Ristorante parte B

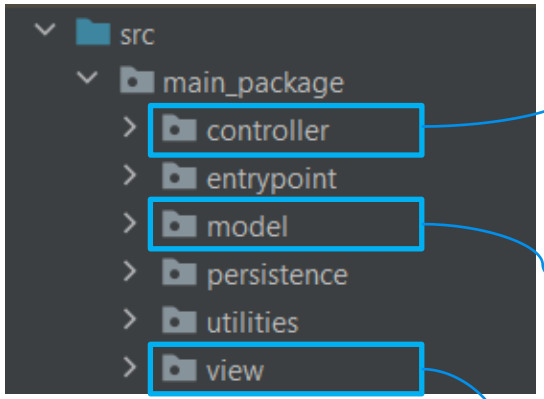
Alessandro Taboni, Matilde Simonini, Leonardo Sivieri

Indice

1.
Separazione Modello – Vista

1.

Separazione Modello – Vista



Package controller

Package contenente le classi che rappresentano i controller MVC dell'applicazione.

Package model

Package contenente le classi atte a modellare il dominio applicativo (ricetta, piatto, prenotazione…).

Package view

Package contenente le classi relative all'interfaccia utente.

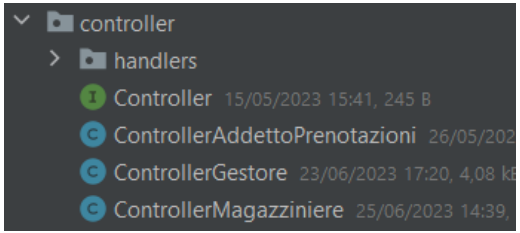
1.
Separazione Modello – Vista

```
public class ControllerGestore implements Controller{
    7 usages
    final private View view;
    7 usages
    private Session session;
    9 usages
    final private List<VoceMenu> vociMenu;
    11 usages
    private GestoreRepository gestoreRepository;

    1 usage
    public ControllerGestore(View view, GestoreRepository gestoreRe
```

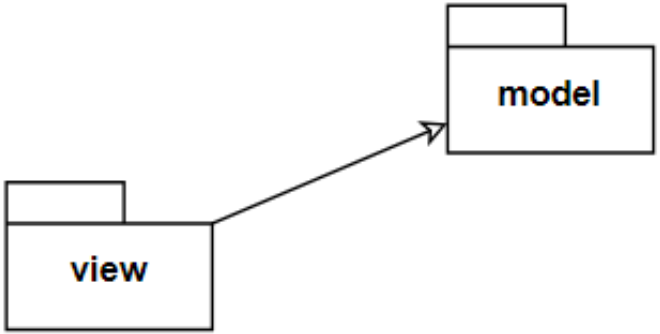
La classe *ControllerGestore* rappresenta un controller MVC che fa riferimento all'interfaccia *View* e delega le varie responsabilità ai vari sotto-controller MVC (quelli che risiedono nel package «handlers»)

```
private void setOption() {
    vociMenu.clear();
    vociMenu.add(new VoceMenu( nomeVoce: "Esci",  handler: null));
    if (session.getState().equals(State.UNLOGGED)) {
        vociMenu.add(new VoceMenu( nomeVoce: "Login", new LoginGestore(gestoreRepository.getUtentiRepository())));
    } else if (session.getState().equals(State.LOGGED)) {
        vociMenu.add(new VoceMenu( nomeVoce: "Logout", new Logout()));
        vociMenu.add(new VoceMenu( nomeVoce: "Inizializza Parametri", new InizializzaParametro(gestoreRepository)));
        vociMenu.add(new VoceMenu( nomeVoce: "Visualizza Parametri", new VisualizzaParametro(gestoreRepository)));
    }
}
```



1.

Separazione Modello – Vista



Le classi dello strato di Presentazione, che nel nostro caso risiedono nel package *view,* dipendono dalle classi di Business, quelle contenute nel package *model*.

