**MINISTERUL EDUCAȚIEI și CERCETĂRII**

**UNIVERSITATEA TEHNICĂ a MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICĂ și MICROELECTRONICĂ**

**DEPARTAMENTUL INGINERIA SOFTWARE și AUTOMATICĂ**

**Lucrare de curs**

Disciplina: Tehnici și Mecanizme de Proiectare Software

Tema: „Analiza și modelarea unei aplicații care va gestiona serviciul de livrari online al unui restaurant”

„Analysis and modeling of an application that will manage the online delivery service of a restaurant.”

A efectuat st.gr. TI-202 Goliș Boris.

A verificat lect.univ. Postaru Andrei.

asis.univ. Gaidau Mihai.

Chișinău 2023

Cuprins:

[Introducere 3](#_Toc137051749)

[1. Modele de design 4](#_Toc137051750)

[1.1 Modele de design creațional 4](#_Toc137051751)

[1.1.1 Singleton Design Pattern 4](#_Toc137051752)

[1.1.2 Builder Design Pattern 5](#_Toc137051753)

[1.1.3 Factory Method Design Pattern 6](#_Toc137051754)

[1.2 Modele de proiectare structurală 7](#_Toc137051755)

[1.2.1 Decorator Design Pattern 7](#_Toc137051756)

[1.2.2 Composite Design Pattern 8](#_Toc137051757)

[1.3 Behavioral design patterns 9](#_Toc137051758)

[1.3.1 Observer Design Pattern 9](#_Toc137051759)

[2 Prezentarea Aplicației 10](#_Toc137051760)

[2.1 Modelele de design creațional 11](#_Toc137051761)

[2.1.1 Factory Method 11](#_Toc137051762)

[2.1.2 Builder 12](#_Toc137051763)

[2.1.3 Singleton 13](#_Toc137051764)

[2.2 Modelele de proiectare structurală 15](#_Toc137051765)

[2.2.1 Decorator 15](#_Toc137051766)

[2.2.2 Composite 16](#_Toc137051767)

[2.3 Behavioral design patterns 17](#_Toc137051768)

[2.3.1 Observer 17](#_Toc137051769)

[3 Diagrama UML Generală: 18](#_Toc137051770)

[Concluzie 19](#_Toc137051771)

[Bibliografie: 20](#_Toc137051772)

# Introducere

În dezvoltarea aplicațiilor software, pattern-urile de design reprezintă un set de soluții recurente și testate, care ne ajută să abordăm problemele comune cu eficiență și eleganță. Acestea oferă structură, organizare și flexibilitate, facilitând crearea și întreținerea unui cod de calitate.

În cadrul acestui raport, vom explora aplicarea unor pattern-uri de design în dezvoltarea unei aplicații Java pentru un serviciu de online food ordering.

Fiecare pattern de design reprezintă o piesă distinctă a unui puzzle complex, contribuind la întregul tablou al aplicației noastre. Vom aduce în discuție câteva dintre aceste pattern-uri și modul în care se integrează în cadrul serviciilor noastre online.

Singleton este un design pattern de creare, care asigură că o clasă are o singură instanță și oferă un punct global de acces la aceasta. Acesta este util în situațiile în care dorim să avem o singură instanță a unei clase în întreaga aplicație, cum ar fi un obiect de configurare sau un manager de resurse.

Builder este un design pattern de creare care permite construirea obiectelor complexe pas cu pas. În loc să utilizăm constructori supraincarcați, Builder ne permite să specificăm opțional diferite componente ale unui obiect și apoi să le combinăm pentru a crea obiectul final. Acesta este util atunci când avem obiecte complexe cu multiple caracteristici și opțiuni de configurare.

Factory este un design pattern de creare care se concentrează pe crearea obiectelor fără a expune logică specifică de instanțiere. O fabrică (factory) abstractă definește o interfață pentru crearea obiectelor, dar clasele derivate decid ce tip de obiecte să creeze. Acesta este util atunci când dorim să abstractizăm procesul de creare a obiectelor și să oferim o modalitate flexibilă de a le instanția.

Observer este un design pattern comportamental care permite unui obiect să notifice și să actualizeze alte obiecte atunci când se întâmplă anumite evenimente. Este bazat pe principiul de dependență inversă, unde observatorii sunt abonați la subiect și primesc actualizări automat atunci când se produc schimbări. Acesta este util în scenarii în care dorim să avem o comunicare slab cuplată între obiecte, cum ar fi într-un sistem de tip eveniment-orientat.

Composite este un design pattern structural care permite gruparea obiectelor într-o structură de arbore, astfel încât acestea pot fi tratate în mod uniform. Compoziția permite manipularea unui obiect individual și a grupurilor de obiecte în aceeași manieră. Acesta este util atunci când avem o structură ierarhică de obiecte și dorim să le tratăm în mod recursiv.

Decorator este un design pattern structural care permite atașarea de comportamente suplimentare la un obiect în timpul rulării. Acesta se bazează pe conceptul de încapsul

# Modele de design

În domeniul ingineriei software, un model de proiectare reprezintă o abordare generală și repetabilă pentru rezolvarea unei probleme comune în procesul de proiectare a software-ului. Un model de proiectare nu este un design final și nu poate fi transformat direct în cod. În schimb, acesta reprezintă o descriere sau un șablon care poate fi aplicat pentru a rezolva o problemă într-o varietate de situații diferite.

