RESTAURANT MANAGEMENT SYSTEM

STUDENT : VESA BIANCA

GRUPA : 30227

1. **Obiectivul temei**

Tema acestui proiect este implementarea unei aplicatii care simuleaza sistemul de gestiune al unui restaurant. Pentru indeplinirea acestei cerinte am urmat modelul pus la dispozitie in suportul lucrarii de laborator, si am impartit obiectivul principal in mai multe obiective secundare :

* Realizarea diagramei UML pusa la dispozitie in indrumatorul de laborator, impartind clasele aplicatiei in cele trei pachete : DataLayer, BusinessLayer, PresentationLayer
* Implementarea interfetelor grafice necesare pentru comunicarea cu utilizatorul
* Utilizarea Composite Design Pattern pentru crearea relatiilor intre clasele MenuItem, BaseProduct, CompositeProduct
* Serializarea clasei Restaurant, pentru salvarea datelor modificate la fiecare rulare a aplicatiei
* Implementarea Observable Design Pattern, care se utilizeaza in cadrul interfetei Chef user, pentru a semnala adaugarea unei comenzi ce contine un CompositeProduct, care trebuie preparat

1. **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Cazul de utilizare a acestei aplicatii este procesarea operatiilor ce pot avea loc in cadrul unui restaurant, separate in functie de fiecare utilizator.

Administratorul poate sa : adauge produse in meniu (de baza sau compuse), stearga produse din meniu (de baza sau compuse), modifice numele sau pretul unui produs, adauge in compozitia unui produs compus un produs de baza, stearga un produs de baza din compozitia unui produs compus, vada meniul restaurantului.

Chelnerul poate sa execute urmatoarele operatii : plasarea unei comenzi cu produse din meniul restaurantului, calcularea pretului total al unei comenzi, generarea unei chitante pentru o comanda, verificarea listei de comenzi.

Bucatarul este anuntat la fiecare plasare de comenzi care contin CompositeProducts, pentru a le pute agati.

Programul pune la dispozitie urmatorul scenariu de utilizare :

* Utilizatorul selecteaza din interfata grafica a restaurantului, daca este administrator, chelner sau bucatar
* La apasarea butonului corespunzator se deschide o alta interfata grafica, care pune la dispozitie operatiile enumerate mai sus, in functie de fiecare tip de utilizator
* Operatiile de adaugare, modificare sau stergere necesita input-uri in casetele text aferente fiecareia. Astfel, in cazul in care input-urile nu sunt valide sau nu corespund meniului restaurantului, se vor afisa notificari in aceasta privinta (ex. “Wrong Input! Try again” sau “Couldn’t find a menu product with this name”)
* Daca datele adaugate sunt valide, operatiile se vor executa cu succes, iar utilizatorii le pot verifica in functie de postul fiecaruia : administratorul poate vedea toate produsele din restaurant, apasand butonul “Show all menu items”, chelnerul poate vedea istoricul comenzilor plasate, apasand butonul “Show all orders”, iar bucatarul poate sa verifice comenzile pe care le are de preparat cu “Open the kitchen”
* Daca utilizatorul este chelner, nu poate plasa comenzi decat daca bucataria a fost deschisa in prealabil ( a fost apasa butonul “Open the kitchen” dupa pornirea aplicatiei)
* Dupa inchiderea aplicatiei se considera ca bucatarul a “preparat” toate comenzile pentru ziua respectiva, iar acestea nu vor mai aparea in tabel la urmatoarea rulare

1. **Proiectare (decizii de proiectare , diagrame UML , structuri de date , proiectare clase , interfete , relatii , packages , algoritmi , interfata utilizator)**

Aplicatia urmeaza diagrama UML pusa la dispozitie in indrumatorul de laborator, cu mici modificari. O prima decizie de proiectare luata este impartirea interfetei de procesare a operatiilor RestaurantProcessing in doua interfete separate: AdminProcessing si WaiterProcessing, care vor fi implementate de clasa Restaurant.

