

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

# Raport

**Tema:** ”Sistem de management al relațiilor cu clienții”

**A efectuat:**

st. grupei TI – 202, Amarfii Eugenia

**Verificat:**

asistent universitar, Gaidău Mihail

**Chișinău 2023**

# Cuprins

Introducere .....	3
1. Analiza domeniului de studiu.....	4
1.1. Cerințe funcționale ale sistemului .....	7
1.2. Cerințe nefuncționale ale sistemului .....	8
2. Modelarea și proiectarea sistemului informatic.....	10
2.1. Diagrama de cazuri de utilizare (Use Case Diagram):.....	10
2.2. Diagrama de clasă (Class Diagram): .....	11
2.3. Diagrama de secvență (Sequence Diagram): .....	11
2.4. Diagrama de stări (State Diagram): .....	12
3. Realizarea sistemului .....	13
3.1. Descrierea limbajului C#.....	13
3.2. Descrierea tehnologiei Entity Framework .....	13
3.3. Descrierea platformei .NET 6.....	15
3.4. Descrierea ASP.NET MVC: .....	16
3.5. Descrierea bazei de date MySQL .....	18
5. Design Patterns .....	21
5.1. Tipuri de Design Patterns .....	22
5.2. Avantajele Design Patterns:.....	23
5.3. Design Patterns implementare .....	24
5.3.1. Singleton.....	24
5.3.2. Factory Method .....	25
5.3.3. Observer Pattern .....	26
5.3.4. Decorator Pattern.....	28
5.3.5. Strategy Pattern .....	29
5.3.6. Proxy Pattern .....	31
5.3.7. Template Method Pattern.....	33
6. Interfața sistemului.....	35
Concluzie.....	39
Bibliografie.....	40

## Introducere

Sistemul de management al relațiilor cu clienții (CRM) este o componentă esențială în industria modernă a afacerilor, care își propune să ușureze și să optimizeze relațiile între organizații și clienți. Acest sistem complex de software oferă o serie de instrumente și funcționalități care permit organizațiilor să gestioneze eficient interacțiunile cu clienții, să optimizeze procesele de vânzări și să îmbunătățească legătura cu clienților.

Într-o lume competitivă și dinamică, construirea și menținerea relațiilor solide cu clienții este esențială pentru succesul unei afaceri. Un sistem de management al relațiilor cu clienții (CRM) oferă instrumente și funcționalități esențiale pentru a ajuta organizațiile să obțină o înțelegere mai bună a clienților săi, să optimizeze interacțiunile și să eficientizeze procesele de vânzări și marketing.

Scopul principal al unui sistem CRM este să ofere o imagine de ansamblu asupra clienților și interacțiunilor cu aceștia. Acesta stochează și gestionează informații despre clienți, cum ar fi date de contact, istoricul achizițiilor, preferințe și feedback-ul clienților. Prin centralizarea acestor informații, organizațiile pot obține o viziune completă a clienților și pot oferi servicii personalizate, adaptate nevoilor și preferințelor individuale.

Astfel, dacă să luăm în considerare și ce beneficii putem urmări în urma utilizării unui sistem CRM, ulterior putem enumera câteva din ele, ce cred că sunt și esențiale. Deci, dacă să vorbim despre îmbunătățirea relației cu clienții: un CRM eficient permite o comunicare mai bună și o mai mare personalizare a serviciilor oferite clienților, ceea ce duce la creșterea satisfacției și loialității acestora. Precum, și creșterea eficienței operaționale, cum ar fi un CRM poate ajuta la automatizarea și optimizarea unor sarcini repetitive și administrative, permițând angajaților să se concentreze mai mult pe activități importante și pe îmbunătățirea experienței clienților.

Care, ulterior va permite analiza datelor și luarea deciziilor, astfel, acest sistem CRM poate oferi analize și rapoarte detaliate despre performanța, tendințe ale clienților și alte informații importante, ajutând organizațiile să ia decizii mai informate și să-și îmbunătățească strategiile de afaceri.

Așadar, acest sistem de management al relațiilor cu clienții (CRM) este un instrument după părerea mea, esențial pentru orice organizație care dorește să își optimizeze interacțiunile cu clienții și să își îmbunătățească performanța în domeniul vânzărilor și al serviciilor oferite. Prin centralizarea și gestionarea informațiilor despre clienți, un CRM facilitează personalizarea serviciilor, îmbunătățește comunicarea și contribuie la creșterea satisfacției clienților. Utilizarea unui CRM poate aduce multiple avantaje organizației, inclusiv creșterea eficienței, îmbunătățirea luării deciziilor ce sunt bazate pe date.

## 1. Analiza domeniului de studiu

Analiza domeniului de studiu al sistemului de management al relațiilor cu clienții (CRM) reprezintă o etapă esențială în dezvoltarea și implementarea unui sistem eficient. Analiza dată are ca scop înțelegerea a nevoilor și cerințelor organizației, precum și a proceselor de afaceri existente pentru a identifica soluțiile potrivite pentru implementarea unui sistem personalizat și optimizat. Astfel, vom cerceta importanța analizei domeniului în implementarea unui astfel de sistem și vom discuta despre aspectele cheie care trebuie luate în considerare în timpul acestei analize.

Astfel, sistemul de management al relațiilor cu clienții (CRM) este o componentă critică, în strategia de afaceri a oricărei organizații, care dorește să își optimizeze interacțiunile cu clienții și să își dezvolte relațiile. Ulterior, ce ține de obiectivele analizei domeniului, ele presupun, înțelegerea nevoilor organizaționale, cum ar fi, identificarea nevoilor și cerințelor organizației în ceea ce privește gestionarea relațiilor cu clienții. Aceasta implică analizarea obiectivelor organizaționale, a modelelor de afaceri și a proceselor existente pentru a identifica lacunele și oportunitățile de îmbunătățire. Deci, următoarea este identificarea proceselor critice, presupune o gestionare eficientă a relațiilor cu clienții, deci ceea ce implică identificarea cerințelor specifice ale organizației în ceea ce privește funcționalitățile, integrările cu alte sisteme, securitatea datelor, personalizarea, raportarea și altele. Aceste cerințe vor ghida selecția și configurarea soluției a unui sistem bine definit și care este adaptat anume cerințelor asupra căror apare necesitatea.

Ulterior, se va concretiza câteva aspecte cheie ce țin de studiul pieței și a concurenței, astfel, aceasta implică pentru a înțelege tendințele actuale, practicile comune și inovațiile din industrie. Aceste informații pot oferi îndrumări valoroase pentru dezvoltarea unei soluții a acestui sistem ce poate fi competitiv și adaptate nevoilor clienților.

Iar, pentru a obține o înțelegere profundă a nevoilor organizației, este esențial să se efectueze interviuri și discuții cu utilizatorii cheie, cum ar fi managerii, reprezentanții serviciului clienți, specialiștii în marketing și alții ce pot aduce o informație concludentă și neapărat cu scop de facilitare a muncii, ce vor ajuta la identificarea proceselor critice și a cerințelor specifice ale fiecărui departament.

Dacă să analizăm datele existente, precum dacă organizația deține deja date relevante despre clienți, analiza acestor date poate oferi insights deci niște perspective importante despre comportamentul clienților, preferințele lor, și altele, din punct de vedere, aceste informații pot ghida configurarea și personalizarea sistemului. Ulterior, ce ține de identificarea obstacolelor și a provocărilor, acestea pot include aspecte legate de schimbarea culturii organizaționale, rezistența angajaților la adoptarea noilor tehnologii sau alte probleme specifice domeniului de activitate.

Analiza domeniului de studiu al sistemului de management al relațiilor cu clienții (CRM) este o etapă esențială în implementarea cu succes a unui sistem. Această analiză permite înțelegerea nevoilor, a proceselor critice și a cerințelor specifice, asigurând dezvoltarea și configurarea unei sistem adaptat nevoilor și obiectivelor companiei. Prin identificarea obstacolelor și provocărilor specifice, se pot lua măsuri pentru a depăși aceste probleme și a asigura succesul implementării sistemului dat.

În continuare, voi descrie trei sisteme de management cu clienții, ce sunt deja existente pe piață și voi realiza o comparație între ele în funcție de diferite criterii relevante.

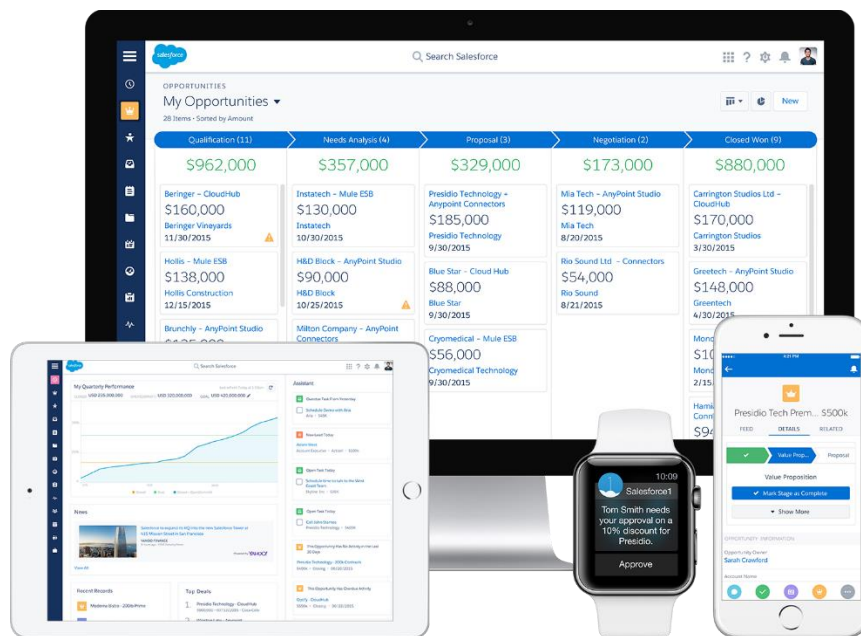


Figura 1.1 – Sistemul de management cu clienții Salesforce

Deci să începem cu unul dintre cele mai populare și puternice sisteme disponibile pe piață Salesforce, ce este reprezentată interfața cu utilizatorul, în figura 1.1 de mai sus. Acest sistem oferă o gamă largă de funcționalități, inclusiv gestionarea datelor clienților, automatizarea vânzărilor, marketingul și serviciul clienți. Salesforce este o soluție cloud-based, ceea ce permite accesul facil la date și funcționalități de oriunde și de pe orice dispozitiv conectat la internet.

Astfel, platforma oferă o interfață intuitivă și personalizabilă, precum și capacitatea de a extinde funcționalitățile prin aplicații și integrări cu alte sisteme. De asemenea, Salesforce oferă o gamă variată de suport și servicii pentru implementarea și personalizarea sistemului.

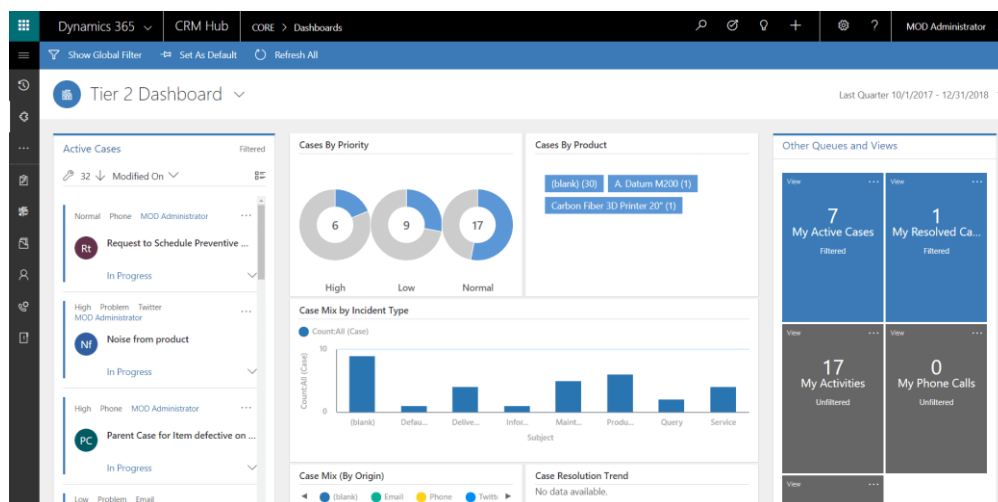


Figura 1.2 – Sistemul de management cu clienții Microsoft Dynamics 365

Următorul sistem este Microsoft Dynamics 365, reprezentat în figura 1.2, de mai sus, astfel este o suită cuprinzătoare de soluții de afaceri, care include și un modul CRM puternic. Această soluție oferă funcționalități de gestionare a relațiilor cu clienții, vânzări, marketing, servicii și alte aspecte relevante. Microsoft Dynamics 365 poate fi personalizat și configurat în funcție de nevoile organizației, iar integrarea cu alte produse Microsoft, cum ar fi Outlook și Excel, facilitează utilizarea și sincronizarea datelor. De asemenea, Microsoft oferă suport și servicii de implementare și mentenanță pentru acest sistem.

