Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

**Proiect de curs**

**Disciplina: Tehnici și metode de proiectare a sistemelor**

**Tema: Proiectarea unei aplicatii de restaurant online de pizza**

**Verificat de: Gaidau Mihai , asist. univ.**

**Executant: Curcubet Ecaterina/, gr. TI-204.**

Chișinău – 2023

Cuprins

[**Introducere** 3](#_Toc137076603)

[**1. Consideratii Teoretice** 4](#_Toc137076604)

[**1.1.** **Principiile SOLID** 4](#_Toc137076605)

[**1.2.** **Modelele de proiectare creational** 7](#_Toc137076606)

[**1.2.1.** **Factory Method** 8](#_Toc137076607)

[**1.2.2.** **Abstract Factory** 9](#_Toc137076608)

[**1.2.3.** **Singleton** 10](#_Toc137076609)

[**1.3.** **Modelele de proiectare structural** 11](#_Toc137076610)

[**1.3.1. Adapter** 12](#_Toc137076611)

[**1.3.2. Decorator** 13](#_Toc137076612)

[**1.3.3. Composite** 14](#_Toc137076613)

[**1.4.** **Modelele de proiectare comportamentale** 15](#_Toc137076614)

[**2. Implementarea modelelor de proiectare intr-un proiect real** 17](#_Toc137076615)

[**2.1.** **Descrierea proiectului** 17](#_Toc137076616)

[**2.2.** **Creational Design Patterns** 18](#_Toc137076617)

[**2.3.** **Structural Design Patterns** 21](#_Toc137076618)

[**2.4.** **Behavioral Design Patterns** 25](#_Toc137076619)

[**Concluzie** 28](#_Toc137076620)

[**Bibliografie** 29](#_Toc137076621)

# **Introducere**

Proiectarea sistemelor reprezintă un aspect esențial în dezvoltarea aplicațiilor software moderne. În cadrul disciplinei "Tehnici și metode de proiectare a sistemelor", am explorat principiile SOLID și modelele de proiectare creational, structural și behavioral, care oferă un cadru solid pentru proiectarea și implementarea sistemelor software robuste, flexibile și ușor de întreținut.

Principiile SOLID, dezvoltate de către Robert C. Martin, reprezintă un set de principii de proiectare care promovează modularitatea, extensibilitatea și reutilizabilitatea în proiectarea sistemelor software. Aceste principii - Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) și Dependency Inversion Principle (DIP) - ne ghidează în îndeplinirea obiectivelor proiectării sistemelor.

Pe lângă principiile SOLID, am explorat și modelele de proiectare creational, care ne ajută să creăm obiecte într-un mod flexibil și eficient, modelele de proiectare structural, care facilitează organizarea și compunerea diferitelor elemente ale sistemului, și modelele de proiectare behavioral, care gestionează interacțiunea între obiectele din cadrul sistemului.

În cadrul acestui raport, ne propunem să aplicăm aceste cunoștințe teoretice în modelarea și proiectarea unei aplicații practice. Vom explora cerințele sistemului, vom analiza diferitele modele de proiectare și vom prezenta soluțiile noastre implementate în cadrul acestor modele. De asemenea, vom evalua avantajele și limitările abordărilor utilizate și vom discuta posibile îmbunătățiri și extensii pentru viitoare dezvoltări.

Prin urmare, acest raport va ilustra aplicarea principiilor SOLID și a modelelor de proiectare creational, structural și behavioral în proiectarea unui sistem software, evidențiind beneficiile aduse de acestea în dezvoltarea aplicațiilor complexe și scalabile.

Continuarea raportului va acoperi detaliile teoretice și practice ale proiectului nostru, modelarea și proiectarea aplicației, concluziile obținute și bibliografia utilizată.

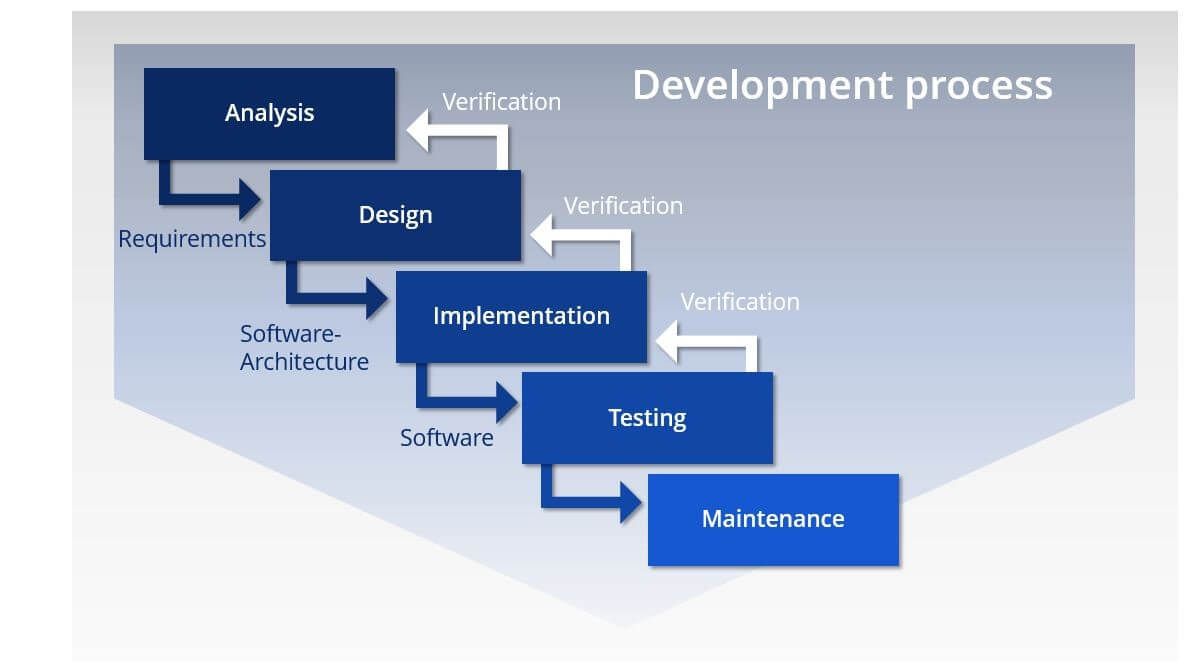


Figura 1 – process de proiectare

4.

# **1. Consideratii Teoretice**

În această secțiune, vom explora principiile SOLID și modelele de proiectare creational, structural și behavioral, care reprezintă fundația teoretică a proiectării sistemelor software eficiente și scalabile.

* 1. **Principiile SOLID**

**Single Responsibility Principle** (SRP), cunoscut și sub denumirea de "un singur motiv pentru a fi modificată", este un principiu fundamental în proiectarea sistemelor software. Acest principiu afirmă că o clasă ar trebui să aibă o singură responsabilitate și să fie responsabilă pentru un singur aspect al sistemului. Prin separarea funcționalității în clase distincte, putem obține un sistem modular, ușor de înțeles, întreținut și extins.

Aplicarea SRP în proiectarea sistemelor implică identificarea responsabilităților clare și bine definite ale fiecărei clase. O clasă ar trebui să se concentreze exclusiv pe o anumită funcționalitate și să nu fie încărcată cu responsabilități suplimentare. Aceasta ajută la crearea unor componente coezive și reutilizabile.

Un exemplu simplu ar fi un sistem de gestionare a unei librării online. Putem avea o clasă "Book" responsabilă exclusiv de manipularea informațiilor legate de cărți, cum ar fi autorul, titlul și ISBN-ul. În același timp, putem avea o altă clasă "Order" responsabilă de gestionarea procesului de plasare a comenzilor și de calcularea prețului total. Prin separarea acestor funcționalități în clase distincte, putem evita încărcarea clasei "Book" cu responsabilități care nu-i aparțin și putem menține coerența și extensibilitatea sistemului.

Beneficiile aplicării SRP sunt multiple. Prin definirea clară a responsabilităților, putem realiza o mai bună înțelegere a codului, o mai mare flexibilitate în modificări și întreținere și posibilitatea de a reutiliza clasele în alte contexte. De asemenea, SRP facilitează testarea unitară, deoarece clasele cu o singură responsabilitate pot fi testate mai ușor și mai independent.

În proiectul nostru la disciplina "Tehnici și metode de proiectare a sistemelor", vom aplica SRP pentru a asigura o proiectare coezivă și modulară a sistemului nostru. Vom identifica și vom separa responsabilitățile, astfel încât fiecare clasă să îndeplinească un singur rol și să fie ușor de înțeles și întreținut.

