**MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare Informatică şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Programul de studii Tehnologia Informației**

**RAPORT**

**Proiect de an la TMPS**

|  |  |  |
| --- | --- | --- |
| **Student:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Țurcanu Mihai,FCIM TI-203** |
| **A verificat:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Mihai Gaidau, asist.univ** |
|  |  |  |

**Chişinău, 2023**

Cuprins

[INTRODUCERE 3](#_Toc136894196)

[CONSIDERATII TEORETICE 4](#_Toc136894197)

[DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU 6](#_Toc136894198)

[MODELAREA SISTEMULUI UML 7](#_Toc136894199)

[CONCLUZII 13](#_Toc136894200)

[BIBLIOGRAFIE 14](#_Toc136894201)

# INTRODUCERE

În acest proiect, am implementat mai multe tipure de proiectare în cadrul unui sistem bancar simplu. Am utilizat următoarele 6 modele de design pentru descrierea sistemului : Builder pattern, Prototype pattern, Decorator pattern, Adapter pattern, Observer pattern și Strategy pattern.

Builder pattern a fost utizilazt pentru a construi obiecte complexe pas cu pas. În acest caz, clasa AccountBuilder este folosită pentru a construi obiecte de tip Account cu diferite caracteristici, precum număr de cont, tip de cont și sold.Prototype pattern ne permite clonarea obiectelor existente pentru a crea obiecte noi. Clasa abstractă AccountPrototype definește metodele pentru clonare și obținerea soldului unui cont, iar clasa Account implementează aceste metode și reprezintă un exemplu concret al unui cont bancar.Decorator pattern ne va permite atașarea de comportamente suplimentare la un obiect într-un mod flexibil. Clasa abstractă AccountDecorator extinde AccountPrototype și are o referință la un obiect AccountPrototype. Clasa OverdraftAccountDecorator extinde AccountDecorator și adaugă comportament suplimentar pentru un cont care permite descoperiri de cont. Acesta este doar un exemplu simplu, dar ar putea exista și alți decoratori care adaugă alte funcționalități.Adapter pattern este folosit pentru a converti interfețe incompatibile între ele. Interfața IBankServiceAdapter este implementată de clasa BankServiceAdapter, care servește ca un adaptor pentru clasa BankService. Adaptorul permite procesarea unui cont de tip AccountPrototype în cadrul serviciului bancar.Observer pattern: Acesta permite notificarea automată a obiectelor observatoare atunci când starea subiectului se schimbă. Interfața IObserver definește metoda Update, iar interfața IObservable definește metodele Attach, Detach și Notify. Clasa AccountObservable implementează IObservable și menține o listă de observatori care sunt notificați atunci când se actualizează soldul contului. Clasa AccountObserver implementează IObserver și afișează mesaje atunci când este notificată despre o actualizare a soldului contului.Strategy pattern are ca scop schimbarea algoritmului folosit într-o anumită situație fără a afecta codul clientului. Interfața IInterestStrategy definește metoda CalculateInterest, iar clasele SimpleInterestStrategy și CompoundInterestStrategy implementează această interfață și oferă diferite strategii de calcul al dobânzii pentru un cont. În plus, codul conține și câteva clase care reprezintă obiecte în cadrul sistemului bancar: BankService pentru procesarea conturilor, Customer pentru reprezentarea unui client bancar și Program care conține funcția Main și demonstrează utilizarea tuturor tiparelor de proiectare în contextul sistemului bancar.

# CONSIDERATII TEORETICE

Builder pattern este utilizat atunci când dorim să construim obiecte complexe pas cu pas, cu opțiuni și configurații variate, oferind în același timp flexibilitate în procesul de construcție. Prin intermediul unui obiect de tip builder, putem specifica și configura diferite caracteristici ale obiectului într-o manieră fluentă și ușor de utilizat. Astfel, putem evita constructori supraincarcați și neclarități în legătură cu parametrii necesari pentru inițializarea unui obiect. Builder pattern ne permite să separăm logica de construcție de implementarea obiectului în sine, permițând construirea obiectelor complexe într-un mod mai intuitiv și controlat.