2.
Pure Fabrication – GRASP

2.
Pure Fabrication – GRASP

```java
public class Ricetta {
    3 usages
    private String nome;
    7 usages
    private Map<String, Double> ingredienti;
    3 usages
    private int porzioni;
    3 usages
    private double caricoLavPerPorzione;

    Matilde Simonini
    public Ricetta(String nome, Map<String, Double> ingredienti, int porzioni, double caricoLavPerPorzione) {
        this.nome = nome;
        this.ingredienti = ingredienti;
        this.porzioni = porzioni;
        this.caricoLavPerPorzione = caricoLavPerPorzione;
    }
```

```java
public class RicetteRepository implements IRicetteRepository {

    9 usages
    private Connection connection;

    1 usage  Matilde Simonini
    public RicetteRepository(Connection connection) { this.connection = connection; }

    5 usages  Matilde Simonini
    @Override
    public void insertRicetta(Ricetta ricetta) {
        String query1 = "INSERT INTO Ricetta (NomeRicetta, Porzioni, CaricoLavoroPorzione) VALUES (?, ?, ?)";

        String query2 = """
                    INSERT INTO Prodotto (NomeProdotto, TipoProdotto, UnitaDiMisura, ConsumoProCapite)
                    SELECT ?, "ingrediente", "kg", -1
                    WHERE NOT EXISTS (
                      SELECT *
```

I frammenti di codice a sinistra sono un esempio dell'applicazione del pattern **GRASP Pure Fabrication**.

In questo caso, il principio Single Responsibility ci porta alla creazione di una classe pure fabrication *RicetteRepository* responsabile della gestione della **persistenza** relativa alla classe di dominio *Ricetta*.

Questa scelta, già operata nella parte A del progetto, ha portato verso una coesione più alta.

2.
Pure Fabrication – GRASP

```java
public class OperazioniSuMappe {
    2 usages  Matilde Simonini
    public static List<String> calcolaIntersezione(Map<Prodotto, Double> map1, Map<Prodotto, Double> map2) {
        List<String> intersection = new ArrayList<>();
        for (Prodotto p : map1.keySet()) {
            if(map2.containsKey(p)) {
                intersection.add(p.getNome());
            }
        }

        return intersection;
    }

    2 usages  Matilde Simonini
    public static Map<Prodotto, Double> calcolaDifferenza(Map<Prodotto, Double> map1, Map<Prodotto, Double> map2) {
        Map<Prodotto, Double> difference = new HashMap<>();
        for (Prodotto p : map1.keySet()) {
            if(!map2.containsKey(p)) {
                difference.put(p, map1.get(p));
            }
        }

        return difference;
    }
}
```

Pure Fabrication è stato applicato anche nella creazione della classe *OperazioniSuMappe* per trattare le *Map* come insiemi e applicare su di esse le tipiche operazioni insiemistiche di **intersezione** e **differenza**.

3.
Indirection – GRASP

3.
Indirection – GRASP

```java
public interface IRicetteRepository {
    5 usages  1 implementation  Matilde Simonini
    void insertRicetta(Ricetta ricetta);
    1 usage  1 implementation  Matilde Simonini
    List<Ricetta> getRicette();
    1 usage  1 implementation  Matilde Simonini
    Ricetta getRicettaFromName(String nome);
    2 usages  1 implementation  Matilde Simonini
    boolean isRicettaPresente(String nome);
    1 usage  1 implementation  Matilde Simonini
    double getCaricoDiLavoroDellaRicetta(String nomeRicetta);
    5 usages  1 implementation  Matilde Simonini
    void deleteRicetta(String nomeRicetta);
}
```

```java
public class RicetteRepository implements IRicetteRepository {

    9 usages
    private Connection connection;

    1 usage  Matilde Simonini
    public RicetteRepository(Connection connection) { this.connection = connection; }

    5 usages  Matilde Simonini
    @Override
    public void insertRicetta(Ricetta ricetta) {
        String query1 = "INSERT INTO Ricetta (NomeRicetta, Porzioni, CaricoLavoroPorzione) VALUES (?, ?, ?)";

        String query2 = """
                INSERT INTO Prodotto (NomeProdotto, TipoProdotto, UnitaDiMisura, ConsumoProCapite)
                SELECT ?, "ingrediente", "kg", -1
                WHERE NOT EXISTS (
                  SELECT *
```

I frammenti di codice a sinistra sono un esempio di applicazione del pattern **GRASP Indirection**.