**Open-Closed Principle** este unul dintre principiile SOLID și are ca obiectiv principal crearea de sisteme software care să fie deschise pentru extensii, dar închise pentru modificări. Acest principiu promovează o arhitectură software flexibilă și ușor de întreținut, în care modificările în funcționalitatea existentă sunt minimizate, iar extensibilitatea este încurajată.

OCP se concentrează pe separarea logicii stabile și neschimbate de logica variabilă, astfel încât o schimbare într-o anumită funcționalitate să nu necesite modificări directe ale codului deja existent. În locul acestora, OCP încurajează extinderea codului prin adăugarea de module, clase sau metode noi care să ofere funcționalitatea adițională dorită.

Pentru a aplica OCP în proiectarea sistemelor, există câteva modalități:

1. Utilizarea de interfețe: Definirea interfețelor pentru componente sau module specifice, astfel încât acestea să poată fi implementate în diferite moduri, fără a modifica codul existent. Aceasta permite adăugarea de implementări noi prin intermediul polimorfismului și reutilizarea interfețelor deja definite.
2. Utilizarea de clase abstracte: Definirea claselor abstracte care conțin funcționalitate comună și stabilă, dar permit extinderea prin implementarea metodelor abstracte în clasele derivate. Astfel, funcționalitatea existentă poate rămâne neschimbată, iar noi funcționalități pot fi adăugate prin clasele derivate.
3. Utilizarea de pattern-uri de proiectare: Există anumite pattern-uri de proiectare care facilitează aplicarea OCP, cum ar fi Factory Method, Abstract Factory și Strategy. Aceste pattern-uri promovează deschiderea pentru extensii prin intermediul interfețelor și a claselor abstracte, în timp ce mențin închiderea pentru modificări prin adăugarea de implementări noi.

Prin aplicarea OCP în proiectarea sistemelor, obținem o arhitectură modulară și ușor de întreținut. Schimbările în funcționalitatea existentă sunt minimizate, iar adăugarea de noi funcționalități se realizează prin extinderea codului existent. Acest lucru duce la un sistem mai flexibil, mai ușor de testat și de extins în viitor.

**Liskov Substitution Principle** este unul dintre principiile SOLID și are ca obiectiv principal asigurarea unei relații corecte și coerente între clasele derivate și clasele lor de bază. Acest principiu definește că obiectele unei clase derivate trebuie să poată fi înlocuite cu obiecte ale clasei lor de bază, fără a afecta corectitudinea și comportamentul programului.

LSP subliniază importanța respectării contractului definit de clasa de bază în ceea ce privește semnăturile metodelor, precondițiile, postcondițiile și invariantele. În cazul în care o clasă derivată nu respectă aceste contracte și introduce comportamente neașteptate sau violă restricțiile clasei de bază, atunci LSP este încălcat.

Pentru a aplica LSP în proiectarea sistemelor, trebuie să ne asigurăm că relația de substituție între clasele derivate și clasele lor de bază este corectă și fără efecte secundare nedorite. Acest lucru poate fi realizat prin respectarea următoarelor principii:

1. Respectarea semnăturilor metodelor: Metodele în clasele derivate trebuie să respecte semnăturile definite în clasele de bază. Aceasta înseamnă că tipurile de date ale parametrilor și tipurile de date returnate trebuie să fie aceleași sau să fie subtipuri ale tipurilor definite în clasa de bază.
2. Respectarea precondițiilor și postcondițiilor: Clasele derivate trebuie să respecte precondițiile și postcondițiile definite în clasele de bază. Aceasta înseamnă că clasele derivate nu trebuie să introducă precondiții mai stricte sau postcondiții mai slabe decât cele definite în clasa de bază.
3. Respectarea invariantele: Clasele derivate trebuie să respecte invariantele definite în clasele de bază. Aceasta înseamnă că invariantele clasei de bază trebuie să rămână valabile și pentru obiectele claselor derivate.

Prin respectarea Liskov Substitution Principle, obținem o ierarhie de clase coerentă și ușor de înțeles. Obiectele claselor derivate pot fi utilizate în locul obiectelor claselor de bază, fără a afecta comportamentul corect al sistemului.

**Interface Segregation Principle** este unul dintre principiile SOLID și are ca obiectiv principal separarea interfețelor în componente mai mici și specializate, astfel încât utilizatorii să nu fie obligați să dependă de metode pe care nu le folosesc. Acest principiu promovează construirea de interfețe specifice pentru clienții lor și evitarea dependenței de interfețe generale sau monolitice.

ISP se bazează pe ideea că clienții nu ar trebui să fie forțați să dependă de metode pe care nu le folosesc. Astfel, interfețele ar trebui să fie proiectate în funcție de nevoile specifice ale clienților și să conțină doar metodele relevante pentru aceștia.

Există câteva principii de bază și motivații ale ISP:

1. Coeziune ridicată: Interfețele trebuie să fie coezive și să conțină doar metodele strâns legate între ele. Acest lucru duce la o înțelegere mai clară și mai simplă a interfeței și facilitează reutilizarea și extensibilitatea componentelor.
2. Cuplare redusă: Prin separarea interfețelor în componente specializate, se reduce cuplarea între acestea. Acest lucru face ca schimbările într-o interfață să nu afecteze și alte componente care nu depind de metodele respective.
3. Evitarea dependențelor nedorite: ISP evită situațiile în care un client este forțat să depindă de metode pe care nu le utilizează. Astfel, clienții pot fi izolați și pot avea interfețe specifice, adaptate nevoilor lor.

ISP poate fi implementat în proiectarea sistemelor prin:

1. Divizarea interfețelor mari în interfețe mai mici și specializate: Aceasta permite clienților să utilizeze doar metodele relevante pentru ei și să evite dependența de funcționalități inutile.
2. Crearea de interfețe specifice pentru clienți: În loc să avem o singură interfață generală pentru toți clienții, putem defini interfețe separate, adaptate nevoilor specifice ale fiecărui client.

Implementarea corectă a ISP aduce numeroase avantaje în proiectarea sistemelor. Acestea includ: reducerea dependențelor inutile între componente, creșterea coeziunii și modularității, ușurarea reutilizării și extensibilității, și facilitarea testării și întreținerii sistemului.

**Dependency Inversion Principle** este unul dintre principiile SOLID și are ca obiectiv principal inversarea dependențelor între modulele și componentele sistemului. Acest principiu promovează construirea de module independente, care depind de abstracțiuni, nu de implementări concrete. Astfel, se realizează o deschidere pentru extensibilitate și reutilizare, iar legăturile strânse între module sunt evitate.

DIP se bazează pe două aspecte fundamentale:

1. Modulele de nivel superior nu ar trebui să depindă direct de modulele de nivel inferior, ci ambele tipuri de module ar trebui să depindă de abstracțiuni. Aceasta înseamnă că modulele de nivel superior trebuie să depindă de interfețe sau clase abstracte, în loc să depindă de implementări concrete.
2. Detaliile de implementare ar trebui să depindă de abstracțiuni, nu invers. Acest lucru înseamnă că modulele concrete ar trebui să depindă de interfețe sau clase abstracte, iar detaliile de implementare specifice să fie ascunse și izolate în interiorul modulelor respective.

Exemple practice de aplicare a Dependency Inversion Principle în proiectarea sistemelor includ:

1. Utilizarea injecției de dependențe (Dependency Injection): Prin injectarea dependențelor prin intermediul constructorilor, metodelor sau proprietăților, modulele de nivel superior primesc obiecte concrete prin intermediul abstracțiunilor. Astfel, se realizează o inversare a dependențelor și se facilitează testarea și extensibilitatea sistemului.
2. Utilizarea inversării controlului (Inversion of Control): Prin utilizarea unor framework-uri sau containere IoC, putem externaliza gestionarea dependențelor către un container specializat. Astfel, modulele de nivel superior nu mai trebuie să creeze direct obiectele dependente, ci le pot obține prin intermediul containerului.
3. Crearea de interfețe și clase abstracte: Definirea interfețelor și claselor abstracte pentru a reprezenta abstracțiunile pe care modulele depind, iar modulele concrete să implementeze aceste abstracțiuni. Astfel, modulele de nivel superior pot depinde de aceste abstracțiuni, în loc să depindă de implementări concrete.

Implementarea corectă a Dependency Inversion Principle aduce numeroase avantaje în proiectarea sistemelor. Acestea includ: decuplarea modulelor, creșterea reutilizării și extensibilității, ușurarea testării și întreținerii, și promovarea unei arhitecturi modulare și ușor de înțeles.

