**Ministerul Educației, Culturii și Cercetări**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Proiect de an**

Disciplina: Tehnici și mecanisme de proiectare software

Tema: „Proiectarea unei aplicații pentru un magazin de Pizza”

A efectuat st.gr. TI-204 Cecan Valerian

A verificat asis.univ. Gaidau Mihai

Chișinău 2023

Cuprins

[**Introducere** 2](#_Toc136927557)

[**1.Proiectarea aplicației** 6](#_Toc136927558)

[**1.1 Descrierea problemei** 6](#_Toc136927559)

[**1.2 Șabloane Creaționale** 7](#_Toc136927560)

[**1.3 Șabloane Structurale** 11](#_Toc136927561)

[**1.4 Șabloane comportamentale** 15](#_Toc136927562)

[Figura 1.7 Șablonul Memento 16](#_Toc136927563)

[**2. Descrierea principiului de lucru** 18](#_Toc136927564)

[**Concluzii** 21](#_Toc136927565)

[**Bibliografie** 22](#_Toc136927566)

# **Introducere**

Modelele de proiectare sunt soluții tipice pentru problemele care apar frecvent în proiectareasoftware.Sunt ca niște planuri pre-făcute pe care le puteți personaliza pentru a rezolva o problemă dedesign recurentă din codul dvs.Nu poate fi găsit doar un model și să fie copiat în programul dvs., așa cum se poate face cu funcțiile sau bibliotecile de-pe-raft. Modelul nu este o bucată de cod specific, ci un concept general pentru rezolvarea unei anumite probleme. Se pot urmări detaliile modelului și se poate implementa o soluție care se potrivește realităților propriului program. Modelele sunt deseori confundate cu algoritmii, deoarece ambele concepte descriu soluții tipice pentru unele probleme cunoscute .În timp ce un algoritm definește întotdeauna un set clar de acțiuni care pot atinge un anumit obiectiv, un model este o descriere mai bună a unei soluții. Codul aceluiași model aplicat la două programe diferite poate fi diferit. O analogie cu un algoritm este o rețetă de gătit: ambii au pași clari pentru atingerea unui obiectiv. Pe de altă parte ,un model seamănă mai mult cu un model: puteți vedea care este rezultatul și caracteristicile acestuia.

ÎN CE CONSTĂ MODELUL?

Majoritatea șabloanelor sunt descrise foarte formal , astfel încât oamenii să le poată reproduce în multe contexte. Iată secțiunile care sunt de obicei prezente într-o descriere a modelului:

* Intenția tiparului de a descrie pe scurt atât problema,cât și soluția.
* Motivația explică în continuare problema și soluția pe care modelul o face posibilă.
* Structura claselor arată fiecare parte a modelului și modul în care acestea sunt legate.
* Exemplul de cod într-unul dintre limbajele de programare populare face mai ușor să înțelegeți ideea din spatele modelului. Unele cataloage de șabloane enumeră alte detalii utile, cum ar fi aplicabilitatea modelului, etapele de implementare și relațiile cu alte tipare.

Un proiectant fara experienta este de multe ori coplesit de multitudinea optiunilor disponibile si are tendinta de a se intoarce la tehnicile non-obiectuale pe care le-a folosit in trecut. Un proiectant experimentat stie ca NU TREBUIE sa rezolve fiecare problema incepand de la zero, ci reutilizand solutii din proiecte anterioare. Atunci cand descopera o solutie buna o va folosi mereu. Acest tip de experienta este o parte din ceea ce confera unui proiectant statutul de expert.

In cadrul sistemelor OO realizate in mod profesionist se pot distinge sabloane de clase si de obiecte in comunicare care rezolva probleme specifice si fac ca sistemele respective sa fie mai flexibile, mai elegante si reutilizabile. Un proiectant familiar cu asemnea sabloane le va putea aplica repede, fara a trebui sa le redescopere.

Sabloanele de proiectare sunt de fapt o memorare pentru posteritate a experientei in domeniul proiectarii sistemelor OO.

***Criterii de clasificare:***

**Dupa scop**: sabloanele pot fi creationale, structurale sau comportamentale.

* + Sabloanele **creationale** (creational patterns) privesc modul de creare a obiectelor.
  + Sabloanele **structurale** (structural patterns) se refera la compozitia claselor sau a obiectelor.
  + Sabloanele **comportamentale** (behavioral patterns) caracterizeaza modul in care obiectele si clasele interactioneaza si isi distribuie responsabilitatile.

