RISTORANTE

PROGETTO INGEGNERIA DEL SOFTWARE — A.A. 2022-23

MATTEO CAMPANELLA, MARTINA CONTESTABILE, ANTONELLO CONSTANTIN RACSAN

INDICE PRESENTAZIONE

PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

LA CHIAVE PER L'ELEGANZA DEL DESIGN: IL CAMMINO VERSO LA MODULARITÀ E L'EFFICIENZA

PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

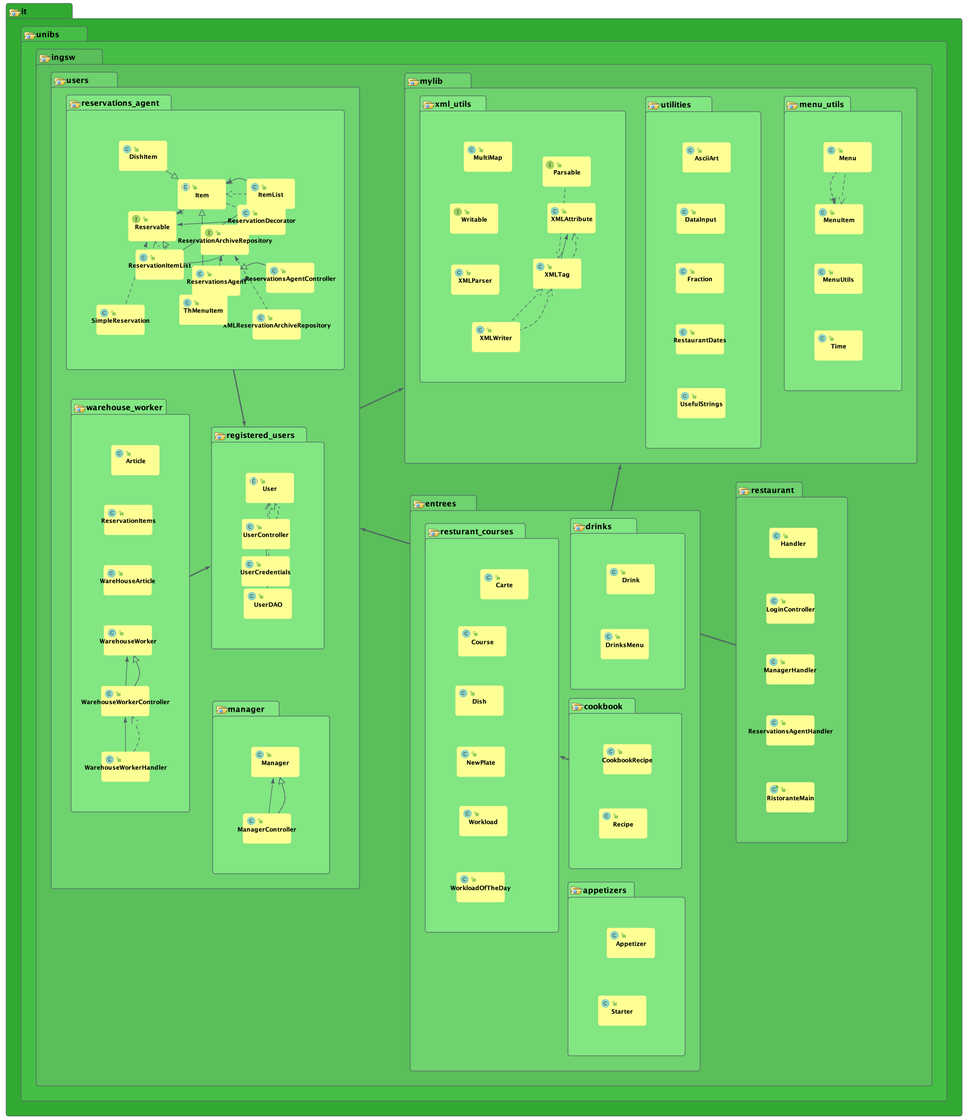
PATTERN APPLICATO AL PROGETTO

Package user, con sottopackage dedicati agli utenti

Package entrees, con sottopackage dedicati ai singoli elementi facenti parte del pasto