* 1. **Modelele de proiectare creational**

Modelele de proiectare creational sunt responsabile de procesul de creare a obiectelor într-un mod flexibil și eficient. Acestea oferă mecanisme pentru a abstrage procesul de creare și pentru a oferi o instanță potrivită a unei clase sau a unei ierarhii de clase. Mai jos sunt enumerate câteva modele de proiectare creational comune:

1. Singleton: Acest model de proiectare asigură că o clasă are o singură instanță și oferă un punct global de acces la aceasta. Singleton este util atunci când avem nevoie ca o clasă să aibă o singură instanță în întregul sistem.
2. Factory Method: Așa cum am descris anterior, acest model de proiectare permite obiectelor să fie create într-o clasă derivată, în loc să fie create direct în clasa de bază. Factory Method abstractizează procesul de creare a obiectelor și permite subclaselor să decidă tipul specific al obiectului pe care îl vor crea.
3. Abstract Factory: Acest model de proiectare furnizează o interfață pentru crearea familiei de obiecte, fără a specifica clasele concrete. Abstract Factory este util atunci când avem mai multe familii de obiecte și dorim să creăm obiecte compatibile din aceeași familie.
4. Builder: Acest model de proiectare permite construirea obiectelor complexe pas cu pas. Builder separă procesul de construcție al obiectului de reprezentarea acestuia și permite același proces de construcție să creeze reprezentări diferite ale aceluiași obiect.
5. Prototype: Acest model de proiectare permite crearea de obiecte noi prin clonarea unui obiect existent, în loc să le construim de la zero. Prototype este util atunci când crearea unui obiect este costisitoare sau complicată, iar obiectele existente pot fi utilizate ca prototipuri pentru a crea noi instanțe.
6. Object Pool: Acest model de proiectare gestionează un set de obiecte pre-instantiate și le oferă clienților atunci când aceștia au nevoie. Object Pool este util în situațiile în care crearea și distrugerea repetată a obiectelor este costisitoare, și se dorește reutilizarea obiectelor existente.
7. Lazy Initialization: Acesta nu este un model de proiectare propriu-zis, ci mai degrabă o tehnică pentru a amâna inițializarea unui obiect până în momentul în care este necesar. Lazy Initialization este util atunci când inițializarea unui obiect este costisitoare și poate fi amânată până în momentul în care obiectul este cu adevărat utilizat.

Acestea sunt câteva exemple de modele de proiectare creational, fiecare având un scop și un context specific în proiectarea sistemelor. Utilizarea potrivită a acestor modele poate contribui la crearea unui sistem mai flexibil, extensibil și ușor de întreținut.

* + 1. **Factory Method**

Factory Method este un pattern de proiectare creational care permite obiectelor să fie create într-o clasă derivată, în loc să fie create direct în clasa de bază. Acest pattern este util atunci când avem o ierarhie de clase și dorim să abstractizăm procesul de creare a obiectelor, permițând fiecărei subclase să decidă tipul specific al obiectului pe care îl va crea.

Utilizarea Factory Method implică următoarele componente:

1. Clasa Creator: Este clasa de bază care definește metoda factoryMethod(), responsabilă de crearea obiectelor. Această clasă poate fi abstractă sau concretă și poate conține și alte metode comune.
2. Clasa ConcreteCreator: Este o clasă derivată a clasei Creator și implementează metoda factoryMethod(). Această clasă decide tipul specific al obiectului pe care îl va crea și returnează o instanță a acestuia.
3. Clasa Product: Reprezintă clasa de bază pentru obiectele create de Factory Method. Această clasă poate fi abstractă sau concretă și definește comportamentul comun al obiectelor create.
4. Clasa ConcreteProduct: Este o clasă derivată a clasei Product și reprezintă tipul specific al obiectului creat de Factory Method. Această clasă implementează comportamentul specific al obiectului.

Exemple concrete de utilizare a Factory Method în proiectarea sistemelor includ:

1. Crearea de obiecte în funcție de configurare: Factory Method poate fi utilizat pentru a crea obiecte în funcție de configurația sistemului sau a aplicației. De exemplu, o aplicație de gestionare a produselor poate utiliza un Factory Method pentru a crea obiecte de tip produs în funcție de setările specifice ale aplicației.

2. Integrarea cu servicii externe: Factory Method poate fi utilizat pentru a crea obiecte care să interacționeze cu servicii externe sau biblioteci. De exemplu, un Factory Method poate fi folosit pentru a crea obiecte care să se conecteze la diferite API-uri de servicii web.

Avantajele utilizării Factory Method în proiectarea sistemelor includ:

1. Flexibilitate și extensibilitate: Factory Method permite adăugarea ușoară de noi clase de produse fără a modifica clasa Creator sau alte clase existente. Astfel, sistemul poate fi extins cu ușurință pentru a suporta noi tipuri de obiecte.
2. Decuplare și abstractizare: Factory Method decuplează procesul de creare a obiectelor de utilizarea acestora, permițând modificări în procesul de creare fără a afecta codul care folosește obiectele create.
3. Reutilizare: Factory Method permite reutilizarea codului de creare a obiectelor în mai multe locuri din sistem, evitând duplicarea de cod.
   * 1. **Abstract Factory**

Principii și scop: Abstract Factory este un pattern de proiectare creational care oferă o interfață pentru crearea familiei de obiecte, fără a specifica clasele concrete. Acest pattern se concentrează pe furnizarea unei modalități de creare a unui grup de obiecte corelate, astfel încât acestea să fie compatibile și să funcționeze împreună. Scopul Abstract Factory este de a abstractiza procesul de creare a obiectelor, permițând sistemului să utilizeze diverse familii de obiecte fără a fi legat de clasele lor concrete.

Implementare: Implementarea Abstract Factory implică următoarele componente:

1. AbstractFactory: Reprezintă interfața abstractă care definește metodele pentru crearea obiectelor din familia de produse. Această interfață poate avea metode pentru crearea fiecărui tip de obiect în familia respectivă.
2. ConcreteFactory: Este o clasă concretă care implementează interfața AbstractFactory și este responsabilă pentru crearea obiectelor specifice din familia de produse.
3. AbstractProduct: Reprezintă interfața abstractă pentru obiectele produse de AbstractFactory. Această interfață poate defini metode comune pentru toate obiectele din familia respectivă.
4. ConcreteProduct: Sunt clase concrete care implementează interfața AbstractProduct și reprezintă tipurile specifice de obiecte din familia de produse.

Avantaje în proiectarea sistemelor:

1. Separarea logică a obiectelor: Abstract Factory permite separarea logică a obiectelor în familii corespunzătoare. Astfel, fiecare familie de obiecte este implementată într-o AbstractFactory și ConcreteProduct-uri asociate, ceea ce facilitează organizarea și înțelegerea sistemului.
2. Flexibilitate și extensibilitate: Utilizarea Abstract Factory permite adăugarea ușoară de noi familii de produse prin adăugarea de noi ConcreteFactory-uri și ConcreteProduct-uri. Aceasta face ca sistemul să fie flexibil și ușor de extins în viitor, pentru a suporta noi tipuri de obiecte sau familii de produse.
3. Deschis pentru modificări, închis pentru utilizare: Abstract Factory urmează principiul OCP (Open-Closed Principle) prin permiterea extensiei sistemului prin adăugarea de noi clase, fără a modifica codul existent care utilizează interfața AbstractFactory și AbstractProduct.
4. Crearea de obiecte coerente: Abstract Factory asigură că obiectele create de o fabrică sunt compatibile între ele și funcționează împreună. Astfel, se evită incompatibilitățile și erorile în sistemul final.
5. Ascunderea detaliilor de implementare: Abstract Factory ascunde detaliile de implementare ale creării obiectelor. Astfel, clasele care utilizează fabrica nu trebuie să cunoască detalii specifice despre cum sunt create obiectele, ci doar să utilizeze interfața comună a AbstractFactory.
   * 1. **Singleton**

Concept și utilizări: Singleton este un pattern de proiectare creational care se asigură că o clasă are o singură instanță și oferă un punct global de acces la aceasta. Scopul Singleton este de a restricționa crearea de multiple instanțe ale unei clase și de a oferi o metodă simplă de acces la instanța unică.

Conceptul de Singleton se bazează pe definirea unei metode statice în clasa Singleton care returnează întotdeauna aceeași instanță a clasei. Această instanță este creată o singură dată și este utilizată de toate apelurile către metoda statică.