L'interfaccia *IRicetteRepository* permette di evitare l'accoppiamento diretto tra l'applicazione e il sistema concreto di persistenza.

In questo caso la persistenza è realizzata tramite SQLite ma grazie a Indirection si potrebbe usare un qualunque altro meccanismo di persistenza senza modificare il resto dell'applicazione.

4.
Single Responsibility – SOLID

4.
Single Responsibility – SOLID

```java
public class InizializzaParametro extends GestioneVociMenu implements Handler{
    8 usages
    private GestoreRepository gestoreRepository;

    2 usages  Matilde Simonini
    public InizializzaParametro(GestoreRepository gestoreRepository){
        super();
        this.gestoreRepository = gestoreRepository;
    }
    Matilde Simonini
    @Override
    public Session execute(Session session, View view) {
        this.inizializzaParametro(session, view);
        return session;
    }

    1 usage  Matilde Simonini
    private void inizializzaParametro(Session session, View view){
        boolean exitFromInizializzaParametro = false;
        inizializzaMenu(session, view);
        do {
            exitFromInizializzaParametro = gestioneSceltaUtente(session, view);
        } while (!exitFromInizializzaParametro);
    }
```

4.
Single Responsibility – SOLID

```java
public class InitCaricoLavoroPerPersona implements Handler {
    2 usages
    private GestoreRepository gestoreRepository;

    2 usages  Matilde Simonini
    public InitCaricoLavoroPerPersona(GestoreRepository gestoreRepository){
        this.gestoreRepository = gestoreRepository;
    }
    Matilde Simonini
    @Override
    public Session execute(Session session, View view) {
        int carico = view.leggiInteroPositivo( s: "Inserisci nuovo carico di lavoro per persona >>  ");
        this.gestoreRepository.getParametriRistoranteRepository().setCaricoLavoroPerPersona(carico);
        return session;
    }
}
```

I frammenti di codice precedenti sono un esempio di applicazione del pattern **SOLID Single Responsibility**.

La classe *InizializzaParametro* delega le varie responsabilità a delle classi specifiche che hanno un solo compito da svolgere.

Nell'esempio la classe *InitCaricoLavoroPerPersona* è dedicata esclusivamente all'inizializzazione del parametro *carico di lavoro per persona*.

5.
Dependency Inversion – SOLID

5.
Dependency Inversion – SOLID

```
public interface View {
    1 implementation  Matilde Simonini
    void print(String str);
    2 usages  1 implementation  Matilde Simonini
    void printTabellaProdottoValore(Map<Prodotto, Double> tabella, String titoloTabella);
    1 usage  1 implementation  Matilde Simonini
    void printRicette(List<Ricetta> listaRicette);
    1 usage  1 implementation  Matilde Simonini
    void printNomiPiatti(List<String> listPiatti);
    no usages  1 implementation  Matilde Simonini
    void printPiatti(List<Piatto> listaPiatti);
    1 usage  1 implementation  Matilde Simonini
    void printMenuTematici(List<MenuTematico> listaMenuTematici);
```

```
public ControllerGestore(View view, GestoreRepository gestoreRepository) {
    super(view);
    this.gestoreRepository = gestoreRepository;
}
```

```
public Session execute(Session session, View view) {
    Map<String, String> corrispondenzePiattiRicetta = gestoreRepository.getPiattiRepository(
    view.printCorrispondenzaPiattoRicetta(corrispondenzePiattiRicetta);
    return session;
}
```

Un esempio di applicazione del pattern **SOLID Dependency Inversion** riguarda l'introduzione dell'interfaccia *View*.

Le classi che necessitano dei servizi di una view fanno riferimento all'interfaccia *View*.