**Dupa domeniul de aplicare**: sabloanele se pot aplica obiectelor sau claselor.

* + Sabloanele **obiectelor** se refera la relatiile dintre obiecte, relatii care au un caracter dinamic.

Sabloanele creationale ale obiectelor acopera situatiile in care o parte din procesul crearii unui obiect cade in sarcina unui alt obiect. Sabloanele structurale ale obiectelor descriu caile prin care se asambleaza obiecte. Sabloanele comportamentale ale obiectelor descriu modul in care un grup de obiecte coopereaza pentru a indeplini o sarcina ce nu ar putea fi efectuata de un singur obiect.

* + Sabloanele **claselor** se refera la relatii dintre clase, relatii stabilite prin mostenire si care sunt statice (fixate la compilare).

Sabloanele creationale ale claselor acopera situatiile in care o parte din procesul crearii unui obiect cade in sarcina subclaselor. Sabloanele structurale ale claselor descriu modul de utilizare a mostenirii in scopul compunerii claselor. Sabloanele comportamentale ale claselor utilizeaza mostenirea pentru descrierea unor algoritmi si fluxuri de control.

|  |  |
| --- | --- |
| **Șabloane creationale** | |
| [Abstract Factory](https://www.dofactory.com/Patterns/PatternAbstract.aspx) | Furnizeaza o interfata pentru crearea unor familii de obiecte |
| [Builder](https://www.dofactory.com/Patterns/PatternBuilder.aspx) | Separa obiectele construite de reprezentarea lor |
| [Factory Method](https://www.dofactory.com/Patterns/PatternFactory.aspx) | Creaza o instanta a diferitelor clase derivate |
| [Prototype](https://www.dofactory.com/Patterns/PatternPrototype.aspx) | O instanta initializata poate fi copiata sau clonata |
| [Singleton](https://www.dofactory.com/Patterns/PatternSingleton.aspx) | O clasa din care doar o singura instanta poate exista |
| **Șabloane structurale** | |
| [Adapter](https://www.dofactory.com/Patterns/PatternAdapter.aspx) | Face ca interfetele diferitelor clase sa corespunda |
| [Bridge](https://www.dofactory.com/Patterns/PatternBridge.aspx) | Separa interfata unui obiect de implementarea acestuia |
| [Composite](https://www.dofactory.com/Patterns/PatternComposite.aspx) | Structura arborescenta de obiecte simple si compuse |
| [Decorator](https://www.dofactory.com/Patterns/PatternDecorator.aspx) | Adauga responsabilitati obiectelor in mod dinamic |
| [Faade](https://www.dofactory.com/Patterns/PatternFacade.aspx) | O singura clasa ce reprezinta un intreg sistem |
| [Flyweight](https://www.dofactory.com/Patterns/PatternFlyweight.aspx) | O instanta folosita pentru un sharing eficient |
| [Proxy](https://www.dofactory.com/Patterns/PatternProxy.aspx) | Un obiect reprezentand un un alt obiect |

Table 1.1 Șabloane creaționale și Șabloane structurale

|  |  |
| --- | --- |
| **Behavioral Patterns** | |
| [Chain of Resp.](https://www.dofactory.com/Patterns/PatternChain.aspx) | O cale de a transmite cereri intre obiecte |
| [Command](https://www.dofactory.com/Patterns/PatternCommand.aspx) | Incapsuleaza o cerere ca un obiect |
| [Interpreter](https://www.dofactory.com/Patterns/PatternInterpreter.aspx) | O cale de a include elemente lingvistice intr-un program |
| [Iterator](https://www.dofactory.com/Patterns/PatternIterator.aspx) | Acces secvential la elementele unei colectii |
| [Mediator](https://www.dofactory.com/Patterns/PatternMediator.aspx) | Defineste comunicarea simplificata intre clase |
| [Memento](https://www.dofactory.com/Patterns/PatternMemento.aspx) | Capteaza si reconstitui starea interna a unui obiect |
| [Observer](https://www.dofactory.com/Patterns/PatternObserver.aspx) | O cale de a anunta schimbarea unui numar de clase |
| [State](https://www.dofactory.com/Patterns/PatternState.aspx) | Modificarea comportamentului unui obiect atunci cand isi schimba starea |
| [Strategy](https://www.dofactory.com/Patterns/PatternStrategy.aspx) | Incapsuleaza un algoritm intr-o clasa |
| [Template Method](https://www.dofactory.com/Patterns/PatternTemplate.aspx) | Transfera pasii exacti ai unui algoritm unei subclase |
| [Visitor](https://www.dofactory.com/Patterns/PatternVisitor.aspx) | Defineste o noua operatie pentru o clasa fara a face schimbari |