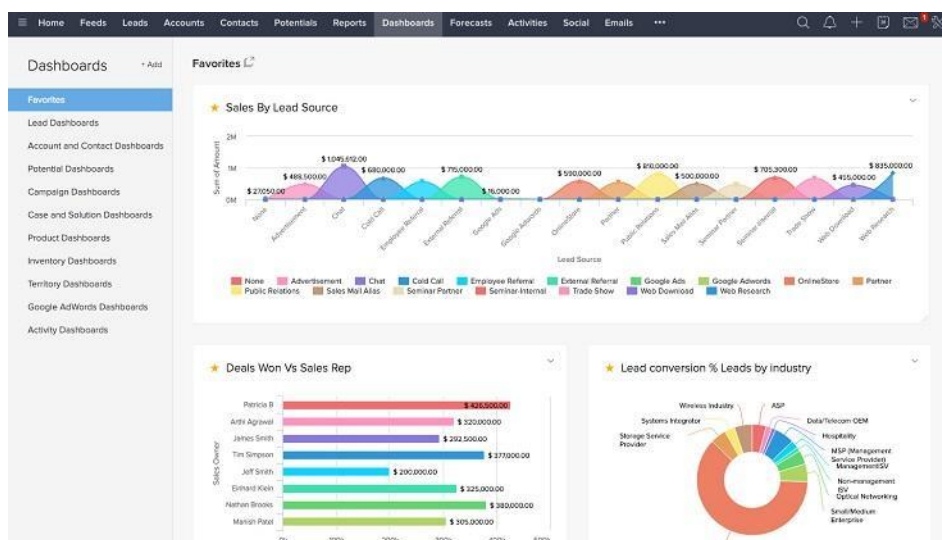


Figura 1.3 – Sistemul de management cu clienții Zoho

În final, sistemul Zoho CRM, care este reprezentată interfața cu utilizatorul în figura 1.3 de mai sus, ce presupune o soluție CRM accesibilă și ușor de utilizat, potrivită pentru organizațiile mici și mijlocii.

Aceasta oferă funcționalități de bază pentru gestionarea relațiilor cu clienții, vânzări, marketing și servicii. Zoho CRM vine cu o interfață intuitivă, personalizabilă și cu un set de instrumente de raportare și analiză. Platforma permite, de asemenea, integrarea cu alte aplicații Zoho și cu alte sisteme terțe. Zoho oferă suport și servicii de asistență tehnică pentru implementare și utilizare.

Astfel, dacă să comparăm sistemele date, după funcționalități, vom identifica că, Salesforce oferă o gamă largă de funcționalități avansate, potrivite pentru organizații de diferite dimensiuni și industrii, iar Microsoft Dynamics 365 oferă o integrare strânsă cu alte produse Microsoft și o suită completă de soluții de afaceri, care ulterior Zoho CRM oferă funcționalități de bază, potrivite pentru organizații mai mici și cu bugete mai reduse.

Ulterior, vom aborda sistemul de personalizare și extensibilitate, a sistemelor Salesforce și Microsoft Dynamics 365, ele oferă opțiuni extinse de personalizare și configurare în funcție de nevoile organizației. Care la rândul său Zoho CRM permite și el personalizarea, însă oferind cu o gamă mai limitată de opțiuni.

Deci, ce ține de prețul acestor sisteme, putem spune că Salesforce este cunoscut pentru a avea prețuri mai ridicate, mai ales pentru organizațiile mari, iar Microsoft Dynamics 365 are o gamă variată de pachete de preț, în funcție de funcționalități și nevoile organizației. Însă, Zoho CRM este mai accesibil ca preț, cu opțiuni de plată flexibile.

Așadar, ceea ce presupune suportul și serviciile acestui sistem precum Salesforce și Microsoft oferă servicii de implementare și suport tehnic extinse, iar Zoho oferă suport și asistență tehnică, dar cu o acoperire mai limitată. Este important faptul că există necesitatea de a se lua în considerare nevoile și cerințele specifice ale organizației în alegerea unui sistem CRM. Fiecare dintre aceste sisteme are punctele sale forte și potrivirea depinde de cerințele și bugetul organizației.

### **1.1. Cerințe funcționale ale sistemului**

Deci, ulterior, vom aborda cerințele funcționale și nefuncționale ale sistemului, precum, gestionarea datelor clienților, și altele.

Astfel, gestionarea datelor clienților, presupune:

- Capacitatea de a colecta, stoca și actualiza informații detaliate despre clienți, inclusiv informații de contact, preferințe, istoricul interacțiunilor.
- Funcționalități de căutare și filtrare pentru a accesa rapid și eficient datele specifice ale clienților.

Automatizarea fluxului de muncă:

- Gestionarea, inclusiv generarea de lead-uri, urmărirea oportunităților.
- Funcționalități de automatizare a fluxurilor de muncă, cum ar fi asignarea automată a sarcinilor, notificările și rapoartele de progres.

Marketing și campanii:

- Gestionarea campaniilor de marketing, inclusiv crearea și urmărirea campaniilor de e-mail, marketingul pe rețele sociale, publicitatea online.
- Segmentarea bazei de date a clienților și personalizarea mesajelor de marketing pentru grupuri specifice de clienți.

Servicii și asistență clienți:

- Gestionarea solicitărilor și problemelor clienților, inclusiv crearea și urmărirea tichetelor de suport, escaladarea problemelor către niveluri superioare.
- Integrarea cu canale de comunicare, cum ar fi chat-ul în timp real, suportul telefonic etc.

Raportare și analiză:

- Generarea de rapoarte și analize detaliate privind performanța de comunicare cu clienții, eficiența campaniilor de marketing.
- Vizualizarea datelor sub formă de grafice, tabele și alte formate ușor de înțeles.

## **1.2. Cerințe nefuncționale ale sistemului**

Cerințe nefuncționale ale sistemului:

Securitate:

- Asigurarea securității și confidențialității datelor clienților prin implementarea măsurilor de securitate adecvate, cum ar fi criptarea datelor, controlul accesului.
- Backup-ul datelor pentru a preveni pierderile sau accesul neautorizat.

Performanță:

- Asigurarea unui timp de răspuns rapid și a unei experiențe fără întreruperi în timpul accesului și utilizării sistemului CRM, indiferent de volumul datelor sau numărul utilizatorilor.
- Capacitatea de a scala sistemul pentru a face față creșterii volumului de date și numărului de utilizatori.

Ușurința în utilizare:

- Interfață intuitivă și ușor de înțeles, astfel încât utilizatorii să poată naviga și utiliza sistemul fără a necesita instruire extensivă.
- Documentație și asistență disponibile pentru a sprijini utilizatorii în utilizarea sistemului CRM.



Integrarea și eficiența sistemului:

- Capacitatea de a se integra cu alte sisteme și aplicații utilizate în organizație, cum ar fi sistemele de contabilitate, e-mail.
- Transferul și partajarea datelor între sistemul CRM și alte aplicații fără pierderea sau coruperea informațiilor.

Astfel, cerințe de descrise mai sus, precum cele funcționale și nefuncționale oferă o bază pentru dezvoltarea și implementarea unui sistem de management cu clienții, eficient și adaptat nevoilor și obiectivelor organizației.

## 2. Modelarea și proiectarea sistemului informatic

Metoda de proiectare UML (Unified Modeling Language) este o metodă standard utilizată pentru a modela și proiecta sistemele software. UML oferă un set de diagrame și notații grafice pentru a reprezenta diferitele aspecte ale unui sistem și relațiile dintre acestea. În cazul proiectării unui sistem CRM, următoarea abordare UML poate fi utilizată:

### 2.1. Diagrama de cazuri de utilizare (Use Case Diagram):

Această diagramă reprezintă interacțiunile între utilizatori și sistemul de management al clienților. Utilizatorii pot fi reprezentați sub formă de actori, iar cazurile de utilizare (use cases) descriu acțiunile specifice pe care utilizatorii le pot realiza în sistem. Această diagramă oferă o perspectivă de ansamblu asupra funcționalităților cheie ale sistemului și interacțiunilor cu utilizatorii. Precum administratorul are posibilitatea de a configura sistemul, administra utilizatorii, care ulterior deține aptitudinea de a administra accesul utilizatorilor. Deci, ambii actori, precum administratorul și utilizatorul dețin accesul de a efectua autentificare și deconectare din sistemul dat.

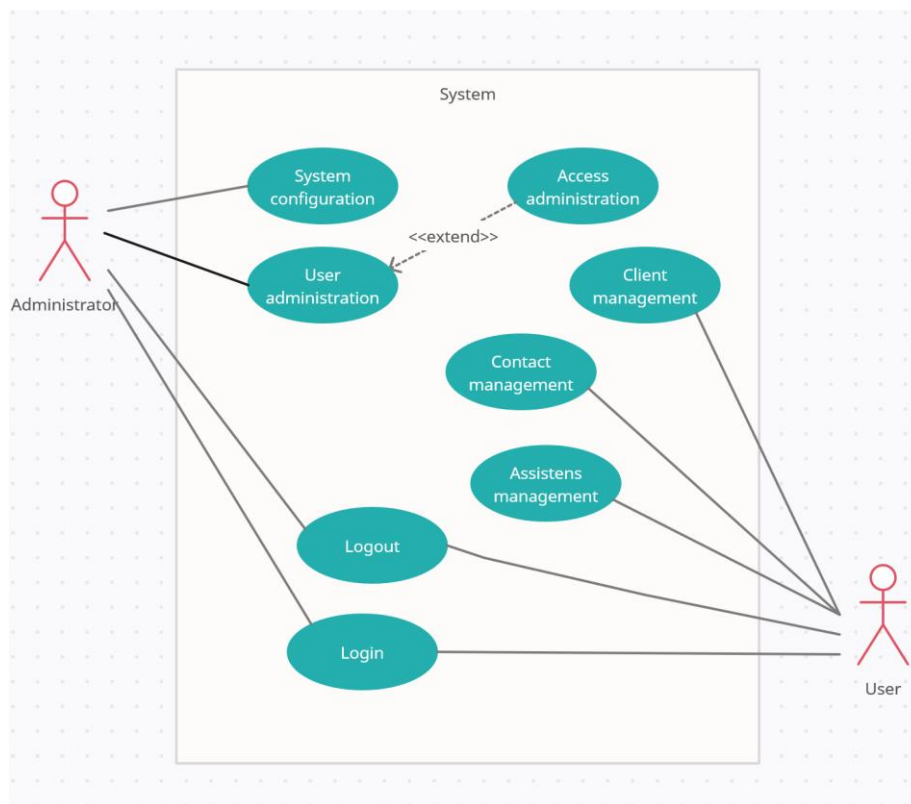


Figura 2.1 – Diagrama de cazul de utilizare

## 2.2. Diagrama de clasă (Class Diagram):

Diagrama de clasă prezintă structura și relațiile dintre diferitele clase și obiecte în cadrul sistemului CRM. Clasele reprezintă entitățile cheie, iar atributele și metodele descriu caracteristicile și comportamentul acestora. Această diagramă ajută la înțelegerea modelului de date și a structurii sistemului CRM. Astfel, în diagrama de clasă reprezentă mai jos, sunt caracterizate următoarele entități cheie, precum clientul, administratorul, raportarea problemelor, tichet sau tag(etichetă), precum și suportul sistemului.

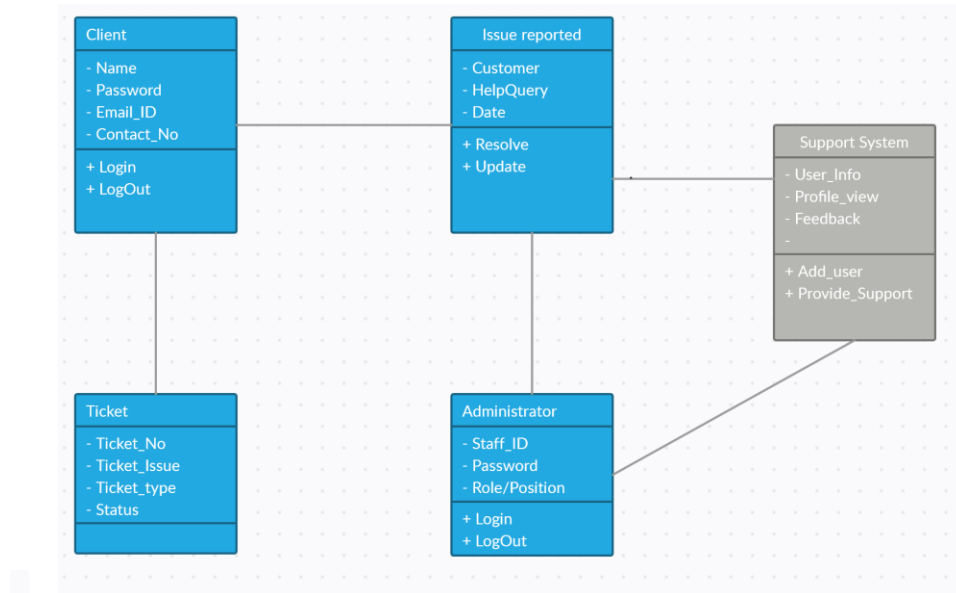


Figura 2.2 – Diagrama de clasă

## 2.3. Diagrama de secvență (Sequence Diagram):

Diagrama de secvență evidențiază interacțiunile dintre diferitele obiecte și entități în cadrul sistemului CRM într-o succesiune temporală. Aceasta descrie fluxul de execuție al operațiunilor și interacțiunile dintre utilizatori, obiecte și servicii externe. Diagrama de secvență oferă o perspectivă detaliată asupra fluxului de lucru și a interacțiunilor în cadrul sistemului CRM.