Prototype pattern este utilizat atunci când dorim să creăm copii ale unui obiect existent, fără a recurge la crearea repetată a obiectelor similare prin intermediul constructorilor. Acest tipar ne permite să clonăm un obiect existent și să creăm rapid copii ale acestuia, păstrând astfel caracteristicile și starea originală. Prototype pattern oferă un mod eficient de a crea obiecte, mai ales atunci când costul creării unui obiect nou este ridicat sau când dorim să creăm obiecte cu configurații similare. În plus, prin utilizarea Prototype pattern, putem crea copii profunde ale obiectelor, astfel încât modificările aduse unui obiect clonat să nu afecteze obiectul original.

Decorator pattern este utilizat atunci când dorim să adăugăm comportamente suplimentare sau să modificăm dinamic comportamentul unui obiect, fără a modifica structura de bază a acestuia. Acest tipar ne permite să atașăm decoratori la un obiect existent, oferindu-i astfel noi funcționalități. Decorator pattern utilizează o abordare compozițională, în care obiectele sunt înconjurate de obiecte înveliș (decoratori) care pot adăuga sau modifica comportamentul obiectului de bază. Astfel, putem extinde funcționalitatea unui obiect fără a modifica codul sursă al acestuia. Decorator pattern oferă flexibilitate și extensibilitate în adăugarea și combinarea diferitelor comportamente la obiecte existente. Într-un scenariu Observer pattern, există două elemente principale: subiectul (obiectul observabil) și observatorii (obiectele care monitorizează subiectul). Subiectul are o listă de observatori la care este înregistrat și le trimite notificări atunci când are loc o schimbare de stare. Observatorii primesc aceste notificări și reacționează în consecință, fie prin preluarea noii stări, fie prin realizarea altor acțiuni în funcție de context.

Adapter pattern este utilizat atunci când dorim să convertim interfețele incompatibile sau să obținem interoperabilitate între două clase sau sisteme existente. Acest tipar ne permite să creăm un adaptor care se comportă ca un intermediar între două interfețe incompatibile, permițându-le să lucreze împreună. Adaptorul traduce cererile și răspunsurile între cele două interfețe, astfel încât să poată comunica și coopera eficient. Adapter pattern ne oferă posibilitatea de a conecta componente existente și de a extinde funcționalitatea lor, fără a modifica codul sursă al acestora.

Observer pattern este utilizat atunci când dorim să implementăm un sistem de notificări automatizate între obiecte, astfel încât schimbările într-un obiect să fie reflectate în mod automat la toți observatorii săi. Acest tipar este util în situațiile în care avem obiecte care trebuie să fie informate despre modificările de stare ale altor obiecte și să reacționeze în consecință. Observer pattern se bazează pe principiul separării celor două elemente principale: subiectul (obiectul observabil) și observatorii (obiectele care monitorizează subiectul). Atunci când subiectul se schimbă, acesta notifică observatorii, care pot reacționa în consecință.

Strategy pattern este utilizat atunci când dorim să selectăm și să schimbăm algoritmul specific utilizat într-un context dat, fără a modifica structura codului clientului. Acest tipar ne permite să definim diferite strategii (algoritmi) într-o ierarhie de clase și să le înlocuim în timpul rulării programului, în funcție de necesități sau preferințe. Strategy pattern promovează flexibilitatea și modularitatea, deoarece separă logica de afaceri de implementarea specifică a algoritmului. Astfel, putem extinde și adăuga ușor noi strategii fără a afecta codul clientului și fără a repeta logica comună. Prin intermediul Strategy pattern, putem izola logica specifică a fiecărei strategii în clase separate, ceea ce facilitează înțelegerea și menținerea codului. În plus, Strategy pattern promovează principiul deschis-închis, permițând adăugarea ușoară a unor noi strategii și extinderea funcționalității, fără a modifica codul deja existent.Utilizând Strategy pattern, putem oferi flexibilitate și modularitate în proiectarea sistemelor, permițând clienților să selecționeze și să schimbe strategiile în funcție de cerințele și preferințele lor. Aceasta face Strategy pattern un instrument puternic pentru gestionarea algoritmilor variabili și adaptarea sistemelor la diverse situații și cerințe de afaceri.