Package restaurant, contenente le classi vista e controllo del ristorante

Package mylib, con sottopackage dedicati a xml, utilità e costruzione menu di dialogo

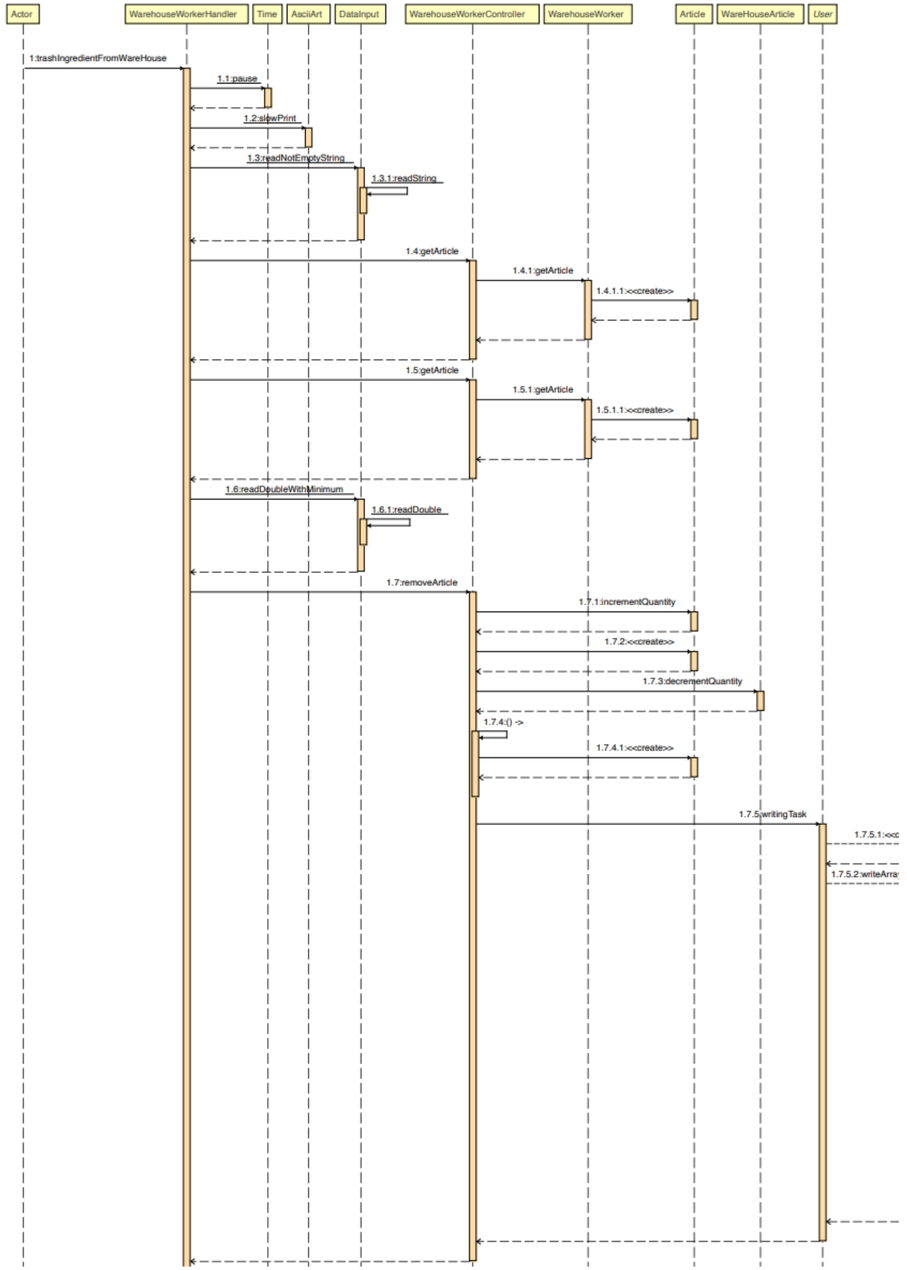


PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

PATTERN APPLICATO AL PROGETTO

Esempio di come avviene la comunicazione fra le classi definite nel programma

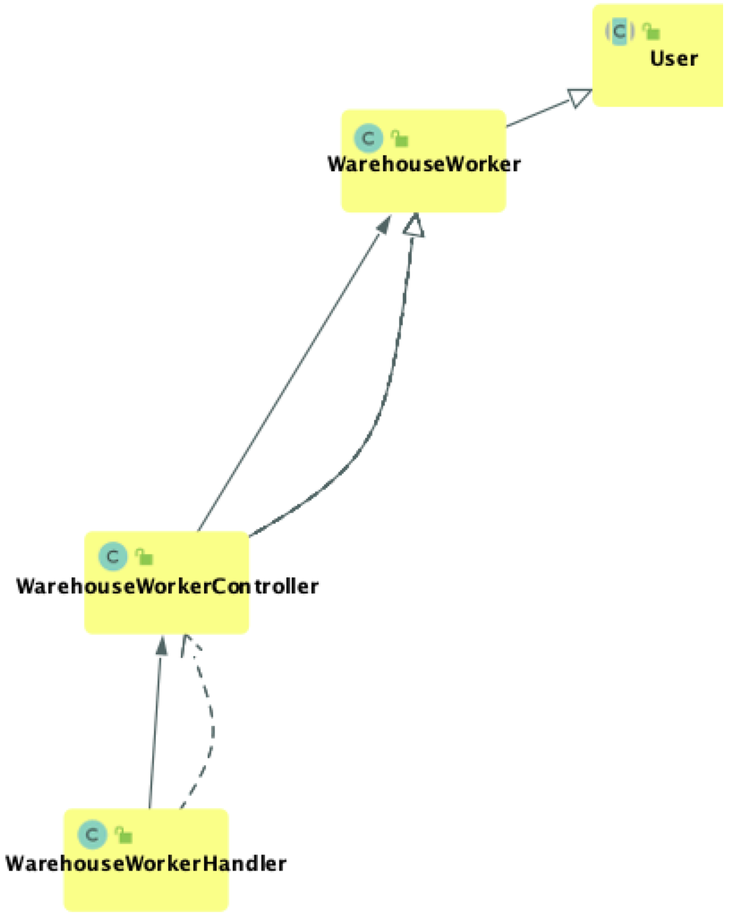
La comunicazione inizia nella view, che comunica gli input al controller, che a sua volta invoca il model, il quale eventualmente lavora con altri oggetti della logica di business



PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