Utilizările Singleton includ:

1. Acces la resurse comune: Singleton este util atunci când avem resurse comune, cum ar fi conexiuni la baze de date, fișiere de configurare sau obiecte de cache. Prin utilizarea unui Singleton, putem asigura că o singură instanță a resursei este creată și utilizată în întregul sistem.
2. Obiecte de configurare: În cazul în care avem obiecte de configurare care trebuie să fie accesibile în mai multe părți ale sistemului, putem utiliza Singleton pentru a asigura că o singură instanță a obiectului de configurare este utilizată.
3. Loggeri și registre: Singleton poate fi utilizat pentru a implementa loggeri sau registre în aplicații, asigurându-ne că există o singură instanță a acestora care poate fi accesată global.

Considerații și restricții de proiectare:

1. Acces global: Singleton oferă un punct global de acces la instanța sa, ceea ce înseamnă că orice parte a sistemului poate utiliza instanța Singleton. Aceasta poate duce la dependențe strânse și poate complica testarea și mentenanța codului.
2. Probleme de concurență: Dacă mai multe fire de execuție încearcă să acceseze sau să modifice instanța Singleton în același timp, pot apărea probleme de concurență. Este important să se implementeze măsuri pentru a asigura accesul concurent corect la instanță.
3. Testarea unitară: Din cauza accesului global și dependențelor strânse, testarea unitară a claselor care utilizează Singleton poate deveni dificilă. Este recomandat să se utilizeze tehnici de mock sau dependency injection pentru a izola clasele de Singleton în timpul testării.
4. Extensibilitate limitată: Singleton nu permite crearea de subclase sau variante diferite ale instanței. Astfel, extinderea sau personalizarea comportamentului instanței Singleton poate fi dificilă.

Este important să se analizeze cu atenție necesitatea utilizării unui Singleton în proiectarea unui sistem. Deși oferă avantaje, trebuie luate în considerare și restricțiile și considerațiile de proiectare asociate cu utilizarea acestui pattern.

* 1. **Modelele de proiectare structural**

Modelele de proiectare structural se concentrează pe organizarea și structurarea obiectelor și relațiilor dintre ele pentru a obține un sistem mai eficient și mai flexibil. Aceste modele se ocupă de compunerea obiectelor în structuri mai mari, precum și de modul în care obiectele comunică și interacționează între ele. Acestea sunt utilizate pentru a obține o structură clară, ușor de înțeles și ușor de întreținut în cadrul sistemelor de software. Unele dintre modelele de proiectare structural comune includ:

1. Adapter: Acest model de proiectare permite obiectelor cu interfețe incompatibile să lucreze împreună prin intermediul unui adaptator care convertește interfața unui obiect într-o altă interfață așteptată de client. Adapter-ul facilitează comunicarea între obiecte care altfel nu ar putea interacționa direct.

2. Bridge: Acest model de proiectare separă abstractizarea de implementare, permițându-le să evolueze independent unul de celălalt. Bridge-ul este utilizat atunci când există multiple implementări posibile pentru o funcționalitate, iar obiectele trebuie să fie decuplate de implementare specifică.

3. Composite: Acest model de proiectare permite tratamentul uniform al obiectelor individuale și al grupurilor de obiecte prin intermediul unei ierarhii de compunere. Composite-ul permite compunerea obiectelor în structuri arbore și oferă metode pentru a manipula atât obiectele individuale, cât și grupurile de obiecte în mod recursiv.

4. Decorator: Acest model de proiectare permite adăugarea de funcționalități suplimentare la un obiect existent fără a modifica structura sa. Decorator-ul utilizează o abordare de învelire pentru a îmbunătăți funcționalitatea obiectului de bază, oferind flexibilitate în adăugarea și eliminarea comportamentelor suplimentare.

5. Facade: Acest model de proiectare oferă o interfață simplificată și unificată pentru un subsistem complex de clase. Facade-ul ascunde complexitatea subsistemului și oferă un punct de acces simplificat pentru clientul care utilizează subsistemul.

6. Flyweight: Acest model de proiectare se concentrează pe partajarea eficientă a unor obiecte mici și ușoare pentru a reduce consumul de memorie. Flyweight-ul utilizează o instanță comună a obiectului și stochează informația specifică în afara obiectului, permițându-i să fie partajat între mai multe entități.

7. Proxy: Acest model de proiectare furnizează un obiect proxy care acționează ca un înlocuitor pentru un alt obiect. Proxy-ul poate controla și gestiona accesul la obiectul real, permițând să se efectueze operații suplimentare, cum ar fi verificarea autorizării sau gestionarea resurselor.

### **1.3.1. Adapter**

Scop și caracteristici:

Pattern-ul Adapter este utilizat atunci când trebuie să conectăm două interfețe incompatibile sau să adaptăm o interfață existentă pentru a fi utilizată într-un context diferit. Scopul său este de a permite comunicarea și colaborarea între obiecte care altfel nu ar putea interacționa direct.

Caracteristicile importante ale pattern-ului Adapter includ:

1. Conversia interfeței: Adapter-ul acționează ca un intermediar între două interfețe incompatibile, convertind cererile primite de la un obiect într-un format acceptabil pentru celălalt obiect. Astfel, permite obiectelor să comunice și să colaboreze fără a fi nevoie de modificări ale codului existent.
2. Flexibilitate și reutilizare: Adapter-ul poate fi reutilizat în diferite contexte pentru a realiza adaptări între diferite interfețe. Astfel, oferă o soluție flexibilă și modulară pentru gestionarea compatibilității între obiecte.
3. Izolarea modificărilor: Prin utilizarea Adapter-ului, modificările într-o interfață sau într-un obiect nu afectează direct obiectele care depind de acea interfață. Adapter-ul poate fi ajustat pentru a se potrivi noii interfețe sau modificări ale obiectului, protejând astfel restul sistemului de schimbări.

Implementare și beneficii în proiectarea sistemelor:

Pentru a implementa pattern-ul Adapter, următoarele componente sunt necesare:

1. Interfața Adaptee: Reprezintă interfața existentă sau clasa care trebuie adaptată pentru a fi utilizată într-un context diferit.
2. Interfața Target: Reprezintă interfața dorită sau așteptată de către clientul care utilizează Adapter-ul.
3. Adapter: Este clasa care implementează interfața Target și utilizează obiectele de tip Adaptee pentru a îndeplini cerințele interfeței Target. Adapter-ul convertește apelurile și cererile de la interfața Target în apeluri corespunzătoare pentru interfața Adaptee.

Beneficiile pattern-ului Adapter în proiectarea sistemelor includ:

1. Reutilizarea codului existent: Adapter-ul permite utilizarea obiectelor existente, care au o interfață incompatibilă cu cea necesară. Astfel, codul existent poate fi reutilizat și integrat în sistemul nou.
2. Integrarea sistemelor: Pattern-ul Adapter facilitează integrarea sistemelor sau modulelor dezvoltate independent. Prin intermediul Adapter-ului, interacțiunea între aceste sisteme devine posibilă, fără a necesita modificări majore ale codului existent.
3. Separarea responsabilităților: Adapter-ul separă responsabilitățile între obiectul care trebuie adaptat și codul care utilizează obiectul adaptat. Astfel, se obține o structură mai modulară și mai clară a sistemului.
4. Flexibilitate și extensibilitate: Adapter-ul oferă flexibilitate în gestionarea schimb

### **1.3.2. Decorator**

Concept și funcționalități:

Pattern-ul Decorator permite adăugarea de funcționalități suplimentare la un obiect existent fără a modifica structura acestuia. Acesta utilizează o abordare de învelire, în care un obiect decorator împachetează un alt obiect și oferă funcționalități suplimentare prin intermediul metodelor sale.

Principalele funcționalități ale pattern-ului Decorator includ:

1. Extinderea comportamentului: Decorator-ul permite adăugarea de funcționalități suplimentare la un obiect existent, fără a afecta obiectul de bază sau alte obiecte similare. Aceasta oferă posibilitatea de a extinde comportamentul obiectului inițial prin învelirea acestuia cu decoratori adiționali.
2. Combinarea flexibilă a funcționalităților: Decorator-ul permite combinarea și compunerea diferitelor funcționalități prin utilizarea multiplelor niveluri de decoratori. Astfel, putem crea combinații personalizate de funcționalități prin adăugarea sau înlăturarea decoratorilor.
3. Transmiterea transparentă a apelurilor: Decorator-ul asigură că toate apelurile către obiectul decorat sunt transmise în mod transparent către obiectul de bază și că rezultatele sunt returnate corect. Astfel, utilizarea unui decorator nu afectează modul în care obiectul inițial este utilizat și apelat.