## Modele de design creațional

Un design pattern crețional este un tip de model de proiectare în ingineria software care se concentrează pe crearea obiectelor într-un mod flexibil și eficient. Scopul principal al design pattern-urilor creționale este de a oferi o modalitate coerentă și simplă de a crea obiecte, fără a le lega strâns de clasele concrete.

### Singleton Design Pattern

Unul dintre cele mai cunoscute design pattern-uri creționale este "Singleton". Acesta are ca scop asigurarea faptului că o clasă poate avea o singură instanță și oferă un punct global de acces la aceasta. Singleton este util în situații în care este necesară existența unei singure instanțe a unei clase, cum ar fi gestionarea unei conexiuni la bază de date sau crearea unui registru global.

**Structura:**

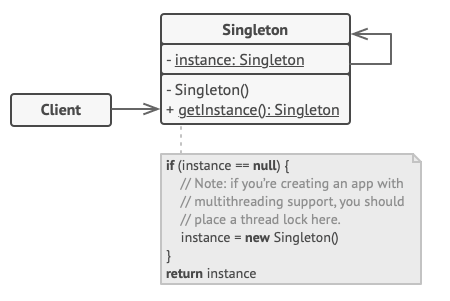


Figura 1.1 – Structura Singleton Design Pattern.

Clasa Singleton declară metoda statică getInstance care returnează aceeași instanță a propriei clase. Constructorul clasei Singleton ar trebui ascuns de codul clientului. Apelarea metodei getInstance ar trebui să fie singura modalitate de a obține obiectul Singleton.

### Builder Design Pattern

Un alt exemplu important de design pattern crețional este "Builder". Acesta este folosit atunci când se dorește crearea de obiecte complexe pas cu pas. Builder separă procesul de construcție al obiectului de reprezentarea sa finală, permițând astfel construirea unui obiect în etape și oferind flexibilitate în alegerea și ordinea pașilor de construcție.

**Structura**

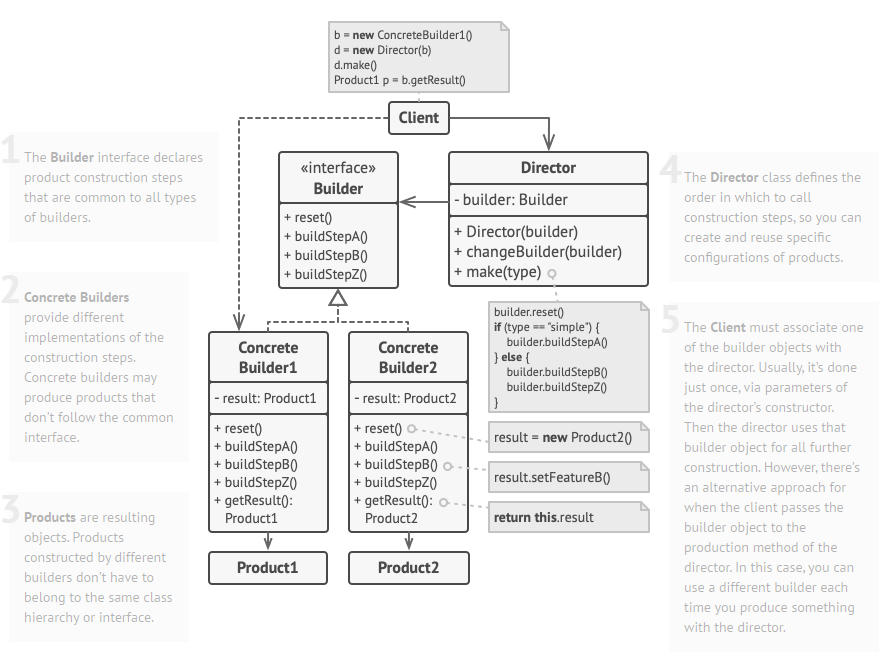


Figura 1.2 – Structura Builder Design Pattern.

1. Interfața Builder declară pașii de construcție a produsului care sunt comuni tuturor tipurilor de constructori.
2. Constructorii concreți oferă implementări diferite ale pașilor de construcție. Constructorii concreți pot produce produse care nu respectă interfața comună.
3. Produsele sunt obiecte rezultate. Produsele construite de constructori diferiți nu trebuie să aparțină aceleiași ierarhii de clase sau interfețe.
4. Clasa Director definește ordinea în care să apeleze pașii de construcție, astfel încât să puteți crea și reutiliza configurații specifice de produse.
5. Clientul trebuie să asocieze unul dintre obiectele constructorului cu directorul. De obicei, acest lucru se face o singură dată, prin parametrii constructorului directorului. Apoi, directorul utilizează acel obiect constructor pentru toate construcțiile ulterioare. Cu toate acestea, există o abordare alternativă în cazul în care clientul transmite obiectul constructor către metoda de producție a directorului. În acest caz, puteți utiliza un constructor diferit de fiecare dată când produceți ceva cu directorul.