Clasele MenuItem, CompositeProduct si BaseProduct respecta Composite Design Pattern, asigurand relatiile de mostenire prezente.

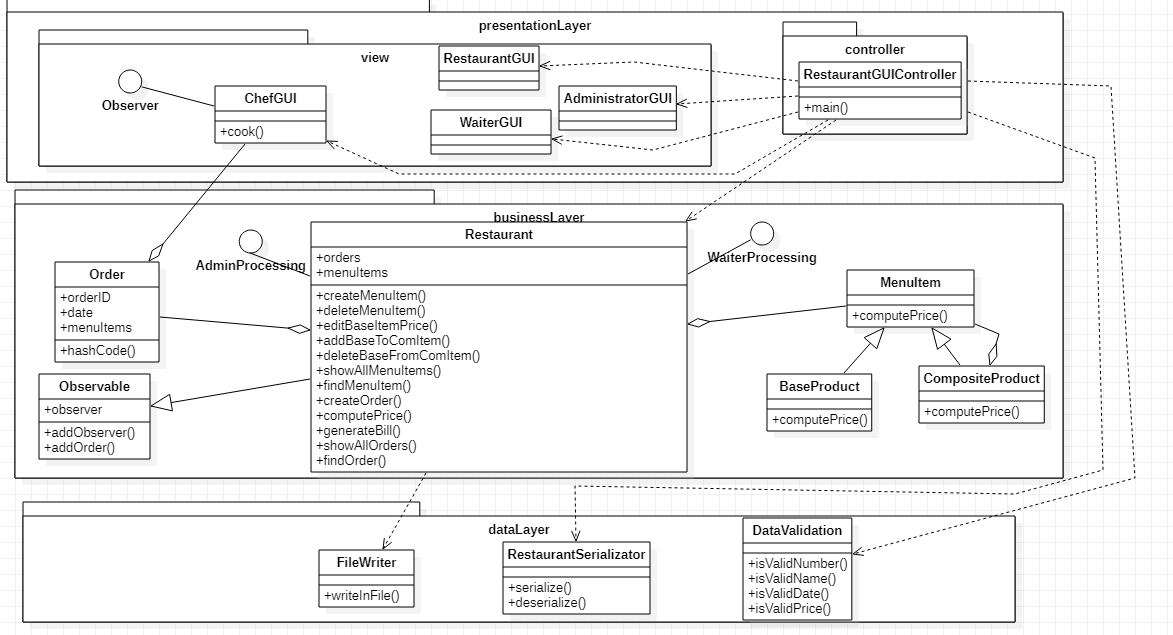
De asemenea, partea de interfatare grafica a aplicatiei contine clasele de tip View (AdministratorGUI, ChefGUI, WaiterGUI, RestaurantGUI), care modeleaza componentele java.swing necesare ferestrei ce comunica cu utilizatorul, precum si clasa tip Controller (RestaurantGUIController), care este de altfel si MainClass pentru intreg programul. Cea din urma controleaza, cu ajutorul claselor View, toate operatiile necesare.

Referitor la impartirea in pachete, aplicatia urmeaza modelul layered architecture:

* Pachetul businessLayer : contine clasele BaseProduct, CompositeProduct, MenuItem, Observable, Order, Restaurant, impreuna cu interfetele pe care le extind
* Pachetul dataLayer : contine clasele DataValidation, FileWriter, RestaurantSerializator
* Pachetul presentationLayer.view : contine AdministratorGUI, ChefGUI, RestaurantGUI, RestaurantGUI si interfata Observer
* Pachetul presentationLayer.view : contine clasa RestaurantGUIController

In ceea priveste structurile de date utilizate,am folosit instante ale colectiei List , pentru a manui cu usurinta un numar mare si variabil de obiecte ale clasei MenuItem si subclase ale acesteia. Fiind o structura dinamica, putem sa adaugam, sa eliminam obiecte din lista, precum si sa o parcurgem eficient , cu ajutorul metodelor pe care le pune la dispozitie. Pentru a stoca un Order am avut nevoie de HashMap, datorita faptului ca o comanda se memoreaza impreuna cu produsele sale. Structura HashMap este foarte utila pentru ca ofera performanta in timp constant pentru operatiunile de baza (adaugare, stergere, cautare).