PATTERN APPLICATO AL PROGETTO

Le dipendenze vanno dalla strato di presentazione allo strato di business



PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

PATTERN APPLICATO AL PROGETTO

Sono controller MVC

MANAGERCONTROLLER

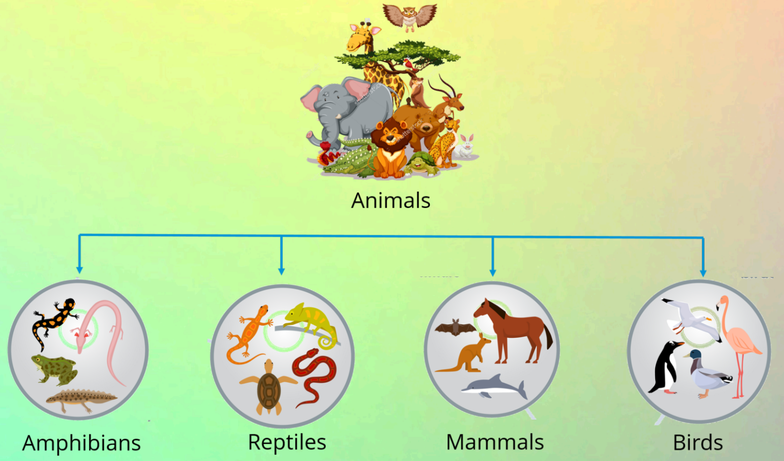
RESERVATIONAGENTRCONTROLLER

WAREHOUSEWORKERCONTROLLER

POLYMORPHISM

Super Class

Child Class



UN'ESPLOSIONE CREATIVA DI FORME E POSSIBILITÀ: IL MANTRA DELL'ADATTABILITÀ ILLIMITATA

POLYMORPHISM GRASP

PATTERN APPLICATO AGLI OGGETTI DI PARSING E WRITING

Definisce metodi per gestire l'analisi e l'elaborazione di tag e attributi da un file XML