### Factory Method Design Pattern

Un alt exemplu de design pattern crețional este "Factory Method". Acesta permite crearea de obiecte într-o clasă de bază, dar delegă procesul de creare către subclasele sale. Astfel, o clasă abstractă sau o interfață definește o metodă prin care se creează obiecte, dar subclasele sale decid concret ce tip de obiect să creeze. Acesta permite o flexibilitate mai mare în crearea obiectelor și deschide posibilitatea extinderii ulterioare a acestora.

**Structura:**

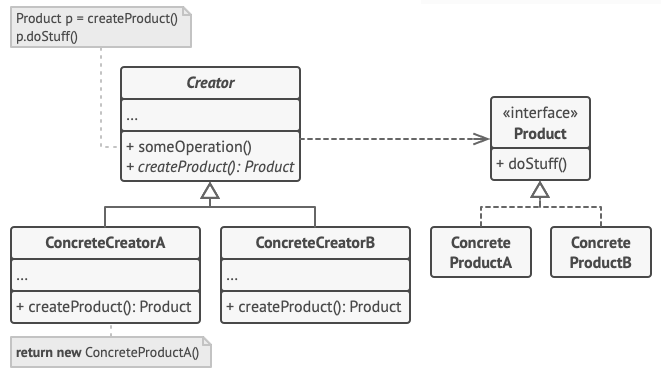


Figura 1.3 – Structura Factory Method Design Pattern.

1. Produsul declară interfața, care este comună tuturor obiectelor care pot fi produse de către creator și subclaselor sale.
2. Produsele concrete sunt implementări diferite ale interfeței produsului.
3. Clasa Creator declară metoda factory care returnează noi obiecte de produs. Este important ca tipul de returnare al acestei metode să se potrivească cu interfața produsului.

Puteți declara metoda factory ca fiind abstractă pentru a forța toate subclasele să implementeze propriile versiuni ale metodei. Ca alternativă, metoda factory de bază poate returna un tip de produs implicit.

Notă: în ciuda numelui său, crearea de produse nu este responsabilitatea primară a creatorului. De obicei, clasa creatorului are deja o anumită logică de afaceri de bază legată de produse. Metoda factory ajută la decuplarea acestei logici de clasele concrete de produs. Iată o analogie: o companie mare de dezvoltare software poate avea un departament de formare pentru programatori. Cu toate acestea, funcția principală a companiei ca întreg rămâne scrierea de cod, nu producerea de programatori.

1. Creatorii concreți suprascriu metoda factory de bază astfel încât să returneze un tip diferit de produs. Rețineți că metoda factory nu trebuie să creeze întotdeauna noi instanțe. Aceasta poate returna și obiecte existente dintr-un cache, o bază de obiecte sau o altă sursă.

## Modele de proiectare structurală

Modelele de proiectare structurale sunt o categorie de modele de proiectare în ingineria software care se concentrează pe organizarea și compunerea claselor și obiectelor pentru a forma structuri mai mari și pentru a oferi funcționalități noi. Aceste modele ajută la asigurarea unui design de sistem flexibil și eficient, promovând reutilizarea codului, modularitatea și extensibilitatea.

### Decorator Design Pattern

Modelul decorator adaugă dinamic comportamente sau responsabilități noi obiectelor prin învelirea acestora în una sau mai multe clase de decorator. Acest model oferă o alternativă flexibilă la subclasare pentru extinderea funcționalității.

**Structura**

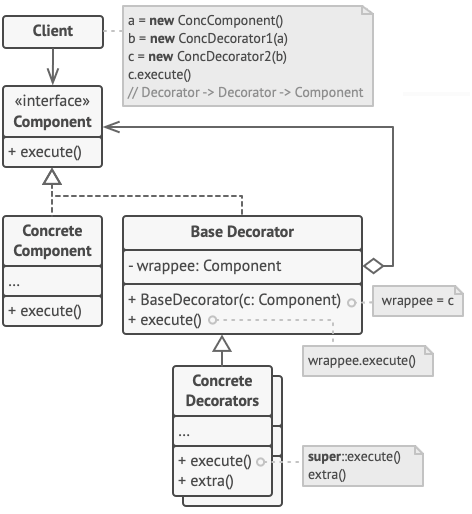


Figura 2.1 – Structura Decorator Design Pattern.

1. Componenta declară interfața comună pentru atât obiectele încapsulate, cât și pentru împachetările acestora.
2. Componenta concretă este o clasă de obiecte care este încapsulată. Aceasta definește comportamentul de bază, care poate fi modificat de decoratori.
3. Clasa Decorator de bază are un câmp pentru referințierea unui obiect încapsulat. Tipul câmpului ar trebui declarat ca fiind interfața componente pentru a putea conține atât componente concrete, cât și decoratori. Decoratorul de bază delegă toate operațiile obiectului încapsulat.
4. Decoratorii concreți definesc comportamente suplimentare care pot fi adăugate dinamic la componente. Decoratorii concreți suprascriu metodele decoratorului de bază și își execută comportamentul înainte sau după apelul metodei părinte.
5. Clientul poate încapsula componente în mai multe straturi de decoratori, atâta timp cât lucrează cu toate obiectele prin intermediul interfeței componente.

### Composite Design Pattern

Modelul compozit definește o structură arborescentă a obiectelor, tratând obiectele individuale și compozițiile de obiecte în mod uniform. Acesta permite clienților să lucreze cu obiecte individuale și grupuri de obiecte într-un mod consistent.