Tabel 1.2 Șabloane de comportament

# **1.Proiectarea aplicației**

## **1.1 Descrierea problemei**

Scopul proiectului este de a proiecta o aplicație software eficientă și ușor de utilizat pentru un magazin de pizza. Aplicația va permite clienților să comande pizza online și să o personalizeze în funcție de preferințele lor, precum și să efectueze plăți și să primească actualizări despre statusul comenzii.

În prezent, magazinul de pizza funcționează în mod tradițional, clienții venind în magazin sau sunând pentru a plasa o comandă. Proiectul urmărește să simplifice procesul de comandă, să eficientizeze fluxul de lucru și să ofere o experiență plăcută pentru clienți.

Principalele provocări și cerințe ale acestei probleme includ:

Personalizarea comenzilor: Aplicația trebuie să ofere clienților opțiuni de personalizare a pizzei, cum ar fi alegerea ingredientelor, dimensiunea, crusta, topping-urile și alte opțiuni suplimentare. Aceasta trebuie să fie ușor de utilizat și intuitivă, astfel încât clienții să poată crea pizza dorită într-un mod flexibil și eficient.

Managementul comenzilor: Sistemul trebuie să ofere o interfață pentru gestionarea comenzilor din partea personalului magazinului. Aceasta ar trebui să permită personalului să vizualizeze comenzile primite, să le confirme sau să le anuleze, să actualizeze statusul comenzilor și să ofere estimări privind timpul de livrare sau ridicare.

Una dintre provocările în proiectarea aplicației pentru magazinul de pizza este crearea unei structuri optimizate pentru crearea și gestionarea obiectelor. În acest context, sabloanele de proiectare creatională pot juca un rol important în asigurarea unui proces eficient și flexibil de creare a instanțelor de obiecte.

Șabloanele de proiectare abordează diferite scenarii de creare a obiectelor și facilitează adaptabilitatea și extensibilitatea sistemului. Iată câteva exemple de șabloane care ar putea fi utilizate în proiectul nostru:

* Factory Method
* Builder
* Prototype
* Adapter
* Decorator
* Memento
* Chain of Responsability

## **1.2 Șabloane Creaționale**

Am implementat următoarele 3 șabloane creaționale: Factory Method,Builder, Prototype,

Pentru început va fi descris Factory Method

**Intent –** Factory Method este un model de design creațional care oferă o interfață pentru crearea de obiecte într-o superclasă, dar permite subclaselor să modifice tipul de obiecte care vor fi create.

-

Figura 1.1 Diagrama UML pentru Factory Method

**Implementare –** Pentru a implementa acest șablon de proiectare, am creat o interfață comună pentru care este Pizza , iar subclasele respectiv :MargherittaPizza, PepperoniPizza,CapriciosaPizza, fiecare avand aceleasi metode comune.

class Pizza\_interface(ABC):  
 def *\_\_init\_\_*(*self*) -> None:  
 *pass*  
  
 def add(*self*, *part*: Any) -> None:  
 *pass*  
  
  
 def add\_p(*self*,*pric* :Any) ->None:  
 *pass*  
  
 def list\_components(*self*) -> None:  
 *pass*  
 @property  
 def get\_components(*self*) -> None:  
 pass  
  
  
 def clone(*self*) -> Pizza:  
 pass

Iar acum urmează un exemplu de subclasă a acestei interfețe:

class Pizza(Pizza\_interface):  
 def *\_\_init\_\_*(*self*) -> None:  
 *self*.components = []  
 *self*.price = None  
  
 def add(*self*, *part*: Any) -> None:  
 *self*.components.append(*part*)  
  
  
 def add\_p(*self*,*pric* :Any) ->None:  
 *self*.price.append(*pric*)  
  
 def list\_components(*self*) -> None:  
 *print*(f"Pizza components: {', '.join(*self*.components)}", *end*="")  
  
 @property  
 def get\_components(*self*) -> None:  
 return *self*.components  
  
  
 def clone(*self*) -> Pizza:  
 return Pizza(*components*=*self*.components,*price*=*self*.price)

Împreună cu acest șablon am implementat și **builder**,care este la fel un șablon creational.

