Ministerul Educației și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Proiect de curs

Disciplina: Tehnici și mecanisme de proiectare software

Tema: Dezvoltarea aplicației inovatoare pentru comenzi în patiserie

Theme: Development of the innovative application for pastry orders

A efectuat: Sadovoi Maria, TI-206

A verificat: asist. univ. Buldumac Oleg

Chișinău 2023

Cuprins

1 Scopul și obiectivele proiectului

În ultimul timp, cererea pe piață de produse care pot fi comandate rapid pentru a fi consumate a crescut pe scară largă datorită creșterii populției pe glob. În patiseriile din orașele mari, unde multe oficii se află pe metru pătrat apare o cerere ridicată de consum de produse rapide, ceea ce reprezintă patiseriile ce conțin o varietate mare de acest fel. Aici apare o problemă pe care ați întâlnit-o cu siguranță în orele de vârf, ce consumă un timp în plus și stat în rând pentru preluarea comenzilor. Una din cunoscutele patiserii din orașul local, Granier, se confruntă cu așa probleme.

Scopul acestei aplicații bazată pe java, este de a soluționa problema clineților de a aștepta în rând mult timp să comande preparatele dorite, astfel vor avea posibilitatea cu ajutorul aplicației de a își seta comanda, care va fi preluată de angajații ce vor prepara bucatele alese prin utilizarea șabloanelor de proiectare structurale, comportamentale și creaționale pentru a avea o aplicație ușor de gestionat pe parcurs în caz că apar unele schimbări sau dacă avem de extins mai multe microservicii de gestionare a bazei de date sau de folosire a unui cloud.

În cadrul aceste drept scop de dezvoltare va fi:

- crearea a două designuri de meniu ce pot fi alese de clienți;
- oferirea posibilității a mai multor clienți de a comanda același meniu;
- crearea unei interfețe pentru meniu;
- implementarea posibilității clientului de a vedea procesul de preparare pe etape a produsului;
- crearea opțiunii de luare la pachet sau pe loc al produsului;
- oferirea clientului posibilitatea de a-și crea propia comandă;
- crearea mai multor tipuri de meniu.

Toate acestea vor duce la dezvoltarea unei aplicațiii ce va oferi clienților posibilitatea de procesare a comenzii mai rapide, astfel creând o eficiență sporită nu doar pentru clienți, dar și pentru angajați, care vor putea vizualiza comenzile la nivel de aplicație.

Pentru o patiserie aceste implementări propuse sunt un avantaj ridicat, ceea ce va scădea rata de clienți care vor pleca din cauza rândului, ce va aduce un nivel de satisfacție ridicat din ambele părți.

2 Reprezentarea problemei

În lumea agitată de astăzi, timpul este prețios, iar comoditatea a devenit o prioritate pentru clienți. Atunci când vizitezi o patiserie aglomerată, bucuria de a te delecta cu deliciile acestora poate fi uneori umbrită de neplăcerile coadă lungă și timp de așteptare. Conștienți de această provocare, vă prezentăm o soluție inovatoare - o aplicație mobilă concepută pentru a transforma modul în care clienții comandă și savurează deserturile preferate.

Patiseriile tradiționale se confruntă adesea cu cozi lungi în orele de vârf, ceea ce determină clienții frustrați și poate duce la pierderea potențială a afacerii. Așteptarea în linie pentru a plasa o comandă poate fi consumatoare de timp, provocând inconveniente și diminuând experiența generală. În plus, clienții care preferă să ia produsele cu ei pot întâmpina dificultăți în găsirea unei modalități rapide și eficiente de a plasa comanda.

Aplicația revoluționară are ca scop soluționarea provocărilor asociate cu comandarea în patiseriile tradiționale. Prin folosirea tehnologiei mobile, aplicația oferă o platformă convenabilă și eficientă pentru clienți, permițându-le să plaseze comenzi fără a mai sta la coadă și să se bucure de o experiență fără bariere.

Este oferită o interfață ușor de utilizat, care permite clienților să navigheze fără efort prin meniul patiseriei, să exploreze diverse opțiuni și să-și personalizeze comenzile în funcție de preferințele lor. Designul intuitiv asigură o experiență de comandă lină și fără stres.

Avem opțiuni flexibile de comandă. Clienții au libertatea de a alege între a lua produsele cu ei sau de a le savura în patiserie. Ei pot specifica preferința lor prin intermediul aplicației, asigurându-se astfel că comenzile lor sunt pregătite în consecință.

Aplicația acesta va fi inovatoare pentru comenzi în patiserie revoluționează modul în care clienții interacționează cu locurile preferate de deserturi. Eliminând necesitatea de a aștepta în cozi lungi, aplicația noastră oferă o soluție convenabilă, economisind timp și oferind o experiență personalizată pentru comandarea deliciilor preferate. Prin interfața intuitivă, opțiunile flexibile și posibilitatea de plată sigură, aplicația noastră promite să îmbunătățească experiența generală în patiserie, asigurându-se că clienții se pot bucura de deliciile preferate fără întârzieri sau neplăceri inutile. Adoptă viitorul comenzilor în patiserie și începe o călătorie a comodității și deliciilor culinare cu aplicația noastră inovatoare.

3 Modelarea și realizarea sistemului

Pentru realizarea sistemului s-au utilizat mai multe tipuri de șabloane: comportamentale, structurale și creaționale. La nivel de creaționale avem abstact factory, builder și prototype. Pentru șabloanele structurale avem implementate facade și decorator, iar pentru o conlucrare și aranjare mai bună la nivel de aspect avem și template method implementat pentru două părți ale aplicației noastre.

Pentru a nu specifica obiectele concrete ale subclaselor, am folosit abstract facatory. Primul lucru pe care îl sugerează este să se declare explicit interfețele pentru fiecare produs distinct din familia de produse. Toate tipurille de cheesecake-uri în cazul nostru cu gust de mango, ciocolată și clasic vor implementa interfața cheesecake. Următorul pas ar fi de făcut o listă a metodelor de creare pentru toate produsele care sunt parte din familia de produse, iar pentru aceasta vom folosi o clasă creator.

Pentru a putea face o comandă clientul, folosim patternul de builder, deoarece avem nevoie să creăm un obiect complex din obiecte simple folosind un approach de pas cu pas. Când dorim să procurăm nu vom alege doar un obiect, dar de regulă vom mai alege o opțiune din meniu, de aceea folosim builderul.

Alt șablon pentru dezvoltarea aplicației este prototype. Dacă cineva dorește aceleași produse ca tine, folosim prototype pentru a clona un obiect deja existent în loc de a crea unul nou. Orice clasă care implementează interfața prototype trebuie să implementeze clonarea ce va apela restul.

Dacă dorim să oferim clientului o experiență de neuitat, care va include procesul de creare a produselor, pașii de producere, prețul și tot ce intră în crearea unui cheesecake, folosim decorator și façade, care ne vor oferi metoda ideală de implementare.

Pentru a avea o interfață grafică în dependență de dorința clientului, în cazul nostru, vor fi două opțiuni din care poate alege fiind în formă de linie sau grilă folosim template method. De asemenea, clientul se poate gândi să comande să ia acasă sau să mănânce în local, pentru aceasta la fel folosim același pattern.

În continuare, avem dezvoltată modelarea fiecărui tip de șablon în parte, pentru început este descrisă ideea șablonului, după care este descrisă proiectarea diagramelor de clasă pentru fiecare șablon implementat în parte și cum lucrează la nivel de cod.

3.1 Șabloane creaționale

Șabloanele de proiectare creaționale sunt un subset al șabloanelor de proiectare care se concentrează pe crearea și inițializarea obiectelor într-un mod flexibil și eficient. Aceste sabloane oferă soluții standardizate pentru probleme comune legate de crearea obiectelor și permit obiectelor să fie create într-un mod independent de clasele lor concrete.

Importanța șabloanelor de proiectare creaționale constă în faptul că ele aduc beneficii semnificative în dezvoltarea software-ului. Iată câteva dintre motivele pentru care aceste sabloane sunt importante:

Abstragerea procesului de creare a obiectelor pentru șabloanele de proiectare creaționale permit programatorilor să abstragă și să izoleze logica de creare a obiectelor într-un mod coerent și modular. Aceasta duce la o mai mare claritate și înțelegere a codului și facilitează reutilizarea și întreținerea ulterioară a acestuia.

Flexibilitate în crearea obiectelor și anume șabloanele de proiectare creaționale oferă flexibilitate în crearea obiectelor, permițând programatorilor să decoupleze clientul de clasele concrete și să utilizeze interfețe și abstractizări pentru a crea obiectele. Acest lucru facilitează schimbul și extensibilitatea ulterioară a implementării obiectelor.

Ascunderea detaliilor de creare prin utilizarea sabloanelor de proiectare creationale permite ascunderea detaliilor de creare a obiectelor în spatele unui nivel de abstractizare. Astfel, codul client nu trebuie să fie conștient de detalii specifice de creare și poate interacționa doar cu interfețele și clasele abstracte, sporind modularitatea și flexibilitatea sistemului.

Promovarea principiilor SOLID, cu șabloanele de proiectare creaționale promovează principiile SOLID, cum ar fi Principiul Deschis/Închis și Principiul Responsabilității Unice. Aceste sabloane facilitează extensibilitatea și modularitatea codului, permițând adăugarea de noi tipuri de obiecte și fabrici fără a afecta codul existent.

Oferă ușurința în testare prin utilizarea sabloanelor de proiectare creationale facilitează testarea unitară și izolarea componentelor de cod. Prin utilizarea unor fabrici și interfețe abstracte, obiectele pot fi înlocuite cu ușurință cu stub-uri sau mock-uri în timpul testării, fără a afecta funcționalitatea globală a sistemului.

Pentru început vom analiza care este scopul șablonului creațional de proiectare Abstract Factory. Scopul e de a furniza o interfață pentru crearea unor familii de obiecte înrudite sau dependente fără precizând clasele lor concrete.

Utilizăm acest șablon când:

- un sistem ar trebui să fie independent de modul în care produsele sale sunt create, compuse, si reprezentat;
- un sistem ar trebui să fie configurat cu una dintre mai multe familii de produse;
- o familie de obiecte de produs înrudite este concepută pentru a fi utilizată împreună și trebuie să aplicați această constrângere;
- doriți să oferiți o bibliotecă de produse de clasă și doriți să dezvăluiți doar interfețele lor, nu implementările lor.