**Structura**

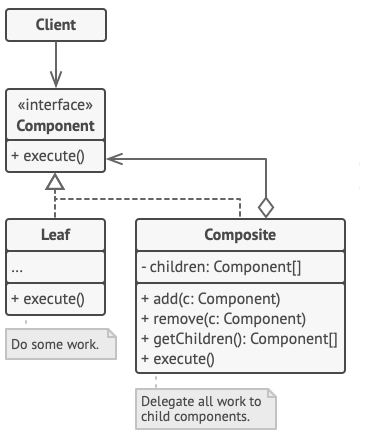


Figura 2.2 – Structura Composite Design Pattern.

1. Interfața Component descrie operațiile comune atât pentru elementele simple, cât și pentru elementele complexe ale arborelui.
2. Frunza (Leaf) este un element de bază al unui arbore care nu are sub-elemente. De obicei, componente frunză fac majoritatea lucrului real, deoarece nu au pe cine să delegheze munca.
3. Containerul (cunoscut și sub numele de compozit) este un element care are sub-elemente: frunze sau alte containere. Un container nu cunoaște clasele concrete ale copiilor săi. El lucrează cu toate sub-elementele doar prin intermediul interfeței componente. La primirea unei solicitări, un container delegă munca la sub-elementele sale, procesează rezultate intermediare și apoi returnează rezultatul final clientului.
4. Clientul lucrează cu toate elementele prin intermediul interfeței componente. Ca rezultat, clientul poate lucra în același mod cu elementele simple sau complexe ale arborelui.

## Behavioral design patterns

Behavior Design Pattern, denumit și "Pattern de proiectare comportamental", este un concept din domeniul ingineriei software care se referă la un set de soluții recurente pentru probleme comune legate de comportamentul și interacțiunea obiectelor și componentelor dintr-un sistem software. Aceste pattern-uri de design furnizează un cadru și o structură pentru organizarea și gestionarea comportamentului aplicațiilor.

### Observer Design Pattern

Acesta permite comunicarea între obiecte într-un mod slab cuplat. Un subiect (obiectul observabil) notifică automat toate obiectele observatoare (ascultătorii) despre orice schimbare de stare.

**Structura**

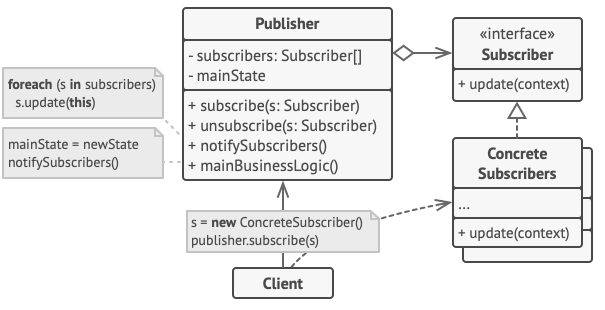


Figura 3.1 – Structura Observer Design Pattern.

1. Editorul (Publisher) emite evenimente de interes pentru alte obiecte. Aceste evenimente apar atunci când editorul își schimbă starea sau execută anumite comportamente. Editorii conțin o infrastructură de abonare care permite noilor abonați să se alăture și abonaților actuali să părăsească lista.
2. Atunci când apare un eveniment nou, editorul trece prin lista de abonamente și apelează metoda de notificare declarată în interfața abonatului pe fiecare obiect abonat.
3. Interfața Subscriber declară interfața de notificare. În majoritatea cazurilor, aceasta constă într-o singură metodă de actualizare (update). Metoda poate avea mai mulți parametri care permit editorului să transmită detalii despre eveniment împreună cu actualizarea.
4. Subscriitorii concreți efectuează anumite acțiuni în răspuns la notificările emise de editor. Toate aceste clase trebuie să implementeze aceeași interfață pentru a nu cupla editorul cu clase concrete.
5. De obicei, abonații au nevoie de anumite informații de context pentru a gestiona corect actualizarea. Din acest motiv, editorii trec adesea date de context ca argumente ale metodei de notificare. Editorul poate transmite chiar el însuși ca argument, permițând abonatului să obțină direct orice date necesare.
6. Clientul creează separat obiectele editor și abonat și apoi înregistrează abonații pentru actualizările editorului.

# Prezentarea Aplicației

Aplicația realizata are că scop primordial gestionarea comenzilor online, care vin pe serverul virtual al restaurantului. Bazandu-ne pe programarea orientata pe obiecte (OOP) am realizat în limbajul Java următoarele pattern-uri:

* Modelele de design creațional:
* Singleton;
* Builder ;
* Factory.
* Modelele de proiectare structurală:
* Decorator;
* Composite.
* Behavioral design patterns:
* Observer.

Utilizând aceste modele de proiectare în dezvoltarea aplicației noastre de consolă pentru produse, am reușit să obținem următoarele beneficii:

- O structură mai modulară: Modelele de proiectare ne-au permis să separăm responsabilitățile și comportamentul diferitelor componente în entități distincte. Aceasta ne-a facilitat organizarea și gestionarea codului într-un mod mai clar și mai structurat.

- Separare clară a responsabilităților: Fiecare clasă și componentă are rolul său bine definit în cadrul modelului de proiectare ales. Acest lucru ne-a ajutat să separăm responsabilitățile și să reducem cuplajul între diferitele părți ale sistemului nostru.