6.
Template Method – GoF

6.
Template Method – GoF

```java
public final void run(String stringaUtente, String accessoNonAbilitato){
    if(isAccessoAbilitato()){
        corpoFissoMetodoRun(stringaUtente);
    } else {
        view.print(accessoNonAbilitato);
    }
    conclusioneVariabileMetodoRun();
}
```

```java
private void corpoFissoMetodoRun(String stringaUtente) {
    String titolo;
    boolean exit = false;
    do {
        impostazioneMenu();
        titolo = isUtenteLoggato(stringaUtente);
        view.setTitoloMenu(titolo);
        exit = gestioneSceltaUtente();
    } while (!exit);
}
```

```java
protected abstract boolean isAccessoAbilitato();
1 usage  3 implementations  atabonidev
protected abstract void conclusioneVariabileMetodoRun();
```

Il metodo *run* della classe astratta *ControllerBase* costituisce un'applicazione del pattern **Template Method** della **Gang of Four**.

Il metodo *run* è *final*, in modo tale che le classi figlie che lo ereditano non possano modificarne la struttura.

Il metodo *run* è costituito da alcune porzioni immutabili (*corpoFissoMetodoRun*) e da parti variabili (*isAccessoAbilitato*, *conclusioneVariabileMetodoRun*) che devono essere implementate nelle classi figlie.

7.
Repository – GoF

7.
Repository – Gof

```
public interface IPrenotazioniRepository {
    2 usages  1 implementation  Matilde Simonini
    void aggiungiPrenotazione(Prenotazione prenotazione);
    1 usage  1 implementation  Matilde Simonini
    void rimuoviPrenotazioniScadute();
    1 usage  1 implementation  Matilde Simonini
    List<Prenotazione> getPrenotazioniPerData(LocalDate data);
    2 usages  1 implementation  Matilde Simonini
    TabellaProdottoQuantita getIngredientiPrenotazioniPerData(LocalDate dataPrenotazione);
    2 usages  1 implementation  Matilde Simonini
    TabellaProdottoQuantita getBevandeServitePrenotazioniPerData(LocalDate now);
    2 usages  1 implementation  Matilde Simonini
    TabellaProdottoQuantita getGeneriExtraServitiPrenotazioniPerData(LocalDate now);
    1 usage  1 implementation  Matilde Simonini
    void clearTable();
}
```

```
public class PrenotazioniRepository implements IPrenotazioniRepository {

    11 usages
    private Connection connection;

    2 usages  Matilde Simonini
    public PrenotazioniRepository(Connection connection) { this.connection = connection; }

    2 usages  Matilde Simonini
    @Override
    public void aggiungiPrenotazione(Prenotazione prenotazione) {
        String query1 = "INSERT INTO Prenotazione (IdPrenotazione, NumeroCoperti, DataDiPreno
```

Per ogni entità dell'applicazione che rappresenta un dato persistente è associata un'interfaccia che espone i metodi di manipolazione dei dati (aggiunta, rimozione, recupero dal sistema di persistenza…).
Le classi *IPrenotazioniRepository* e *PrenotazioniRepository* costituiscono un esempio di utilizzo del pattern **Repository**.

Nell'applicazione la persistenza è realizzata con SQLite ma è possibile, grazie alla presenza delle interfacce come *IPrenotazioniRepository*, usufruire di altre implementazioni.