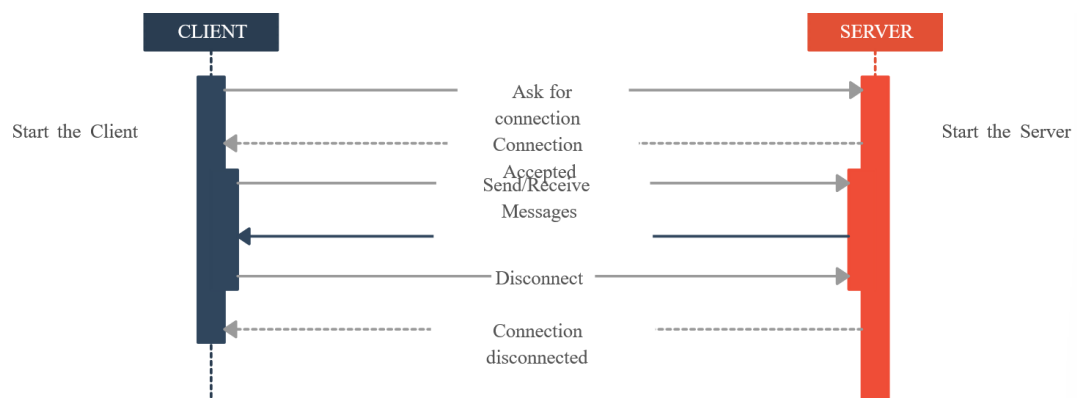


Figura 2.3 – Diagrama de secvență

Astfel, în figura 2.3 de mai sus avem reprezentată diagrama de secvență, în care este abordată interacțiunea dintre client și server, descriind fluxul de execuție a operațiunilor și interacțiunilor, precum sunt, cere conexiune, la care răspunsul de conectare poate fi acceptat sau rejectat, ulterior în urma stabilirii conexiunii se va putea tine o legătură, pentru a transmite date, mesaje. Așadar, în urma manipulării, executăm deconectarea clientului de la server.

#### 2.4. Diagrama de stări (State Diagram):

Această diagramă este utilă în modelarea proceselor și stărilor entităților din sistemul CRM. Ea prezintă tranzițiile între diferitele stări și evenimentele care declanșează aceste tranziții. De exemplu, o entitate CRM poate avea stări precum "Inserarea datelor de autentificare", "Autentificare reușită", "Finalizat", iar dacă datele de autentificare sunt inserate greșit, atunci va fi o nouă stare precum "Date de autentificare greșite", urmată de "Autentificare incorectă", "Ieșire", "Finalizat". Diagrama de stări ajută la înțelegerea comportamentului și fluxului de lucru al entităților din sistemul CRM.

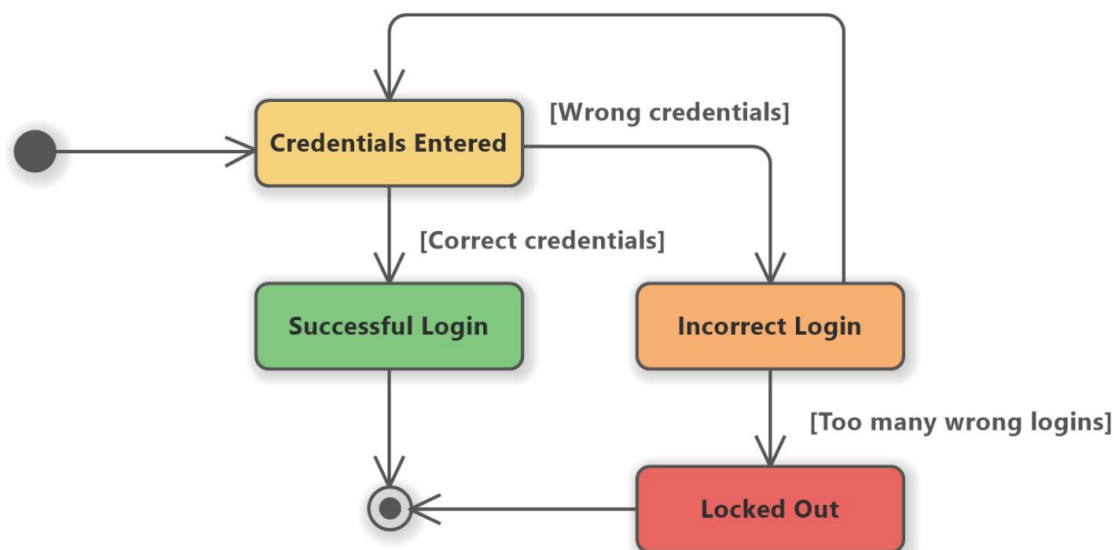


Figura 2.3 – Diagrama de stare

Acestea sunt doar câteva dintre diagramele UML care pot fi utilizate în proiectarea unui sistem CRM. Utilizarea acestor diagrame și notațiile UML oferă o modalitate clară și structurată de a reprezenta diferitele aspecte ale sistemului, relațiile și interacțiunile dintre entități.

### 3. Realizarea sistemului

Sistemul de management al relațiilor cu clienții (CRM) este o componentă esențială pentru orice afacere care dorește să gestioneze eficient relațiile cu clienții săi. În continuare, pentru realizarea unui astfel de sistem este nevoie de abordarea unor tehnologii precum Entity Framework, .NET 6, ASP.NET MVC, AutoMapper, Identity Framework și o bază de date MySQL. Ulterior, se va analiza modul în care aceste tehnologii sunt integrate și exemple de cod din sistemul dat scrise în C#.

#### 3.1. Descrierea limbajului C#

C# este un limbaj de programare modern, orientat pe obiect, dezvoltat de către Microsoft. A fost lansat pentru prima dată în anul 2000 și a devenit unul dintre limbajele de programare populare în dezvoltarea aplicațiilor .NET.

Caracteristici cheie ale limbajului C#:

- Orientat pe obiect: C# este un limbaj de programare orientat pe obiect, ceea ce înseamnă că permite definirea și utilizarea claselor, obiectelor, moștenirii, încapsulării și polimorfismului.
- Tipizare statică: C# este un limbaj cu tipizare statică, ceea ce înseamnă că tipurile de date sunt verificate în timpul compilării, asigurându-se astfel de siguranța tipurilor.
- Platformă .NET: C# este un limbaj de programare care rulează pe platforma .NET, oferind acces la o bibliotecă bogată și puternică de funcționalități, cunoscută sub denumirea de Framework .NET.
- Gestionează automat memorie: C# utilizează sistemul de gestionare a memoriei garbage collector, care se ocupă automat de eliberarea memoriei ocupate de obiectele care nu mai sunt utilizate.
- Sintaxă similară cu C++ și Java: C# a fost creat cu scopul de a combina caracteristicile bune ale limbajelor C++ și Java, astfel încât programatorii cu experiență în aceste limbaje să poată trece ușor la C#.

C# este utilizat în dezvoltarea unei game variate de aplicații, inclusiv aplicații de desktop, aplicații web, servicii web, aplicații mobile și multe altele. Este un limbaj puternic, care oferă suport pentru programare funcțională, LINQ (Language Integrated Query), asincronitate și multe alte funcționalități moderne.

#### 3.2. Descrierea tehnologiei Entity Framework

Entity Framework este un ORM (Object-Relational Mapping) pentru platforma .NET care facilitează interacțiunea cu baza de date utilizând obiecte și abstracții în loc de cod SQL direct. Acesta oferă o modalitate simplă și eficientă de a gestiona operațiile de acces la date, inclusiv crearea, citirea, actualizarea și ștergerea datelor. Prin intermediul Entity Framework, dezvoltatorii pot defini modele de obiecte în codul lor și să interacționeze cu aceste modele, în timp ce Entity Framework se ocupă de generarea și executarea instrucțiunilor SQL corespunzătoare pentru a efectua operațiile dorite în baza de date.

Astfel, având la dispoziție câteva concepte cheie și caracteristici ale Entity Framework, precum

- Modele de obiecte: Entity Framework permite dezvoltatorilor să definească modele de obiecte reprezentând entitățile și relațiile din baza de date. Aceste modele sunt definite utilizând clase și proprietăți în codul C#. De exemplu, o clasă "Client" poate fi definită cu proprietăți precum "Id", "Nume", "Adresa" etc.
- Mape de obiect-relațional: Entity Framework facilitează maparea între modelele de obiecte și tabelele din baza de date. Acesta utilizează convenții sau adnotări pentru a defini cum se realizează maparea între proprietățile claselor și coloanele tabelor.
- Contextul de bază de date: Dezvoltatorii trebuie să creeze un context de bază de date care acționează ca o punte între modelele de obiecte și baza de date. Contextul de bază de date oferă metode și proprietăți pentru a interacționa cu baza de date, cum ar fi DbSet pentru a reprezenta entitățile și metode precum Add, Update și Delete pentru a efectua operații CRUD.
- Migrări: Entity Framework vine cu suport integrat pentru migrări, ceea ce facilitează gestionarea modificărilor structurale ale bazei de date. Dezvoltatorii pot crea și aplica migrări pentru a adăuga, modifica sau șterge tabele și coloane în baza de date, fără a pierde datele existente.
- Interogări LINQ: Entity Framework permite dezvoltatorilor să utilizeze expresii LINQ (Language Integrated Query) pentru a formula interogări complexe către baza de date. Acest lucru oferă o modalitate ușor de înțeles și tipizată de a interoga datele și de a aplica filtre, sortări și alte operații.

Entity Framework aduce numeroase beneficii, cum ar fi reducerea cantității de cod SQL scris manual, eliminarea preocupărilor de mapare obiect-relațională și creșterea productivității dezvoltatorilor. Este important de menționat că Entity Framework este compatibil cu o gamă largă de baze de date, inclusiv SQL Server, MySQL, PostgreSQL și Oracle, ceea ce oferă flexibilitate în a alege baza de date potrivită pentru sistemul dat este MySQL.

Astfel, Entity Framework este un ORM (Object-Relational Mapping) care facilitează interacțiunea cu baza de date utilizând obiecte, iar baza de date care a fost . Sa utilizat Entity Framework pentru a gestiona entitățile legate de sistem, cum ar fi entitatea Client.

Exemplu de cod din sistemul dat pentru definirea entității Client în Entity Framework:

```
public class Client
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Adress { get; set; }
    public string Telephon { get; set; }
    public string Email { get; set; }
}
```

### 3.3. Descrierea platformei .NET 6

Platforma .NET 6 este o versiune a framework-ului de dezvoltare software .NET, dezvoltat de Microsoft. Această versiune aduce o serie de îmbunătățiri semnificative și noi funcționalități pentru dezvoltarea aplicațiilor web, mobile și desktop în mediul .NET.

Astfel, .NET 6 este o platformă de dezvoltare software open-source și multiplatformă, care oferă un mediu robust și scalabil pentru dezvoltarea de aplicații. Aceasta combină limbajele de programare, framework-urile și bibliotecile necesare pentru a construi aplicații moderne și eficiente.

Caracteristici cheie ale platformei .NET 6, presupune performanța îmbunătățiri semnificative a aplicațiilor, datorită optimizărilor aduse în compilator și motorul de execuție. Platforma .NET 6 este compatibilă cu o gamă largă de sisteme de operare și arhitecturi hardware, inclusiv Windows, macOS, Linux. Platforma dată reprezintă o dezvoltare hibridă, ceea ce facilitează dezvoltarea de aplicații hibride care rulează pe mai multe platforme, cum ar fi aplicații desktop, aplicații web și aplicații mobile.

Ulterior, platforma .NET 6 oferă suport îmbunătățit pentru containerizarea aplicațiilor, ceea ce facilitează implementarea și scalabilitatea în medii de producție moderne, cum ar fi Kubernetes. Precum și integrarea cu cloud-ul, deci platforma .NET 6 se integrează bine cu serviciile de cloud, inclusiv Azure, oferind funcționalități avansate pentru dezvoltarea de aplicații cloud-native.

Iar ce ține de securitate, .NET 6 aduce noi caracteristici și instrumente pentru a securiza aplicațiile, cum ar fi suportul îmbunătățit pentru autentificare și autorizare. Precum și dezvoltare web modernă, .NET 6 include ASP.NET Core, care oferă un framework puternic și flexibil pentru dezvoltarea de aplicații web și API-uri.