Aplicații și avantaje în proiectarea sistemelor:

Pattern-ul Decorator poate fi utilizat în diverse situații și aduce următoarele avantaje în proiectarea sistemelor:

1. Extensibilitate: Decorator-ul oferă o alternativă flexibilă la moștenirea clasică, permițând adăugarea de funcționalități suplimentare la un obiect fără a crea o clasă derivată pentru fiecare combinație posibilă. Acest lucru face sistemul mai extensibil și mai ușor de întreținut.
2. Reutilizare: Decorator-ul permite reutilizarea obiectelor existente, adăugând funcționalități suplimentare prin învelirea acestora în decoratori. Astfel, putem obține comportamente diferite și personalizate utilizând aceleași obiecte de bază.
3. Separarea preocupărilor: Pattern-ul Decorator ajută la separarea preocupărilor specifice și oferă o structură mai modulară a sistemului. Funcționalitățile suplimentare pot fi adăugate și eliminate independent, fără a afecta restul sistemului.
4. Flexibilitate în timpul execuției: Deoarece decoratorii pot fi adăugați sau înlăturați în timpul execuției, acest pattern permite configurarea dinamică a obiectelor și funcționalităților acestora în funcție de nevoile și contextul sistemului.

### **1.3.3. Composite**

Definiție și structură: Pattern-ul Composite permite tratarea uniformă a obiectelor individuale și a grupurilor de obiecte prin intermediul unei ierarhii de compunere. Acesta transformă obiectele individuale și grupurile de obiecte într-o structură arbore, astfel încât acestea să poată fi tratate în mod similar. Pattern-ul Composite este compus din următoarele componente:

1. Component: Reprezintă interfața comună pentru toate obiectele din ierarhia compusă. Definește metodele comune pe care le pot implementa atât obiectele individuale, cât și grupurile de obiecte.
2. Leaf: Reprezintă obiectele individuale din ierarhia compusă. Acestea sunt obiecte elementare care nu pot avea subobiecte. Implementează metodele specifice obiectelor individuale.
3. Composite: Reprezintă grupurile de obiecte din ierarhia compusă. Un composite este compus dintr-o colecție de componente, inclusiv obiecte leaf și obiecte composite. Implementează metodele specifice pentru gestionarea subobiectelor.

Utilizări și eficiență în proiectarea sistemelor: Pattern-ul Composite este util în următoarele situații și aduce multiple beneficii în proiectarea sistemelor:

1. Tratarea uniformă a obiectelor: Composite-ul permite tratarea uniformă a obiectelor individuale și a grupurilor de obiecte în cadrul unei ierarhii. Astfel, putem trata obiectele compuse în același mod în care tratăm obiectele individuale, fără a face distincție între ele.
2. Compoziție ierarhică: Pattern-ul Composite permite crearea de structuri ierarhice complexe prin compunerea obiectelor individuale și a grupurilor de obiecte. Acesta oferă flexibilitate în construirea și gestionarea ierarhiilor compuse, permițând adăugarea și eliminarea obiectelor într-un mod dinamic.
3. Simplificarea codului: Prin utilizarea pattern-ului Composite, putem evita duplicarea codului pentru tratarea obiectelor individuale și a grupurilor de obiecte. Astfel, se obține un cod mai simplu și mai clar, iar operațiile comune sunt definite o singură dată și pot fi aplicate recursiv în întreaga ierarhie.
4. Reutilizare și extensibilitate: Composite-ul facilitează reutilizarea și extensibilitatea codului, deoarece obiectele individuale și grupurile de obiecte pot fi utilizate și combinate în diferite moduri pentru a obține noi structuri compuse. Acest pattern permite crearea de ierarhii compuse complexe prin reutilizarea și combinarea obiectelor existente.
5. Operații recursive: Pattern-ul Composite permite aplicarea operațiilor și metodelor asupra întregii ierarhii compuse, utilizând recursivitatea. Aceasta facilitează implementarea de funcționalități care necesită operații pe întreaga structura.
   1. **Modelele de proiectare comportamentale**

Modelele de proiectare comportamentale (Behavioral Design Patterns) sunt un set de tipare utilizate în proiectarea sistemelor software pentru a defini interacțiunile și comportamentul între diferite obiecte și entități ale sistemului. Aceste tipare se concentrează pe modul în care obiectele comunică între ele și cum sunt gestionate fluxurile de date și acțiunile în cadrul sistemului.

Există mai multe modele de proiectare comportamentale, fiecare având un scop și o utilizare specifică. Iată câteva exemple comune:

1. Observer Pattern (Patternul Observer): Acest pattern definește o relație de tip unu-la-mulți între obiecte, astfel încât atunci când un obiect suferă o modificare, toate obiectele dependente sunt notificate și actualizate automat.

2. Strategy Pattern (Patternul Strategy): Acest pattern permite definirea unui set de algoritmi diferiți care pot fi schimbați în timpul execuției. Obiectele pot utiliza un algoritm specific în funcție de nevoi sau condiții.

3. Command Pattern (Patternul Command): Acest pattern încapsulează o cerere sub forma unui obiect, permițând parametrizarea clientului cu cererea dorită și oferind flexibilitate în gestionarea comenzilor și acțiunilor.

4. Template Method Pattern (Patternul Template Method): Acest pattern definește o structură schelet pentru o metodă, lăsând implementarea detaliilor specifice subclaselor. Aceasta permite reutilizarea codului și definirea de comportamente specifice în subclasă.

5. Iterator Pattern (Patternul Iterator): Acest pattern oferă o modalitate de a accesa și itera prin elementele unei colecții fără a expune detaliile interne ale colecției. Astfel, se oferă o abstracție eficientă pentru accesul la elementele unei colecții.

6. State Pattern (Patternul State): Acest pattern permite obiectelor să își schimbe comportamentul în funcție de starea internă. În loc să utilizeze instrucțiuni condiționale, se utilizează obiecte de stare separate pentru a reprezenta și gestiona stările diferite.

Acestea sunt doar câteva exemple de modele de proiectare comportamentale. Fiecare dintre aceste tipare oferă un set de soluții și abstracții pentru a îmbunătăți modul în care obiectele interacționează și se comportă în cadrul sistemelor software.

1. **Implementarea modelelor de proiectare intr-un proiect real**
   1. **Descrierea proiectului**

Proiectul implică dezvoltarea unei aplicații pentru un magazin de pizza, în care au fost implementate diverse tipare de proiectare pentru a îmbunătăți structura și comportamentul sistemului.

Pentru partea de creare a pizzei, ai folosit tiparul de proiectare Builder. Acesta a fost implementat prin intermediul clasei PizzaBuilder, care oferă metode necesare pentru crearea diferitelor tipuri de pizza. Aceasta permite construirea pizzei pas cu pas și oferă flexibilitate în adăugarea opțională a diferitelor ingrediente.

Pentru a asigura existența unei singure instanțe pentru meniu și stoc, ai implementat tiparul de proiectare Singleton. Clasele Menu și Storage sunt ambele singleton-uri, pentru că un magazin de pizza trebuie să aibă doar un singur meniu și un singur stoc. Ai folosit metodele statice și implementarea Lazy pentru a asigura siguranța în ceea ce privește thread-urile.

Pentru gestionarea plăților în moduri diferite (Card, Cash și Bitcoin), ai utilizat tiparul de proiectare Strategy. Ai creat o clasă PaymentContext, care primește strategia de plată dorită de la client. Contextul nu știe detalii specifice despre strategii, ci interacționează cu toate strategiile prin intermediul unei interfețe generice IPaymentStrategy. Aceasta permite selectarea și aplicarea corespunzătoare a algoritmului de plată ales.

Pentru a controla accesul la serviciile de livrare, ai implementat tiparul de proiectare Proxy. Proxy-ul acționează ca un substitut pentru obiectul original, controlând accesul la serviciile de livrare și permițând executarea acțiunilor suplimentare înainte sau după cererea către obiectul original.

În plus, ai utilizat și tiparul de proiectare Facade pentru a simplifica interacțiunea cu sistemul de gestionare a magazinului de pizza. Clasa PizzaShop acționează ca o interfață simplificată către funcționalitățile complexe ale sistemului, permițând clienților să interacționeze cu acestea fără a cunoaște detaliile interne ale implementării.

Aceste tipare de proiectare au fost implementate în proiectul tău pentru a îmbunătăți structura, flexibilitatea și reutilizarea codului, precum și pentru a facilita extinderea și modificarea sistemului în viitor.

Tiparele de proiectare sunt un grup de soluții predefinite care ajută la rezolvarea diferitelor probleme comune de programare. În principal, tiparele de proiectare sunt împărțite în 3 categorii: Creational (de creare), Structural (structurale) și Behavioral (de comportament).