- Flexibilitate crescută în gestionarea diferitelor tipuri de produse: Prin utilizarea modelelor de proiectare, am putut adăuga și gestiona ușor noi tipuri de produse în aplicația noastră. De exemplu, utilizând modelul Builder, am putut crea diferite implementări ale pașilor de construcție, permițându-ne să construim programul necesar.

Aceste îmbunătățiri ne-au permis să dezvoltăm și să gestionăm mai eficient aplicația noastră de consolă pentru serviciul online de food ordering , oferindu-ne o structură modulară, o separare clară a responsabilităților și o flexibilitate crescută în gestionarea comenzilor.

## Modelele de design creațional

### Factory Method

Secventa codului din proiectul dat are o implementare a unui model Factory în Java, care creează și gestionează obiecte de tipul `FoodItem`. Această abordare permite crearea și manipularea obiectelor de tipul `Pizza` și `Burger` folosind aceeași interfață `FoodItem`.

`FoodItem` este o interfață care definește trei metode: `prepare()`, `cook()` și `serve()`. Aceste metode sunt implementate în clasele `Pizza` și `Burger`, care implementează interfața `FoodItem`. Implementarea acestor metode presupune afișarea unor mesaje specifice și așteptarea unei perioade de timp folosind `TimeUnit.SECONDS.sleep()`.

Clasa `FoodItemFactory` este responsabilă pentru crearea obiectelor de tip `FoodItem`. Aceasta are o metodă `createFoodItem()` care primește un tip de aliment (`type`) și returnează un obiect de tip `FoodItem` corespunzător. În implementarea actuală, metoda verifică dacă `type` este "pizza" sau "burger" și returnează o instanță corespunzătoare a clasei respective. Dacă `type` nu corespunde niciunui tip de aliment cunoscut, se returnează `null`.

Această abordare permite crearea și utilizarea obiectelor de tip `Pizza` și `Burger` într-un mod flexibil și ușor extensibil. De exemplu, dacă se dorește adăugarea unui nou tip de aliment, este suficient să se creeze o nouă clasă care implementează interfața `FoodItem` și să se adauge un caz corespunzător în metoda `createFoodItem()` din clasa `FoodItemFactory`.

Este important să menționez că, în implementarea actuală, metodele `prepare()`, `cook()` și `serve()` aruncă o excepție `InterruptedException` atunci când firul de execuție este întrerupt în timpul așteptării. Această excepție poate fi tratată și gestionată corespunzător în codul care utilizează aceste metode.

În concluzie, secventa codului din proiect implementează un model Factory pentru a crea și manipula obiecte de tipul `Pizza` și `Burger`, folosind o interfață comună `FoodItem`. Aceasta este o abordare eficientă și flexibilă pentru gestionarea obiectelor similare din punct de vedere funcțional, dar cu implementări specifice diferite.

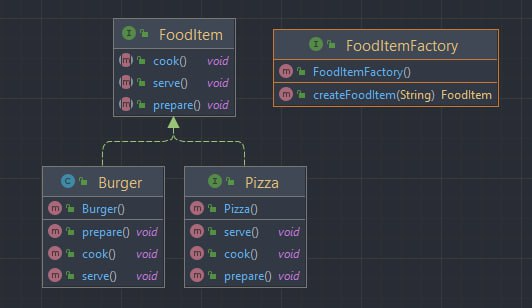


Figura 4.1 – Schema UML pentru Metoda Factory.

* Există o interfață FoodItem care definește metodele prepare(), cook() și serve().
* Clasa Pizza și clasa Burger implementează interfața FoodItem și oferă implementări specifice pentru metodele respective.
* Clasa FoodItemFactory are o metodă createFoodItem() care primește un tip de aliment și returnează un obiect de tipul FoodItem.
* Relația dintre FoodItemFactory și FoodItem este de asociere, deoarece FoodItemFactory utilizează și returnează obiecte de tipul FoodItem.
* Ambele clase, Pizza și Burger, sunt asociate cu interfața FoodItem prin implementare (implements), ceea ce înseamnă că acestea oferă implementări concrete pentru metodele definite în interfață.

### Builder

Secventa codului din proiectul dat este o implementare simplă a unui design pattern Builder pentru a construi obiecte de tip Order într-un mod mai flexibil și mai ușor de utilizat. Clasa `Order` reprezintă obiectul final pe care dorim să-l construim. Aceasta are câmpuri pentru `foodItem` (elementul de mâncare comandat), `quantity` (cantitatea comandată), `options` (opțiunile suplimentare ale comenzii), `observers` (o listă de observatori de starea comenzii) și `status` (starea comenzii). Metoda `Order` este construită cu ajutorul clasei `OrderBuilder`. `OrderBuilder` este o clasă separată care permite configurarea simplă a obiectului `Order`. Aceasta are metode pentru a seta `foodItem`, `quantity` și pentru a adăuga `options`. Metodele returnează întotdeauna o referință la instanța curentă a `OrderBuilder`, astfel încât să putem encadena mai multe metode într-o singură linie. Metoda `build()` este folosită pentru a crea și returna obiectul `Order` final. Clasa `Order` are și metode pentru adăugarea și eliminarea de observatori, setarea stării comenzii și notificarea observatorilor atunci când starea se schimbă. Utilizarea design pattern-ului Builder în acest cod facilitează crearea și configurarea obiectelor de tip `Order`, înlocuind nevoia de a utiliza constructori cu un număr mare de parametri sau de a folosi metode setter individuale. Astfel, se obține un cod mai curat și mai ușor de întreținut.