7.
Repository – Gof

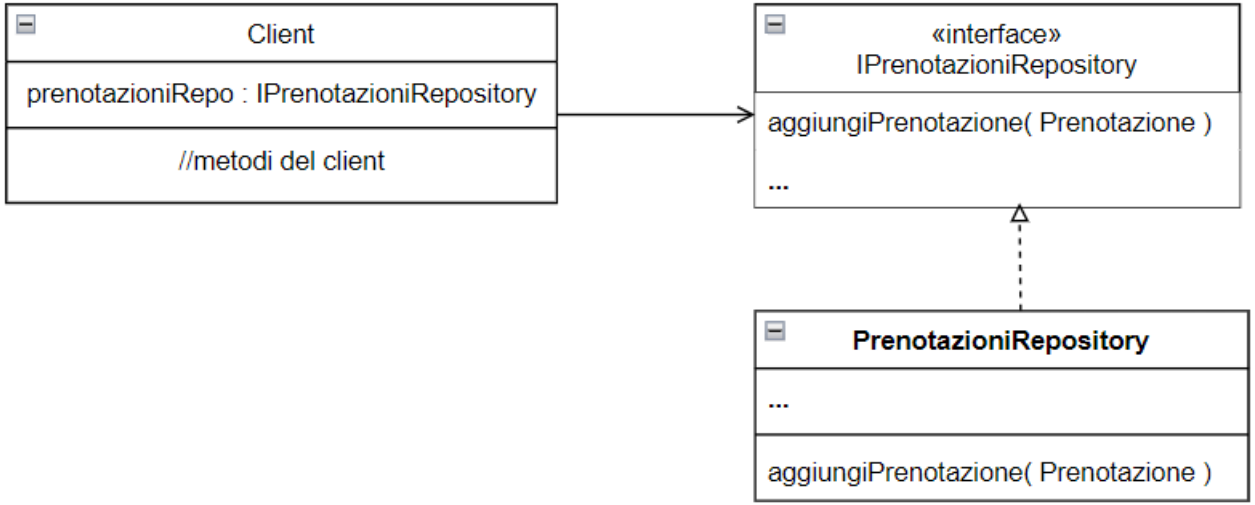


Diagramma delle classi per *PrenotazioniRepository*

7.
Repository – Gof

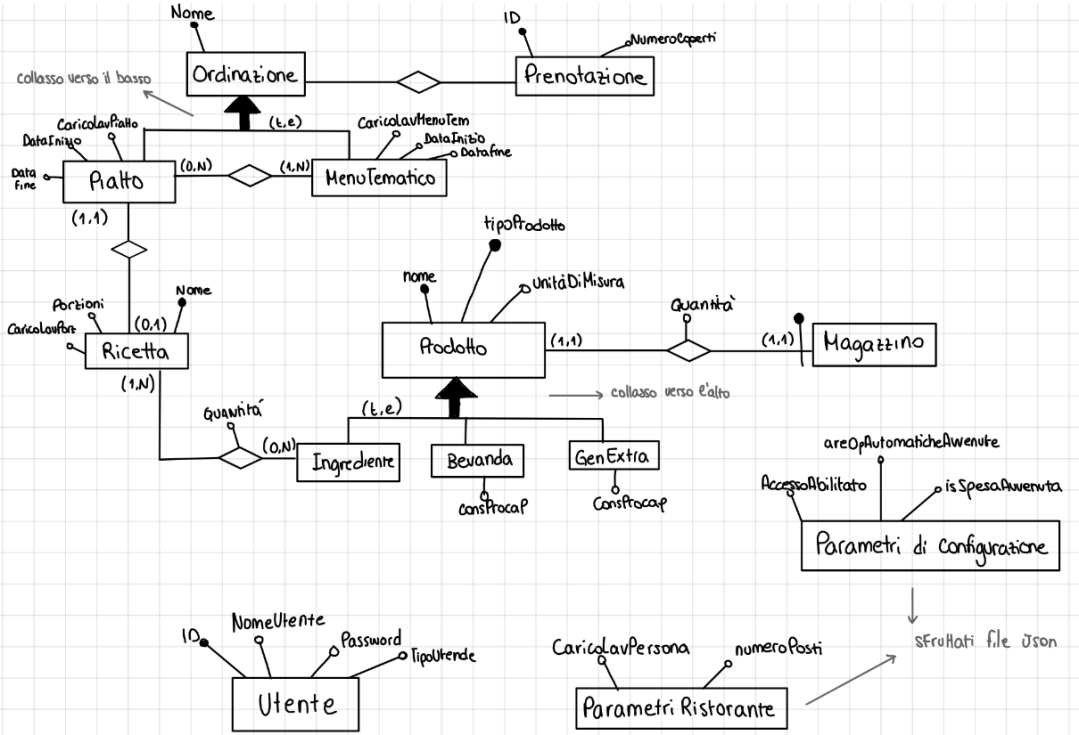


Diagramma E-R della base di dati.