* 1. **Creational Design Patterns**

Pattern-urile de design creatioal sunt soluții reutilizabile pentru problemele legate de crearea obiectelor într-un mod flexibil și decuplat de clasele specifice. Aceste pattern-uri facilitează procesul de creare a obiectelor, promovând principii precum encapsularea, modularitatea și reutilizarea codului.

Iată o descriere generală a câtorva pattern-uri de design creatioal:

Pattern-ul **Singleton** asigură că o singură instanță a unei clase poate exista în cadrul unei aplicații. Acesta oferă un punct centralizat de acces la instanța respectivă și limitează crearea de noi instanțe.

Pattern-ul **Factory Method** furnizează o metodă abstractă într-o clasă de bază, care permite claselor derivate să creeze și să returneze instanțe ale unor clase concrețe. Astfel, clientul nu trebuie să cunoască clasa exactă a obiectului pe care îl primește.

Pattern-ul **Abstract Factory** furnizează o interfață pentru crearea familiilor de obiecte conexe, fără a specifica clasele concrete. Aceasta permite schimbarea facilă a familiei de obiecte create, păstrând totodată cuplarea redusă între client și clasele concrete.

Pattern-ul **Builder** permite crearea unui obiect complex pas cu pas, oferind un proces de construcție flexibil. Prin intermediul unei clase de builder, se pot specifica diferite etape de construcție și se obține un obiect final într-un mod controlat și modular.

Pattern-ul **Prototype** permite crearea de noi obiecte prin clonarea unui prototip existent. Astfel, se evită crearea unor obiecte costisitoare în mod repetat, iar clonarea poate fi personalizată în funcție de necesități.

Pattern-ul **Object Pool** menține un set de obiecte pre-instanțiate și le oferă disponibilitate în mod reutilizabil. Astfel, se evită costurile asociate cu crearea și distrugerea repetată a obiectelor, optimizând performanța aplicației.

Acestea sunt doar câteva exemple de pattern-uri de design creatioal. Fiecare pattern abordează diferite aspecte ale creării obiectelor și poate fi aplicat în funcție de specificul și nevoile aplicației dezvoltate. Utilizarea acestor pattern-uri facilitează dezvoltarea unui cod modular, ușor de întreținut și reutilizabil.

În acest proiect am utilizat 3 creatioanal design patterns: **Builder**, **Prototype** și **Singleton**. Singleton Design Pattern este utilizat pentru a asigura că aveți o singură instanță a unui anumit obiect în întreaga aplicație. În acest proiect, clasa Menu și clasa Storage sunt ambele singletons, deoarece un magazin de pizza poate avea doar un meniu și un depozit. Pentru clasa Menu am implementat tiparul Singleton utilizând implementarea constructorului static, iar pentru Storage am utilizat implementarea Lazy. Ambele sunt sigure în ceea ce privește thread-urile.

*sealed class Menu*

*{*

*public List<MenuItem> Items { get; set; } = new();*

*private static readonly Menu \_menuInstance = new();*

*private Menu() { }*

*public static Menu Instance*

*{*

*get*

*{*

*return \_menuInstance;*

*}*

*}*

*}*

*sealed class Storage*

*{*

*private static readonly Lazy<Storage> \_storageIntance = new(() => new Storage());*

*private Storage() { }*

*private static List<IIngredient> \_ingredients = new();*

*public List<IIngredient> Ingredients*

*{*

*get => \_ingredients;*

*set => \_ingredients = value;*

*}*

*public static Storage Intance*

*{*

*get => \_storageIntance.Value;*

*}*

*}*

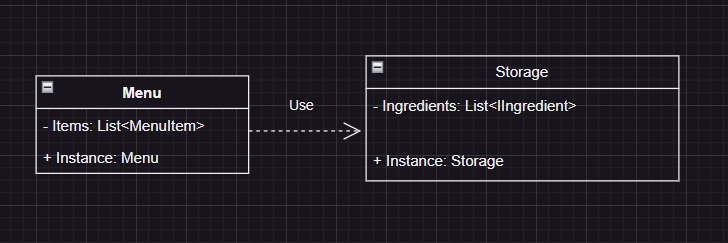
**

Figura 2 – diagrama singleton.

Relația dintre cele două clase este de utilizare (uses), ceea ce înseamnă că clasa Menu utilizează clasa Storage pentru a accesa și gestiona ingredientele. Săgeata indică direcția utilizării.

Ambele clase sunt implementate ca singleton-uri, având o singură instanță în întreaga aplicație. Clasa Menu folosește un membru static Instance pentru a accesa instanța singleton-ului, în timp ce clasa Storage utilizează un membru static Instance bazat pe implementarea Lazy pentru a obține instanța singleton-ului.

Design pattern **Builder** a fost utilizat pentru crearea Pizzei, deoarece o Pizza poate avea o listă lungă de ingrediente opționale și setarea tuturor acestora în constructor ca valori nullable va face codul să pară dezordonat și greu de citit. Pentru implementarea tiparului Builder, am creat clasa PizzaBuilder care furnizează toate metodele necesare pentru crearea unui număr mare de pizza diferite. Există diferite implementări ale builder-ului, dar cea utilizată în acest proiect permite concatenarea metodelor builder, deoarece fiecare metodă din clasa Builder returnează "this" (instanța builder-ului), cu excepția metodei Bake() care returnează obiectul construit concret.

*interface IPizzaBuilder*

*{*

*IPizzaBuilder AddDough(bool isGlutenFree = false);*

*IPizzaBuilder AddCheese();*

*IPizzaBuilder AddCapsicum(int units);*

*IPizzaBuilder AddOlive();*

*IPizzaBuilder AddSalami(int units);*

*Pizza Bake();*

*}*

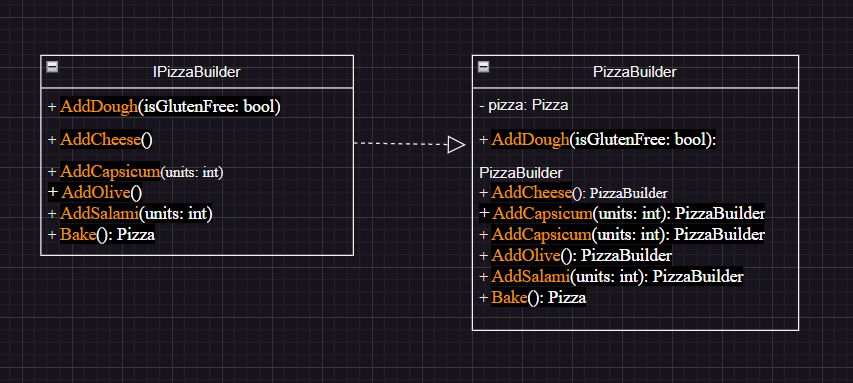
**

Figura 3 - diagrama Builder.

Clasa PizzaBuilder oferă metode pentru a adăuga diferite ingrediente și a construi o pizza. Aceste metode returnează o instanță a propriei clase (PizzaBuilder), permițând astfel concatenarea metodelor pentru a forma un șir de instrucțiuni de construire a pizzei. Metoda Bake() este responsabilă de finalizarea construcției și returnarea obiectului rezultat de tip Pizza.

Această abordare folosind builder pattern permite o construcție flexibilă și ușor de citit a obiectelor de tip Pizza, evitându-se un constructor complicat cu multe parametri opționali.

Ultimul desing pattern crețional implementat în acest proiect este **Prototype**. A fost implementat utilizând interfața IProduct care are metoda Clone și fiecare produs o implementează. Astfel, atunci când un client dorește să comande un produs, nu-i pasă de implementarea concretă, utilizează interfața IProduct.

*public IProduct OrderMenuItem(int id)*

*{*

*var orderedItem = \_menu.Items.Find(item => item.Id == id);*

*if(orderedItem == null)*

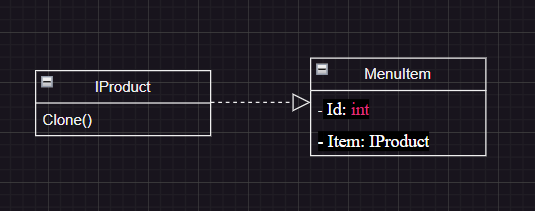
*{*

*return null;*

*}*

*return orderedItem.Item.Clone();*

*}*

**Figura 4 – diagrama prototype

Această diagramă indică faptul că clasa MenuItem are o asociere cu interfața IProduct, iar clasa MenuItem utilizează metoda Clone() pentru a obține o copie a obiectului Item de tip IProduct.