**Builder** este un model de design creațional care vă permite să construiți obiecte complexe pas cu pas. Modelul vă permite să produceți diferite tipuri și reprezentări ale unui obiect folosind același cod de construcție**.**

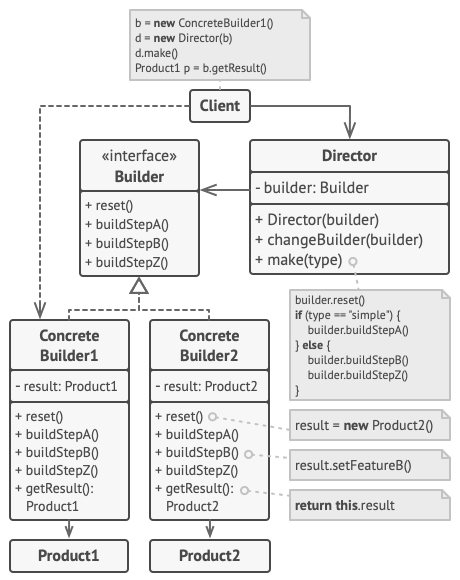


Figura 1.2 Șablonul Builder

Pentru implementarea acestui șablon am creat o interfață builder,care posedă metode abstracte ,prin care sunt descriși pașii de construcție ce urmează.

Apoi am realizat constructorii concreți,pentru 3 tipuri de Pizza, avand în comun majoritatea ingredientelor, iar la crearea obictelor, obținem lucruri diferite dar care aparțin aceleiași interfațe. Am definit clasa Baker, pentru a defini ordinea de construcție a obiectelor și configurarea lor.

class Builder(ABC):  
 @property  
 @abstractmethod  
 def pizza(*self*) -> None:  
 pass  
  
 @abstractmethod  
 def make\_crust(*self*) -> None:  
 pass  
  
 @abstractmethod  
 def add\_sauce(*self*) -> None:  
 pass  
  
 @abstractmethod  
 def add\_mushroms(*self*) -> None:  
 pass  
  
 @abstractmethod  
 def add\_olives(*self*) -> None:  
 pass

O segvență de cod din interfața Builder,unde definim mai intai tipul de blat,apoi tipul de sos și deja toppingurile,deoarece folosim acest șablon creational, putem adăuga unele ingredient suplimentare.

class Baker:  
 def *\_\_init\_\_*(*self*) -> None:  
 *self*.\_builder = None  
  
 @property  
 def builder(*self*) -> Builder:  
 return *self*.\_builder  
  
 @*builder*.setter  
 def builder(*self*, *builder*: Builder) -> None:  
 *self*.\_builder = *builder* def bake\_margheritta(*self*) -> None:  
 *self*.builder.make\_crust()  
 *self*.builder.add\_sauce()  
 *self*.builder.add\_tomato()  
 *self*.builder.add\_cheese()  
 *self*.builder.add\_basilic()  
 *self*.builder.add\_8()

În această segvență de cod este reprezentată clasa Baker care definește caracteristicile fiecărui obiect. Însă pentru a nu se crea de fiecare data obiect nou de la 0, am implementat și șablonul **Prototype.**

**Prototype** este un model de design creațional care permite copierea obiectelor existente fără a face codul să depindă de clasele lor.

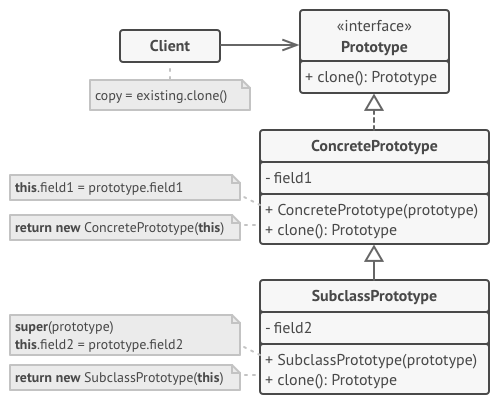


Figura 1.3 Șablonul Prototype

Pentru implementarea acestuia a fost implementată interfața Prototype cu metoda clone, iar apoi pentru fiecare subclasă a fost realizată prototipul concret