În continuare vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.1:

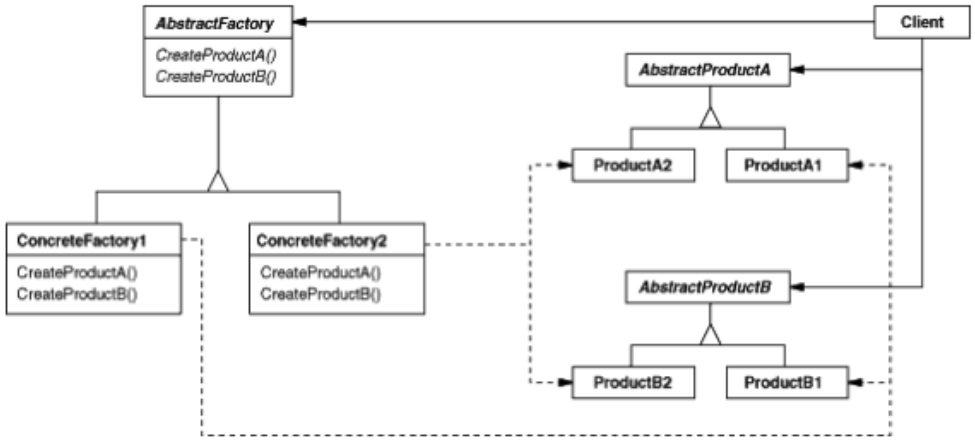

Figura 3.1 – Structura șablonului de proiectare Abstract Factory

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- abstractFactory declară o interfață pentru operațiuni care creează un produs abstract obiecte;
- concreteFactory implementează operațiunile de creare a obiectelor de produs concrete;
- abstractProduct declară o interfață pentru un tip de obiect produs;

- concreteProduct definește un obiect produs care urmează să fie creat de betonul corespunzător fabrică și implementează interfața AbstractProduct;
- client folosește numai interfețele declarate de AbstractFactory și clasele AbstractProduct.

Colaborările de care are parte acest pattern sunt în mod normal, o singură instanță a unei clase ConcreteFactory este creată în timpul execuției.Această Concrete Factory creează obiecte de produs având un particularimplementare. Pentru a crea diferite obiecte de produs, clienții ar trebui să folosească o altă Concrete Factory. AbstractFactory amână crearea de obiecte de produs către ConcreteFactory subclasă.

Drept consecințe a implementării și aplicării în practică a șablonului dar, avem drept rezultat unele.

Izolearea claselor concrete și anume modelul Abstract Factory vă ajută să controlați clasele de obiecte pe care le creează o aplicație. Pentru că o fabrică încapsulează responsabilitatea și procesul de creare a obiectelor de produs, izolează clienții de clasele de implementare. Clienții manipulează instanțe prin interfețele lor abstracte. Numele claselor de produse sunt izolat în implementarea fabricii de beton; nu apar în codul clientului.

Facilitează schimbul de familii de produse. Clasa unei fabrici de beton apare o singură dată într-o aplicație, adică acolo unde este instanțiată. Acest ușurează schimbarea fabricii de beton pe care o folosește o aplicație. Se poate utilizați diferite configurații de produs prin simpla schimbare a betonului fabrică. Deoarece o fabrică abstractă creează o familie completă de produse, întreaga familie de produse se schimbă deodată. În exemplul nostru de interfață cu utilizatorul, putem trece pur și simplu de la widget-urile Motif la widget-urile Presentation Manager prin comutarea obiectelor din fabrică corespunzătoare și recrearea interfata.

Promovează coerența între produse. Când produsul obiectează într-o familie sunt concepute pentru a lucra împreună, este important ca o aplicație să folosească obiecte dintr-o singură familie la un moment dat. AbstractFactory face acest lucru ușor a impune.

Sprijinirea noilor tipuri de produse este dificilă. Extinderea fabricilor abstracte a produce noi tipuri de produse nu este ușor. Asta pentru că Interfața AbstractFactory fixează setul de produse care pot fi create. Sprijinirea noilor tipuri de produse necesită extinderea interfeței din fabrică, care

implică schimbarea clasei AbstractFactory și a tuturor subclaselor acesteia. Discutăm o soluție la această problemă în secțiunea Implementare.

Patternuri care relaționează cu acest șablon sunt adesea implementate cu Factory Method, dar pot fi implementate și folosind Prototype.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată.

Pentru a nu specifica subclase de obiecte concrete dintr-o familie, am folosit modelul AbstractFactory. Primul lucru pe care îl sugerează este să declarați în mod explicit interfețele pentru fiecare produs distinct al familiei de produse. Toate variantele de cheesecake (ClassicCheesecake, ChocolateCheesecake, MangoCheesecake) implementează interfața Cheesecake. Următoarea mișcare este de a face o listă de metode de creare pentru toate produsele care fac parte din familia de produse (createCheesecake).

În continuare avem structura, sau mai bine zis implementarea la nivel de diagramă UML a sistemului pentru șablonul dat în care sun menționate toate relațiile care sunt pentru codul care a fost implementat și pentru o percepere maiușoară a logicii la nivel de implementare pentru orice persoană. Acesta reprezentând în figura 4.2 participanții și anume clasele și interfețele care depind și implementeaza unele de la altele.

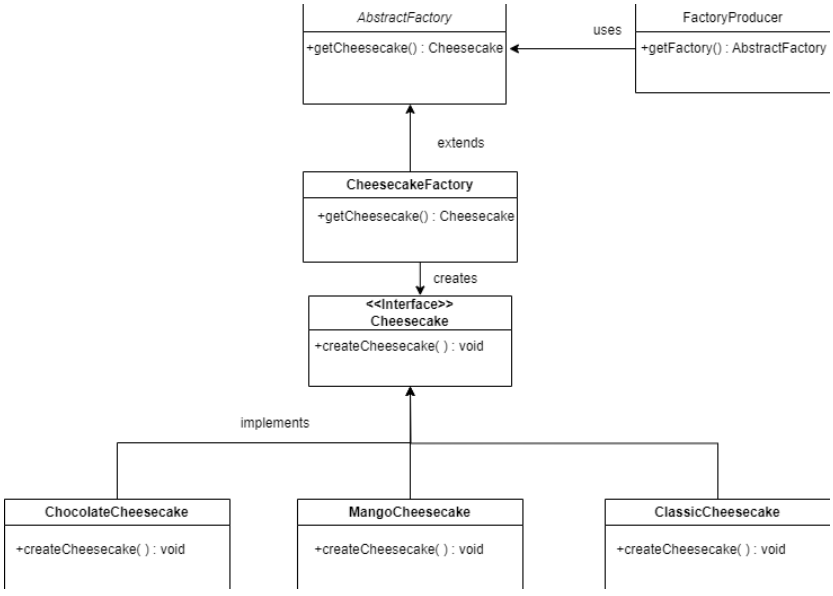

Figura 3.2 – Diagrama șablonului Abstract Factory

Paricipanții la nivel de acească diagramă sunt delegate prin asociere:

- AbstractFactory este participantul abstractFactory;
- Cheesecake este participantul interfața produsului;
- CheesecakeFactory este participantul Concrete Factory;
- Cheesecake cu mango, Cheesecake cu ciocolată și Classic Cheesecake sunt participanții concreți;
- Factory Producer are rolul de creator;
- AbstractFactoryPatternDemo are rolul de client.

În continuare vom realiza o explicație a fiecărei clase și a rolului său în cadrul modelului de fabrică abstractă:

- AbstractFactory aceasta este o clasă abstractă care declară metoda abstractă `getCheesecake(String cheesecakeType)`. Ea servește ca bază pentru clasele concrete de fabrică. Rolul său este de a defini interfața pentru crearea obiectelor de produs (cheesecake-uri) fără a specifica clasele lor concrete.
- Cheesecake aceasta este o interfață care declară metoda `createCheesecake()`. Ea reprezintă interfața de produs pentru diferite tipuri de cheesecake-uri. Rolul său este de a defini interfața pentru obiectele de produs pe care fabrica abstractă le creează.
- CheesecakeFactory această clasă extinde clasa AbstractFactory și suprascrie metoda `getCheesecake(String cheesecakeType)`. Ea implementează logica pentru crearea de tipuri specifice de cheesecake-uri bazate pe parametrul `cheesecakeType` furnizat. Rolul său este de a oferi implementarea concretă a fabricii abstracte prin crearea obiectelor de produs specifice (MangoCheesecake, ChocolateCheesecake, ClassicCheesecake) în funcție de tipul furnizat.
- MangoCheesecake, ChocolateCheesecake și ClassicCheesecake aceste clase implementează interfața Cheesecake și oferă implementări specifice pentru metoda `createCheesecake()`. Fiecare clasă reprezintă un tip diferit de cheesecake. Rolul lor este de a defini obiectele concrete de produs pe care fabrica concretă (CheesecakeFactory) le creează.
- FactoryProducer această clasă furnizează o metodă statică `getFactory(String choice)` care returnează o instanță a clasei AbstractFactory în funcție de alegerea furnizată. Rolul său

este de a încapsula crearea obiectului de fabrică concret (CheesecakeFactory) și de a oferi o modalitate centralizată de obținere a fabricii potrivite în funcție de alegere.

- AbstractFactoryPatternDemo este clasa principală în care este demonstrat modelul de fabrică abstractă. Rolul său este de a utiliza fabrica abstractă și obiectele de produs pentru a crea familii de obiecte conexe (cheesecake-uri). Aceasta obține o instanță a clasei AbstractFactory prin intermediul FactoryProducer și o utilizează pentru a crea diferite tipuri de cheesecake-uri, apelând metoda `getCheesecake(String cheesecakeType)` a fabricii. În final, aceasta apelează metoda `createCheesecake()` pe fiecare obiect de cheesecake pentru a le crea.

Prin atribuirea unor roluri specifice fiecărei clase, modelul de fabrică abstractă promovează crearea de familii de obiecte conexe fără a le cupla strâns la clasele lor concrete în codul client. Acest lucru permite o extensibilitate ușoară și adăugarea de noi tipuri de produse sau fabrici în viitor fără a modifica codul client existent.