Codice generico che può essere applicato a oggetti di diverse classi di parsing e writing

```java
public interface Parsable {
    HashMap<String, Consumer<String>> setters = new HashMap<>();

    default void setAttribute(@NotNull XMLTag xmlTag) {
        setSetters();
        Consumer<String> method = setters.get(xmlTag.getTagName());
        if (method != null) method.accept(xmlTag.getTagValue());
        XMLAttribute[] attributes = xmlTag.getAttributes();
        if (attributes != null) {
            for (XMLAttribute a : attributes) {
                method = setters.get(a.getName());
                if (method != null) method.accept(a.getValue());
            }
        }
    }

    default boolean containsAttribute(String tag) {
        return setters.get(tag) != null;
    }

    void setSetters();

    /**
     * Ritorna, sotto formato di stringa, il valore del primo tag.
     */
    String getStartString();
}
```

POLYMORPHISM GRASP

PATTERN APPLICATO AGLI OGGETTI DI PARSING E WRITING

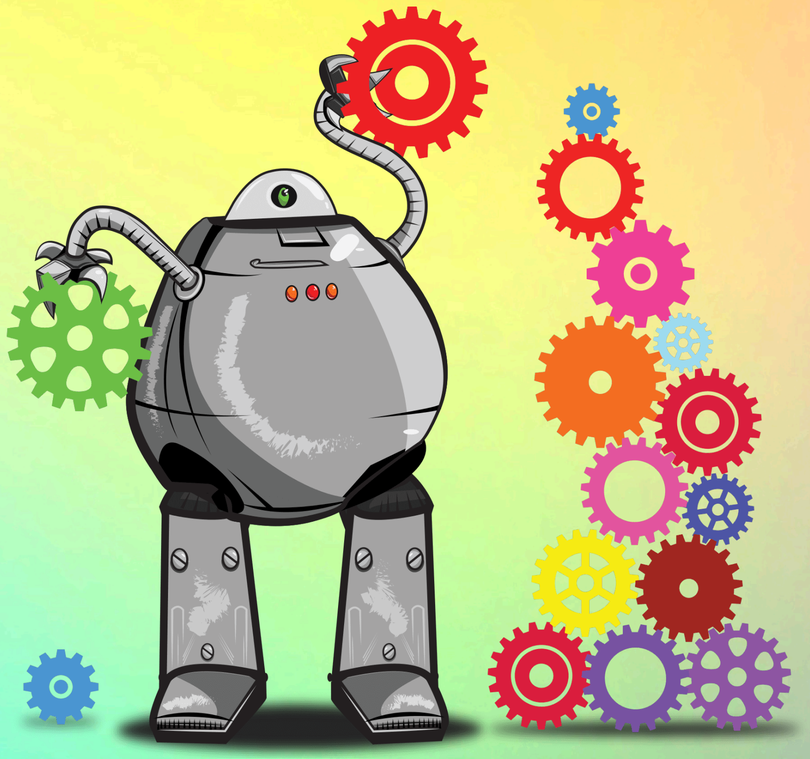
Information Expert **Responsabile di gestire l'analisi dei tag e degli attributi del file XML. Sa come assegnare i valori agli attributi corrispondenti negli oggetti che implementano l'interfaccia**

Low Coupling **Maggiore modularità e facilita eventuali modifiche o estensioni future**

High Cohesion **Raggruppare in modo coerente le responsabilità correlate migliorando la coesione all'interno del sistema**

```java
public class Appetizer implements Parsable {
    private String genre;
    private Double quantity;
    public static final String START_STRING = "starter";
    private static final List<String> ATTRIBUTE_STRINGS = new ArrayList<>();

    static {
        ATTRIBUTE_STRINGS.add("genre");
        ATTRIBUTE_STRINGS.add("quantity");
    }

    @Override
    public void setSetters() {
        setters.put(ATTRIBUTE_STRINGS.get(0), this::setGenre);
        setters.put(ATTRIBUTE_STRINGS.get(1), this::setQuantity);
    }

    @Override
    public String getStartString() {
        return START_STRING;
    }

    public void setGenre(String genre) {this.genre = genre;}

    public void setQuantity(String quantity) {this.quantity = Double.parseDouble(quantity);}

    public String getGenre() {return genre;}

    public String getQuantity() {return Double.toString(quantity);}

    public double getQuantityDouble() {return quantity;}
}
```