Dacă să abordăm componente cheie ale .NET 6, ele sunt .NET SDK: SDK-ul .NET 6 (Software Development Kit) furnizează instrumentele necesare pentru dezvoltarea, compilarea și testarea aplicațiilor .NET, ulterior ASP.NET Core fiind un framework pentru dezvoltarea aplicațiilor web și API-uri. Acesta oferă suport pentru rutare, model-view-controller, middleware-uri și multe altele, precum, Entity Framework Core: Este un ORM (Object-Relational Mapping) pentru lucrul cu baze de date relaționale în aplicații .NET. Acesta facilitează interacțiunea cu baza de date prin utilizarea modelelor de obiecte, și Blazor care la rândul său este un framework pentru dezvoltarea aplicațiilor web interactive utilizând C# și HTML. Blazor permite dezvoltatorilor să construiască interfețe de utilizator reactiv și dinamice într-un mod similar cu dezvoltarea aplicațiilor client-side.

Astfel, beneficii și avantaje ale utilizării .NET 6:

- Productivitate crescută: Platforma .NET 6 oferă un set de instrumente și biblioteci puternice care facilitează dezvoltarea rapidă și eficientă a aplicațiilor.
- Compatibilitate înapoi: .NET 6 este proiectat să fie compatibil cu versiunile anterioare ale .NET, ceea ce facilitează migrarea și actualizarea aplicațiilor existente.
- Ecosistem puternic: .NET beneficiază de un ecosistem bogat de biblioteci și framework-uri dezvoltate de comunitatea open-source, care extind funcționalitățile și posibilitățile de dezvoltare.
- Suport și documentație extinsă: Microsoft oferă suport și documentație detaliată pentru platforma .NET, ceea ce facilitează învățarea și rezolvarea problemelor întâmpinate în dezvoltare.

În concluzie, platforma .NET 6 oferă un mediu puternic și flexibil pentru dezvoltarea de aplicații în diferite domenii, de la aplicații web și mobile la aplicații desktop și cloud-native. Aceasta aduce îmbunătățiri semnificative în performanță, productivitate și compatibilitate, facilitând astfel dezvoltarea de aplicații de înaltă calitate în mediul .NET.

.NET 6 este o platformă de dezvoltare care oferă un mediu robust pentru crearea aplicațiilor, să utilizezi .NET 6 pentru a dezvolta componentele sistemului CRM, inclusiv controlerele și serviciile.

Exemplu de cod din sistem pentru un controler Client în ASP.NET MVC utilizând .NET 6:

```
[ApiController]
[Route("api/[controller]")]
public class ClientController : ControllerBase
{
    private readonly IClientService _clientService;

    public ClientController(IClientService clientService)
    {
        _clientService = clientService;
    }

    [HttpGet]
    public IActionResult GetClients()
    {
        var clients = _clientService.GetClients();
        return Ok(clients);
    }

    [HttpPost]
    public IActionResult CreateClient(ClientDto clientDto)
    {
        var createdClient = _clientService.CreateClient(clientDto);
        return CreatedAtAction(nameof(GetClient), new { id = createdClient.Id }, createdClient);
    }
}
```

### 3.4. Descrierea ASP.NET MVC:

Platforma ASP.NET MVC (Model-View-Controller) este un framework de dezvoltare web utilizat pentru construirea și implementarea aplicațiilor web scalabile, robuste și ușor de întreținut. Acesta este parte integrantă a platformei .NET și oferă un set de șabloane și convenții pentru a organiza codul și pentru a facilita dezvoltarea aplicațiilor web.



Ulterior, sunt descrise câteva aspecte importante ale platformei ASP.NET MVC:

- Arhitectura Model-View-Controller (MVC): Platforma ASP.NET MVC se bazează pe modelul MVC, care separă logica de prezentare (View), logica de business (Model) și logica de control (Controller). Această separare clară a responsabilităților facilitează dezvoltarea și testarea separată a fiecărei componente.
- Rutare flexibilă și gestionarea URL-urilor: Platforma ASP.NET MVC oferă un sistem de rutare flexibil care mapează URL-urile la acțiuni și metode în controlere. Aceasta facilitează crearea de URL-uri prietenoase cu motoarele de căutare și personalizarea traseelor aplicației.
- Motorul de vizualizare și șabloanele: ASP.NET MVC oferă un motor de vizualizare puternic, care permite utilizarea diferitelor șabloane de vizualizare, cum ar fi Razor sau ASPX, pentru a genera markup-ul HTML pentru afișarea datelor. Acesta facilitează separarea clară a codului HTML și a logicii de afișare.
- Integrare cu framework-uri și biblioteci populare: Platforma ASP.NET MVC se integrează bine cu alte framework-uri și biblioteci din ecosistemul .NET. De exemplu, se poate integra cu Entity Framework pentru gestionarea accesului la date, AutoMapper pentru maparea obiect-relațională și Identity Framework pentru autentificare și autorizare.
- Testare unitară și TDD: Platforma ASP.NET MVC încurajează dezvoltarea bazată pe testare și dezvoltarea condusă de teste (TDD). Componentele MVC pot fi testate separat, folosind framework-uri precum NUnit sau Microsoft.VisualStudio.TestTools.UnitTesting, pentru a asigura calitatea și stabilitatea aplicației.
- Extensibilitate și personalizare: Platforma ASP.NET MVC este extrem de extensibilă și permite dezvoltatorilor să adauge funcționalități personalizate prin intermediul filtrelor, atributele, a ajutoarelor și a interceptorilor de solicitări. Aceasta oferă o flexibilitate sporită pentru adaptarea aplicației la cerințele specifice.
- Suport pentru REST full API: Platforma ASP.NET MVC oferă suport nativ pentru crearea și implementarea serviciilor web bazate pe arhitectura RESTful. Aceasta facilitează dezvoltarea și expunerea API-urilor web pentru comunicarea cu alte aplicații și dispozitive.

ASP.NET MVC oferă o abordare structurată și flexibilă pentru dezvoltarea aplicațiilor web, cu accent pe separarea clară a responsabilităților și pe testabilitate. Prin intermediul acestei platforme, dezvoltatorii pot construi aplicații web robuste, scalabile și ușor de întreținut, care pot satisface cerințele complexe ale clienților.

ASP.NET MVC este un framework de dezvoltare web care facilitează construirea de aplicații web scalabile și flexibile, din acest motiv a fost utilizată platforma dată, ASP.NET MVC pentru a crea interfața utilizator a sistemului și pentru a gestiona rutele și acțiunile. Ulterior, este prezentat un exemplu de cod pentru o vedere `Index.cshtml` în ASP.NET MVC:

```
@model List<ClientDto>
<h1>List of clients</h1>
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Adress</th>
            <th>Telephon</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var client in Model)
        {
            <tr>
                <td>@client.Name</td>
                <td>@client.Adress</td>
                <td>@client.Telephon</td>
                <td>@client.Email</td>
            </tr>
        }
    </tbody>
</table>
```

### 3.5. Descrierea bazei de date MySQL

MySQL este un sistem de gestionare a bazelor de date relaționale open-source și foarte popular, utilizat într-o varietate largă de aplicații și scenarii de dezvoltare. A fost creat inițial de compania suedeza MySQL AB și este acum dezvoltat și întreținut de Oracle Corporation.

Ulterior, ce ține de caracteristicile și avantajele oferite de MySQL includ:

- Performanță și scalabilitate: MySQL este cunoscut pentru performanța sa ridicată și capacitatea de scalabilitate. Poate gestiona volum mare de date și tranzacții concurente cu eficiență și fără întreruperi. De asemenea, suportă îmbunătățiri precum indexare avansată, cache și optimizare a interogărilor pentru a asigura o execuție rapidă a operațiunilor în baza de date.
- Fiabilitate și stabilitate: MySQL este recunoscut pentru fiabilitatea sa. Acesta oferă funcționalități de recuperare a datelor și mecanisme de backup pentru a asigura că datele sunt protejate și disponibile în caz de probleme sau eșecuri. De-a lungul anilor, MySQL a fost testat și utilizat într-un număr mare de aplicații, ceea ce contribuie la stabilitatea și maturitatea sa.
- Ușurință de utilizare și administrare: MySQL oferă o interfață intuitivă și ușor de utilizat, atât pentru dezvoltatori, cât și pentru administratorii de baze de date. Acesta vine cu un set complet de instrumente

și utilitare pentru gestionarea și monitorizarea bazelor de date, facilitând sarcinile de instalare, configurare, optimizare și întreținere.

- Flexibilitate și compatibilitate: MySQL suportă standardul SQL și oferă o gamă largă de funcționalități, cum ar fi triggeri, funcții stocate și proceduri stocate. De asemenea, suportă diferite tipuri de date și oferă extensibilitate prin intermediul plugin-urilor. MySQL este compatibil cu o varietate de limbaje de programare și framework-uri, inclusiv .NET, PHP, Python și Java, ceea ce facilitează integrarea cu diferite aplicații și tehnologii.
- Comunitate activă și suport extins: MySQL are o comunitate largă și activă de dezvoltatori și utilizatori care oferă suport și resurse utile. Există forumuri de discuții, grupuri de utilizatori și documentație bogată disponibilă pentru a ajuta la rezolvarea problemelor și pentru a găsi răspunsuri la întrebări.

MySQL poate fi utilizat într-o gamă largă de scenarii, de la aplicații web și mobile la soluții de afaceri și enterprise. Este o opțiune populară și fiabilă pentru dezvoltatorii care doresc să implementeze și să gestioneze baze de date relaționale în proiectele lor.

Astfel, pentru a utiliza MySQL într-un proiect, trebuie fie instalat serverul MySQL și să fie configurată o bază de date. Apoi, se poate utiliza limbajul SQL pentru a crea, modifica și interoga tabelele și datele din baza de date. Există, de asemenea, biblioteci și drivere disponibile pentru diferite limbaje de programare, care facilitează interacțiunea cu MySQL în cod. Este important de menționat faptul că descrierea prezentată mai sus este o prezentare generală a caracteristicilor și beneficiilor MySQL.

Ulterior, entitatea care pot fi prezentă într-o bază de date pentru un sistem este tabelul Client. Deci, tabelul "Client" stochează informațiile esențiale despre clienți, cum ar fi nume, adresă, număr de telefon, adresă de e-mail, etc. Fiecare înregistrare din tabel îi aparține un identificator unic, deci unui ID.

```
-- Create table "Clients"
CREATE TABLE Clients (
    Id INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Adress VARCHAR(100),
    Telephone VARCHAR(20),
    Email VARCHAR(100)
);
```

Așadar, aceasta este doar o prezentare a modului în care sa realizat acest sistem utilizând Entity Framework, .NET 6, ASP.NET MVC, AutoMapper, Identity Framework și o bază de date MySQL. Baza de date este un element crucial într-un sistem, deoarece este responsabilă pentru stocarea și gestionarea informațiilor despre clienți, contacte și alte date relevante. Este important să se proiecteze și să se implementeze o structură de bază de date eficientă pentru a asigura o performanță optimă și o gestionare corectă a datelor.

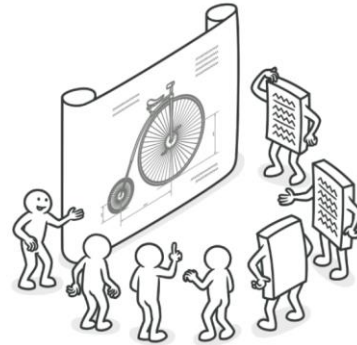
Ulterior, prin utilizarea acestor tehnologii și principii de dezvoltare, am avut posibilitatea de a crea un sistem de management de relații cu clienții, care facilitează gestionarea eficientă a relațiilor cu clienții și oferă o interfață prietenoasă utilizatorilor.

Un alt aspect crucial este utilizarea unei baze de date adecvate pentru a stoca și gestiona informațiile despre clienți și interacțiunile cu aceștia. Am explorat utilizarea bazei de date MySQL și am prezentat un exemplu de cod SQL pentru crearea tabelor în baza de date.

## 5. Design Patterns

# DESIGN PATTERNS

**Design patterns** are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.



Design Patterns sunt soluții tipice la probleme comune în proiectarea software-ului. Fiecare model este ca un plan pe care le putem personaliza pentru a rezolva o anumită problemă de proiectare. Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucată de cod ce poate fi aplicată direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Design pattern-urile nu trebuie privite drept niște rețete care pot fi aplicate direct pentru a rezolva o problemă din design-ul aplicației, pentru că de multe ori pot complica inutil arhitectura. Trebuie întâi înțeles dacă este cazul să fie aplicat un anumit pattern, și de-abia apoi adaptat pentru situația respectivă. Este foarte probabil chiar să folosiți un pattern (sau o abordare foarte similară acestuia) fără să vă dați seama sau să îl numiți explicit. Ce e important de reținut după studierea acestor pattern-uri este un mod de a aborda o problemă de design.

