodotti presenti nel carrello |
| Pre-condizione | **context** CarrelloService :: svuota()  **pre:** prodotti -> size() > 0 |
| Post-condizione | **context** CarrelloService :: svuota()  **post:** prodotti -> size () = 0 |
| Eccezioni |  |

# Glossario