class MargherittaPizzaPrototype(PizzaPrototype):  
 def *\_\_init\_\_*(*self*):  
 *self*.\_pizza = *self*.create\_pizza()  
  
 def create\_pizza(*self*) -> Pizza:  
 builder = Pizza\_builder()  
 baker = Baker()  
 baker.builder = builder  
 baker.bake\_margheritta()  
 return builder.pizza  
  
 def clone(*self*) -> Pizza:  
 return deepcopy(*self*.\_pizza)

## **1.3 Șabloane Structurale**

Modelele de proiectare structurală explică modul de asamblare a obiectelor și claselor în structuri mai mari, păstrând în același timp aceste structuri flexibile și eficiente.

În realizarea proiectului am folosit 2 șabloane structurale:Adapter și Decorator.

**Adapter** este un model de design structural care permite obiectelor cu interfețe incompatibile să colaboreze.

Convertează interfața unei clase într-o altă interfață și așteptată de la clienți. Adaptorul permite claselor să lucreze împreună, ceea ce altfel nu ar putea din cauza interfețelor incompatibile.

Conectează o clasă existentă cu o interfață nouă.

Potrivirea impedanței unei componente vechi cu un sistem nou.

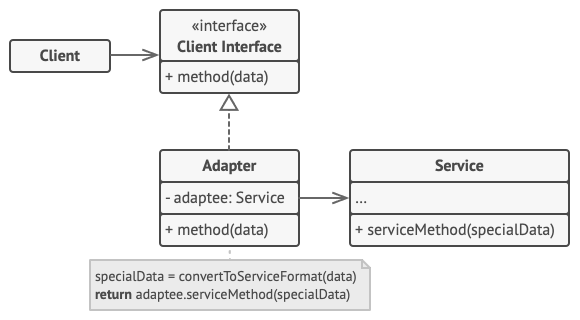


Figura 1.4 Șablonul Adapter

În cazul acestui proiect ,Adapter-ul a fost implementat pentru e exclude unele produse din pizza pentru persoanele care suferă de alergii la produsele respective, pentru aceasta s-a realizat o interfață care să fie compatibilă cu interfața de creare, ele sunt compatibile deoarece ambele operează cu proprietățile obiectului, primul adaugă iar adapterul va elimina,pentru început implementăm interfața

from abc import ABC, abstractmethod  
from builder import Pizza  
  
class PizzaIngredientRemover(ABC):  
 @abstractmethod

def remove(*self*, *pizza*: Pizza, *ingredient*: *str*) -> Pizza:  
 pass

Apoi se implementează însuși adapterul:

class PizzaIngredientRemoverAdapter(PizzaIngredientRemover):  
 def remove(*self*, *pizza*: Pizza, *allergen*: *str*) -> Pizza:  
 if *allergen* == 'gluten':  
 return *super*().remove(*pizza*, 'flour')  
 elif *allergen* == 'dairy':  
 return *super*().remove(*pizza*, 'cheese')  
 elif *allergen* == 'nuts':  
 return *super*().remove(*pizza*, 'nuts')  
 elif *allergen* == 'basilic':  
 return *super*().remove(*pizza*, 'basilic')  
 else:  
 return *super*().remove(*pizza*, *allergen*)  
 *#return pizza*

În cazul în care oaspetele specifică un allergen din cei specificați în cod, ingredientele respective sunt eliminate.

După ce este realizată comanda urmează partea de achitare, iar în cazul în care numarul de pizze comandate este mai mare decat 3,atunci se aplică un sistem de reduceri realizat prin Decorator.

**Decorator** permite unui utilizator să adauge o nouă funcționalitate unui obiect existent fără a-i modifica structura. Acest tip de model de design face parte din modelul structural, deoarece acest model acționează ca un înveliș pentru clasa existentă.

Acest model creează o clasă de decorator care înglobează clasa originală și oferă funcționalitate suplimentară, păstrând semnătura metodelor de clasă intactă.