# **DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU**

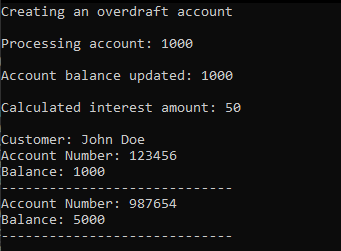
Aplicația data relevă o implementare simplificată a mai multor tipare de proiectare într-un sistem bancar. Scopul acestui cod este de a ilustra modul în care funcționează și cum pot fi utilizate tiparele de proiectare Builder, Prototype, Decorator, Adapter, Observer și Strategy.

Datele de intrare în acest cod includ informații despre :

* clienți,
* conturi bancare ,
* strategii de dobândă

Aceste date sunt utilizate pentru a crea și configura obiecte corespunzătoare, precum și pentru a realiza operații specifice pe acestea, cum ar fi calculul dobânzii și afișarea informațiilor despre conturi.

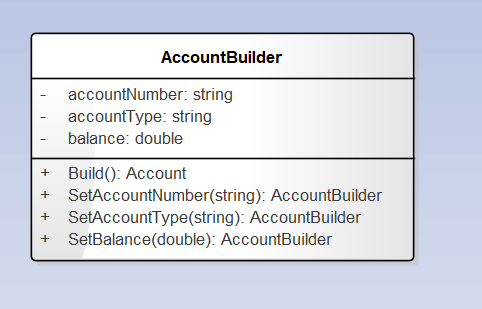
Datele de ieșire ale acestui cod includ afișarea detaliilor despre clienți și conturile lor, precum și afișarea dobânzii calculate pe baza strategiei selectate.



**Figura 3.1** Rezultatul final

Aplicația oferă o implementare simplă a mai multor tipare de proiectare, prezentând modul în care acestea pot fi utilizate într-un sistem bancar. Aceste tipare facilitează crearea și configurarea obiectelor, extinderea funcționalității prin decorare, adaptarea interfețelor existente și gestionarea observării și notificărilor între obiecte. Acest cod ilustrează modul în care tiparele de proiectare pot îmbunătăți structura și modularitatea sistemelor software.