In cele din urma , voi atasa diagrama UML aferenta proiectului :



1. **Implementare**

* **Clasele BaseProduct, CompositeProduct si MenuItem**

Aceste trei clase functioneaza conform modelului Composite Design Pattern.

Restaurantul are in meniu diferite produse, care sunt reprezentate prin clasa MenuItem, iar aceste produse pot fi simple (BaseProduct) sau complexe, adica formate din mai multe produse (CompositeProduct). Cele doua clase, BaseProduct si CompositeProduct mostenesc MenuItem, avand in comun metoda computePrice(). Avand nevoie de aceasta metoda, design pattern-ul ales este util pentru a trata in acelasi mod produsele compuse cu cele simple, fara a verifica la fiecare operatie din ce categorie face parte obiectul cu care se lucreaza. Astfel, un MenuItem are ca parametri nume si pret, iar unui CompositeProduct i se adauga o lista de MenuItem, deci computePrice va itera aceasta lista pentru a calcula suma tuturor produselor componente.

CompositeProduct are in plus urmatoarele metode:

* addBaseProduct(MenuItem) : adauga un produs nou in lista de MenuItem
* delBaseProduct(MenuItem) : sterge un produs din lista de MenuItem
* setBaseProductName(String oldName, String newName) : se apeleaza atunci cand modificam numele unui BaseProduct care este in componenta unui CompositeProduct
* setBaseProductPrice(String name, float newPrice) : se apeleaza atunci cand modificam pretul unui BaseProduct care este in componenta unui CompositeProduct
* hasBaseProduct(String name) : returneaza true daca exista in compozitie un MenuItem cu numele *name*
* getBaseProducts() : Resturneaza un String compus din numele fiecauri MenuItem din compozitie
* toString()
* **Clasa Order**

Parametri clasei Order sunt orderID (numar intreg dupa care se identifica unic fiecare Order), date (data la care a fost plasata comanda), si o colectie de MenuItem. Constructorul le initializeaza cu valorile primite.

Metodele descrise, exceptand gettere si settere sunt:

* hashCode() : returneaza ID-ul unic al fiecarei comenzi, ce reprezinta cheia la care va fi stocata comanda in structura HashMap a clasei Restaurant
* equals() : metoda suprascrisa care sa corespunda obiectelor de tip Order
* computePrice() : calculeaza suma tuturor produselor corespunzatoare unei comenzi
* showMenuItems() : returneaza un String compus din numele fiecarui MenuItem din comanda
* showCompositeProducts() : returneaza un String compus din numele fiecarui CompositeProduct din comanda, pentru a-l putea transmite interfetei ce afiseaza toate CompositeProducts pe care bucatarul le va gati
* toString()
* **Clasele Restaurant si Observable**

Clasa Restaurant este cea mai importanta in contextul acestei aplicatii si inglobeaza toate clasele mentionate mai sus. Interfetele pe care le implementeaza sunt: AdminProcessing (defineste operatiile pe care le poate efectua administratorul), WaiterProcessing (defineste operatiile pe care le poate efectua chelnerul) si Serializable (pentru a putea serializa obiectul Restaurant). De asemenea, se incadreaza in modelul Observer Design Pattern si extinde clasa abstracta Observable.

Pentru efectuarea cerintei propuse, am implementat clasa Observable, care are ca parametru un Observer.

Un Observable are rolul de a notifica Observer-ul atunci cand are loc un anumit eveniment. In cazul de fata, atunci cand chelnerul adauga o comanda ce contine un CompositeProduct, bucatarul trebuie sa il prepare, iar clasa Observable se ocupa cu apelarea metodei cook(Order) pe obiectul Observer, semnaland acest lucru.