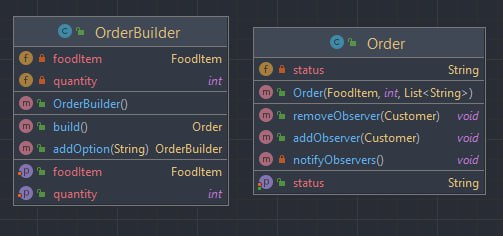


Figura 4.2 – Schema UML pentru Builder.

* Clasa FoodItem reprezintă un element de mâncare și este folosită în construirea comenzii (Order).
* Clasa Customer reprezintă un observator al stării comenzii. Aceasta este notificată atunci când starea comenzii se schimbă.
* Clasa OrderStatusObserver este o interfață sau clasă abstractă care definește metoda observe() pentru observatorii stării comenzii.
* Clasa Order reprezintă comanda și are o relație de asociere cu FoodItem. Are, de asemenea, o listă de observatori (observers) și un câmp pentru starea comenzii (status). Metodele addObserver(), removeObserver(), setStatus() și getStatus() permit manipularea stării comenzii și gestionarea observatorilor.
* Clasa OrderBuilder este responsabilă pentru construirea comenzii. Aceasta are câmpuri pentru foodItem, quantity și options. Metodele setFoodItem(), setQuantity() și addOption() sunt utilizate pentru configurarea comenzii. Metoda build() este folosită pentru a crea și returna obiectul Order final.

### Singleton

Secventa din codul programului implementează un Singleton pentru clasa OrderManager. Iată o analiză a codului și un set de comentarii relevante:

* private static OrderManager instance: Aceasta este variabila statică care stochează o singură instanță a clasei OrderManager.
* private OrderManager(): Acesta este constructorul privat care împiedică crearea de instanțe directe ale clasei OrderManager.
* public static OrderManager getInstance(): Aceasta este metoda statică care returnează instanța unică a clasei OrderManager. Dacă instanța nu a fost creată încă, se creează o nouă instanță și este returnată. Dacă instanța există deja, se returnează instanța existentă.
* public void UserOrder(int idUser, String[] foods, String address): Aceasta este o metodă care preia un ID de utilizator, o listă de mâncăruri și o adresă și procesează comanda. Pentru fiecare mâncare din lista foods, se folosește un FoodItemFactory pentru a crea un obiect FoodItem. Apoi, se apelează metodele prepare(), cook() și serve() pe obiectul FoodItem creat pentru a pregăti, găti și servi mâncarea. În cazul în care apare o excepție în timpul procesării mâncării, se afișează un mesaj de eroare.

Comentarii generale:

* Singleton pattern este folosit pentru a permite o singură instanță a unei clase să existe în întregul sistem și pentru a oferi un punct de acces global la acea instanță.
* Constructorul privat asigură că instanțele clasei pot fi create doar din interiorul clasei.
* Metoda getInstance() oferă o modalitate de a obține instanța unică a clasei OrderManager.
* Metoda UserOrder() procesează comenzile utilizatorului și pregătește, gătește și servește mâncarea utilizând un FoodItemFactory.

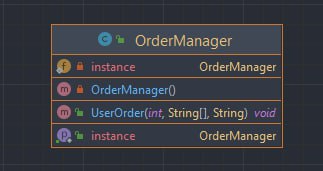


Figura 4.3 – Schema UML pentru Singleton.

* OrderManager este reprezentat ca o clasă cu o singură metodă getInstance() pentru a obține instanța singleton și o metodă UserOrder() pentru a procesa comenzile utilizatorului.
* FoodItemFactory este o altă clasă care este utilizată în metoda UserOrder(). Aceasta are o singură metodă createFoodItem(String) pentru a crea obiecte FoodItem.
* FoodItem este o clasă care conține metodele prepare(), cook() și serve(), care sunt apelate în metoda UserOrder() pentru a pregăti, găti și servi mâncarea.

## Modelele de proiectare structurală

### Decorator

Pattern-ul Decorator permite adăugarea de funcționalități suplimentare la un obiect existent, fără a afecta structura acestuia. Clasa FoodItemDecorator este o clasă abstractă care implementează interfața FoodItem. Aceasta conține un membru decoratedFoodItem de tip FoodItem, care reprezintă obiectul original la care se adaugă funcționalități suplimentare. Constructorul primește acest obiect decorat și îl stochează în membrul decoratedFoodItem. Metodele prepare(), cook() și serve() sunt suprascrise din interfața FoodItem și apelează metodele corespunzătoare ale obiectului decorat. Clasa ToppingDecorator extinde clasa FoodItemDecorator și adaugă funcționalități suplimentare la obiectul decorat. În metoda prepare(), se apelează mai întâi metoda prepare() a obiectului decorat pentru a păstra comportamentul original, iar apoi se adaugă logica specifică de pregătire a topping-ului. Această implementare permite adăugarea de mai multe decoratori, fiecare adăugând funcționalități suplimentare la obiectul decorat. Astfel, se poate crea o structură în formă de lanț, în care fiecare decorator adaugă un nou comportament la obiectul decorat.