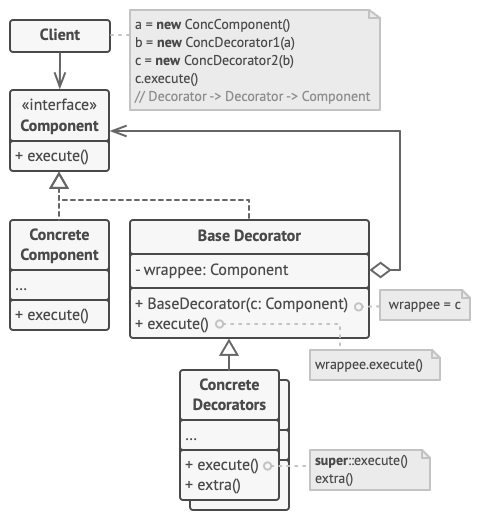


Figura 1.5 Șablon Decorator

Pentru implementarea acestui șablon creăm o clasă nouă „Order” care va avea ca atribute lista cu pizze comandate și prețul total. Iar apoi creăm clasa DiscountDecorator, pentru aplicarea reducerilor, iar calcularea noului preț se efectuează prin metoda „get\_price”

class DiscountDecorator:  
 def *\_\_init\_\_*(*self*, *component*, *discount\_percentage*):  
 *self*.component = *component  
 self*.discount\_percentage = *discount\_percentage* def get\_price(*self*):  
 original\_price = *self*.component.get\_price()  
 discounted\_price = original\_price - (original\_price \* *self*.discount\_percentage)  
 return discounted\_price

## **1.4 Șabloane comportamentale**

În ingineria software, modelele de proiectare comportamentală sunt modele de proiectare care identifică modele comune de comunicare între obiecte și realizează aceste modele. Procedând astfel, aceste modele cresc flexibilitatea în realizarea acestei comunicări.

La proiectarea acestei aplicații au fost folosite 2 șabloane de comportament: Memento și Chain of responsability.

**Chain of Responsability** este un model de design comportamental care vă permite să transmiteți cereri de-a lungul unui lanț de manipulatori. La primirea unei cereri, fiecare handler decide fie să proceseze cererea, fie să o transmită următorului handler din lanț.

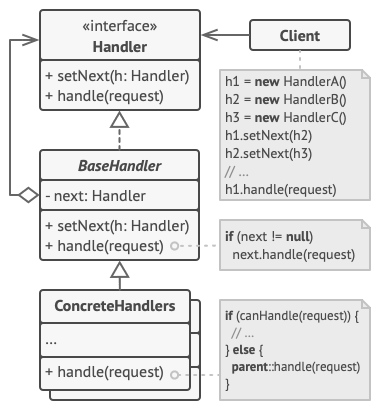


Figura 1.6 Șablonul Chain of Responsability

În acest proiect lanțul este creat din 3 manipulatori,primul este pentru adaugarea ingredientelor extra, al 2-lea manipulator verifică pasul de eliminare a unor produse,iar ultimul pas este cel de comandă.

class Handler(ABC):  
  
 @abstractmethod  
 def set\_next(*self*, *handler*: Handler) -> Handler:  
 pass  
  
 @abstractmethod  
 def handle(*self*, *request*) -> Optional[*str*]:  
 pass

Clasa abstractă handler, pentru a manipula fiecare pas din lanț,după aceaste se realizează altă clasă cu metode concrete,iar la final se implementează câte o clasă pentru fiecare pas.

class AddToppingHandler(AbstractHandler):  
 def handle(*self*, *request*: Any) -> *str*:  
 if *request* == "extra-topping":  
 return f"Extra toping was added {*request*}"  
 else:  
 return *super*().handle(*request*)

**Memento** este un model de design comportamental care vă permite să salvați și să restabiliți starea anterioară a unui obiect fără a dezvălui detaliile implementării acestuia.

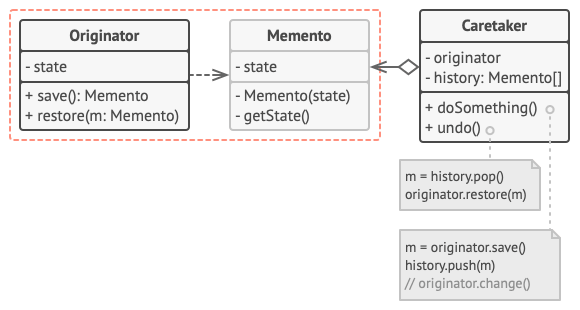


Figura 1.7 Șablonul Memento

În această aplicație fost implementat cu scopul de a realiza un pas înapoi în caz de greșeli în comandă sau dacă prețul este exagerat.