CONTROLLER



GUIDA E GOVERNA CON MAESTRIA: L'ARCHITETTO DEL CONTROLLO INTELLIGENTE CHE DÀ VITA ALL'ORDINE NELL'APPLICAZIONE

CONTROLLER GRASP

PATTERN APPLICATO ALLE CLASSI LOGINCONTROLLER E USERCONTROLLER

LoginController si occupa di coordinare l'autenticazione dell'utente

Riceve le credenziali dell'utente e interagisce con UserController per autenticare l'utente

```
public class LoginController {
    private UserController userController;

    public LoginController(UserController userController) {
        this.userController = userController;
    }

    public User authenticateUser(String username, String password) {
        User user = userController.authenticateUser(username, password);
        return user;
    }
}
```

CONTROLLER GRASP

PATTERN APPLICATO ALLE CLASSI LOGINCONTROLLER E USERCONTROLLER

Low Coupling **Le classi dipendono il meno possibile dalle altre classi**

High Cohesion **Le responsabilità all'interno delle classi sono ben definite e correlate tra loro**

Prima del refactoring **Handler aveva la responsabilità di autenticare l'utente e di interagire direttamente con il Model per ottenere le credenziali degli utenti**

Dopo il refactoring **Responsabilità riassegnata a UserController, ogni classe ha una responsabilità specifica e ben definita**

OPEN-CLOSED PRINCIPLE

INNOVAZIONE SENZA LIMITI:LA CHIAVE CHE APRE LE PORTE DEL PROGRESSO, CONSENTENDO L'ESTENSIONE SENZA LA NECESSITÀ DI MODIFICHE, TRASFORMANDO IL CODICE IN UN TERRENO FERTILE PER NUOVE IDEE

OCP SOLID

PATTERN APPLICATO ALLE CLASSI DEGLI UTENTI NEL SISTEMA ATTRAVERSO USER

Contratto comune che stabilisce metodi e proprietà che ogni tipo di utente deve implementare.
I lavoratori estendono il comportamento di base dato da User e aggiungono le peculiarità che li contraddistinguono

Estensione per supportare nuovi tipi di utenti senza dover modificare il codice sorgente originale

OCP SOLID

PATTERN APPLICATO ALLE CLASSI DEGLI UTENTI NEL SISTEMA ATTRAVERSO USER

Apertura all'estensione **aggiungere nuovi tipi di utenti creando nuove classi che implementano User, senza cambiare quelli esistenti**

MODULARITÀ DEL SISTEMA

SEMPLIFICA L'AGGIUNTA

RIDUCE L'IMPATTO DELLE MODIFICHE

MANUTENIBILITÀ DEL SISTEMA

SINGLE RESPONSIBILITY PRINCIPLE

MANTENERE IL POTERE DELLA FOCALIZZAZIONE: ILLUMINA LA STRADA VERSO LA COESIONE E L'EFFICIENZA DEL CODICE

SRP SOLID

PATTERN APPLICATO AD HANDLER

Si concentra esclusivamente sulla gestione delle interazioni i rispettivi utenti

Codice generico che può essere applicato a oggetti di diverse classi di parsing e writing

SRP SOLID

PATTERN APPLICATO AD HANDLER

Prima del refactoring Handler aveva la responsabilità gestire l'interazione con tutti gli utenti

Dopo il refactoring suddivisione più chiara delle responsabilità all'interno del sistema