În continuare vom analiza care este scopul șablonului creațional de proiectare Prototype. Scopul de a specifica tipurile de obiecte de creat folosind o instanță prototip și creare deobiecte noi prin copierea acestui prototip.

Utilizăm acest șablon când:

- când clasele de instanțiat sunt specificate în timpul rulării, de exemplu, prin încărcare dinamică;
- pentru a evita construirea unei ierarhii de clasă a fabricilor care să fie paralelă cu clasa ierarhia produselor;
- când instanțe ale unei clase pot avea una dintre câteva combinații diferite de stat. Poate fi mai convenabil să instalați un număr corespunzător de prototipuri și clonează-le mai degrabă decât instanțiarea manuală a clasei, de fiecare dată cu starea corespunzătoare.

Vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.5:

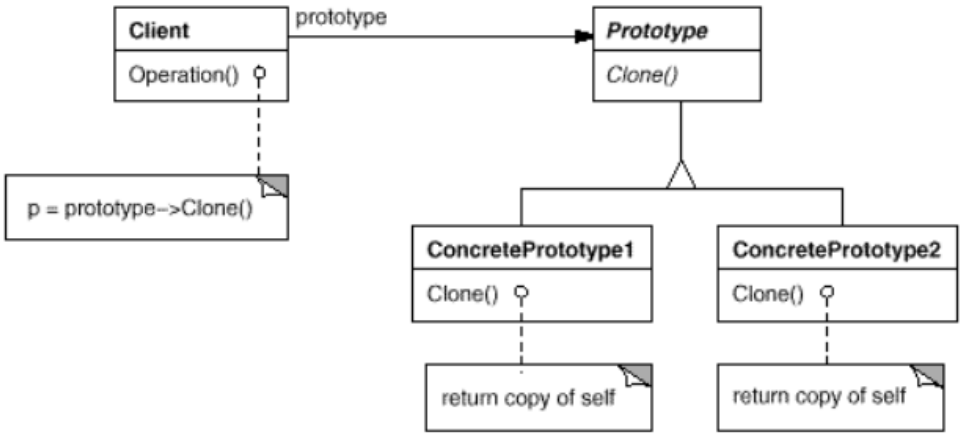

Figura 3.4 – Structura șablonului de proiectare Prototype

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- Prototype declară o interfață pentru clonarea în sine;
- ConcretePrototype implementează o operație de clonare în sine;
- Client creează un nou obiect solicitând unui prototip să se cloneze.

Beneficiile suplimentare ale modelului Prototype sunt enumerate mai jos.

- Adăugarea și eliminarea produselor în timpul rulării. Prototipurile vă permit să încorporați o nouă clasă de produse concrete într-un sistem pur și simplu prin înregistrarea a instanță prototip cu clientul. Este puțin mai flexibil decât altele modele de creație, deoarece un client poate instala și elimina prototipuri în timpul rulării.
- Specificarea de noi obiecte prin variarea valorilor. Sistemele foarte dinamice vă permit definiți un comportament nou prin compoziția obiectului — prin specificarea valorilor pentru variabilele unui obiect, de exemplu, și nu prin definirea unor noi clase. Tu definiți în mod eficient noi tipuri de obiecte prin instanțierea claselor existente și înregistrarea instanțelor ca prototipuri ale obiectelor client. Un client poate prezenta un comportament nou prin delegarea responsabilității prototipului. Acest tip de design permite utilizatorilor să definească noi „clase” fără programare. De fapt, clonarea unui prototip este similară cu instanțiarea unei clase.

- Subclasare redusă. Metoda fabricii produce adesea o ierarhie a Clase de creatori care sunt paralele cu ierarhia claselor de produse. Prototipul modelul vă permite să clonați un prototip în loc să solicitați o metodă din fabricăface un obiect nou. Prin urmare, nu aveți nevoie deloc de o ierarhie a clasei Creator.
- Configurarea unei aplicații cu clase în mod dinamic. Un timp de rulare mediile vă permit să încărcați dinamic clase într-o aplicație.

Prototype și Abstract Factory sunt modele concurente în anumite privințe, așa cum noi discutați la sfârșitul acestui capitol. Totuși, pot fi folosite împreună. Abstract Factory ar putea stoca un set de prototipuri din care să cloneze și să se întoarcă obiecte de produs.Modele care folosesc intens modelele Compozite și Decorator. adesea pot beneficia și de Prototype.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată.

În cazul în care cineva dorește aceleași produse ca și tine, folosim sablonul de proiectare Prototype pentru clonarea unui obiect existent (comandă) în loc de crearea unuia nou. Orice clasă care implementează interfața Prototype ar trebui să implementeze metoda clone() și astfel obiectul poate fi clonat. În următoarea imagine este reprezentat diagrama de clasă pentru sablonul de proiectare Prototype diagrama de clasă:

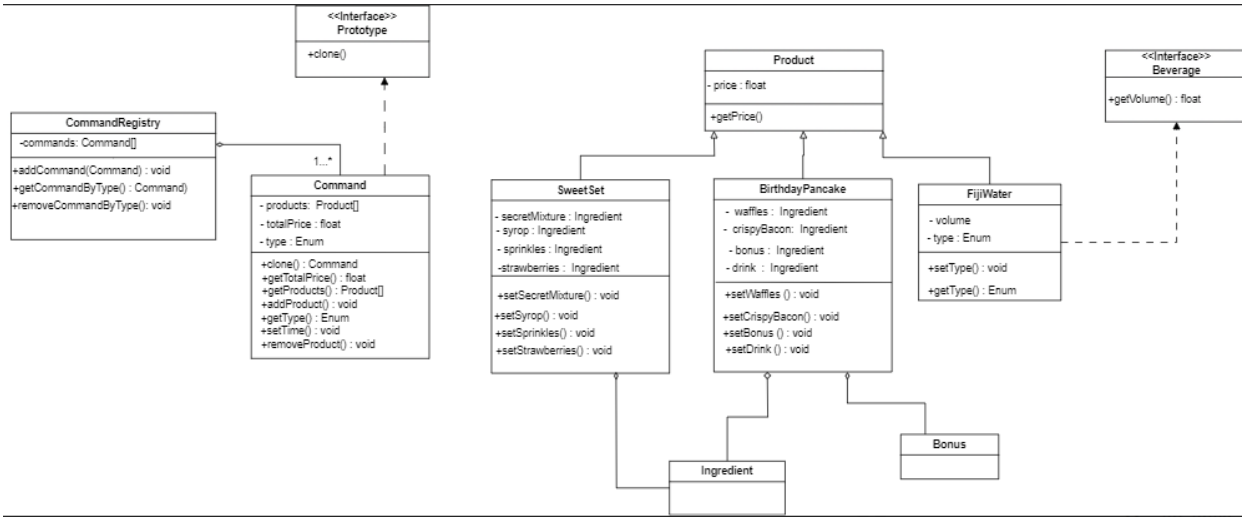

Figura 3.5 – Diagrama șablonului Prototype

Paricipanții la nivel de acească diagramă sunt reprezentate prin agregare,dependență și moștenire:

- interfața Prototype este participant ce definește interfața pentru clonarea obiectelor. Rolul fiind de a declara metoda clone() care trebuie implementată de clasele care susțin clonarea;
- clasa Command este un participant de prototip concret.Are rolul de a reprezinta obiectul complex care trebuie clonat;
- Clasa CommandRegistry are rolul de participant drept client. Și acționează ca un registru pentru gestionarea comenzilor;
- clasa Product este participant componentă. Reprezintă clasa de bază pentru implementările specifice de produse;
- clasa BirthdayPancake are rolul de produs concret;
- clasa FijiWater are rolul de produs concret. Reprezintă un tip specific de produs (Fiji Water);
- clasa SweetSet este la fel un produs concret. Reprezintă un tip specific de produs (Sweet Set);
- clasa Ingrediente participantul ce reprezintă obiectele de ingrediente utilizate în produse;
- clasa Bonus reprezintă obiectele de elemente bonus care pot face parte din Sweet Set.

În rezumat, participanții în fiecare clasă sau interfață sunt definiți pe baza rolurilor lor în cadrul sablonului Prototype. Clasa Prototype (Command) reprezintă obiectul complex care trebuie clonat, în timp ce Produsele Concretele (BirthdayPancake, FijiWater, SweetSet) reprezintă tipuri specifice de produse. CommandRegistry acționează ca un client și gestionează comenzile, iar clasele Ingredient și Bonus reprezintă ingredientele și elementele bonus utilizate în produse.

În continuare vom realiza o explicație a fiecărei clase și a rolului său în cadrul șablonului prototype:

Scopul șablonului de proiectare Prototype în această implementare este de a permite crearea de copii ale obiectelor existente, în loc de a crea noi obiecte, atunci când este necesar unul similar. Prin clonarea obiectelor, putem evita crearea unui obiect de la zero și putem economisi timp și resurse.

Șablonul de proiectare Prototype este folosit în următoarele moduri în implementarea dată:

- interfața `Prototype` definește metoda `clone()` pe care orice clasă care o implementează trebuie să o ofere;

- în acest caz, clasa `Command` implementează `Prototype` și, prin urmare, trebuie să ofere o implementare a metodei `clone()`;

- clasa `Command` este obiectul complex pe care dorim să îl clonăm.

- prin implementarea interfeței `Prototype` și metodei `clone()`, putem crea o copie a obiectului `Command`;

- clonarea se realizează prin utilizarea unui constructor de copiere care primește un obiect `Command` existent și creează un nou obiect `Command` cu aceleași caracteristici;

- acest lucru permite crearea unor copii independente ale comenzilor existente;

- clasa `Command` este utilizată în `CommandRegistry`, care păstrează un registru de comenzi;

- în loc să creeze noi obiecte de tip `Command` de fiecare dată când este necesară o comandă, putem utiliza clonarea pentru a obține copii ale comenzilor existente din registru;

- aceasta economisește timp și resurse, deoarece nu mai este nevoie să se realizeze operațiuni costisitoare pentru crearea unei comenzi noi;

În concluzie, utilizarea sablonului de proiectare Prototype în implementarea dată permite clonarea obiectelor existente pentru a crea copii independente. Aceasta oferă avantaje precum economisirea timpului și resurselor, evitând nevoia de a crea obiecte noi de la zero și oferind posibilitatea de a obține copii ale obiectelor existente într-un mod eficient și simplu.