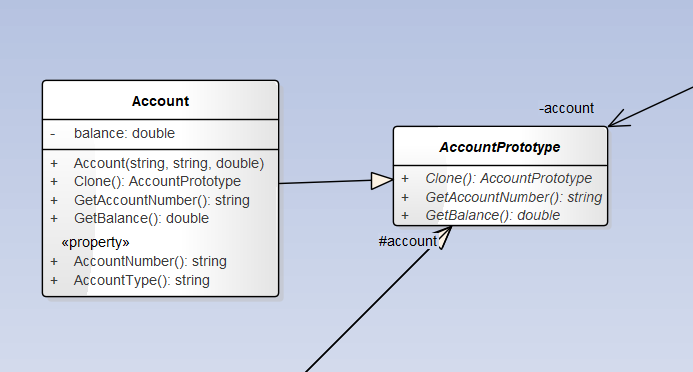
# MODELAREA SISTEMULUI UML



**Fig. 4.1** Builder Design pattern

Diagrama reprezintă implementarea unui builder pentru obiectele de tip Account. Builder-ul este o clasă care facilitează construirea obiectelor complexe într-un mod pas cu pas, permițând setarea individuală a diferitelor proprietăți ale obiectului.Clasa AccountBuilder are câteva câmpuri private, cum ar fi accountNumber, accountType și balance, care reprezintă proprietățile unui obiect de tip Account. Metodele SetAccountNumber, SetAccountType și SetBalance sunt utilizate pentru a seta valorile acestor proprietăți într-un mod incremental. Aceste metode returnează obiectul AccountBuilder în sine, permițând concatenarea lor într-o manieră fluentă.Metoda Build este responsabilă pentru crearea efectivă a obiectului de tip Account, utilizând valorile setate anterior. Ea instantiază un nou obiect Account cu ajutorul constructorului acestuia, folosind valorile furnizate pentru numărul de cont, tipul de cont și soldul.Acest builder permite crearea ușoară a obiectelor de tip Account, oferind un mod flexibil și clar de a specifica valorile proprietăților obiectului în timpul construirii.

Prin utilizarea unui builder, dezvoltatorii pot evita construirea obiectelor cu mulți parametri sau folosirea constructorilor supraincarcati, ceea ce duce la un cod mai lizibil și mai ușor de întreținut. Astfel, se poate obține un obiect Account complet inițializat, cu valori specifice pentru numărul de cont, tipul de cont și soldul.

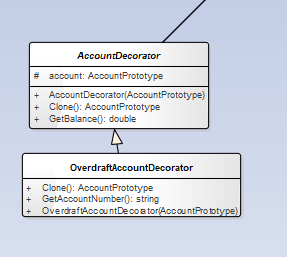


**Fig. 4.2** Prototype pattern

Clasa Account extinde clasa abstractă AccountPrototype și implementează metodele abstracte definite în aceasta. Această clasă conține câmpuri pentru numărul de cont (AccountNumber), tipul de cont (AccountType) și sold (balance). Constructorul clasei Account primește aceste valori și le atribuie câmpurilor corespunzătoare. Metoda Clone este suprascrisă pentru a returna o copie shallow (prin apelul metodei MemberwiseClone()) a obiectului curent, ceea ce permite clonarea obiectului Account. Metoda GetBalance returnează soldul asociat contului, iar metoda GetAccountNumber returnează numărul de cont.Clasa abstractă AccountPrototype definește metodele abstracte Clone, GetBalance și GetAccountNumber, care vor fi implementate în clasele derivate. Aceasta servește ca un tip de bază pentru obiectele care pot fi clonate și oferă o interfață comună pentru acestea.

Utilizarea acestor două clase în combinație permite crearea și clonarea ușoară a obiectelor de tip Account. Clasa Account definește comportamentul specific al unui cont, iar clasa abstractă AccountPrototype definește contractul pe care orice clasă derivată trebuie să-l respecte.

Prin utilizarea Prototype Pattern, dezvoltatorii pot crea copii ale obiectelor existente fără a depinde de detaliile implementării și pot evita crearea obiectelor dinamic, reducând astfel costurile de creare a obiectelor. Clonarea este utilă atunci când se dorește crearea unui nou obiect cu aceleași proprietăți precum un alt obiect existent, fără a folosi constructorul și inițializând astfel obiectul mai rapid și mai eficient. Prin utilizarea Prototype Pattern, dezvoltatorii pot crea copii ale obiectelor existente fără a depinde de detaliile implementării și pot evita crearea obiectelor dinamic, reducând astfel costurile de creare a obiectelor. Clonarea este utilă atunci când se dorește crearea unui nou obiect cu aceleași proprietăți precum un alt obiect existent, fără a folosi constructorul și inițializând astfel obiectul mai rapid și mai eficient.