Proiectarea modelelor necesită o combinație de abilități, inclusiv creativitate, atenție la detalii și cunoștințe tehnice. Un designer de model de succes trebuie să fie capabil să vizualizeze produsul finit și să traducă acea viziune într-un model precis și precis.

Un model de design oferă o soluție generală reutilizabilă pentru problemele comune care apar în proiectarea Scopul și utilizarea fiecărui model de design pentru a alege și implementa modelul corect, după cum este necesar.

### **5.1. Tipuri de Design Patterns**

Există în principal trei tipuri de modele de design:

#### *1. Creational(Creativ)*

Aceste modele de design sunt toate despre instanțierea clasei sau crearea de obiecte. Aceste modele pot fi clasificate în continuare în modele de creație de clasă și modele de creație obiect. În timp ce modelele de creare a clasei folosesc moștenirea în mod eficient în procesul de instanțiere, modelele de creare a obiectelor folosesc delegarea în mod eficient pentru a finaliza treaba. Structural

#### *2. Structural*

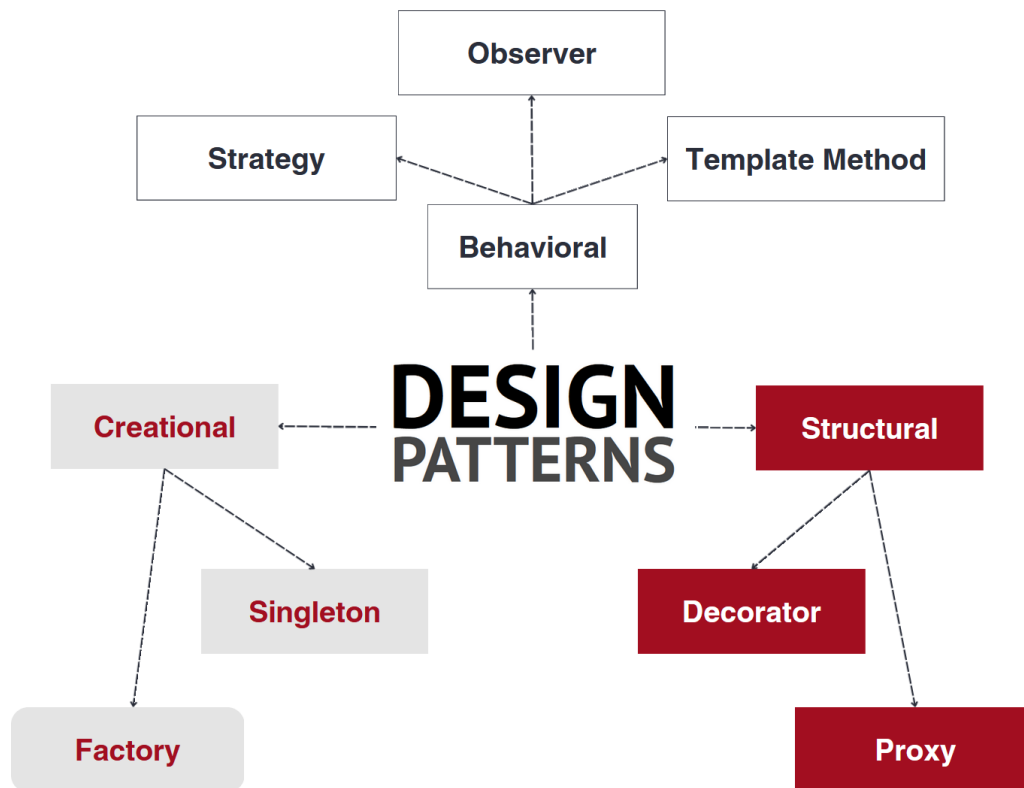
Aceste modele de design sunt despre organizarea diferitelor clase și obiecte pentru a forma structuri mai mari și pentru a oferi noi funcționalități. Modelele de proiectare structurală sunt Adaptor, Bridge, Compozit, Decorator, Fațadă, Flyweight, Private Class Data și Proxy.

Cazul de utilizare al Design Patterns Structural: când 2 interfețe nu sunt compatibile între ele și doresc să stabilească o relație între ele printr-un adaptor, se numește model de proiectare a adaptorului. Modelul adaptorului convertește interfața unei clase într-o altă interfață sau clasă la care se așteaptă clientul, adică adaptorul permite claselor să lucreze împreună, care altfel nu ar putea din cauza incompatibilității. deci, în aceste tipuri de scenarii incompatibile, putem alege modelul adaptorului.

#### *3. Behavioral (Comportamental)*

Tiparele comportamentale sunt despre identificarea tiparelor comune de comunicare între obiecte și realizarea acestor tipare. Tiparele comportamentale sunt Lanț de responsabilitate, Comandă, Interpret, Iterator, Mediator, Memento, Obiect nul, Observator, Stat, Strategie, Metodă șablon, Vizitator

Cazul de utilizare al Design Patterns comportamental - modelul skeleton (șablon) definește scheletul unui algoritm într-o operație care amână unii pași către subclase. Template method permite subclaselor să redefinească anumiți pași ai unui algoritm fără a modifica structura algoritmului. De exemplu, în proiectul dvs., doriți ca comportamentul modulului să se poată extinde, astfel încât să putem face modulul să se comporte în moduri noi și diferite pe măsură ce cerințele aplicației se modifică sau pentru a satisface nevoile aplicațiilor noi. Cu toate acestea, nimeni nu are voie să facă modificări la codul sursă, adică puteți adăuga, dar nu puteți modifica structura în acele scenarii.



## 5.2. Avantajele Design Patterns:

**Precizie:** Avantajul principal al proiectării modelelor este că permite măsurători și calcule precise și precise, rezultând o îmbrăcăminte bine adaptată.

**Eficiență:** Crearea unui model economisește timp și efort în procesul de producție. Odată creat modelul, acesta poate fi folosit pentru a realiza mai multe articole de îmbrăcăminte, reducând timpul și resursele necesare pentru a crea fiecare.

**Creativitate:** proiectarea modelelor permite exprimarea creativă și experimentarea, permițând designerilor să exploreze diferite stiluri, forme și tehnici.

**Consecvență:** Un model asigură consistența în procesul de producție, rezultând o uniformitate a produsului finit, care este esențială pentru producția la scară largă.

**Replicabilitate:** modelele pot fi replicate și utilizate pentru diferite dimensiuni, stiluri și țesături, făcându-le un atu valoros pentru designeri și producători.

Dezavantajele proiectării modelelor:

**Cost:** Proiectarea modelelor poate fi costisitoare, mai ales dacă implică software sau echipamente specializate.

**Abilități:** proiectarea modelelor necesită cunoștințe și abilități specializate, care este posibil să nu fie disponibile pentru toată lumea.

Consumatoare de timp: Crearea unui model poate fi un proces consumator de timp, care necesită atenție la detalii și numeroase ajustări pentru a asigura o potrivire adecvată.

Creativitate limitată: lucrul în limitele unui model poate limita creativitatea designerului, făcând dificilă crearea de modele unice sau inovatoare.

Durabilitate: producția de modele, în special pe hârtie, poate contribui la risipă și daune mediului.

### 5.3. Design Patterns implementare

#### 5.3.1. Singleton

Paternul Singleton asigură că este creată o singură instanță a unei clase și oferă un punct global de acces la acea instanță.

În fragmentul de cod, clasa `DatabaseConnection` reprezintă o conexiune la baza de date.

```
public class DatabaseConnection
{
    private static DatabaseConnection instance;

    private DatabaseConnection() { }

    public static DatabaseConnection Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new DatabaseConnection();
            }
            return instance;
        }
    }

    public void Connect()
    {
        try
        {
            // Code to establish a database connection
            string connectionString = "your_database_connection_string";
            // Establish the connection using the connection string
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                Console.WriteLine("Database connection established.");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error connecting to the database: " + ex.Message);
        }
    }
}
```

Astfel, clasa are un câmp static privat numit `instance` de tip `DatabaseConnection`. Acest câmp conține o singură instanță a clasei `DatabaseConnection`. Constructorul clasei este marcat ca privat, prevenind instanțierea directă a clasei `DatabaseConnection` din afara clasei în sine.



Clasa are o proprietate statică publică numită „Instanță”. Această proprietate oferă acces la o singură instanță a clasei. Utilizează inițializarea leneșă, ceea ce înseamnă că instanța este creată numai atunci când este accesată pentru prima dată. În cadrul getter-ului proprietății `Instanță`, codul verifică dacă câmpul `instanță` este null. Dacă este nulă, indicând că nicio instanță nu a fost creată încă, o nouă instanță a `DatabaseConnection` este creată și alocată câmpului `instanță`.

Metoda `Connect` din clasa `DatabaseConnection` reprezintă funcționalitatea de a stabili o conexiune la baza de date. Această metodă nu este statică și poate fi apelată pe instanța singleton. Accesând proprietatea `Instance`, puteți obține o singură instanță a clasei `DatabaseConnection` și apoi puteți apela metoda `Connect` pentru a stabili o conexiune la baza de date.

Modelul Singleton asigură existența unei singure instanțe a „DatabaseConnection” pe toată durata de viață a aplicației. Acest lucru poate fi benefic atunci când doriți să partajați o singură conexiune la bază de date între mai multe componente sau atunci când trebuie să vă asigurați că este utilizată o singură conexiune într-un mediu cu mai multe fire.

### 5.3.2. Factory Method

Factory Method este un model de creație care oferă o interfață pentru crearea de obiecte, dar permite subclaselor sau claselor derivate să decidă ce clasă să instanțieze.

```
public class ExportFactory
{
    public enum ExportType
    {
        Pdf,
        Excel
    }
    public static ExportBase CreateExport(ExportType type)
    {
        switch (type)
        {
            case ExportType.Pdf:
                return new PdfExport();
            case ExportType.Excel:
                return new ExcelExport();
            default:
                throw new ArgumentException("Invalid export type.");
        }
    }
}
```

În fragmentul de cod, avem următoarele componente: codul prezentat este o clasă numită "ExportFactory", care are rolul de a crea și returna obiecte de tip "ExportBase" în funcție de un anumit tip de export specificat. Clasa "ExportFactory" conține, de asemenea, un enum numit "ExportType", care enumeră diferite tipuri de export disponibile: "Pdf" și "Excel".

Metoda statică "CreateExport" primește un argument de tip "ExportType" și utilizează o structură "switch" pentru a decide ce tip de export trebuie creat și returnat. În cazul în care tipul de export specificat este

"ExportType.Pdf", metoda returnează o instanță a clasei "PdfExport". Dacă tipul de export este "ExportType.Excel", metoda returnează o instanță a clasei "ExcelExport". În cazul în care se specifică un alt tip de export, se aruncă o excepție de tip "ArgumentException" cu mesajul "Invalid export type."

Clasa "ExportFactory" are un rol de fabrică (factory) pentru crearea obiectelor de export, oferind o abstracție în ceea ce privește crearea și gestionarea acestora. Prin intermediul acestei fabrici, utilizatorul poate solicita un anumit tip de export și va primi obiectul corespunzător. Acest design pattern (șablon de proiectare) este util pentru a asigura o creare flexibilă și ușor de extins a obiectelor de export în viitor, adăugând noi tipuri de export fără a afecta utilizatorii existenți ai fabricii.

### 5.3.3. Observer Pattern

**Modelul Observer** este un model comportamental care stabilește o dependență unu-la-mulți între obiecte, în care mai mulți observatori sunt notificați cu privire la orice schimbare de stare a unui subiect sau obiect observabil.

```
public interface IObservable
{
    void Update();
}

public interface IObservable
{
    void Attach(IObservable observer);
    void Detach(IObservable observer);
    void Notify();
}

public class CustomerService : IObservable
{
    private List<IObservable> observers = new List<IObservable>();

    public void Attach(IObservable observer)
    {
        observers.Add(observer);
    }

    public void Detach(IObservable observer)
    {
        observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (var observer in observers)
        {
            observer.Update();
        }
    }
}

public class EmailNotifier : IObservable
{
    public void Update()
    {
        try
        {
            // Code to send an email notification
        }
    }
}
```

```

        string smtpServer = "your smtp_server";
        string senderEmail = "sender@example.com";
        string recipientEmail = "recipient@example.com";
        string subject = "Notification";
        string body = "This is a notification email.";

        using (SmtpClient client = new SmtpClient(smtpServer))
        {
            using (MailMessage message = new MailMessage(senderEmail, recipientEmail, subject,
body))
            {
                client.Send(message);
            }
        }

        Console.WriteLine("Email notification sent.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error sending email notification: " + ex.Message);
    }
}

```

În fragmentul de cod, avem următoarele componente: Interfață `IObserver`: Această interfață definește contractul pentru un observator. Include o singură metodă `Update()`, care este apelată de obiectul observabil atunci când are loc o schimbare de stare.

Interfață `IObservable`: Această interfață definește contractul pentru un obiect observabil. Acesta include trei metode: `Attach(IObserver observator)`, `Detach(IObserver observator)` și `Notify()`. `Attach` este folosit pentru a adăuga un observator la lista de abonați, `Detach` este folosit pentru a elimina un observator din listă, iar `Notify` este folosit pentru a notifica toți observatorii abonați când are loc o schimbare de stare.