Pentru a realiza această sarcină,creăm 3 clase:originator,caretaker și Memento, în prima se realizează toți pașii când vrem să salvăm progresul, în Memento se stochează informația despre progres, iar în caretaker acțiunile de undo ,history sau backup.

class Originator():  
  
  
 \_state = None  
  
  
 def *\_\_init\_\_*(*self*, *state*: *str*) -> None:  
 *self*.\_state = *state  
 print*(f"Originator: My initial state is: {*self*.\_state}")  
  
 def add\_pizza(*self*) -> None:  
 *print*("Originator: new was added")  
 *self*.\_state = *self*.\_generate\_random\_string(30)  
 *print*(f"Originator: and my state has changed to: {*self*.\_state}")

Este prezentată doar o metodă, iar în continuare urmează clasa Caretaker:

class Caretaker():  
  
  
 def *\_\_init\_\_*(*self*, *originator*: Originator) -> None:  
 *self*.\_mementos = []  
 *self*.\_originator = *originator* def backup(*self*) -> None:  
 *print*("\nCaretaker: Saving Originator's state...")  
 *self*.\_mementos.append(*self*.\_originator.save())  
  
 def undo(*self*) -> None:  
 if not *len*(*self*.\_mementos):  
 return

Iar clasa Memento salveaă informația despre progress într-un anumit pas

class ConcreteMemento(Memento):  
 def *\_\_init\_\_*(*self*, *state*: *str*) -> None:  
 *self*.\_state = *state  
 self*.\_date = *str*(datetime.now())[:19]  
  
 def get\_state(*self*) -> *str*:  
  
 return *self*.\_state  
  
 def get\_name(*self*) -> *str*:  
  
  
 return f"{*self*.\_date} / ({*self*.\_state[0:9]}...)"  
  
 def get\_date(*self*) -> *str*:  
 return *self*.\_date

# **2. Descrierea principiului de lucru**

Principiile de muncă ale sabloanelor creatională facilitează crearea și utilizarea obiectelor într-un mod modular, flexibil și eficient. Aceste sabloane ne permit să separăm responsabilitățile legate de crearea și configurarea obiectelor de alte aspecte ale sistemului, precum logica de afaceri sau interacțiunea cu utilizatorul. Astfel, putem obține un cod mai modular, mai ușor de întreținut și cu o arhitectură mai flexibilă.

Aplicația are un funcțional simplu,dar suficient pentru realizarea unei comenzi depline.

La început este prezent un menu

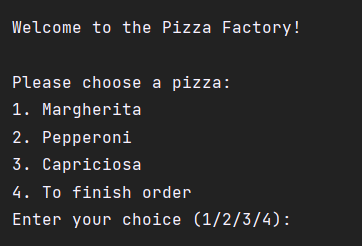


Figura 2.1 Meniul aplicației

În acest menu putem observa utilizarea șablonului Factory prin împărțirea pe tipuri. După alegerea uneia dintre aceste pizze, urmează alt pas,alegerea ingredientelor extra

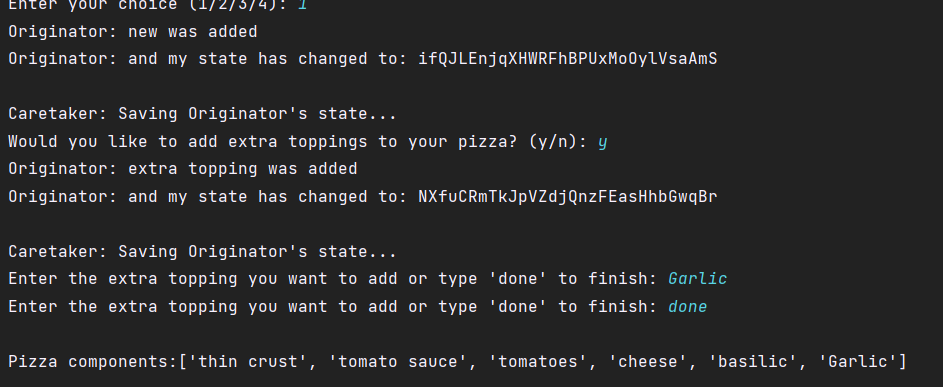


Figura 2.2 Alegerea ingredientelor extra

În imaginea anterioară pe lângă builder se observă și Memento Pattern , care salvează fiecare pas în realizarea comenzii.Dpă realizarea acestui lucru, urmează sa specificăm daca suntem alergici,pentru a înlătura unele ingrediente