De exemplu, putem avea un decorator CheeseDecorator care adaugă brânză la mâncărurile decorate, sau un decorator SauceDecorator care adaugă sosuri. Aceste decoratoare pot fi adăugate în lanț, astfel încât să se obțină mâncăruri cu multiple topping-uri și sosuri.

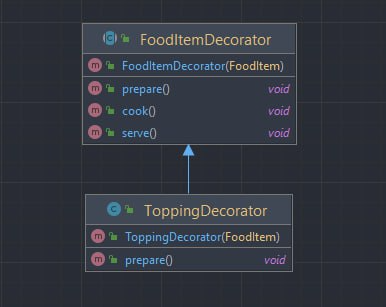


Figura 5.1 – Schema UML pentru Decorator.

* Clasa FoodItem reprezintă interfața care definește metodele prepare(), cook() și serve().
* Clasa abstractă FoodItemDecorator implementează interfața FoodItem și conține un membru decoratedFoodItem de tip FoodItem. Această clasă este punctul de plecare pentru toți decoratori.
* Clasa ToppingDecorator extinde clasa FoodItemDecorator și adaugă logica specifică pentru pregătirea topping-urilor.
* Săgețile dintre clase reprezintă relația de moștenire (extindere) sau de agregare (compoziție). În cazul nostru, ToppingDecorator extinde FoodItemDecorator, iar FoodItemDecorator conține un membru de tip FoodItem.

### Composite

Pattern-ul Composite este utilizat pentru a crea structuri ierarhice de obiecte, astfel încât obiectele individuale și grupurile de obiecte să fie tratate în mod uniform.

În codul de mai sus, avem trei clase:

* MenuCategory este clasa compozită care implementează interfața MenuComponent. Aceasta reprezintă o categorie de meniu care poate conține sub-componente (elemente de meniu individuale sau alte categorii de meniu). Metodele addComponent și removeComponent sunt utilizate pentru a adăuga și elimina sub-componente. Metoda display parcurge toate sub-componentele și le afișează pe rând.
* MenuComponent este interfața care definește metoda display. Toate componentele meniului (atât categoriile, cât și elementele de meniu) implementează această interfață și trebuie să ofere o implementare pentru metoda display.
* MenuItem este clasa frunză care reprezintă un element de meniu individual. Aceasta implementează interfața MenuComponent și oferă o implementare pentru metoda display. Clasa conține o denumire și un preț pentru elementul de meniu.

În utilizarea acestui cod, puteți crea o structură ierarhică a meniului adăugând categorii de meniu și elemente de meniu la categorii. Apoi, puteți apela metoda display pe categoria principală pentru a afișa întregul meniu.

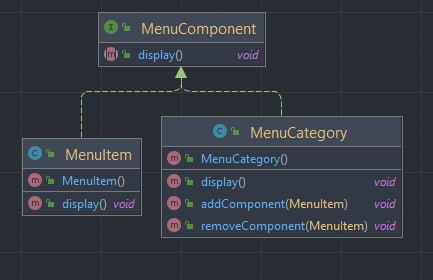


Figura 5.2 – Schema UML pentru Composite.

* MenuComponent este interfața care definește metoda display().
* MenuItem este clasa care implementează MenuComponent. Are două atribute: name de tip String și price de tip int.
* MenuCategory este clasa compozită care implementează MenuComponent. Are un atribut subComponents de tip List<MenuItem> pentru a stoca sub-componentele sale.
* Relația între MenuComponent și MenuItem este o asociație de tip compoziție, deoarece MenuCategory deține o listă de sub-componente MenuItem. De asemenea, MenuComponent are o asociere bidirecțională cu MenuItem prin metoda display(), care este implementată de ambele clase.

## Behavioral design patterns

### Observer

Interfața OrderStatusObserver definește un contract pentru observatorii de stări ale comenzilor. Are o metodă update care primește un obiect Order și este responsabilă de actualizarea observatorului cu starea comenzii.

Clasa Customer implementează interfața OrderStatusObserver și reprezintă un client. Are un câmp name care stochează numele clientului și un constructor pentru a seta acest câmp. Metoda update este suprascrisă pentru a afișa starea comenzii primite ca argument.

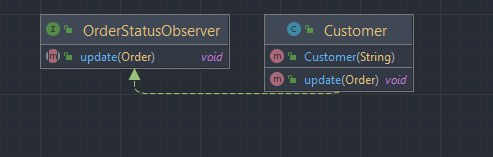


Figura 6.1 – Schema UML pentru Decorator.

* Există o interfață OrderStatusObserver cu o metodă update(Order) care este implementată de clasa Customer.
* Clasa Customer are un câmp privat name pentru a stoca numele clientului.
* Customer implementează metoda update(Order) pentru a gestiona actualizările stării comenzii pentru client.
* Interfața OrderStatusObserver este asociată cu clasa Order prințe o asociere de tipul <> (multiplicitate 1 la 1), ceea ce înseamnă că un observator poate fi asociat cu o singură comandă și o comandă poate avea un singur observator.
* Clasa Order are un câmp privat status pentru a stoca starea comenzii.