* 1. **Structural Design Patterns**

Pattern-urile de design structural se concentrează pe organizarea și relațiile dintre obiecte și clase într-un sistem software. Aceste pattern-uri ajută la crearea unei structuri eficiente, flexibile și ușor de înțeles în cadrul unei aplicații, prin încapsularea și combinarea obiectelor pentru a forma componente mai mari și complexe.

Iată o descriere generală a câtorva pattern-uri de design structural:

Pattern-ul **Adapter** convertește interfața unui obiect într-o altă interfață așteptată de un client. Acesta permite colaborarea între obiecte cu interfețe incompatibile și facilitează reutilizarea codului existent.

Pattern-ul **Decorator** permite atașarea de funcționalități suplimentare la un obiect într-un mod dinamic. Acesta extinde comportamentul unui obiect fără a afecta alte instanțe ale aceleiași clase și oferă flexibilitate în funcție de nevoile specifice.

Pattern-ul **Composite** permite tratarea unui grup de obiecte similare într-un mod uniform ca și cum ar fi un singur obiect. Acesta formează o ierarhie de arbori compuși, unde atât obiectele individuale, cât și grupurile de obiecte pot fi manipulate într-un mod consistent.

Pattern-ul **Proxy** furnizează un obiect de substituție care controlează accesul la obiectul real. Acesta poate fi folosit pentru a controla accesul la obiecte sensibile, pentru a realiza evaluări leneșe sau pentru a gestiona resursele limitate.

Pattern-ul **Facade** furnizează o interfață simplificată și unificată pentru un set de interfețe și clase complexe. Acesta ascunde complexitatea sistemului sub o interfață simplă, ușurând utilizarea și integrarea cu alte componente.

Pattern-ul **Bridge** separă o abstractizare de implementare, astfel încât acestea să poată varia independent. Acesta permite decuplarea unei ierarhii de clase prin intermediul unei ierarhii separate de interfețe, permițând modificarea și extinderea lor în mod independent.

Acestea sunt doar câteva exemple de pattern-uri de design structural. Fiecare pattern adresează diferite aspecte ale organizării și relațiilor între obiecte și clase într-un sistem software. Utilizarea acestor pattern-uri facilitează dezvoltarea unui cod modular, flexibil și ușor de întreținut, prin separarea responsabilităților și promovarea reutilizării și extensibilității.

În acest proiect, am utilizat 3 tipare structurale desing patterns: **Bridge**, **Proxy** și **Facade**. Facade este un design pattern structural care furnizează o interfață simplificată către o bibliotecă, un framework sau orice alt set complex de clase. În acest caz, clasa Facade este reprezentată de clasa PizzaShop. Clientul comunică cu clasa Facade utilizând interfața IPizzaShop. Utilizând acest tipar, clientul nu trebuie să știe nimic despre sistemul PizzaShop în afara metodelor expuse de clasa PizzaShop. În acest fel, tiparul Facade simplifică interacțiunea cu sistemul nostru.

*interface IPizzaShop*

*{*

*List<MenuItem> GetMenuItems();*

*IProduct OrderMenuItem(int id);*

*void GetDelivery(List<IProduct> order);*

*}*

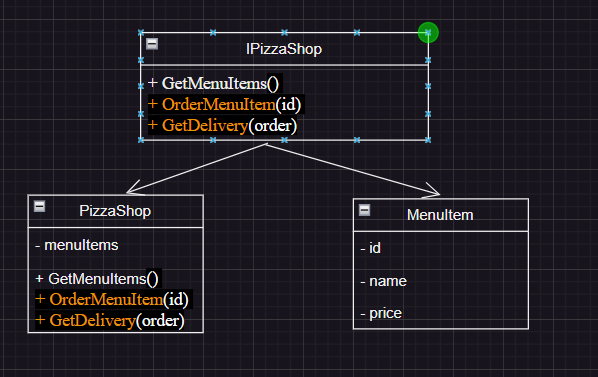
**

Figura 5 – diagrama facade

Relația între PizzaShop și IPizzaShop este reprezentată prin implementarea interfeței IPizzaShop de către clasa PizzaShop. Astfel, clasa PizzaShop expune metodele definite în interfață pentru a comunica cu clientul.

De asemenea, există o relație de asociere între PizzaShop și MenuItem, deoarece PizzaShop are o listă de elemente de meniu (menuItems) care sunt obiecte de tip MenuItem.

Aceasta este o reprezentare simplificată a relațiilor între clase în cadrul pattern-ului Facade.

**Bridge** este un design pattern structural care vă permite să separați o clasă mare sau un set de clase strâns legate în două ierarhii separate - abstractizare și implementare - care pot fi dezvoltate independent una de cealaltă. În acest caz, tiparul de proiectare Bridge este reprezentat de relațiile IProduct și IIngredient. În acest fel, am separat complet acest set de clase strâns legate în două ierarhii.

*interface IIngredient*

*{*

*IngredientTypeEnum Type { get; }*

*float Price { get; }*

*int Supply { get; set; }*

*}*

*interface IProduct*

*{*

*List<IIngredient> Ingredients { get; }*

*public double GetPrice();*

*public IProduct Clone();*

*}*

Utilizând aceste interfețe, putem crea mai multe implementări concrete, cum ar fi Pizza și PizzaDough, care extind IProduct, sau Ingredient, care extinde IIngredient.

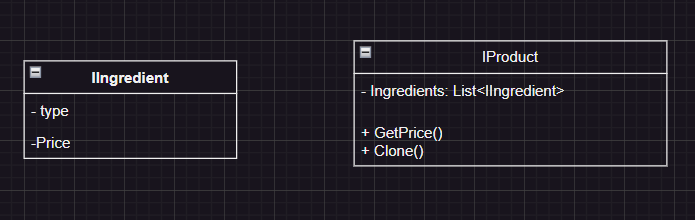


Figura 6 – diagrama bridge

Există o relație de implementare între interfețele IIngredient și IProduct și clasele concrete Ingredient, Pizza și PizzaDough. Aceasta indică faptul că clasele concrete implementează metodele și proprietățile definite în interfețe.

Clasele Pizza și PizzaDough extind interfața IProduct, iar clasa Ingredient extinde interfața IIngredient. Aceasta permite dezvoltarea independentă a ierarhiilor IProduct și IIngredient, permițând flexibilitate în crearea și combinarea produselor și ingredientelor.

**Proxy** este un tipar de proiectare structural care vă permite să furnizați un înlocuitor sau un substitut pentru un alt obiect. Un proxy controlează accesul la obiectul original, permițându-vă să efectuați ceva înainte sau după ce cererea ajunge la obiectul original. În cazul sistemului nostru, proxy-ul este utilizat pentru a controla accesul la serviciile de livrare.

*class ProxyDeliveryService*

*{*

*IDeliveryService \_carDeliveryService = new CarDeliveryService();*

*IDeliveryService \_bikeDeliveryService = new BikeDeliveryService();*

*public void Deliver(List<IProduct> products)*

*{*

*if(products.Count > 5)*

*{*

*\_carDeliveryService.Deliver(products);*

*return;*

*}*

*\_bikeDeliveryService.Deliver(products);*

*}*

*}*

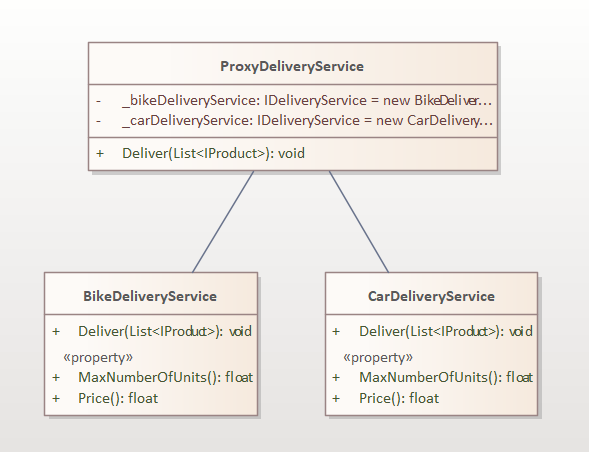
**

Figura 6 – diagrama proxy.

Clasa ProxyDeliveryService acționează ca un proxy pentru serviciile de livrare CarDeliveryService și BikeDeliveryService. Implementarea sa controlează accesul la serviciile de livrare în funcție de numărul de produse specificat.

Interfața IDeliveryService definește metoda Deliver() care este implementată de clasele CarDeliveryService și BikeDeliveryService. Aceste clase reprezintă servicii concrete de livrare, iar ProxyDeliveryService le utilizează pentru a efectua livrarea în funcție de condițiile specifice.