Pentru început vom analiza care este scopul șablonului creațional de proiectare Builder. Scopul e de a separa construcția unui obiect complex de reprezentarea acestuia astfel încât același proces de construcție poate crea reprezentări diferite.

Utilizăm acest șablon când:

- algoritmul pentru crearea unui obiect complex ar trebui să fie independent de piesele care alcătuiesc obiectul și modul în care sunt asamblate.

- procesul de construcţie trebuie să permită reprezentări diferite pentruobiectul care este construit.

În continuare vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.7:

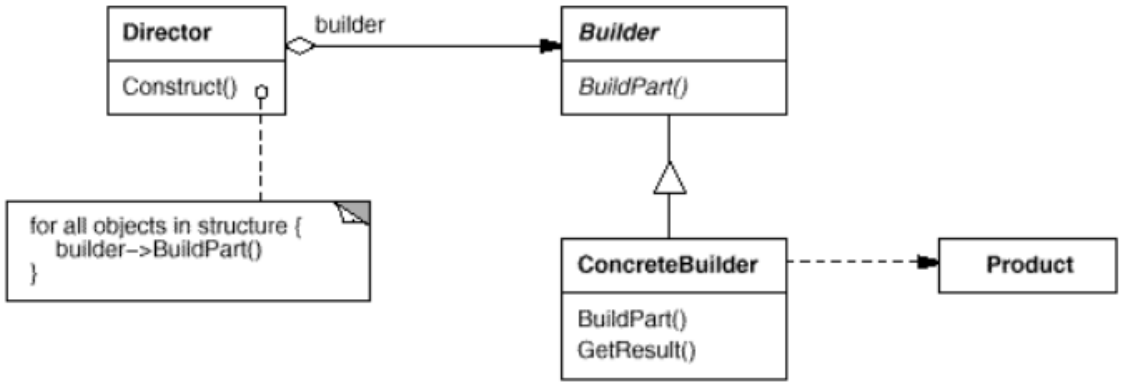

Figura 3.6 – Structura șablonului de proiectare Builder

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- builder specifică o interfață abstractă pentru crearea părților unui Produs obiect;
- concreteBuilder construiește și asamblează părți ale produsului prin implementarea
- interfața Builder definește și ține evidența reprezentării pe care o creează și oferă o interfață pentru preluarea produsului;
- director construiește un obiect folosind interfața Builder;
- produs reprezintă obiectul complex în construcție. ConcreteBuilder construiește reprezentarea internă a produsului și definește procesul rin care este asamblat.Include clase care definesc părțile constitutive, inclusiv interfețe pentru asamblarea pieselor în rezultatul final.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată.

Pentru a realiza o comandă, am folosit modelul de builder, deoarece era necesar pentru a construi un obiect complex din obiecte simple folosind abordarea pas cu pas. Când vrei să mănânci, nu vei alege doar un cheesecake sau doar apă. Vom comanda și un cheesecake ca primă porție, și

poate o inghețată de ce nu ceva de băut. Deci, modelul builder sugerează să extrageți codul de construcție a obiectului din propria sa clasă și să îl mutați în obiecte separate numite constructori. Modelul organizează construcția obiectului într-un set de pași (buildCheesecake, buildIceCream, buildBiscuits etc.).

În continuare avem structura, sau mai bine zis implementarea la nivel de diagramă UML a sistemului pentru șablonul dat în care sun menționate toate relațiile care sunt pentru codul care a fost implementat și pentru o percepere maiușoară a logicii la nivel de implementare pentru orice persoană. Acesta reprezentând în figura 4.8 participanții și anume clasele și interfețele care depind și implementeaza unele de la altele.

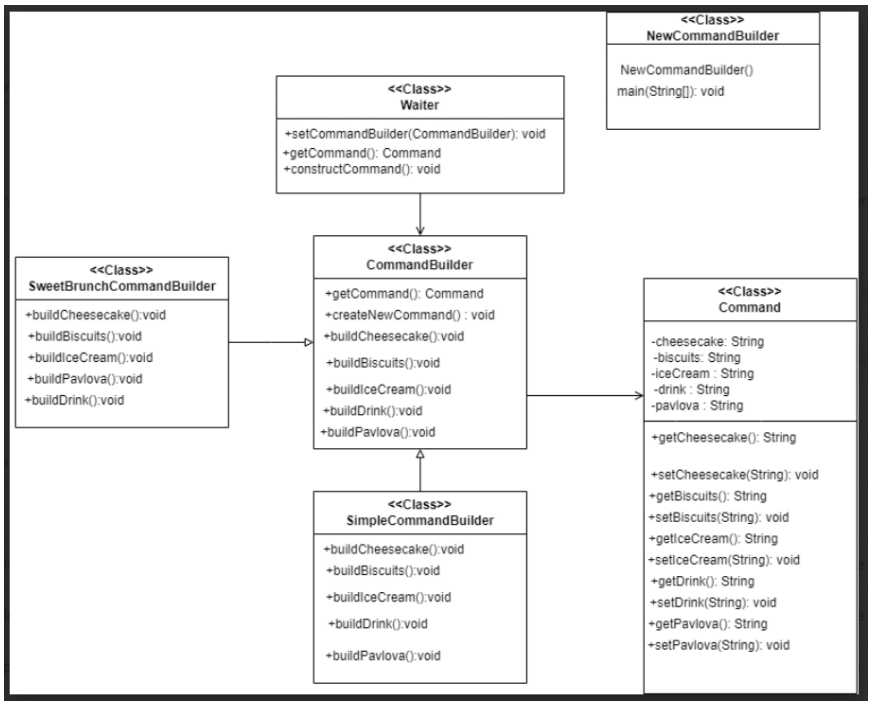

Figura 3.7 – Diagrama șablonului Builder

Paricipanții la nivel de acească diagramă sun reprezentate prin moștenire și asociere:

- clasaCommand este participantul Produsul;
- commandBuilder este constructorul;
- simpleCommandBuilder și SweetBrunchCommandBuilder sunt participanții la consturctorii concreți;
- clasa Waiter are rolul de Director;

- NewCommandBuilder are rolul de client.

În continuare vom realiza o explicație a fiecărei clase și a rolului său în cadrul modelului de fabrică abstractă:

Codul implementează șablonul de proiectare Builder în Java. Șablonul de proiectare Builder este un șablon de proiectare creațional care permite construirea de obiecte complexe pas cu pas. Acesta separă construcția unui obiect de reprezentarea acestuia, permițând același proces de construcție să creeze reprezentări diferit:

Avem produsul (clasa Command):

- această clasă reprezintă obiectul complex care este construit;

- conține mai multe proprietăți (cheesecake, biscuits, ice cream, drink, pavlova) și metodele corespunzătoare de getter și setter;

- metoda toString() este suprascrisă pentru a afișa valorile proprietăților.

Este constructorul abstract (clasa CommandBuilder):

- Această clasă definește interfața abstractă pentru crearea părților produsului.

- Conține o referință la produsul în curs de construire (Command).

- Furnizează metode pentru a crea o nouă comandă, precum și metode abstracte pentru construirea fiecărei părți a comenzii (cheesecake, biscuits, ice cream, drink, pavlova).

- Constructorii concreți moștenesc această clasă și furnizează implementarea pentru metodele abstracte.

Constructorii concreți (clasele SimpleCommandBuilder și SweetBrunchCommandBuilder):

- Aceste clase moștenesc CommandBuilder și furnizează implementarea pentru construirea diferitelor variații ale comenzii.

- Fiecare constructor concret suprascrie metodele abstracte pentru a seta valori specifice pentru fiecare proprietate a clasei Command.

Directorul (clasa Waiter):

- această clasă este responsabilă de gestionarea procesului de construcție.

- deține o referință la CommandBuilder.

- metoda setCommandBuilder() este utilizată pentru a seta constructorul specific care va fi utilizat.

- metoda constructCommand() inițiază procesul de construcție prin apelarea metodelor constructorului pentru a construi diferitele părți ale comenzii.

- metoda getCommand() recuperează comanda finală construită din constructor

Clientul (clasa NewCommandBuilder):

- Această clasă reprezintă codul client care utilizează șablonul de proiectare Builder.

- Demonstrează crearea diferitelor tipuri de comenzi folosind constructori diferiți (SweetBrunchCommandBuilder și SimpleCommandBuilder) prin intermediul unui Waiter.

În ansamblu, codul demonstrează cum este implementat șablonul de proiectare Builder prin utilizarea constructorilor abstracti și concreți pentru a construi obiecte complexe (Command) pas cu pas sub îndrumarea unui Director (Waiter).

3.2 Șabloane structurale

Șabloanele structurale sunt preocupate de modul în care sunt compuse clasele și obiectele formează structuri mai mari. Modelele de clasă structurală folosesc moștenirea pentru a compune interfețe sau implementări. Ca exemplu simplu, luați în considerare cum moștenirea multiplă combină două sau mai multe clase într-una singură. Rezultatul este o clasă care combină proprietățile claselor sale părinte.

Acest model este deosebit de util pentru a face ca bibliotecile de clasă dezvoltate independent să lucreze împreună. Alt exemplu este forma de clasă a modelului Adaptor (157). În general, un adaptor face o interfață (adaptul) se conformează cu alta, oferind astfel o uniformitate abstractizarea diferitelor interfețe. Un adaptor de clasă realizează acest lucru prin moștenind în mod privat dintr-o clasă de adaptat. Adaptorul își exprimă apoi interfață în ceea ce privește adaptatul.

În loc să compună interfețe sau implementări, modele de obiecte structurale descrieți modalități de a compune obiecte pentru a realiza o nouă funcționalitate. Adăugat flexibilitatea compoziției obiectului provine din capacitatea de a schimba compoziția la run-time, ceea ce este imposibil cu compoziția statică a clasei.

În continuare vom analiza care este scopul șablonului de proiectare Facade. Are scopul de a atașa responsabilități suplimentare unui obiect în mod dinamic. Decoratorii oferă o alternativă flexibilă la subclasare pentru extinderea funcționalității.