Clasa `CustomerService`: Această clasă implementează interfața `IObservable` și reprezintă un serviciu pentru clienți care poate fi observat pentru schimbările de stare. Menține o listă de observatori (câmpul `observatori`) și furnizează metodele `Atașare`, `Detașare` și `Notificare` așa cum sunt definite în interfața `IObservable`. Când metoda `Notify` este apelată, aceasta iterează peste toți observatorii abonați și invocă metoda lor `Update`.

Clasa `EmailNotifier`: Această clasă implementează interfața `IObserver` și reprezintă un observator care trimite notificări prin e-mail. Include metoda `Update`, care reprezintă acțiunea întreprinsă de acest observator atunci când este notificat despre o schimbare de stare. Puteți adăuga codul specific trimiterii notificărilor prin e-mail în cadrul metodei `Actualizare`.

Clasa `SMSNotifier`: Această clasă implementează și interfața `IObserver` și reprezintă un observator care trimite notificări prin SMS. Include metoda `Update`, care reprezintă acțiunea întreprinsă de acest observator atunci când este notificat despre o schimbare de stare. Puteți adăuga codul specific trimiterii notificărilor prin SMS în cadrul metodei `Actualizare`.

Modelul Observator permite o cuplare liberă între observabil și observatori, deoarece observabilul nu trebuie să cunoască detaliile specifice ale observatorilor. Permite o relație flexibilă și dinamică între obiecte,

permițând adăugarea sau eliminarea cu ușurință a observatorilor fără a afecta logica de bază a obiectului observabil.

### 5.3.4. Decorator Pattern

Modelul Decorator vă permite să adăugați în mod dinamic un comportament sau responsabilități suplimentare la un obiect, învelindu-l într-una sau mai multe clase de decorator. Fiecare clasă de decorator extinde sau modifică comportamentul obiectului decorat fără a-i schimba interfața de bază.

```
public class CRMService : ICRMService
{
    public void Process()
    {
        try
        {
            Console.WriteLine("Processing CRM data...");
            Console.WriteLine("CRM data processed successfully.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error processing CRM data: " + ex.Message);
        }
    }
}

public abstract class CRMServiceDecorator : ICRMService
{
    protected ICRMService decoratedService;

    public CRMServiceDecorator(ICRMService decoratedService)
    {
        this.decoratedService = decoratedService;
    }

    public virtual void Process()
    {
        decoratedService.Process();
    }
}

public class LoggingDecorator : CRMServiceDecorator
{
    public LoggingDecorator(ICRMService decoratedService) : base(decoratedService) { }

    public override void Process()
    {
        Console.WriteLine("Logging before processing");
        base.Process();
        Console.WriteLine("Logging after processing");
    }
}

public class AuthorizationDecorator : CRMServiceDecorator
{
    public AuthorizationDecorator(ICRMService decoratedService) : base(decoratedService) { }

    public override void Process()
    {
        Console.WriteLine("Authorizing before processing");
        base.Process();
        Console.WriteLine("Authorizing after processing");
    }
}
```

Codul prezentat definește o serie de clase care implementează un design pattern decorator pentru serviciul CRM. Clasa "CRMService" implementează interfața "ICRMService" și conține metoda "Process". În această metodă, se încearcă procesarea datelor CRM prin afișarea unui mesaj de procesare cu succes în cazul în care nu apare nicio excepție. În caz contrar, se afișează un mesaj de eroare.

Clasa abstractă "CRMServiceDecorator" este o clasă bază pentru decoratori și implementează, de asemenea, interfața "ICRMService". Această clasă are un membru protejat "decoratedService" de tip "ICRMService" care reprezintă serviciul decorat. Constructorul clasei primește serviciul decorat și îl atribuie membrului "decoratedService". Metoda "Process" este implementată virtual și apelează metoda "Process" a serviciului decorat.

Clasa "LoggingDecorator" este un decorator care extinde clasa "CRMServiceDecorator". Constructorul clasei primește serviciul decorat și îl transmite constructorului clasei de bază. Suprascrie metoda "Process" pentru a adăuga funcționalitatea de logare înainte și după apelul metodei "Process" a serviciului decorat. Astfel, se afișează un mesaj de logare înainte de procesare și un mesaj de logare după procesare.

Clasa "AuthorizationDecorator" este un alt decorator care extinde clasa "CRMServiceDecorator". Constructorul clasei primește serviciul decorat și îl transmite constructorului clasei de bază. Suprascrie metoda "Process" pentru a adăuga funcționalitatea de autorizare înainte și după apelul metodei "Process" a serviciului decorat. Astfel, se afișează un mesaj de autorizare înainte de procesare și un mesaj de autorizare după procesare.

Aceste clase permit adăugarea de funcționalități suplimentare înainte și după apelul metodei "Process" a serviciului CRM de bază. Decoratorii pot fi înlanțuiți în diferite combinații pentru a oferi comportamente suplimentare, cum ar fi logarea, autorizarea sau orice altă funcționalitate necesară, fără a modifica direct codul serviciului CRM de bază.

### 5.3.5. Strategy Pattern

Pattern-ul Strategy este un design pattern comportamental care permite definirea și schimbarea dinamică a comportamentului unui obiect în timpul execuției. Acesta separă comportamentul din clasa principală și îl încapsulează în clase separate numite strategii. Astfel, se obține o mai mare flexibilitate și modularitate în implementarea și utilizarea diferitelor strategii.

```
public interface IEmailStrategy
{
    void SendEmail(string email, string message);
}
public class GmailEmailStrategy : IEmailStrategy
{
    public void SendEmail(string email, string message)
    {
        try
```

```

        {
            string smtpServer = "smtp.gmail.com";
            int smtpPort = 587;
            string senderEmail = "your_email@gmail.com";
            string senderPassword = "your_password";
            string recipientEmail = email;
            string subject = "Email Subject";
            string body = message;

            using (SmtpClient client = new SmtpClient(smtpServer, smtpPort))
            {
                client.EnableSsl = true;
                client.UseDefaultCredentials = false;
                client.Credentials = new NetworkCredential(senderEmail, senderPassword);

                using (MailMessage mailMessage = new MailMessage(senderEmail, recipientEmail,
subject, body))
                {
                    client.Send(mailMessage);
                }
            }

            Console.WriteLine("Email sent using Gmail SMTP.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error sending email using Gmail SMTP: " + ex.Message);
        }
    }
}

public class EmailSender
{
    private IEmailStrategy emailStrategy;

    public EmailSender(IEmailStrategy emailStrategy)
    {
        this.emailStrategy = emailStrategy;
    }

    public void SendEmail(string email, string message)
    {
        emailStrategy.SendEmail(email, message);
    }
}

```

În fragmentul de cod, avem următoarele componente: Interfață `ICRMService`: Această interfață definește contractul pentru un serviciu CRM și include o singură metodă `Process()`. Clasa `CRMSERVICE`: Această clasă implementează interfața `ICRMService` și reprezintă implementarea de bază a serviciului CRM. Oferă funcționalitatea de bază pentru procesarea datelor CRM.

Clasa `CRMSERVICEDecorator`: Aceasta este o clasă abstractă care implementează interfața `ICRMService` și servește ca clasă de decorare de bază. Menține o referință la o instanță a `ICRMService` (`decoratedService`) și delegă apelul metodei `Process()` serviciului decorat. Oferă o metodă virtuală `Process()` care poate fi suprascrisă de clase de decorator concret.

Clasa `LoggingDecorator`: Această clasă extinde clasa `CRMSERVICEDecorator` și adaugă funcționalitate de logare la serviciul decorat. Acesta suprascrie metoda `Process()` pentru a adăuga instrucțiuni

de înregistrare înainte și după apelarea metodei `Process()` a serviciului decorat. Apelul `base.Process()` invocă metoda `Process()` a serviciului decorat.

Clasa `AuthorizationDecorator`: Această clasă extinde clasa `CRMServiceDecorator` și adaugă funcționalitate de autorizare serviciului decorat. Acesta suprascrie metoda `Process()` pentru a adăuga verificări de autorizare înainte și după apelarea metodei `Process()` a serviciului decorat. Apelul `base.Process()` invocă metoda `Process()` a serviciului decorat.

Modelul Decorator permite îmbunătățirea flexibilă și modulară a comportamentului obiectului fără a modifica implementarea sau interfața de bază. Puteți stivui mai mulți decoratori unul peste altul, iar ordinea în care sunt aplicați decoratorii poate fi semnificativă. Decoratorii pot fi adăugați sau eliminați dinamic, oferind extensie dinamică și capabilități de personalizare obiectelor în timpul execuției

### **5.3.6. Proxy Pattern**

Pattern-ul Proxy este un design pattern structural care permite crearea unui obiect intermediar (proxy) care controlează accesul la un alt obiect (subiect). Proxy-ul acționează ca o învelitoare pentru subiect, permițându-i să controleze și să gestioneze cererile către subiect înainte de a fi procesate sau să ofere funcționalități suplimentare în jurul obiectului de bază.

Principalele componente ale Pattern-ului Proxy sunt:

- Subiect (Subject): Acesta este obiectul real care oferă funcționalitatea specifică. Proxy-ul acționează ca un înlocuitor al subiectului și trebuie să implementeze aceeași interfață sau să fie înrudit cu aceeași clasă ca subiectul.
- Proxy: Acesta este obiectul intermediar care înconjoară subiectul și controlează accesul la acesta. Proxy-ul implementează aceeași interfață ca și subiectul și redirecționează apelurile către subiect în funcție de necesități. Proxy-ul poate furniza, de asemenea, funcționalități suplimentare, cum ar fi gestionarea accesului, caching-ul rezultatelor sau înregistrarea de statistici.

Utilizarea Pattern-ului Proxy oferă mai multe avantaje, cum ar fi:

- Controlul accesului: Proxy-ul poate aplica reguli de acces și permisiuni pentru a restricționa sau a gestiona accesul la obiectul de bază.
- Lazy loading: Proxy-ul poate amâna încărcarea obiectului de bază până când acesta este solicitat efectiv, permițând economisirea resurselor și o încărcare mai rapidă a aplicației.
- Caching: Proxy-ul poate stoca rezultatele operațiilor anterioare și le poate furniza direct în viitor, evitând astfel repetarea costisitoare a acelorași operații.
- Simplificarea interfeței: Proxy-ul poate ascunde complexitatea subiectului și poate oferi o interfață simplificată și mai ușor de utilizat.

În concluzie, Pattern-ul Proxy oferă o abordare flexibilă pentru controlul și gestionarea accesului la obiecte, permițând adăugarea de funcționalități suplimentare sau restricționarea accesului la obiectul de bază într-un mod transparent pentru client.

```
public class CRMDDataProvider : ICRMDDataProvider
{
    public List<Customer> GetCustomers()
    {
        try
        {
            string connectionString = "your_database_connection_string";
            List<Customer> customers = new List<Customer>();

            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                string query = "SELECT Id, Name, Email, Phone FROM Customers";
                using (SqlCommand command = new SqlCommand(query, connection))
                {
                    using (SqlDataReader reader = command.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            int id = reader.GetInt32(0);
                            string name = reader.GetString(1);
                            string email = reader.GetString(2);
                            string phone = reader.GetString(3);

                            Customer customer = new Customer(id, name, email, phone);
                            customers.Add(customer);
                        }
                    }
                }
            }

            return customers;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error retrieving customers from CRM system: " + ex.Message);
            return new List<Customer>();
        }
    }
}

public class CRMDDataProviderProxy : ICRMDDataProvider
{
    private CRMDDataProvider realDataProvider;
    private List<Customer> cachedCustomers;

    public List<Customer> GetCustomers()
    {
        if (realDataProvider == null)
        {
            realDataProvider = new CRMDDataProvider();
        }

        if (cachedCustomers == null)
        {
            cachedCustomers = realDataProvider.GetCustomers();
        }
    }
}
```



```

        return cachedCustomers;
    }
}

```

În codul prezentat, avem două clase: `CRMDDataProvider` și `CRMDDataProviderProxy`. Aceste clase implementează interfața `ICRMDDataProvider`, care definește metoda `GetCustomers` pentru a obține o listă de clienți dintr-un sistem CRM. Clasa `CRMDDataProvider` reprezintă furnizorul real de date CRM.

În metoda `GetCustomers`, se realizează conexiunea la baza de date specificată prin `connectionString`. Apoi, se construiește un obiect `SqlCommand` cu o interogare SQL pentru a selecta informațiile despre clienți din tabela "Customers". Utilizând un obiect `SqlDataReader`, se iterează prin rezultatele interogării și se construiesc obiecte `Customer` pe baza valorilor citite din baza de date. Aceste obiecte `Customer` sunt adăugate într-o listă, care este returnată la finalul metodei. În cazul în care apare o excepție în timpul accesării datelor, este afișat un mesaj de eroare și este returnată o listă goală.