Figura 2.3 Eliminarea ingredientelor

În acest pas este vizibilă implementarea adapterului,prin intermediul căruia se elimină careva ingrediente,iar după realizarea acestui pas ,apare din nou meniul inițial,pentru a comanda mai departe,iar daca repetăm înca de 2 ori aceleași acțiuni,v-o vedea alt meniu

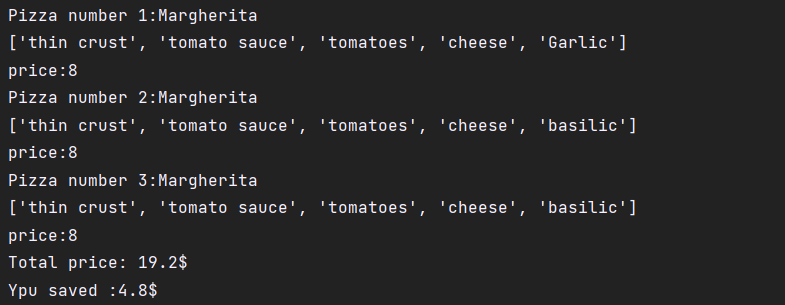


Figura 2.4 Rezultatul comenzii

La final se poate observa lista de obiecte cmandate, ingredientele fiecăreia, prețul total și cat a fost reducerea. Și avem și istoria de back-upuri a șablonului Memento

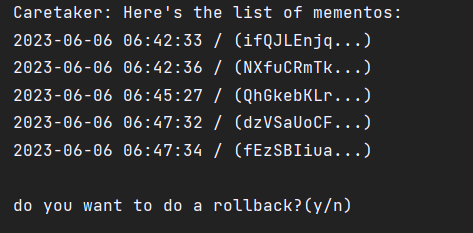


Figura 2.5 Lista de Memento-uri

# 

# **Concluzii**

În concluzie, proiectul de curs la disciplina "Tehnici și Mecanisme de Proiectare Software" pentru proiectarea unei aplicații pentru un magazin de pizzerie o oportunitate de aplicare și aprofundare a cunoștințelor teoretice în proiectarea software. Prin abordarea acestui proiect, s-a avut posibilitatea de a explora a diverselor aspecte și principii importante în procesul de proiectare a unei aplicații.

Proiectul a început prin identificarea problemelor, respectiv nevoia de dezvoltare a unei aplicații care să permită gestionarea comenzilor într-un magazin de pizza. S-a elaborat o descriere detaliată a problemelor și a cerințelor de stabilitate și funcționalităților necesare pentru a satisface această nevoie.

În continuare, s-au aplicat diverse sabloane de proiectare, inclusiv sabloane creaționale, pentru optimizarea procesului de creare și de gestionare a obiectelor în aplicare. S-au folosit sabloane precum Singleton, Factory și Builder pentru asigurarea unei creări eficiente și flexibile a obiectelor, facilitând astfel extensibilitatea și adaptabilitatea sistemului în viitor. Pe proiect, s-au explorat și alte principii și concepte importante în proiectarea software, cum ar fi separat responsabilitățile, modularitatea, extensibilitatea și flexibilitatea. S-a avut ocazia de a lucra cu arhitecturi de software și s-a înțeles importanța aplicării acestor principii pentru obținerea unui cod coerent, ușor de întreținut și scalabil.

În final, realizarea acestui proiect a condus la dobândirea unei mai bune dezvoltări a tehnologiilor și tehnicilor de proiectare software, precum și a importanței aplicării principiilor și a sabloanelor de proiectare în contextul dezvoltării unei aplicații complexe. Acest proiect a oferit oportunitatea de aplicare a cunoștințelor teoretice într-un proiect practic, dezvoltând abilități de proiectare și gândire critică în domeniul software.

# **Bibliografie**

* + 1. Șabloane Creaționale , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/creational-patterns>
    2. Șabloane Structurale , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/structural-patterns>
    3. Șabloane de Comportament , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/behavioral-patterns>

# 