8.
Testing del Login

8.
Testing del Login

```
@Test
void doLogin() {
    Login login = new Login(utentiRepository,  tipoUtente: "gestore");
    session = login.execute(session, cliView);

    Utente gestore = session.getUtente();

    assertNotNull(gestore);
    assertEquals(gestore.getUsername(),  actual: "g");
    assertEquals(gestore.getUserType(),  actual: "gestore");
    assertEquals(gestore.getPassword(),  actual: "pg");
}
```

Il frammento di codice rappresenta il metodo che si occupa di testare la classe *Login* e in particolare il metodo *doLogin* della stessa, attraverso la metodologia black box.

Il test è condotto sulla base dello scenario principale del caso d'uso «Login».

8.
Testing del Login

```
public class Test_Login_P {
    1 usage
    IUtentiRepository utentiRepository = TestServices.getInstance().ge
```

```
private TestServices() {
    this.servicesFactory = new ServicesFactory( connectionDBString: "jdbc:sqlite:RistoranteTest.db",
}

 atabonidev
public static TestServices getInstance() {
    TestServices result = instance;
    if (result == null) {
        synchronized (TestServices.class) {
            result = instance;
            if (result == null) {
                instance = result = new TestServices();
            }
        }
    }
    return result;
}
```

Eseguendo tutti i test insieme abbiamo riscontrato un errore dovuto al tentativo di accesso alla base di dati avvenuto da parte di connessioni diverse (una per ogni classe di test) contemporaneamente.

Abbiamo risolto il problema sfruttando il pattern **Gof Singleton** per creare un'unica connessione condivisa dalle diverse classi di test.

Nota: i metodi di test fanno riferimento ad una base di dati apposita, diversa da quella di produzione.

8.
Testing del Login

```java
@BeforeEach
public void setUp(){
    File file = new File( pathname: "./src/testing/input_cases/case_AggiungiPrenotazione_S.txt");
    try {
        inputDati = new InputDati(new Scanner(new BufferedReader(new FileReader(file))), new OutputUtils(new PrintWriter(outputStreamCaptor)));
        cliView = new CLIView(inputDati);
    } catch (FileNotFoundException e) {
        System.out.println("File non trovato");
    }
}
```

Per poter **leggere l'output** prodotto da un test e per potergli **fornire gli input** in modo automatico è stata configurata opportunamente la classe ***CLIView***.

Gli input utili ad un test specifico vengono prelevati dal file di testo ad esso associato.
Allo stesso modo l'output viene letto dall'istanza di una classe che sostituisce la console.

9.
Extract Class – Refactoring

9.
Extraxt Class – Refactoring

```
private Session showCaricoLavoroPerPersona(Session session, View view) {
    int caricoLavPerPersona = gestoreRepository.getParametriRistoranteRepository().getCaricoLavoroPerPersona();
    view.print(String.valueOf(caricoLavPerPersona));
    return session;
}
```

```
public class ShowCaricoLavoroPerPersona implements Handler {
    2 usages
    private GestoreRepository gestoreRepository;

    1 usage   Matilde Simonini
    public ShowCaricoLavoroPerPersona(GestoreRepository gestoreRepository){
        this.gestoreRepository = gestoreRepository;
    }
    Matilde Simonini
    @Override
    public Session execute(Session session, View view) {
        int caricoLavPerPersona = gestoreRepository.getParametriRistoranteRepository().getCaricoLavoroPerPersona();
        view.print(String.valueOf(caricoLavPerPersona));
        return session;
    }
}
```

Un esempio di pattern applicato durante la fase di refactoring del codice è **Extract Class**.

Il metodo *showCaricoLavoroPerPersona* è stato inizialmente messo all'interno della classe *VisualizzaParametro* e successivamente è stato estratto per costituire la classe *ShowCaricoLavoroPerPersona*.

Operando in questo modo abbiamo seguito anche il principio SOLID Single Responsibility.

Grazie per l’ascolto