Aceasta este o reprezentare simplificată a relațiilor între clase în cadrul pattern-ului Proxy.

* 1. **Behavioral Design Patterns**

Pattern-urile de design comportamentale se concentrează pe gestionarea interacțiunilor și comunicării între diferite obiecte într-un sistem software. Aceste pattern-uri ajută la definirea modului în care obiectele cooperează și se interconectează pentru a rezolva problemele specifice legate de comportamentul și fluxul datelor într-o aplicație.

Iată o descriere generală a câtorva pattern-uri de design comportamentale:

Pattern-ul **Observer** definește o relație de tip unu-la-mulți între obiecte, astfel încât atunci când un obiect suferă o modificare, toate obiectele dependente sunt notificate și actualizate automat. Acest pattern este util pentru implementarea evenimentelor și notificărilor în aplicații.

Pattern-ul **Strategy** permite definirea unei familii de algoritmi, încapsulându-le în obiecte separate și făcându-le interschimbabile. Astfel, se poate modifica comportamentul unui obiect în timpul rulării, fără a afecta codul client care utilizează acel obiect.

Pattern-ul **Template Method** definește o structură de bază pentru o anumită operațiune, permițând claselor derivate să-și implementeze detaliile specifice ale acelei operațiuni. Aceasta promovează reutilizarea codului și permite flexibilitate în personalizarea comportamentului.

Pattern-ul **Chain of Responsibility** permite crearea unei lanțuri de obiecte care pot procesa o solicitare primită. Fiecare obiect din lanț decide dacă va trata solicitarea sau o va transmite mai departe în lanț către un alt obiect. Acest pattern facilitează decuplarea între obiectele care solicită și cele care procesează.

Pattern-ul **Command** încapsulează o solicitare ca obiect, permițând parametrizarea clienților cu diverse solicitări și gestionarea lor în mod flexibil. Acest pattern permite separarea solicitării de execuția acesteia și poate fi util pentru implementarea unui sistem de undo/redo.

Pattern-ul **Iterator** furnizează o modalitate de a accesa elementele unei colecții fără a expune structura internă a colecției. Acest pattern oferă o iterație uniformă asupra elementelor colecției și simplifică interacțiunea cu aceasta.

Acestea sunt doar câteva exemple de pattern-uri de design comportamentale. Fiecare pattern adresează diferite aspecte ale comportamentului și interacțiunii obiectelor într-un sistem software. Utilizarea acestor pattern-uri facilitează dezvoltarea unui cod modular, ușor de înțeles și de întreținut, prin separarea responsabilităților și promovarea flexibilității în comportamentul aplicației.

În acest proiect, am utilizat tiparul de proiectare comportamental **Strategy** pentru a implementa plățile în diferite moduri: Card, Cash și Bitcoin.

Tiparul Strategy sugerează că trebuie să luați o clasă care realizează o acțiune specifică în multe moduri diferite și să extrageți toți acești algoritmi în clase separate numite strategii.

Clasa originală, numită context, trebuie să aibă un câmp pentru a stoca o referință către una dintre strategii. Contextul delegă munca către un obiect de strategie legat în loc să o execute de unul singur.

*class PaymentContext*

*{*

*private IPaymentStrategy \_paymentStrategy;*

*public void SetStrategy(IPaymentStrategy strategy)*

*{*

*\_paymentStrategy = strategy;*

*}*

*public void ExecuteStrategy(double amount)*

*{*

*\_paymentStrategy.Pay(amount);*

*}}*

PaymentContext nu este responsabil pentru selectarea unei modalități adecvate de plată. În schimb, clientul transmite strategia de plată dorită către context. De fapt, contextul nu știe prea multe despre strategii. Lucrează cu toate strategiile prin intermediul aceleiași interfețe generice IPaymentStrategy, care expune doar o singură metodă (Pay) pentru declanșarea algoritmului încapsulat în strategia selectată.

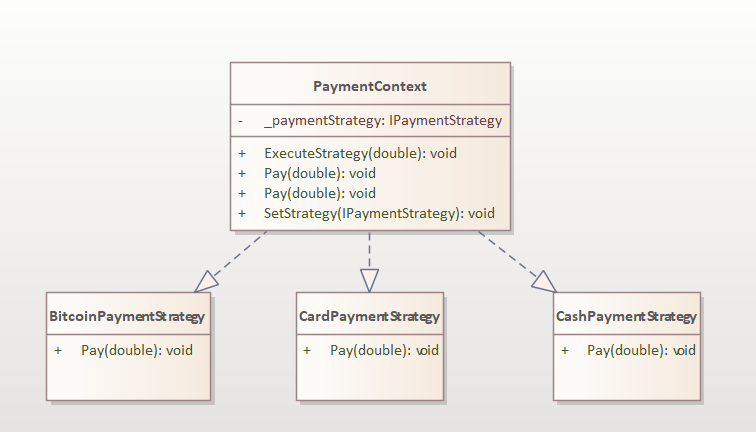


Figura 7 – diagrama Strategy.

În diagrama de mai sus, avem clasa PaymentContext, care reprezintă contextul utilizat pentru efectuarea plăților. Aceasta are un câmp privat \_paymentStrategy de tipul IPaymentStrategy, care reprezintă strategia curentă de plată selectată.

Interfața IPaymentStrategy definește operația comună Pay, care este implementată de clasele concrete CardPayment și CashPayment. Acestea sunt strategiile individuale care implementează algoritmul specific de plată.

Clientul poate utiliza PaymentContext pentru a selecta și executa o strategie specifică de plată. Metoda SetStrategy este utilizată pentru a seta strategia dorită, în timp ce metoda ExecuteStrategy declanșează algoritmul de plată specificat de strategie.

Astfel, tiparul Strategy permite separarea logicii de plată în strategii individuale și delegarea acestei responsabilități contextului, oferind flexibilitate și posibilitatea de a schimba strategiile de plată într-un mod transparent pentru client.

# **Concluzie**

În concluzie, utilizarea tiparelor de proiectare, cum ar fi cele creționale, structurale și comportamentale, în proiectul tău pentru magazinul de pizza a adus mai multe beneficii.

Tiparele de proiectare creționale, cum ar fi Singleton, Builder și Prototype, au ajutat la gestionarea obiectelor și crearea acestora într-un mod flexibil și eficient. Singleton a asigurat că există o singură instanță a anumitor obiecte critice în întreaga aplicație, cum ar fi meniul și stocul. Builder a permis crearea ușoară a unor obiecte complexe, cum ar fi pizza, cu o mulțime de opțiuni opționale, fără a complica constructorul. Prototype a oferit posibilitatea de a crea copii ale unui obiect existent, economisind timp și resurse în crearea de noi obiecte similare.

Tiparele de proiectare structurale, cum ar fi Bridge, Proxy și Facade, au contribuit la organizarea și structurarea sistemului tău. Bridge a separațat ierarhiile de abstractizare și implementare, oferind flexibilitate în modificarea și extinderea funcționalității. Proxy a permis controlul accesului la serviciile de livrare, adăugând logica suplimentară înainte sau după cereri. Facade a furnizat o interfață simplificată către funcționalitățile complexe ale sistemului, permițând clienților să interacționeze fără a cunoaște detaliile interne.

Tiparele de proiectare comportamentale, cum ar fi Strategy, au facilitat implementarea unor opțiuni flexibile de plată în sistemul tău. Prin separarea algoritmilor de plată în clase separate, ai putut selecta și utiliza strategia de plată dorită fără a afecta contextul sau interfața clienților.

Importanța utilizării acestor tipare de proiectare într-un proiect real constă în faptul că ele aduc structură, modularitate și flexibilitate în dezvoltarea software-ului. Ele permit separarea funcționalităților, reducerea cuplării între componente și ușurarea extinderii și modificării sistemului în viitor. De asemenea, ele îmbunătățesc citibilitatea și întreținerea codului, permițând dezvoltatorilor să lucreze mai eficient și să colaboreze mai ușor într-un proiect complex. Utilizarea tiparelor de proiectare într-un mod adecvat poate contribui la dezvoltarea unui sistem software mai robust și scalabil.

# **Bibliografie**

https://refactoring.guru/design-patterns

[ideas/resources/guide-to-uml-diagramming-and-database-modeling](https://www.microsoft.com/ro-ro/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling)

<https://glennbouchard.com/ro/260-pengertian-dan-tipe-diagram-uml.html>

https://biblioteca.regielive.ro/referate/limbaje-de-programare/diagrame-uml-198821.html

https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/