Utilizăm acest șablon pentru ca:

- să adauge responsabilităţi obiectelor individuale în mod dinamic şi transparent, adică fără a afecta alte obiecte;
- pentru responsabilități care pot fi retrase;
- când extinderea prin subclasare este impracticabilă. Uneori un număr mare de sunt posibile extensii independente și ar produce o explozie de subclase pentru a sprijini fiecare combinație. Sau o definiție de clasă poate fi ascunsă sau altfel indisponibil pentru subclasare.

În următoarea reprezentare vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.9:

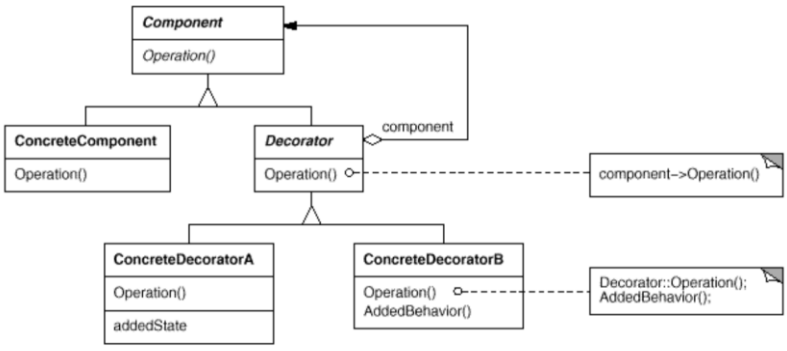

Figura 3.8 – Structura șablonului de proiectare Decorator

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- componenta definește interfața pentru obiectele care pot avea responsabilități adăugate acestora în mod dinamic;
- concreteComponent definește un obiect față de care pot fi responsabilități suplimentare atașat;
- decorator menține o referință la un obiect Component și definește o interfață care se conformează cu interfața componentei.
- concreteDecorator adaugă responsabilități la componentă.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată.

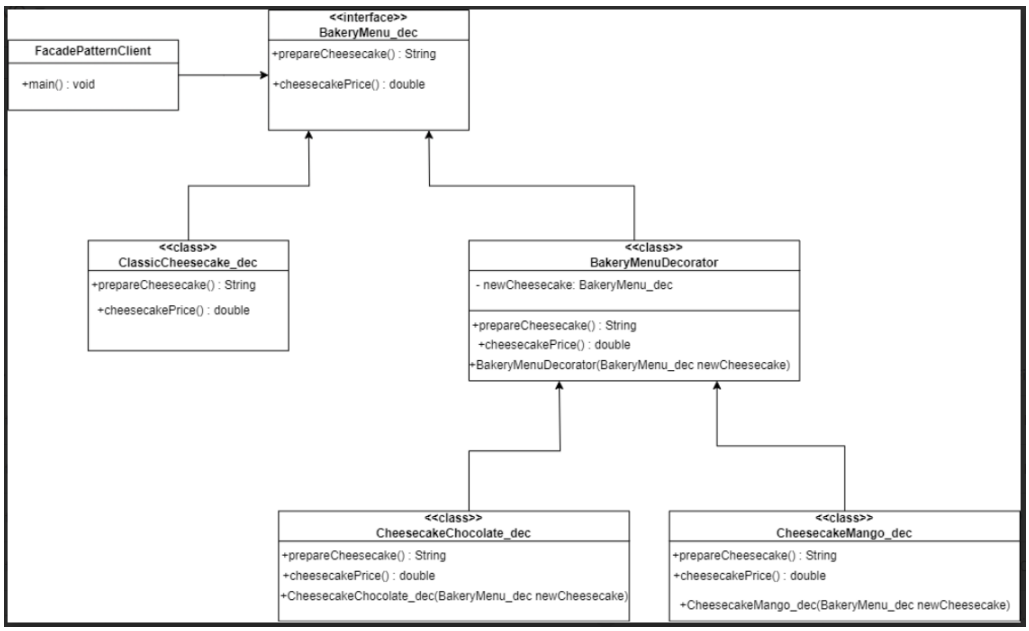

Figura 3.9 – Diagrama șablonului Decorator

Paricipanții la nivel de acesă diagramă sunt:

În urma codului furnizat, iată participanții din cadrul pattern-ului Decorator:

- componenta (BakeryMenu_dec): Aceasta este o interfață care definește operațiile ce pot fi decorate. În acest caz, furnizează metodele `prepareCheesecake()` și `cheesecakePrice()`;
- componenta concretă (CheesecakeClassic_dec): Această clasă implementează interfața `BakeryMenu_dec` și reprezintă componenta de bază ce poate fi decorată. Oferă

implementarea metodelor `prepareCheesecake()` și `cheesecakePrice()` specifice pentru cheesecake-ul clasic;

- decorator (BakeryMenuDecorator): Aceasta este o clasă abstractă care implementează interfața `BakeryMenu_dec`. Menține o referință către un obiect de tip `BakeryMenu_dec` și furnizează o interfață comună pentru toți decoratorii. Servește și ca clasă de bază pentru decoratorii concreți;

- decoratori concreți (CheesecakeChocolate_dec, CheesecakeMango_dec): Aceste clase extind clasa `BakeryMenuDecorator` și furnizează funcționalități suplimentare prin adăugarea de comportament propriu înainte sau după apelarea metodelor obiectului decorat. Suprascriu metodele `prepareCheesecake()` și `cheesecakePrice()` pentru a modifica comportamentul obiectului decorat.

În acest cod, pattern-ul Decorator este implementat prin utilizarea interfeței `BakeryMenu_dec` ca și componentă, clasa `BakeryMenuDecorator` ca decorator, și clasele `CheesecakeClassic_dec`, `CheesecakeChocolate_dec` și `CheesecakeMango_dec` ca și componente și decoratori concreți. Decoratorii adaugă funcționalități suplimentare (de exemplu, modificarea descrierii și prețului) componentei de bază (`CheesecakeClassic_dec`) prin delegarea către obiectul decorat și îmbunătățirea comportamentului acestuia.

Clasa `BakeryMenuDecorator` acționează ca decorator de bază și menține o referință către obiectul decorat. Decoratorii concreți (`CheesecakeChocolate_dec` și `CheesecakeMango_dec`) extind această clasă și suprascriu metodele sale pentru a oferi decorări specifice. Ei îmbunătățesc funcționalitatea obiectului `CheesecakeClassic_dec` prin adăugarea de descrieri și modificarea prețurilor.

Clasa `FacadePatternClient` servește drept client care interacționează cu decoratori și cu obiectul decorat prin intermediul interfeței `BakeryMenu_dec`, creând instanțe ale decoratorilor și utilizând comportamentul decorat oferit de aceștia.

În continuare vom analiza care este scopul șablonului creațional de proiectare facade. Scopul de a furniza o interfață unificată unui set de interfețe dintr-un subsistem. Fațada definește o interfață de nivel superior care face subsistemul mai ușor de utilizat.

Utilizăm acest șablon când:

- când doriți să oferiți o interfață simplă unui subsistem complex. Subsistemedevin adesea mai complexe pe măsură ce evoluează. Cele mai multe modele, atunci când sunt aplicate, rezultă în clase tot mai mici;
- când doriți să vă stratificați subsistemele. Utilizați o fațadă pentru a defini un punct de intrare la fiecare nivel de subsistem;
- la momentul când există multe dependențe între clienți și clasele de implementare a unei abstractii.

În continuare vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.11:

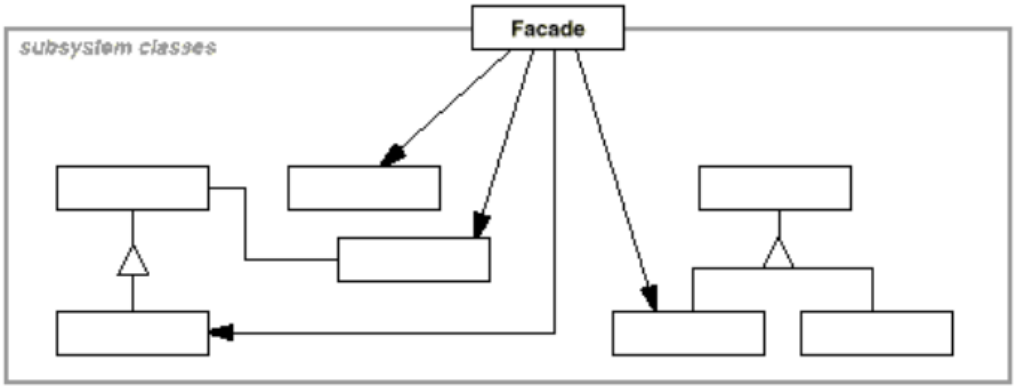

Figura 3.10 – Structura șablonului de proiectare Facade

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- fațada știe ce clase de subsistem sunt responsabile pentru o cerere și delegă cererile clientului către obiectele subsistemului corespunzătoare;
- clase de subsistem ajută la implementarea funcționalității subsistemului și gestionarea lucrărilor atribuite de obiectul Fațadă.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată.