Clasa `CRMDDataProviderProxy` reprezintă un proxy pentru `CRMDDataProvider`. Aceasta implementează, de asemenea, metoda `GetCustomers`, dar folosește un obiect de tip `CRMDDataProvider` real doar atunci când este necesar. În momentul primei apelări a metodei `GetCustomers`, este creat un obiect `CRMDDataProvider` și se apelează metoda `GetCustomers` a acestuia pentru a obține lista de clienți. Lista de clienți este salvată într-o variabilă `cachedCustomers`. În apelurile ulterioare ale metodei `GetCustomers`, se returnează lista salvată în `cachedCustomers` fără a accesa din nou obiectul `CRMDDataProvider`.

Aceasta este o implementare simplă a pattern-ului Proxy. Prin intermediul proxy-ului, se realizează gestionarea accesului și caching-ul rezultatelor obținute de la furnizorul real de date. Astfel, se evită accesarea repetată a bazei de date în cazul în care datele nu s-au modificat și se îmbunătățește performanța sistemului.

### 5.3.7. Template Method Pattern

Pattern-ul Template Method este un design pattern comportamental care definește scheletul unui algoritm într-o clasă de bază, permițând claselor derivate să implementeze anumite etape ale acestui algoritm în mod specific. Acest pattern promovează reutilizarea codului și separarea responsabilităților între clase.

Prin utilizarea Template Method Pattern, se realizează o separare clară între scheletul algoritmului și detaliile specifice ale implementării. Aceasta permite clasei de bază să ofere o implementare generică și reutilizabilă a algoritmului, în timp ce clasele derivate pot oferi comportamente specifice și personalizate pentru etapele individuale ale algoritmului.

```

public abstract class ExportBase
{
    public void Export(DataGridView dgv, string fileName)
    {
        try
        {

```

```

PrepareData(dgv);
SaveFile(dgv, fileName);
MessageBox.Show(GetSuccessMessage(), "", MessageBoxButtons.OK, MessageBoxIcon.Information);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message, "Error", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);
}
}
protected abstract void PrepareData(DataGridView dgv);
protected abstract void SaveFile(DataGridView dgv, string fileName);
protected abstract string GetSuccessMessage();
}

```

Codul prezentat reprezintă o clasă abstractă numită "ExportBase" care servește drept bază pentru clasele de export specifice. Această clasă conține o metodă publică "Export" care primește două argumente: un obiect "DataGridView" și un șir de caractere "fileName". Metoda "Export" încearcă să exporteze datele din obiectul "DataGridView" într-un fișier specificat prin "fileName".

În interiorul metodei "Export", se folosește un bloc "try-catch" pentru a captura eventualele excepții care pot apărea în timpul procesului de export. În primul rând, se apelează metoda "PrepareData", care este o metodă abstractă protejată definită în clasă și trebuie implementată în clasele derivate. Această metodă are rolul de a pregăti datele din obiectul "DataGridView" într-un format adecvat pentru export.

Următorul pas este apelarea metodei "SaveFile", care este de asemenea o metodă abstractă protejată și trebuie implementată în clasele derivate. Această metodă primește obiectul "DataGridView" și numele fișierului și se ocupă de salvarea efectivă a datelor în fișier.

După salvarea fișierului, se afișează o fereastră de mesaj cu un mesaj de succes. Mesajul de succes este obținut prin apelul metodei "GetSuccessMessage", care este, de asemenea, o metodă abstractă protejată și trebuie implementată în clasele derivate.

În cazul în care apare o excepție în timpul procesului de export, se capturează această excepție în blocul "catch" și se afișează un mesaj de eroare corespunzător utilizatorului prin intermediul unei ferestre de mesaj.

Această clasă abstractă "ExportBase" oferă un schelet general pentru funcționalitatea de export, precizând metodele pe care clasele derivate trebuie să le implementeze. Aceasta permite o extensibilitate și flexibilitate în implementarea diferitelor tipuri de export și permite gestionarea erorilor într-un mod consistent.

## 6. Interfața sistemului

Interfața sistemului CRM este un aspect crucial al experienței utilizatorului și ar trebui să fie intuitivă, ușor de utilizat și să ofere funcționalități relevante pentru nevoile utilizatorilor. Iată o scurtă descriere a componentelor cheie ale interfeței.

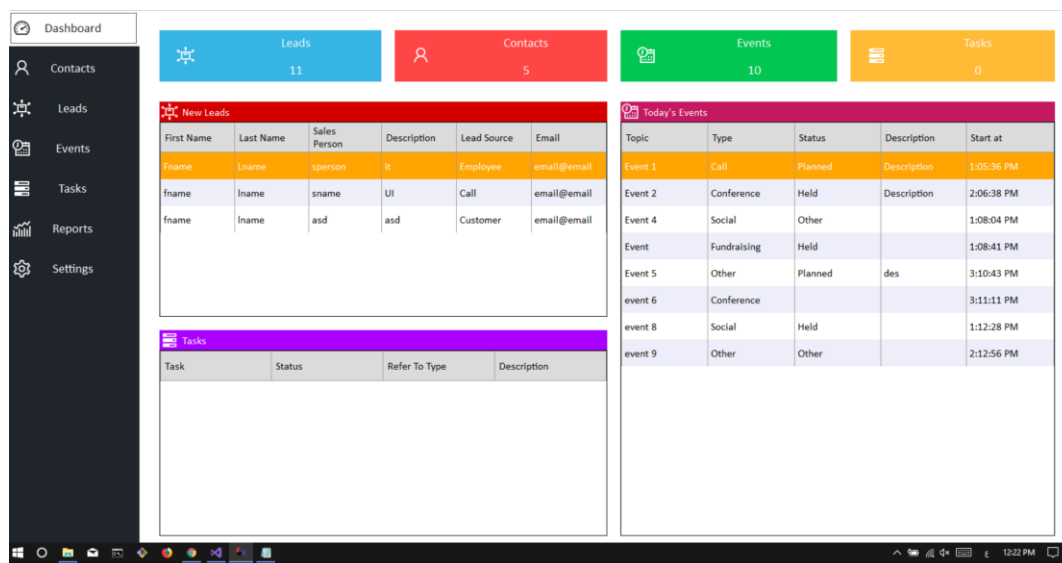


Figura 6.1 – Interfața sistemului panoul de control

Astfel, dacă să abordăm interfața Dashboard sau Panoul de control, care este reprezentată în figura 4.1 de mai sus, reprezintă pagina principală a sistemului și oferă o privire de ansamblu asupra activităților și performanței generale a companiei. Aici utilizatorii pot vizualiza grafice, statistici, rapoarte și alte informații relevante pentru a evalua performanța echipei, interacțiunile cu clienții și altele.

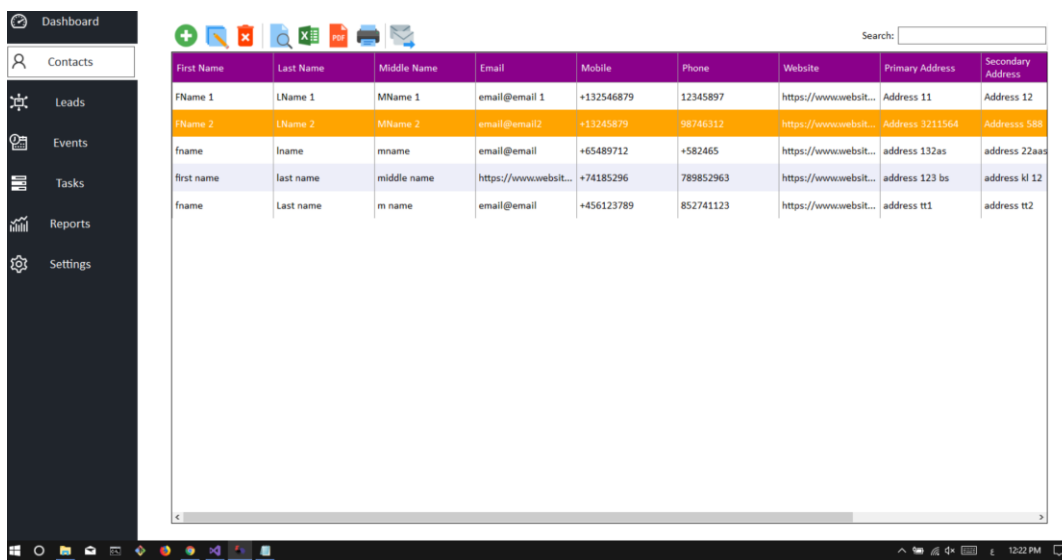
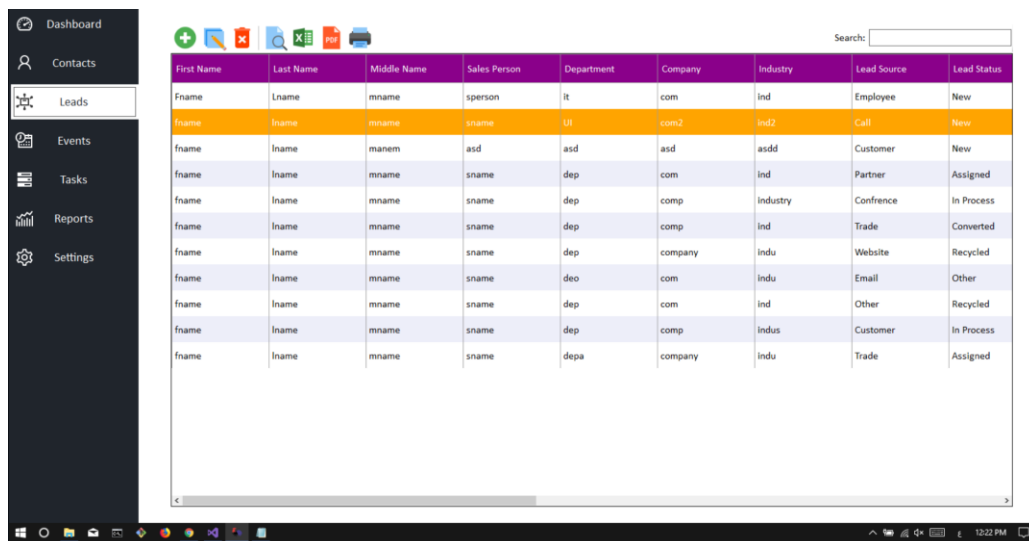


Figura 6.2 – Interfața sistemului panoul Contacts

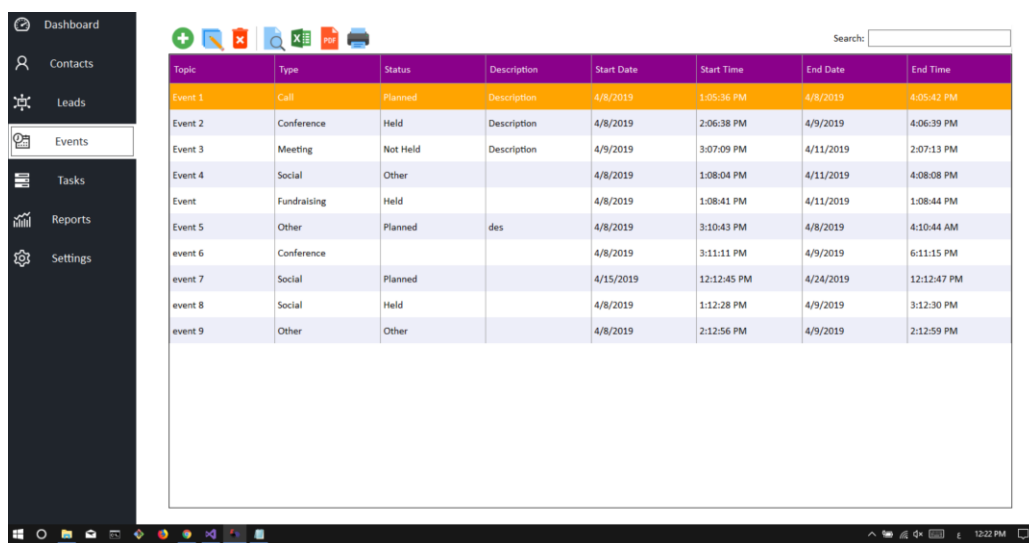
Ulterior, secțiunea panoului Contacte, reprezentat în figura 4.2 de mai sus, permite utilizatorilor să gestioneze informațiile referitoare la clienți și contactele asociate. Aceasta poate include detalii precum nume, adrese, numere de telefon, adrese de e-mail și alte informații de contact relevante. Utilizatorii pot adăuga, edita și șterge contacte, precum și să le atribuie anumite categorii sau etichete pentru o organizare mai eficientă.