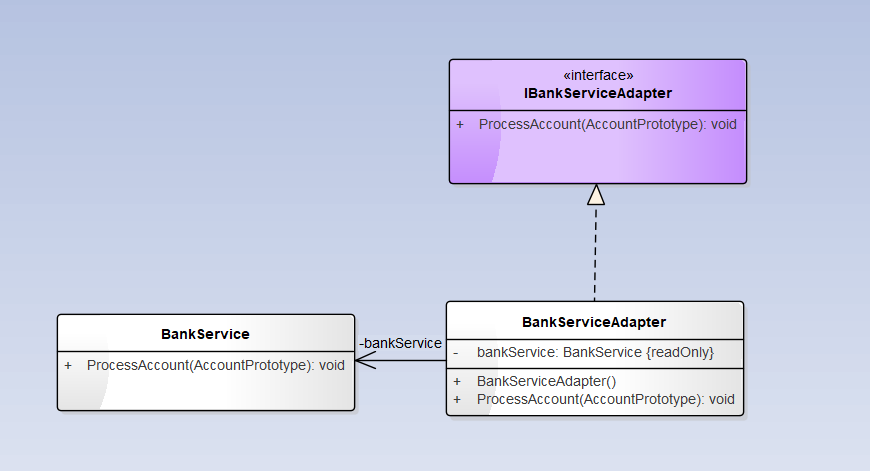


**Fig. 4.3** Decorator pattern

Observam că se utilizează conceptul de Decorator Pattern pentru a extinde comportamentul unui obiect de tip AccountPrototype. Clasa abstractă AccountDecorator extinde clasa AccountPrototype și primește un obiect de tip AccountPrototype ca parametru în constructor. Această clasă conține un câmp account care reține referința către obiectul pe care îl decorează. Metodele Clone și GetBalance sunt suprascrise pentru a redirecționa apelurile către obiectul decorat (account). Astfel, orice apel la metodele Clone și GetBalance ale obiectului decorator vor fi pasate în mod transparent către obiectul decorat.

Clasa OverdraftAccountDecorator extinde clasa AccountDecorator și adaugă comportament suplimentar pentru un cont cu depășire de limită. Constructorul clasei primește un obiect AccountPrototype și apelează constructorul clasei de bază cu acest obiect. Metoda Clone suprascrie metoda de bază și, în plus față de clonarea obiectului decorat, efectuează și o logică adițională specifică contului cu depășire de limită. În acest exemplu, se afișează mesajul "Creating an overdraft account" pentru a indica faptul că se creează un cont cu depășire de limită. Metoda GetAccountNumber este suprascrisă pentru a redirecționa apelurile către obiectul decorat (account).

Prin utilizarea Decorator Pattern, putem extinde comportamentul unui obiect în mod dinamic, adăugând funcționalități suplimentare la obiectul de bază fără a modifica direct clasa de bază. Obiectul decorat poate fi utilizat în aceeași manieră ca și obiectul de bază, dar cu funcționalități suplimentare adăugate de decorator.



**Fig. 4.4** Adapter pattern

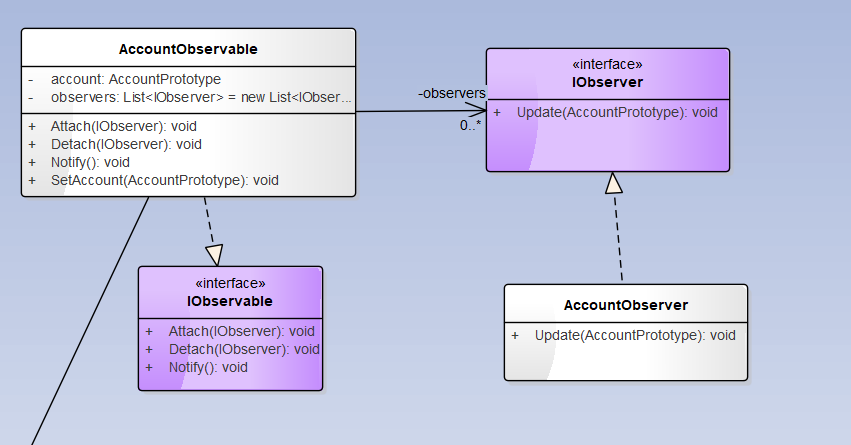
Aceste trei clase ilustrează utilizarea pattern-ului Adapter pentru a adapta interfața unui serviciu bancar existent pentru a putea procesa obiecte de tip AccountPrototype.