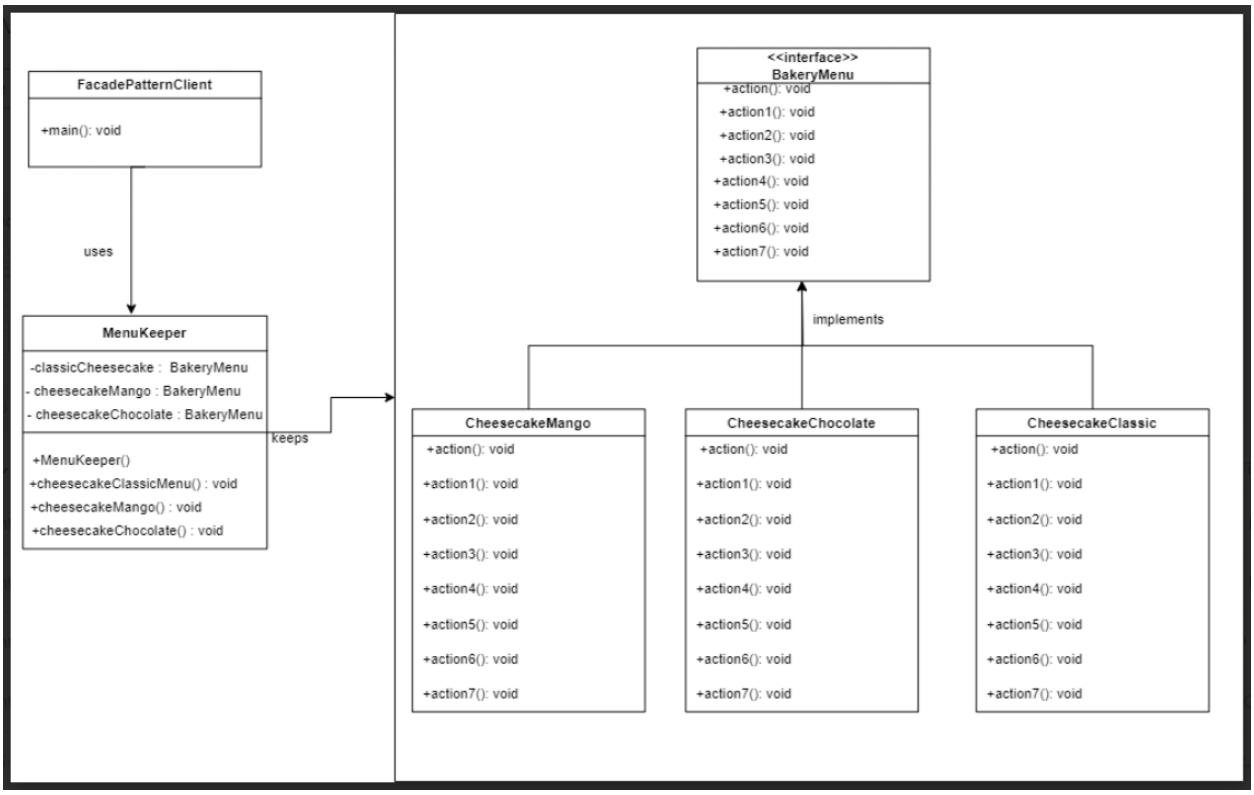

Figura 3.11 – Diagrama șablonului facade

Paricipanții la nivel de acească diagramă sunt:

- facade (MenuKeeper): Furnizează o interfață simplificată pentru client în vederea interacțiunii cu subsistemele (gusturile de cheesecake). Clasa MenuKeeper încapsulează interacțiunile complexe și expune metode de nivel superior pentru a selecta și executa acțiuni legate de diferitele gusturi de cheesecake;
- clasele subsistem (CheesecakeClassic, CheesecakeChocolate, CheesecakeMango): Aceste clase reprezintă gusturile individuale de cheesecake și implementează interfața BakeryMenu. Fiecare subclasă conține acțiuni specifice legate de gustul respectiv;
- Client (FacadePatternClient): Interacționează cu fațada (MenuKeeper) pentru a efectua operații asupra subsistemelor. Clasa FacadePatternClient prezintă un meniu utilizatorului și gestionează intrările utilizatorului pentru a executa acțiunile dorite.

În acest cod, design pattern-ul Facade este respectat deoarece clasa MenuKeeper acționează ca o fațadă prin furnizarea de metode simplificate (cheesecakeClassicMenu(), cheesecakeChocolateMenu(), cheesecakeMangoMenu()) pentru a accesa subsistemele (gusturi de

cheesecake). Clientul interacționează cu MenuKeeper pentru a selecta un gust de cheesecake și a executa acțiunile aferente. Clasa MenuKeeper ascunde complexitățile creării și interacțiunii cu diferitele gusturi de cheesecake și oferă o interfață unificată pentru client.

În continuare vom realiza o explicare pentru fiecare clasă. Pentu început se crează o interfață BakeryMenu. După care creăm o clasă de implementare ClassicCheesecake care va implementa interfața BakeryMenu. Avem o clasă de implementare CheesecakeChocolate și una cheesecakeMango, care va implementa interfața BakeryMenu. După care avem o clasă concretă MenuKeeper care va utiliza interfața BakeryMenu. La final, se creează un client care poate achiziționa produse de pe meniul BakeryMenu prin intermediul clasei MenuKeeper.

În loc să dezvoltăm codul să lucreze cu zeci de clase ale framework-ului direct, am creat o clasă facade care încapsulează acea funcționalitate și o ascunde restului codului. Această structură ajută și să minimizăm efortul de a face upgrade la versiuni viitoare ale framework-ului sau de a-l înlocui cu altul. Singurul lucru pe care ar trebui să-l schimbăm în aplicația ta ar fi implementarea metodelor facadei.

BakeryMenu oferă acces facil la o anumită parte a funcționalității subsistemului. Știe unde să redirecționeze cererea clientului și cum să opereze toate componentele mobile.

Se poate crea o clasă MenuKeeper pentru a evita încărcarea unei singure fasete cu caracteristici nerelevante care ar putea face din ea încă o structură complexă. Fasetele suplimentare pot fi utilizate atât de clienți, cât și de alte fasete.

Subsistemul complex este alcătuit din zeci de obiecte diferite, cum ar fi chocolateCheesecake, classicCheesecake, mangoCheesecake. Pentru a le face pe toate să facă ceva semnificativ, trebuie să ne aprofundăm în detalii de implementare ale subsistemului, cum ar fi inițializarea obiectelor în ordinea corectă și furnizarea datelor în formatul potrivit.

Clientul FacadePatternClient folosește fațada în loc să apeleze direct obiectele subsistemului.

3.3 Șabloane comportamentale

Șabloanele comportamentale sunt un set de sabloane de proiectare care se concentrează pe organizarea și gestionarea comportamentului și interacțiunii între obiecte și entități într-un sistem

software. Aceste sabloane abordează modul în care obiectele și componentele software cooperează și se comunică între ele pentru a realiza anumite funcționalități și comportamente.

Importanța șabloanelor comportamentale constă în faptul că oferă soluții reutilizabile și flexibile pentru gestionarea interacțiunilor dintre obiecte într-un mod structurat și eficient. Prin aplicarea acestor sabloane, dezvoltatorii pot obține următoarele beneficii:

Șabloanele comportamentale ajută la separarea responsabilităților și comportamentelor specifice în entități distincte, facilitând astfel structurarea clară și modularitatea sistemului.

Reutilizare și extensibilitate: Aceste sabloane permit reutilizarea codului existent și extinderea funcționalității fără a afecta structura de bază a sistemului. Aceasta facilitează dezvoltarea și întreținerea ulterioară a aplicațiilor software.

Prin utilizarea șabloanelor comportamentale, se poate adapta și configura comportamentul obiectelor în timpul execuției, permițând astfel sistemului să se adapteze la schimbările cerințelor și să ofere soluții flexibile.

Aceste șabloane promovează cuplarea redusă între obiecte, deoarece definește interfețe clare pentru interacțiunea și comunicarea între ele. Acest lucru face ca sistemele să fie mai flexibile și mai ușor de întreținut.

Prin separarea comportamentului în entități separate, teste unitare și teste de integrare pot fi efectuate mai ușor și mai eficient, permițând detectarea și rezolvarea erorilor mai rapid.

Unele exemple de șabloane comportamentale includ:

- strategy pattern ce permite definirea unei familii de algoritmi și încapsularea fiecărui algoritm într-o clasă separată, astfel încât să poată fi schimbați între ei în mod dinamic.

- observer patten care stabilește o relație unu-la-mulți între obiecte, astfel încât atunci când un obiect se schimbă, toate obiectele dependente sunt notificate automat și actualizate

- command pattern ce încapsulează o solicitare sub formă de obiect, permițând astfel parametrizarea clienților cu diferite solicitări și gestionarea acestora în mod flexibil.

- iterator pattern ce furnizează o modalitate de a accesa elementele unui obiect compus într-un mod secvențial, fără a expune structura internă a obiectului

- state patten ne permite unui obiect să-și schimbe comportamentul în funcție de starea internă și să păstreze aceste tranziții de stare încapsulate în obiecte separate.

Acestea sunt doar câteva exemple de șabloane comportamentale. Utilizarea corectă a acestor sabloane poate îmbunătăți structura, flexibilitatea și extensibilitatea aplicațiilor software, permițând dezvoltatorilor să scrie cod mai modular și mai ușor de întreținut.

Pentru început vom analiza care este scopul șablonului template method ce e folosit cand definiți scheletul unui algoritm într-o operație, amânând câțiva pași la subclasele. Metoda șablon permite subclaselor să redefinească anumiți pași ai unui algoritm fără a modifica structura algoritmului.

Utilizăm acest șablon când:

- pentru a implementa părțile invariante ale unui algoritm o dată și lăsați-l pe seama subclase pentru a implementa comportamentul care poate varia;
- când comportamentul comun între subclase ar trebui să fie factorizat și localizat;
- clasă comună pentru a evita duplicarea codului;

Mai întâi identificați diferențele în codul existent și apoi separați diferențele în operațiuni noi. În cele din urmă, înlocuiți codul diferit cu metoda șablon care apelează una dintre aceste noi operații.

- pentru a controla extensiile subclaselor. Puteți defini o metodă de șablon pe care o apelează, permițând astfelextensii numai în acele puncte.

În continuare vom analiza structura șablonului de proiectare, care este reprezentată în figura 4.13:

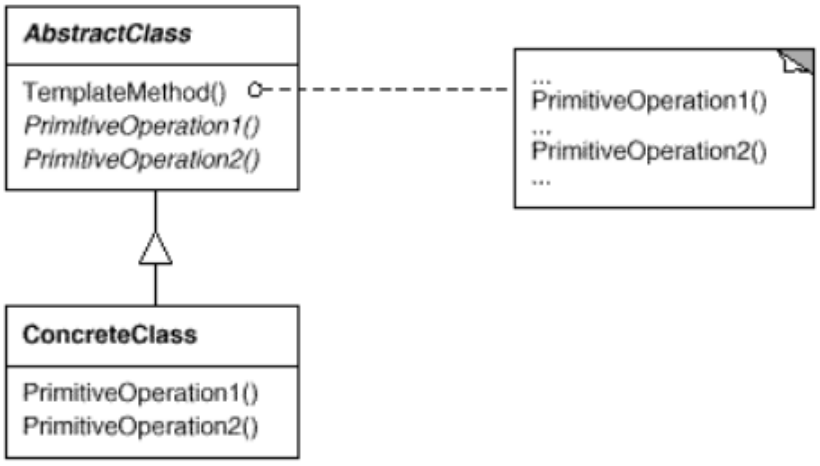

Figura 3.12 – Structura șablonului de proiectare Template Method

Participanții șablonului date de proiectare, care îl putem observa mai sus sunt următorii:

- AbstractClass definește operații primitive abstracte care concretizează subclasele definiți pentru a implementa pașii unui algoritm.Implementează o metodă șablon care definește scheletul unui algoritm.Metoda șablon apelează și operații primitive ca operațiuni definite în AbstractClass sau cele ale altor obiecte;
- ConcreteClass ce implementează operațiile primitive de a transporta.