MAGGIORE COESIONE ***OGNI CLASSE SI CONCENTRA SUL PROPRIO COMPITO SPECIFICO***

FACILITÀ DI MANUTENZIONE ***BASE DI CODICE PIÙ PICCOLA E FOCALIZZATA, MODIFICHE CON MENO ERRORI***

RIUSO DEL CODICE ***CLASSI PROGETTATE MODULARMENTE, CONSENTENDO RIUSO DEL CODICE***

SCALABILITÀ ***NUOVE FUNZIONALITÀ AGGIUNGIBILI IN MODO PIÙ AGEVOLE, LE CLASSI SONO CONCENTRATE SUI LORO COMPITI E NON RICHIEDONO MODIFICHE ESTENSIVE***

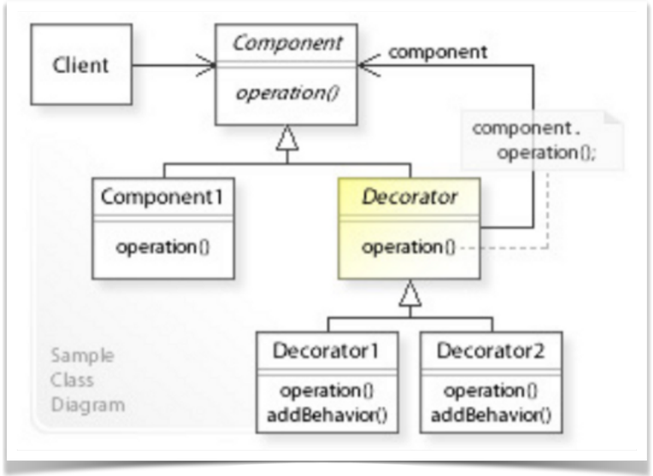
DECORATOR

SCATENA LA CREATIVITÀ DEL TUO CODICE: IL PENNELLO MAGICO CHE PERMETTE DI ADORNARE GLI OGGETTI CON INFINITI STRATI DI FUNZIONALITÀ, CREANDO OPERE D'ARTE SOFTWARE

GOF DECORATOR

PATTERN APPLICATO ALLE PRENOTAZIONI

- **Reservable**
  (interfaccia Component)
- **ReservationAgent**
  (Client)
- **SimpleReservation**
  (Component1)
- **ReservationDecorator**
  (Decorator)
- **ReservationItemList**
  (Decorator1)



GOF DECORATOR

PATTERN APPLICATO ALLE PRENOTAZIONI

Separare la logica di gestione di una prenotazione base (che si suppone non cambi spesso), dalla logica di una prenotazione completa di una lista di menu/piatti.

Modifiche alla gestione della lista di item senza andare ad intaccare una SimpleReservation base

```java
/**
 * Costruttore. La ReservationItemList sarà una prenotazione decorata
 * con una lista di item.
 *
 * @param decoratedReservation reservation da decorare.
 * @param itemList lista di item.
 */
5 usages  anton +1
public ReservationItemList(Reservable decoratedReservation, ItemList itemList) {
    super(decoratedReservation);
    this.itemList = itemList;
}
```

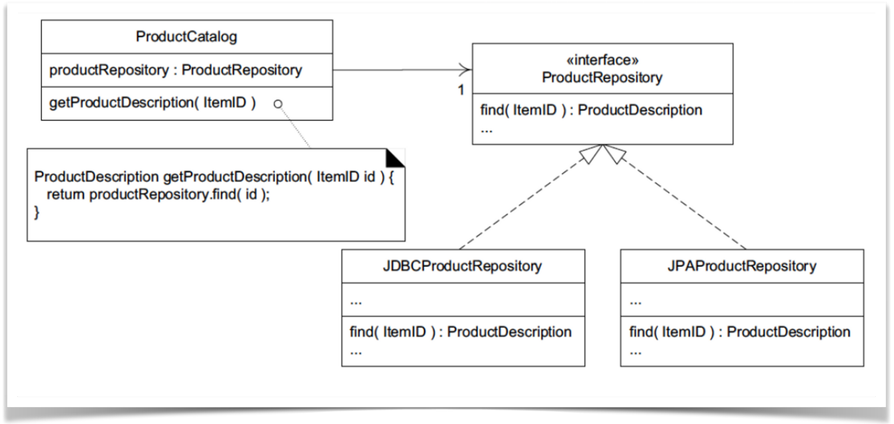
REPOSITORY

LA CASSAFORTE DEI DATI CHE RIVOLUZIONA L'ACCESSO: IL PORTALE VERSO L'ARCHIVIAZIONE SICURA E L'INTERROGAZIONE POTENTE DEI DATI, SBLOCCANDO IL POTENZIALE ILLIMITATO DELL'INFORMAZIONE