# Diagrama UML Generală:

Diagrama UML (Unified Modeling Language) este o formă de reprezentare grafică a sistemelor software, folosită pentru modelarea și proiectarea acestora. UML oferă un set standard de simboluri și convenții pentru a ilustra structura, comportamentul și interacțiunile componentelor unui sistem software.

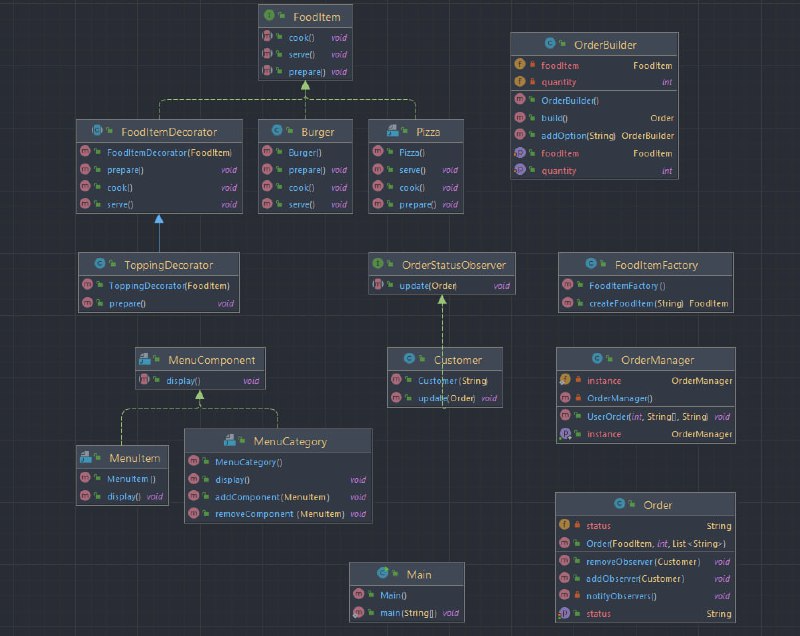


Figura 3.1 – Diagrama UML generală.

# Concluzie

În cadrul acestui proiect, am analizat un sistem de gestionare al comenzilor în care am identificat utilizarea mai multor paternuri de proiectare pentru a obține o structură coerentă și flexibilă. Codul furnizat a evidențiat următoarele tipare de proiectare. S-a folosit un Factory pentru a crea instanțe de obiecte de tip FoodItem. Clasa FoodItemFactory conține o metodă createFoodItem(), care primește un tip specific (cum ar fi "pizza" sau "burger" sau orice altă clasă ce implementează instanța FoodItem ) și returnează un obiect de tip corespunzător. Am observat utilizarea unui Builder pentru crearea obiectelor de tip Order. Clasa OrderBuilder oferă metode pentru setarea FoodItem-ului, cantității și opțiunilor, iar metoda build() construiește obiectul Order final. OrderManager implementează un Singleton pentru a asigura că există o singură instanță a clasei în întregul sistem. Această clasă gestionează comenzile utilizatorului și folosește Factory pentru a crea și trata obiecte de tip FoodItem. Clasa Customer acționează ca un observator în sistemul de gestionare a comenzilor. Implementând interfața OrderStatusObserver, Customer poate fi adăugat ca observator al unei comenzi și va fi notificat atunci când starea comenzii se schimbă. S-a utilizat un Decorator pentru a extinde funcționalitatea obiectelor de tip FoodItem. Clasa FoodItemDecorator oferă o bază pentru decoratori și este extinsă de clasa ToppingDecorator, care adaugă logică suplimentară pentru prepararea topping-urilor. Clasele MenuCategory și MenuItem implementează patern-ul Composite. MenuCategory este o categorie de meniu care poate conține mai multe obiecte MenuItem. Clasa MenuCategory oferă metode pentru adăugarea și eliminarea componentelor și pentru afișarea categoriei și a sub-componentelor sale.

Prin utilizarea acestor tipare de proiectare, sistemul de gestionare al comenzilor devine mai modular, flexibil și ușor de extins. Se separă responsabilitățile între diferitele clase și se asigură că modificările aduse într-o parte a sistemului nu afectează în mod neașteptat alte părți.

În concluzie, implementarea acestor tipare de proiectare în sistemul de gestionare al comenzilor contribuie la creșterea calității, ușurinței în dezvoltare și întreținere, și permite o flexibilitate sporită în adăugarea de noi funcționalități sau modificarea celor existente. Utilizarea acestor tipare demonstrează o înțelegere solidă a conceptelor de proiectare și abordări eficiente în dezvoltarea unui sistem complex.

# Bibliografie:

1. Codul sursă:

<https://github.com/DarkWeider11/TMPS>

1. Design Patterns:

<https://refactoring.guru/design-patterns>

1. Singleton Design Pattern:

<https://refactoring.guru/design-patterns/singleton>

1. Factory Method Design Pattern:

<https://refactoring.guru/design-patterns/factory-method>

1. Builder Design Pattern:

<https://refactoring.guru/design-patterns/builder>

1. Decorator Design Pattern:

<https://refactoring.guru/design-patterns/decorator>

1. Composite Design Pattern:

<https://refactoring.guru/design-patterns/composite>

1. Observer Design Pattern:

<https://refactoring.guru/design-patterns/observer>