Ca să avem o imagine mai clară asupra acestui pattern, el a fos implementat la nivel de cod în cadrul aplicației care a fost dezvoltată. Ca un utilizator poți alege un șablon pentru vizionarea meniului sub formă de linie sau unul grilă. Mai mult, poți alege dacă vrei să faci comanda pe local sau să o iei acasă. De asemenea, puteți alege opțiunea de ieșire, în cazul în care nu doriți să alegeți un șablon pentru meniu.

În continuare avem structura, sau mai bine zis implementarea la nivel de diagramă UML a sistemului pentru șablonul dat în care sun menționate toate relațiile care sunt pentru codul care a fost implementat și pentru o percepere maiușoară a logicii la nivel de implementare pentru orice persoană. Acesta reprezentând în figura 4.14 și figura 4.15 participanții și anume clasele și interfețele care depind și implementeaza unele de la altele.

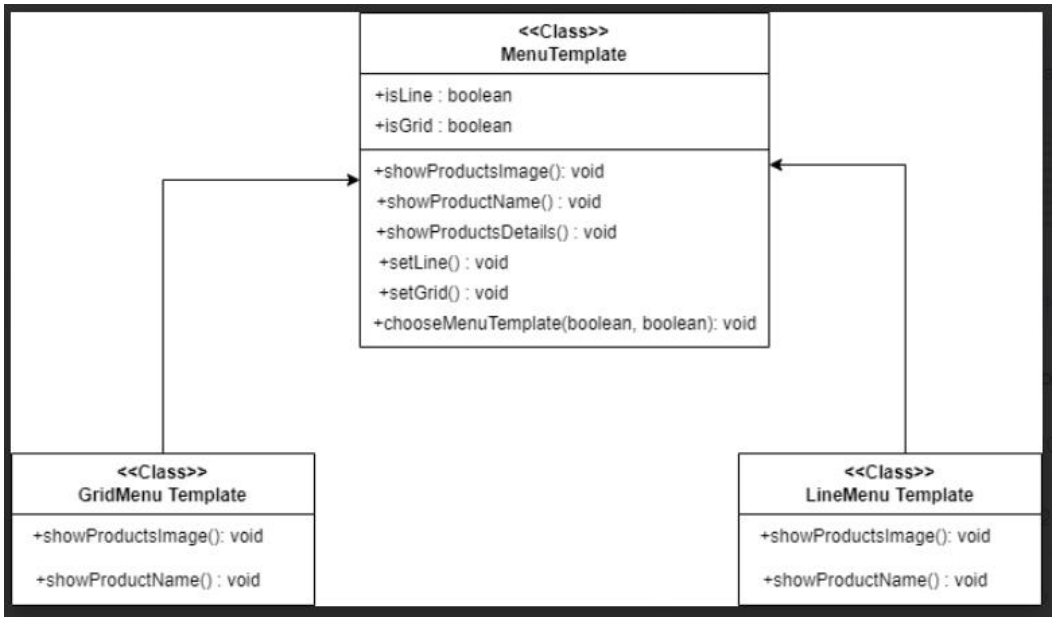

Figura 3.13 – Diagrama al doilea șablon Template Method

Paricipanții la nivel de acească diagramă sunt prin moștenire:

- MenuTemplate este clasa abstracta care ne definește template method;
- LineMenuTemplate și GridMenuTemplate sunt clase concrete.

Codul demonstrează implementarea design pattern-ului Template Method. Pattern-ul Template Method este un pattern de design comportamental care definește scheletul unei algoritmi într-o clasă de bază și permite subclaselor să ofere implementări specifice pentru anumite etape ale algoritmului.

În cod, clasa "MenuTemplate" reprezintă clasa abstractă de bază care definește metoda template "chooseMenuTemplate()". Această metodă acționează ca scheletul algoritmului pentru afișarea unui meniu cu diferite aspecte. Variabilele "isLine" și "isGrid" sunt utilizate pentru a determina tipul de template pentru meniu.

Metodele "showProductsImage()" și "showProductsName()" sunt metode abstracte care reprezintă etape în algoritm care trebuie implementate de subclase concrete. Aceste metode definesc modul în care imaginile și numele produselor sunt afișate.

Metoda "showProductsDetails()" este o metodă finală, ceea ce înseamnă că nu poate fi suprascrisă de subclase. Ea oferă o implementare implicită pentru afișarea detaliilor produsului, dar subclasele nu au permisiunea să schimbe acest comportament.

Metodele "chooseLine()" și "chooseGrid()" sunt, de asemenea, metode finale, oferind implementări implicite pentru selectarea aspectului cu linii sau cu grilă. Subclasele nu pot schimba acest comportament.

Clasele concrete "LineMenuTemplate" și "GridMenuTemplate" sunt subclase care extind clasa "MenuTemplate". Ele suprascriu metodele abstracte "showProductsImage()" și "showProductsName()" pentru a oferi implementări specifice pentru afișarea imaginilor și numelor, bazate pe template-ul ales.

În rezumat, pattern-ul Template Method permite subclaselor să definească anumite etape ale unui algoritm în timp ce menține structura generală a algoritmului în clasa de bază. Acesta promovează reutilizarea codului și flexibilitatea oferind o modalitate standard de implementare și modificare a părților unui algoritm.

Participanții la pattern-ul Template Method sunt:

- clasa abstractă ("MenuTemplate") reprezintă clasa abstractă de bază care definește metoda template și declară metode abstracte pe care subclasele trebuie să le implementeze;

- claseele concrete ("LineMenuTemplate" și "GridMenuTemplate") implementează metodele abstracte definite în clasa abstractă și oferă implementări specifice pentru afișarea imaginilor și a numelor, în funcție de template-ul ales;

- metoda template ("chooseMenuTemplate()") definește structura generală a algoritmului și ordinea etapelor de executat. Aceasta apelează metodele abstracte și metodele finale după nevoie pentru a executa algoritmul;

- metodele abstracte ("showProductsImage()" și "showProductsName()") care sunt declarate în clasa abstractă și implementate de către clasele concrete. Aceste metode reprezintă etape ale algoritmului care trebuie personalizate de către subclase;

- metodele finale ("showProductsDetails()", "chooseLine()" și "chooseGrid()") oferite de clasa abstractă cu o implementare implicită care nu poate fi suprascrisă de subclase. Aceste metode reprezintă etape ale algoritmului care sunt comune tuturor subclaselor și nu trebuie schimbate;

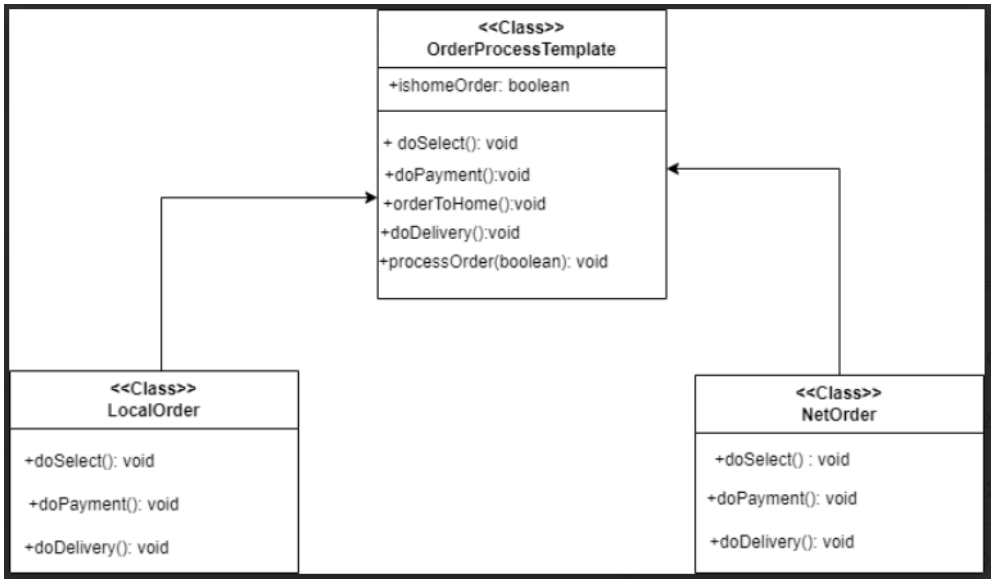

Figura 3.14 – Diagrama primului șablon Template Method

Paricipanții la nivel de acească diagramă sunt reprezentate prin moștenire:

- OrderProcessTemplate este clasa abstracta care ne definește template method;
- LocalOrder și NetOrder sunt clase concrete.

Codul demonstrează implementarea design pattern-ului Template Method. Pattern-ul Template Method este un pattern de design comportamental care definește scheletul unei algoritmi într-o clasă de bază și permite subclaselor să ofere implementări specifice pentru anumite etape ale algoritmului.

În cod, clasa "OrderProcessTemplate" reprezintă clasa abstractă de bază care definește metoda template "processOrder()". Această metodă acționează ca scheletul algoritmului pentru procesarea unei comenzi. Variabila "isOrder" este utilizată pentru a determina dacă comanda este plasată sau nu.

Metodele "doSelect()" și "doPayment()" sunt metode abstracte care reprezintă etape în algoritm care trebuie implementate de subclase concrete. Aceste metode definesc modul în care selecția produselor și plata sunt efectuate.