GOF REPOSITORY

SALVATAGGIO NELL'ARCHIVIO PRENOTAZIONI

- **ReservationAgent** *(ProductCatalog)*
- **ReservationArchiveRepository** *(ProductRepository)*
- **XMLReservationArchiveRepository** *(implementazione)*



NUOVA MODALITÀ DI SALVATAGGIO?

CREARE NUOVA CLASSE

```java
/**
 * Metodo che si occupa del salvataggio delle prenotazioni nell'apposito archivio.
 */
1 usage  anton
public void saveInReservationArchive(){
    agent.getReservationArchiveRepository().save(RestaurantDates.workingDay.format(RestaurantDates.formatter));
}
```

TESTING REQUISITO

SVELA IL POTERE DEI TUOI REQUISITI: IL FARO CHE ILLUMINA IL CAMMINO PER ASSICURARE L'ACCURATEZZA, L'ADERENZA E IL SUCCESSO DELLE TUE VISIONI PROGETTUALI

BOUNDARY VALUE ANALYSIS

TESTING REQUISITO: INSERIMENTO PRENOTAZIONE

Test su valori limite di coperti di una prenotazione

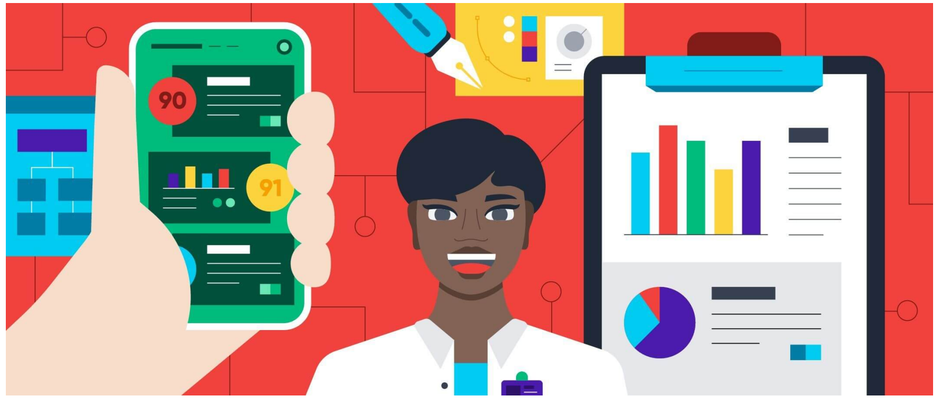
Test su valori limite di occorrenze di un item

Test superamento workload massimo raggiungibile

Test ristorante pieno

Test ripetitività nome prenotazione

Test ripetitività nome item



BOUNDARY VALUE ANALYSIS

TESTING REQUISITO: INSERIMENTO PRENOTAZIONE

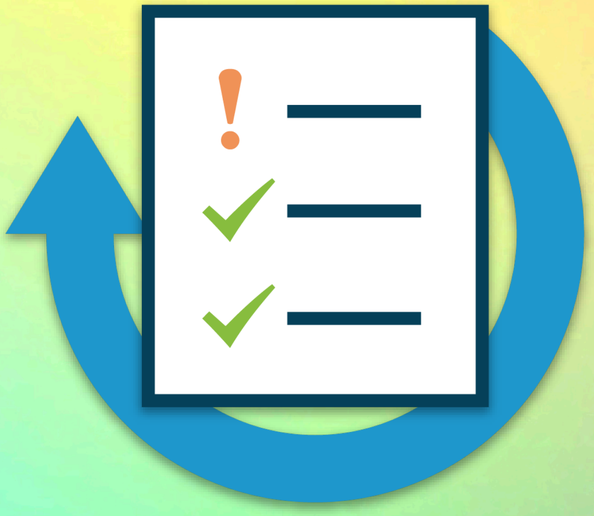
massimo + 1

```java
@Test
public void resCover_restaurantEmpty_maxWorkloadPlusOne(){
    int restaurantWorkload = 50;
    agent.setCopertiRaggiunti(0);
    int resCover = restaurantWorkload+1;

    Reservable res1 = controller.createReservationItemList(new SimpleReservation( name: "test", resCover), itemList);

    if(!controller.exceedsCover(resCover))
        controller.insertReservation(res1);

    assertThat(agent.getReservations().size(), is(equalTo( operand: 0)));
}
```