Operatiunea descrisa anterior este necesara in cadrul metodei createOrder din Restaurant, pe care o voi descrie in cele ce urmeaza.

Variabilele folosite in Restaurant sunt lista de MenuItem, care pastreaza toate produsele din stocul restaurantului si un HashMap cu chei de tip Order si valori de tip colectie de MenuItem, care va salva fiecare comanda impreuna cu lista de MenuItem corespunzatoare.

Metodele mostenite de la cele doua interfete, precum si altele noi definite sunt:

* createMenuItem(MenuItem) : adauga un produs nou in lista de produse a restaurantului
* deleteMenuItem(MenuItem) : sterge un produs din meniul restaurantului; daca alte produse compuse au acel produs in compozitie, vor fi sterse apeland in mod recursiv aceasta metoda
* editBaseItemPrice(MenuItem menuItem, float newPrice) : modifica pretul unui produs simplu din meniu; daca el se afla in compozitia unui produs compus, si pretul lui va fi de asemenea modificat apeland metoda setBaseProductPrice
* editMenuItemName(MenuItem menuItem, String newName) : modifica numele unui produs din meniu; daca menuItem se afla in compozitia unui produs compus, va fi modificat si in lista lui de produse
* addBaseToComItem(MenuItem menuItem, MenuItem baseProduct) : adauga un nou MenuItem in compozitia unui produs compus
* deleteBaseFromComItem(MenuItem menuItem, MenuItem baseProduct) : sterge un produs din compozitia unui produs compus
* showAllMenuItems() : itereaza lista de MenuItem a restaurantului si adauga toate produsele intr-un tabel, care mai apoi va fi afisat in interfata grafica
* findMenuItem(String) : cauta dupa nume un produs in meniul restaurantului; daca produsul exista va fi returnat, in caz contrar se returneaza referinta la null
* createOrder(Order) : creeaza o comanda noua, iar daca aceasta contine un produs compus, o adauga in lista de comenzi a bucatarului ( observer )
* computePrice(Order) : calculeaza si returneaza pretul unei comenzi
* generateBill(Order) : apeleaza metoda statica a clasei FileWriter writeInFile, oferind ca parametri numele fisierului ce va contine chitanta unei comenzi si textul respectivei chitante
* showAllOrders() : itereaza colectia de Order a restaurantului si adauga toate comenzile intr-un tabel, care mai apoi va fi afisat in interfata grafica
* findOrder(int orderID) : cauta o comanda plasata de chelner dupa ID; returneaza comanda daca exista in istoricul restaurantului sau null in caz contrar
* **Clasele DataValidation, FileWriter si RestaurantSerializator**

Aceste trei clase se ocupa cu validarea datelor primite de la interfata cu utilizatorul, generarea fisierelor pentru chitantele comenzilor, respectiv serializarea/deserializarea obiectului Restauran.

Metodele implementate in fiecare dintre acestea sunt:

* DataValidation contine patru metode care valideaza datele pe care le folosim pentru a crea obiecte : isValidDate, isValidNumber, isValidName verifica cu ajutorul pattern-urilor regex daca numele produselor, datele comenzilor si ID-ul dupa care cautam o comanda sunt corecte, iar isValidPrice indica daca pretul introdus este numar negativ
* FileWriter are o singura metoda, care creeaza obiecte PrintWriter cu fisiere pentru chitantele comenzilor, si scrie in acele fisiere detaliile corespunzatoare
* RestaurantSerializator descrie doua metode : serialize si deserialize, care vor salva in/ incarca din fisierul “restaurant.ser” obiectul restaurant cu care lucreaza aplicatia, pentru a putea pastra rezultatele operatiilor efectuate la fiecare rulare
* **Clasele AdministratorGUI, WaiterGUI, RestaurantGUI**

Scopul principal indeplinit de cele trei este de a defini elementele java.swing necesare interfetei cu utilizatorul (JFrame, JPanel, JLabel, JTextField, JButton). Aceste componente sunt create in constructorii claselor si adaugate in interfata grafica cu ajutorul obiectelor ActionListener, definite doar in clasa controller.