Metoda "orderWrap()" este o metodă finală, ceea ce înseamnă că nu poate fi suprascrisă de subclase. Ea oferă o implementare implicită pentru încheierea comenzii, dar subclasele nu au permisiunea să schimbe acest comportament.

Metoda "doDelivery()" este o metodă abstractă care trebuie implementată de subclasă și reprezintă etapa de livrare a comenzii.

Metoda "processOrder()" este o metodă finală care apelează etapele algoritmului în ordinea specificată. Aceasta apelează metodele "doSelect()", "doPayment()", "orderWrap()" (dacă comanda este plasată) și "doDelivery()".

Clasele concrete "NetOrder" și "LocalOrder" sunt subclase care extind clasa "OrderProcessTemplate". Ele suprascriu metodele abstracte pentru a oferi implementări specifice pentru selecția produselor, plata și livrarea, în funcție de tipul comenzii (comandă online sau comandă locală).

Putem afirma că pattern-ul Template Method permite subclaselor să definească anumite etape ale unui algoritm în timp ce menține structura generală a algoritmului în clasa de bază. Acesta promovează reutilizarea codului și flexibilitatea oferind o modalitate standard de implementare și modificare a părților unui algoritm.

4 Interfața grafică a aplicației

În aplicația furnizată, este utilizată o interfață grafică a consolei pentru a interacționa cu utilizatorul. Această interfață grafică a consolei oferă un set simplu de opțiuni pe care utilizatorul le poate selecta pentru a alege tipul de cheesecake dorit sau pentru a ieși din aplicație.

Utilizarea unei interfețe grafice în acest context aduce mai multă claritate și ușurință în utilizarea aplicației pentru utilizator. În loc să introducă manual cod sau date în linia de comandă, utilizatorul are acum posibilitatea de a interacționa cu un meniu vizual reprezentat în consolă. Aceasta reduce riscul de a introduce erori sau de a nu înțelege opțiunile disponibile.

În continuare avem reprezentată interfața grafică unde clientul își poate alege tipul de meniu în figura 4.1:

```
=========Choose Bakery's Menu Template============
            1. Line Template.
            2. Grid Template.
            3. Exit.
Enter your choice: 1
Set to line successfully
the images are displayed in form of lines
the name of the products are displayed near the image
the product details
```

Figura 4.1 – Meniul de selectare a template-ului

Interfața grafică a consolei oferă utilizatorului o experiență mai intuitivă și ușor de utilizat. Utilizatorul poate naviga prin meniu, selectând opțiunile dorite prin intermediul tastaturii. De aceea, în continuare avem opțiunea utilizatorului de a comanda pe loc sau la pachet produsele, reprezentat în figura 4.2:

```
========= Order Type ============
            1. Home.
            2. Local.
            3. Exit.
Enter your choice: 2
Customer chooses the item from menu at MC.
Pays at counter through cash/POS
Order done successfull
No delivery is required
```

Figura 4.2 – Meniul de selectare a tipului de comandă

Prin afișarea opțiunilor disponibile și cerând utilizatorului să selecteze opțiunea dorită, se reduce necesitatea de a memora sau de a introduce manual comenzi și argumente. Această abordare face ca interacțiunea cu aplicația să fie mai prietenoasă pentru utilizatorii neexperimentați sau pentru cei care nu sunt familiarizați cu linia de comandă. Interfața grafică a consolei oferă un nivel mai înalt de abstractizare și ascunde detaliile tehnice, permițând utilizatorului să se concentreze mai mult pe alegerea și interacțiunea cu opțiunile disponibile.

În figura 4.3 avem afișat meniul de procesare și prețul comenzii:

```
========= Bakery's Menu===============
            1. Classic cheesecake.
            2. Chocolate cheesecake.
            3. Mango cheesecake
            4. Exit.
Enter your choice: 3
Taking a pan...
Crushing biscuits and pressing them into the bottom of a pan....
Beating cream cheese until smooth and creamy...
Adding mango syrop iin the filling...
Pouring the cream cheese mixture over the crust in the pan....
Baking the cheesecake in a preheated oven at a specified temperature for a designated time...
Letting the cheesecake to cool...
Serving the cheesecake.
Consistence: This is Classic Cheesecake Without chocolate or classic filling.
The price is:
20.0
```

Figura 4.3 - Procesarea comenzii

Importanța unei interfețe grafice în acest context constă în faptul că face aplicația mai accesibilă și mai ușor de utilizat pentru diferiți utilizatori. Prin oferirea unui mediu interactiv și intuitiv, interfața grafică a consolei îmbunătățește experiența utilizatorului și crește eficiența interacțiunii cu aplicația. Utilizatorii pot naviga prin meniu și pot face alegeri rapid și fără efort suplimentar, oferindu-le o modalitate comodă de a interacționa cu funcționalitățile oferite de aplicație.

Utilizarea unei interfețe grafice a consolei în acest context aduce un nivel suplimentar de utilizabilitate și accesibilitate, permițând utilizatorilor să interacționeze cu aplicația într-un mod mai natural și mai intuitiv.

Concluzii

În concluzie, dezvoltarea unei aplicații cu intenția de a simplifica procesul de comandă la o brutărie a reușit să utilizeze cu succes mai multe șabloane de proiectare, inclusiv șabloanele Abstract Factory, Builder, Prototype, Template Method, Facade și Decorator. Aceste șabloane de proiectare au îmbunătățit semnificativ funcționalitatea și experiența utilizatorului în ceea ce privește plasarea comenzilor la brutărie, fără a mai fi nevoie să aștepte în rânduri lungi.

Șablonul Abstract Factory a fost implementat pentru a furniza o modalitate de a crea familii de produse de brutărie conexe, cum ar fi prăjituri, torturi și pâine, printr-o interfață comună. Acest lucru permite aplicației să gestioneze cu ușurință diferite tipuri de produse de brutărie și asigură crearea coerentă și unitară a produselor conexe.

Șablonul Builder a fost utilizat pentru a construi comenzi complexe de produse de brutărie pas cu pas, oferind o abordare flexibilă și personalizabilă. Clientii pot alege dintr-o varietate de produse de brutărie, specifica cantități, adăuga personalizări și selecta opțiuni de livrare sau ridicare. Șablonul Builder simplifică construcția acestor comenzi, asigurându-se că procesul este structurat și ușor de urmărit.

Șablonul Prototype a fost utilizat pentru a permite aplicației să creeze și să gestioneze eficient mai multe instanțe ale produselor de brutărie. Prin utilizarea prototipurilor, aplicația poate clona și personaliza instanțe existente de produse, reducând costurile de creare a unor obiecte noi. Acest șablon contribuie la îmbunătățirea performanței și la reducerea consumului de memorie.

Șablonul Template Method a fost integrat în fluxul de lucru al aplicației de comandă. Acesta definește scheletul procesului de comandă, permițând implementarea pașilor specifici de către subclase. Acest șablon asigură coerența procesului de comandă pentru diferite tipuri de produse de brutărie, în timp ce permite personalizarea în funcție de cerințe specifice sau variații ale produselor.

Șablonul Facade a fost implementat pentru a oferi o interfață simplificată care ascunde complexitatea sistemului de comandă subiacent. Prin încapsularea funcționalității interne a aplicației și furnizarea unei interfețe unificate, șablonul Facade permite clienților să navigheze fără efort prin procesul de comandă, făcându-l mai intuitiv și prietenos utilizatorului.

În cele din urmă, șablonul Decorator a fost aplicat pentru a îmbunătăți produsele de brutărie cu caracteristici sau personalizări suplimentare. Clienții pot alege să adauge ingrediente, decoruri sau modificări suplimentare la comenzile lor de produse de brutărie, care sunt integrate fără probleme în produsul final. Șablonul Decorator permite modificări dinamice și flexibile fără a afecta structura de bază a produselor de brutărie.

În ansamblu, combinația acestor șabloane de proiectare a contribuit în mare măsură la succesul aplicației de comandă de la brutărie, oferind o soluție robustă, scalabilă și ușor de utilizat. Clienții pot plasa acum comenzi în mod convenabil și eficient, evitând cozi lungi și bucurându-se de o experiență de comandă fără probleme. Șabloanele de proiectare utilizate nu numai că au îmbunătățit funcționalitatea aplicației, dar au și facilitat întreținerea, extensibilitatea și dezvoltarea ulterioară a noi funcționalități.

În concluzie, utilizarea și implementarea șabloanelor de proiectare în aplicația de comandă la brutărie au demonstrat utilitatea și importanța acestor concepte în dezvoltarea software.Prin utilizarea șabloanelor de proiectare, cum ar fi Fabrică Abstractă, Builder, Prototype, Template Method, Facade și Decorator, aplicația a obținut numeroase avantaje și beneficii. Acestea includ:

Șabloanele de proiectare oferă o structură bine definită și coerență în ceea ce privește arhitectura aplicației. Aceasta facilitează înțelegerea, dezvoltarea și întreținerea sistemului, permițând echipei de dezvoltare să lucreze într-un mod mai organizat și eficient.Ne permit reutilizarea codului și componentelor existente, reducând timpul și efortul necesare pentru dezvoltarea de funcționalități similare. Acestea oferă, de asemenea, o modalitate simplă de a extinde aplicația cu noi caracteristici sau comportamente, fără a afecta structura și funcționalitatea existente. Ele oferă un cadru flexibil și modular care poate fi ajustat în mod ușor pentru a satisface nevoile variate ale clienților.

În concluzie, utilizarea și implementarea șabloanelor de proiectare în aplicația de comandă la brutărie au adus numeroase beneficii și au contribuit la crearea unei aplicații robuste, scalabile și ușor de utilizat. Acestea au facilitat dezvoltarea unui sistem coerent, flexibil și extensibil, care îmbunătățește semnificativ experiența utilizatorului și eficiența procesului de comandă.

Bibliografie

1. Freeman, E., & Freeman, E. (2004). Head First Design Patterns: A Brain-Friendly Guide. O'Reilly Media, Inc.
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
3. Gang of Four. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
4. Design Patterns[citat 16.05.23]. Disponibil: https://refactoring.guru/design-patterns
5. Design Pattttterns Java[citat 25.05.23]. Disponibil:https://www.javatpoint.com/design-patterns-in-java
6. Repozitoriul github : https://github.com/mariareee/TMPS_PROJECT