First Name	Last Name	Middle Name	Sales Person	Department	Company	Industry	Lead Source	Lead Status
fname	lname	mname	sperson	it	com	ind	Employee	New
fname	lname	mname	sname	UI	com2	ind2	Call	New
fname	lname	manem	asd	asd	asd	asdd	Customer	New
fname	lname	mname	sname	dep	com	ind	Partner	Assigned
fname	lname	mname	sname	dep	comp	Industry	Conference	In Process
fname	lname	mname	sname	dep	comp	ind	Trade	Converted
fname	lname	mname	sname	dep	company	Indu	Website	Recycled
fname	lname	mname	sname	deo	com	Indu	Email	Other
fname	lname	mname	sname	dep	com	ind	Other	Recycled
fname	lname	mname	sname	dep	comp	Indus	Customer	In Process
fname	lname	mname	sname	depa	company	Indu	Trade	Assigned

Figura 6.3– Interfața sistemului panelul Leads

Așadar, secțiunea Potențiali clienți (Leads) reprezentată în figura 4.3 de mai sus, este dedicată gestionării potențialilor clienți care încă nu au devenit clienți efectivi. Utilizatorii pot adăuga informații despre potențiali clienți, precum nume, detalii de contact, interesul lor în produsele sau serviciile oferite și alte informații relevante. Aceasta permite echipei să urmărească și să gestioneze în mod eficient potențialele oportunități de afaceri.



Topic	Type	Status	Description	Start Date	Start Time	End Date	End Time
Event 1	Call	Planned	Description	4/8/2019	1:05:36 PM	4/8/2019	4:05:42 PM
Event 2	Conference	Held	Description	4/8/2019	2:06:38 PM	4/9/2019	4:06:39 PM
Event 3	Meeting	Not Held	Description	4/9/2019	3:07:09 PM	4/11/2019	2:07:13 PM
Event 4	Social	Other		4/8/2019	1:08:04 PM	4/11/2019	4:08:08 PM
Event	Fundraising	Held		4/8/2019	1:08:41 PM	4/11/2019	1:08:44 PM
Event 5	Other	Planned	des	4/8/2019	3:10:43 PM	4/8/2019	4:10:44 AM
event 6	Conference			4/8/2019	3:11:11 PM	4/9/2019	6:11:15 PM
event 7	Social	Planned		4/15/2019	12:12:45 PM	4/24/2019	12:12:47 PM
event 8	Social	Held		4/8/2019	1:12:28 PM	4/9/2019	3:12:30 PM
event 9	Other	Other		4/8/2019	2:12:56 PM	4/9/2019	2:12:59 PM

Figura 6.4 – Interfața sistemului panelul Events

Astfel, secțiunea Evenimente(Events) reprezentată în figura 4.4 de mai sus, este folosită pentru planificarea și urmărirea evenimentelor, întâlnirilor și activităților asociate clienților și oportunităților. Utilizatorii pot crea și gestiona evenimente, să le atribuie clienți sau specifice și să adauge detalii despre locație, orar și alte informații relevante. Aceasta permite o gestionare mai eficientă a întâlnirilor și interacțiunilor cu clienții.

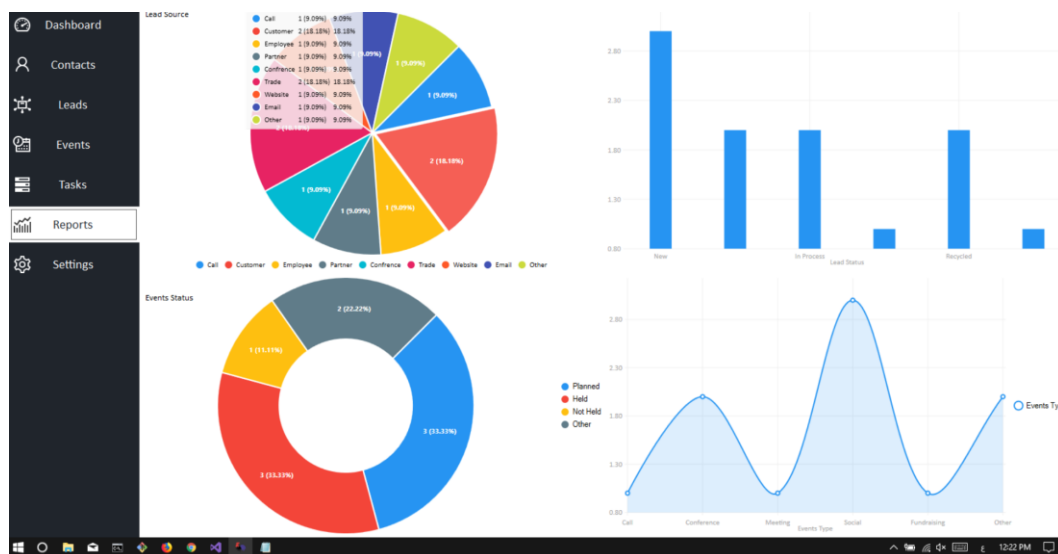


Figura 6.5 – Interfața sistemului panelul Reports

Ulterior, despre secțiunea Rapoarte reprezentată în figura 4.5 de mai sus, oferă utilizatorilor posibilitatea de a genera rapoarte personalizate și analize asupra datelor din sistem. Utilizatorii pot selecta criterii de filtrare, cum ar fi perioada, segmentul de clienți, și pot vizualiza grafice, tabele și alte tipuri de vizualizări a datelor. Acest lucru facilitează monitorizarea și evaluarea performanței, identificarea tendințelor și luarea deciziilor bazate pe date concrete.

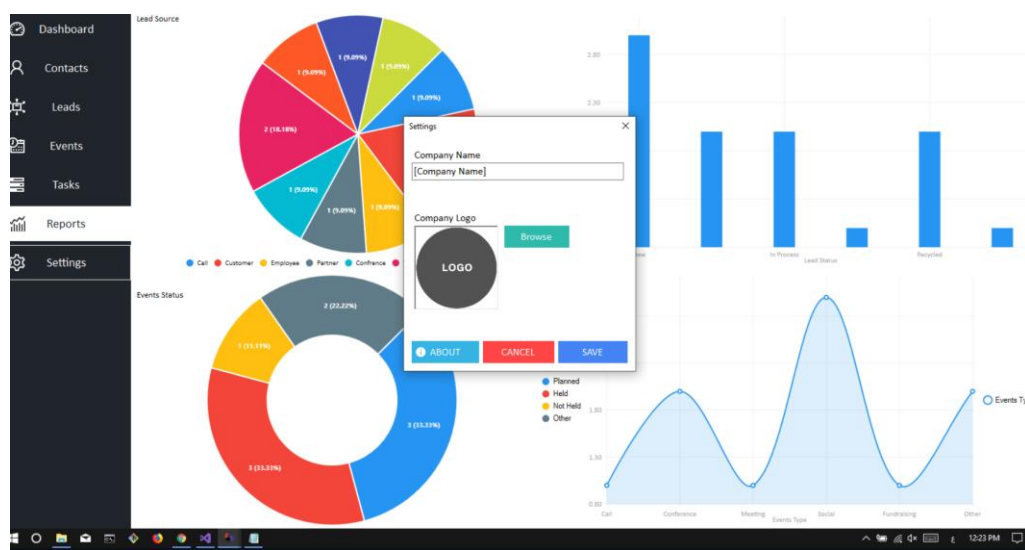


Figura 6.6 – Interfața sistemului panelul Settings

Iar, secțiunea Setări, reprezentată în figura 4.6 de mai sus, permite utilizatorilor să personalizeze și să configureze sistemul conform nevoilor. Aceasta poate include opțiuni precum configurarea profilului utilizatorului, personalizarea câmpurilor și etichetelor, setarea permisiunilor pentru utilizatori, integrarea cu alte aplicații și multe altele. Setările permit adaptarea sistemului pentru a se potrivi proceselor și fluxurilor de lucru specifice ale companiei.

Astfel, este important ca interfața sistemului proiectat să concludă nevoile și preferințele utilizatorilor, astfel încât să ofere o experiență eficientă și plăcută. Elementele de design, organizarea informațiilor și funcționalitățile sunt intuitive și permit utilizatorilor să navigheze ușor și să efectueze sarcinile specifice fără dificultate.

## Concluzie

În urma implementării a sistemului de management al relațiilor cu clienții, care este necesar să fie eficient, implică utilizarea unui set de tehnologii moderne și potente care să faciliteze procesul de gestionare a datelor și a interacțiunilor cu clienții. Am explorat, de asemenea, tehnologii și instrumente cheie utilizate în dezvoltarea unui sistem, cum ar fi limbajul C#, Entity Framework, .NET 6, ASP.NET MVC, AutoMapper și Identity Framework. Aceste tehnologii oferă o fundație solidă pentru crearea și implementarea unui sistem.

Prin utilizarea acestor tehnologii, există posibilitatea de a construi un sistem puternic, flexibil și ușor de întreținut, care să îndeplinească cerințele specifice și să îmbunătățească relația cu clienții. Este important de menționat că tehnologiile menționate sunt doar câteva exemple dintre cele disponibile, iar alegerea lor depinde de cerințele specifice ale proiectului și de cunoștințele și experiența echipei de dezvoltare.

În concluzie Design Patterns reprezintă soluții și abordări comune la problemele de proiectare software recurente. Acestea furnizează un set de principii și direcții care pot fi aplicate în proiectarea și implementarea sistemelor software pentru a obține un cod mai modular, mai flexibil, mai ușor de înțeles și de întreținut.

Astfel, proiectul CRM folosește mai multe Design Patterns pentru a oferi o arhitectură modulară și extensibilă. Iată o concluzie sumară despre fiecare pattern utilizat, deci, Singleton: A fost folosit pentru a asigura că există o singură instanță a clasei CRMService disponibilă în întregul sistem. Factory Method: Clasa ExportFactory utilizează acest pattern pentru a crea și returna obiecte de export în funcție de tipul specificat. Proxy: Nu există implementare a pattern-ului Proxy în codul prezentat. Decorator: Sunt utilizate clasele CRMServiceDecorator, LoggingDecorator și AuthorizationDecorator pentru a adăuga funcționalități suplimentare înainte și după apelul metodei Process a serviciului CRM. Observer: Nu există implementare a pattern-ului Observer în codul prezentat. Strategy: Nu există implementare a pattern-ului Strategy în codul prezentat. Template Method: Clasa ExportBase folosește acest pattern pentru a defini un schelet de algoritmi în metoda Export, lăsând implementarea detaliilor specifice claselor derivate.

Utilizarea acestor pattern-uri în proiectul CRM oferă o structură modulară și flexibilă, permițând adăugarea și extinderea funcționalităților într-un mod ușor de gestionat și de adaptat la cerințe viitoare. Aceste pattern-uri contribuie la crearea unui cod mai curat, mai flexibil și mai ușor de întreținut. În procesul de proiectare și dezvoltare a sistemului, am evidențiat importanța modelării și proiectării, inclusiv utilizarea diagramelor UML pentru a înțelege cerințele și fluxurile de lucru ale sistemului. Am prezentat exemple de diagrame, cum ar fi diagrama de cazuri de utilizare, diagrama de clase și diagrama de secvență, care sunt instrumente valoroase în proiectarea unui sistem eficient. În final, am subliniat importanța unei interfețe de utilizator prietenoase și intuitive într-un sistem, care să faciliteze navigarea și utilizarea eficientă a funcționalităților oferite. Am descris componente cheie ale interfeței, cum ar fi panoul de control, secțiunile de contacte, potențiali clienți, evenimente, rapoarte și setări.

Sistemul de Management al Relațiilor cu Clienții este o soluție indispensabilă pentru companii în efortul lor de a dezvolta și menține relații puternice cu clienții. Implementarea unui sistem bine conceput și personalizat poate contribui semnificativ la creșterea afacerii și îmbunătățirea satisfacției clienților. Prin utilizarea tehnologiilor moderne, modelelor de proiectare adecvate și unei baze de date eficiente, un sistem poate deveni un instrument valoros pentru orice organizație în gestionarea și dezvoltarea relațiilor cu clienții.

## Bibliografie

1. "CRM at the Speed of Light: Essential Customer Strategies for the 21st Century" de Paul Greenberg
2. "Customer Relationship Management: Concepts and Technologies" de Francis Buttle
3. "The CRM Handbook: A Business Guide to Customer Relationship Management" de Jill Dyché
4. "Mastering the Requirements Process: Getting Requirements Right" de Suzanne Robertson și James Robertson
5. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development: Build applications with C#, .NET Core, Entity Framework Core, ASP.NET Core, and ML.NET using Visual Studio Code, 4th Edition 4th Edition by Mark J.Price
6. "Domain-Driven Design: Tackling Complexity in the Heart of Software" de Eric Evans
7. "Pro ASP.NET MVC 5" de Adam Freeman
8. "Entity Framework Core in Action" de Jon P. Smith
9. "ASP.NET Core in Action" de Andrew Lock și Filip Wojcieszyn
10. "Clean Code: A Handbook of Agile Software Craftsmanship" de Robert C. Martin
11. "The Pragmatic Programmer: Your Journey to Mastery" de Andrew Hunt și David Thomas
12. [https://sourcemaking.com/design\\_patterns](https://sourcemaking.com/design_patterns)
13. <https://refactoring.guru/design-patterns>