* **Clasa ChefGUI**

Incadrandu-se alaturi de Restaurant si Observable in modelul Observer Design Pattern, ChefGUI este fereastra care joaca rolul de Observer si implementeaza interfata cu acelasi nume. La fel ca AdministratorGUI, WaiterGUI si RestaurantGUI, elementele grafice sunt initializate in constructor, insa are in plus fata de acestea un ArrayList<Order>, unde pastram toate comenzile care trebuie sa fie preparate de catre bucatar la apelarea metodei cook(Order).

* **Clasa RestaurantGUIController**

Este clasa Main a aplicatiei si se ocupa de toate operatiile care se efectueaza asupra claselor Restaurant si cele din pachetul view. La pornirea aplicatiei se deserializeaza obiectul Restaurant din “restaurant.ser”, iar apoi se creeaza ActionListeners pentru toate elementele grafice din interfata. La apasarea fiecarui buton care apare se executa operatii de validare, iar mai apoi cele existente in clasa Restaurant. Constrangerea existenta este ca Observer-ul restaurantului nu se initializeaza decat la apasarea butonului “Open the kitchen”, semnificand faptul ca un chelner nu poate plasa comenzi decat daca bucatarul este pregatit sa le primeasca.

La inchiderea aplicatiei, obiectul Restaurant este incarcat inapoi in “restaurant.ser” cu noile valori.

1. **Rezultate**

Dupa cum se poate observa in urma rularii programului, operatiile pe Orders si MenuItems se executa cu succes. Datorita faptului ca am implementat clasa DataValidation si am folosit-o la fiecare citire a input-urilor, inainte de a le transmite mai departe pentru creare de obiecte, m-am asigurat ca niciodata nu se vor transmite parametri care sa nu fie valizi. Din acest motiv, pentru a testa partea de Design by Contract, respectiv instructiunile assert, am eliminat temporar partile de validare, iar rezultatele preconditiilor, postconditiilor si a invariantului (care se asigura ca restaurantul sa aiba mereu produse in stoc) sunt cele asteptate.

Totodata, la pornirea aplicatiei, obiectul Restaurant este incarcat din fisierul “restaurant.ser”, urmand ca dupa executarea operatiilor dorite, la inchiderea ferestrei interfetei grafice Restaurant, sa fie serializat inapoi in fisierul “restaurant.ser”.

1. **Concluzii**

Dupa finisarea acestui proiect pot sa afirm ca am dobandit informatii noi cu privire la Observable Design Pattern, Composite Design Pattern, Design by Contract si Serialization, fiind concepte pe care nu le intalnisem in trecut. Am inteles utilitatea acestora si cum sa le implementez cat mai bine pentru a le folosi in cadrul proietului. De asemenea, am consolidat cunostintele pe care le aveam legate de utilizarea structurii de date Map si de eficienta pe care o ofera.

In ceea ce priveste dezvoltarile ulterioare, consider ca programul poate fi imbunatatit prin impartirea task-urilor pe care le are clasa RestaurantGUIController mai multor clase, pentru a evita aglomerarea de cod relativ asemanator. O metoda de inregistrare in sistemul de gestiune al restaurantului ar asigura ca operatiile executate de un angajat nu interfereaza cu cele ale altor angajati. Aici as putea sa adaug si ca aplicatia ar putea fi extinsa la un numar mai mare de chelneri si bucatari.

1. **Bibliografie**

* Composite Design Pattern

<https://www.geeksforgeeks.org/composite-design-pattern/>

<https://www.baeldung.com/java-composite-pattern>

* HashMap

<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

<https://stackoverflow.com/questions/1066589/iterate-through-a-hashmap>

* Observer Design Pattern

<https://www.baeldung.com/java-observer-pattern>

* Graphical User Interface

<https://docs.oracle.com/javase/tutorial/uiswing/layout/using.html>

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html>

* Serialization

<https://www.tutorialspoint.com/java/java_serialization.htm>