Prima clasă, BankService, reprezintă un serviciu bancar existent și conține o metodă ProcessAccount care primește un obiect de tip AccountPrototype și afișează un mesaj de procesare a contului.

A doua clasă, BankServiceAdapter, implementează interfața IBankServiceAdapter, care definește o metodă ProcessAccount. Această clasă servește ca adaptor între serviciul bancar existent (BankService) și interfața IBankServiceAdapter. Constructorul clasei BankServiceAdapter creează o instanță a serviciului bancar existent și înregistrează-o într-un câmp privat. Metoda ProcessAccount a adaptorului redirecționează apelul către metoda ProcessAccount a serviciului bancar existent.

Ultima clasă, IBankServiceAdapter, este o interfață care definește metoda ProcessAccount. Această interfață servește ca contract între adaptor și orice client care folosește adaptorul.

Prin utilizarea pattern-ului Adapter, se permite adaptarea unui serviciu existent la o interfață specifică, astfel încât să poată fi utilizat într-un context diferit sau cu alte obiecte. În acest caz, adaptorul BankServiceAdapter permite utilizarea serviciului bancar existent (BankService) pentru a procesa obiecte de tip AccountPrototype, în conformitate cu interfața IBankServiceAdapter. Aceasta facilitează integrarea serviciului bancar existent într-un sistem care utilizează obiecte de tip AccountPrototype, fără a modifica direct serviciul bancar.

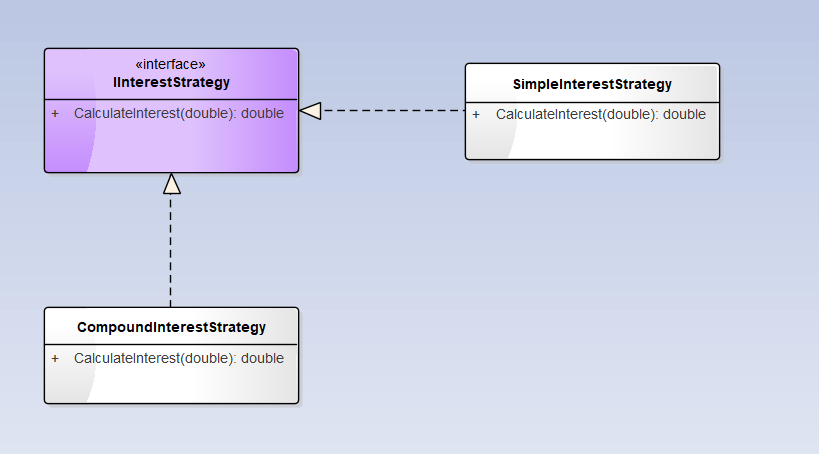


**Fig. 4.5** Observer pattern

Interfața IObservable definește metodele necesare pentru gestionarea observatorilor. Metodele Attach și Detach sunt utilizate pentru adăugarea și eliminarea observatorilor din lista de observatori. Metoda Notify este responsabilă pentru notificarea tuturor observatorilor înregistrati atunci când se produce o actualizare.Interfața IObserver definește metoda Update, care este apelată atunci când subiectul trimite o notificare observatorilor. În implementarea dată, clasa AccountObserver reprezintă un observator care afișează un mesaj în momentul actualizării unui cont.

Clasa AccountObservable implementează interfața IObservable și gestionează lista de observatori și subiectul observat, care este de tip AccountPrototype. Metoda Attach adaugă un observator în lista de observatori, Detach îl elimină, iar Notify parcurge lista de observatori și apelează metoda Update a fiecărui observator înregistrat. Metoda SetAccount actualizează subiectul observat și apoi apelează metoda Notify pentru a trimite notificarea la toți observatorii înregistrați.