nome duplicato

```java
@Test
public void duplicatedReservationNameTest(){
    Reservable res1;
    Reservable res2;

    res1 = controller.createReservationItemList(new SimpleReservation( name: "test",  resCover: 4), itemList);
    controller.insertReservation(res1);

    res2 = controller.createReservationItemList(new SimpleReservation( name: "test",  resCover: 2), itemList);

    if(!controller.isAlreadyIn(res2.getName(), controller.getReservationNameList()))
        controller.insertReservation(res2);

    assertThat(agent.getReservations().size(), is(equalTo( operand: 1)));
}
```

TESTING METODO



RIVOLUZIONA IL CODICE CON UN'ARMA SEGRETA: IL MARTELLO CHE FORGIA L'ECCELLENZA, GARANTENDO LA SOLIDITÀ E L'AFFIDABILITÀ DELLE FUNZIONI

TESTING METODO

TEST SU CHECKMENUWORKLOAD DI MANAGERCONTROLLER

testCheckMenuWorkload() verifica che il metodo restituisca true quando il carico di lavoro del menu è compatibile con il carico di lavoro impostato per persona

testCheckIncorrectMenuWorkload() verifica che il metodo restituisca false quando il carico di lavoro del menu non è compatibile con il carico di lavoro impostato per persona

```
@Test
public void testCheckMenuWorkload() throws Exception {
    Manager manager = new Manager("gestore", "esempio", true);
    Queue<User> users = new LinkedList<>();
    users.add(new Manager("gestore", "esempio", true));

    UserController controller = new ManagerController(users, manager);

    Fraction menuWorkload = new Fraction(2, 3);

    manager.setWorkloadPerPerson(3);

    assertThat(((ManagerController) controller).checkMenuWorkload(menuWorkload), is(true));
}

@Test
public void testCheckIncorrectMenuWorkload() throws Exception {
    Manager manager = new Manager("gestore", "esempio", true);
    Queue<User> users = new LinkedList<>();
    users.add(new Manager("gestore", "esempio", true));

    UserController controller = new ManagerController(users, manager);

    Fraction menuWorkload = new Fraction(7, 3);

    manager.setWorkloadPerPerson(2);

    assertThat(((ManagerController) controller).checkMenuWorkload(menuWorkload), is(false));
}
```

POLYMORPHISM GRASP

PATTERN APPLICATO AGLI OGGETTI DI PARSING E WRITING

Information Expert **Responsabile di gestire l'analisi dei tag e degli attributi del file XML. Sa come assegnare i valori agli attributi corrispondenti negli oggetti che implementano l'interfaccia**

Low Coupling **Maggiore modularità e facilita eventuali modifiche o estensioni future**

High Cohesion **Raggruppare in modo coerente le responsabilità correlate migliorando la coesione all'interno del sistema**

REFACTORING

SCATENA LA POTENZA DELLA RINASCITA: L'ARTE CHE TRASFORMA IL CAOS IN ORDINE, RIVELANDO IL VERO POTENZIALE DEL CODICE E APRENDO LE PORTE ALLA MANUTENIBILITÀ E ALL'INNOVAZIONE

REFACTORING

REFACTORING DI HANDLER

Handler era la classe di interfacciamento fra utente e sistema

È stata spezzata in classi più piccole per essere leggibile, manutenibile e coesa

REFACTORING

REFACTORING DI HANDLER

Extract Class Creazione di ManagerHandler, ReservationAgentHandler e WareHouseWorkerHandler

Move Method Spostati i metodi dei relativi utenti nelle rispettive classi

Extract Method Spezzettato alcuni metodi dell'addetto alle prenotazioni