În acest fel, implementarea pattern-ului Observer permite observatorilor să se aboneze și să primească notificări atunci când se produce o actualizare într-un subiect. În exemplul dat, atunci când un cont este actualizat prin apelul metodei SetAccount, observatorii înregistrați vor primi notificarea și vor afișa un mesaj corespunzător, în cazul clasei AccountObserver.



**Fig. 4.6** Strategy pattern.

Interfața IInterestStrategy definește metoda CalculateInterest care este implementată în clasele SimpleInterestStrategy și CompoundInterestStrategy. Aceste clase reprezintă strategiile specifice de calcul al dobânzii.Clasa SimpleInterestStrategy implementează metoda CalculateInterest într-un mod specific, în cazul dat, calculând dobânda ca un procent fix de 5% din soldul contului.

Clasa CompoundInterestStrategy implementează, de asemenea, metoda CalculateInterest, dar într-un mod diferit față de SimpleInterestStrategy. În cazul dat, calculul dobânzii se face prin înmulțirea soldului contului cu 10%, reprezentând o dobândă compusă.

Prin utilizarea acestor strategii, se permite flexibilitatea de a alege și schimba strategia de calcul al dobânzii în funcție de nevoile specifice ale aplicației sau ale utilizatorului. De exemplu, un cont poate utiliza SimpleInterestStrategy într-un anumit moment și apoi poate trece la CompoundInterestStrategy în alt moment, fără a afecta structura de bază a codului.

# CONCLUZII

În elaborării acest proiect, am implementat mai multe tipure de proiectare în cadrul unui sistem bancar. Am utilizat următoarele 6 modele de design pentru descrierea sistemului : Builder pattern, Prototype pattern, Decorator pattern, Adapter pattern, Observer pattern și Strategy pattern.

Pattern-ul Builder este folosit pentru a construi obiecte de tip cont bancar. Clasa AccountBuilder permite setarea diferitelor proprietăți ale unui cont, precum numărul de cont, tipul de cont și soldul, și construiește un obiect Account în final.

Pattern-ul Prototype este folosit pentru a clona obiecte de tip cont bancar. Clasa Account implementează interfața AccountPrototype și definește o metodă de clonare, care creează o copie superficială a obiectului.

Pattern-ul Decorator este folosit pentru a extinde comportamentul unui cont bancar existent. Clasa abstractă AccountDecorator este extinsă de clasa OverdraftAccountDecorator, care adaugă funcționalitatea de cont cu descoperire de sold.Pattern-ul Adapter este utilizat pentru a adapta o clasă existentă BankService la interfața IBankServiceAdapter. Clasa BankServiceAdapter permite utilizarea metodei ProcessAccount din BankService într-un mod adaptat la interfața AccountPrototype.Pattern-ul Observer este utilizat pentru a notifica observatorii atunci când un cont bancar este actualizat. Clasele AccountObservable și AccountObserver permit observarea și actualizarea conturilor bancare.

Pattern-ul Strategy este utilizat pentru a implementa diferite strategii de calcul al dobânzii pentru un cont bancar. Interfața IInterestStrategy este implementată de clasele SimpleInterestStrategy și CompoundInterestStrategy, permițând calculul dobânzii în funcție de strategia specifică.

În plus, aplicația definește și o clasă Customer care reprezintă un client bancar și permite adăugarea, eliminarea și afișarea conturilor bancare ale clientului.Prin utilizarea acestor pattern-uri de proiectare, aplicația devine flexibilă, ușor de extins și permite gestionarea diferitelor tipuri de conturi bancare și strategii de calcul al dobânzii.

# BIBLIOGRAFIE

Design Patterns [citat 03.06.2023] Disponibil:

<https://sourcemaking.com/design_patterns>

Refactoring Guru [citat 05.06.2023] Disponibil:

<https://refactoring.guru/>

C# Design Patterns [citat 05.06.2023] Disponibil:

<https://www.dofactory.com/net/design-patterns>