UNIVERSITETI I MITROVICËS "ISA BOLETINI"
FAKULTETI I INXHINIERISË MEKANIKE DHE KOMPJUTERIKE
DEPARTAMENTI: INFORMATIKË INXHINIERIKE

PUNIM DIPLOME

Mentori
MSc. Halil Sadiku, PhD Cand.

Kandidati
Arbnor Jusufi

Mitrovicë 2021

UNIVERSITETI I MITROVICËS "ISA BOLETINI"
FAKULTETI I INXHINIERISË MEKANIKE DHE KOMPJUTERIKE
DEPARTAMENTI: INFORMATIKË INXHINIERIKE

PUNIM DIPLOME

VLERËSIMI I MODELEVE TË ZHVILLIMIT TË APLIKACIONEVE NË GJUHËN PROGRAMUESE C#

Mentori
MSc. Halil Sadiku, PhD Cand.

Kandidati
Arbnor Jusufi

Mitrovicë 2021

DEKLARATË E ORIGJINALITETIT

Me anë të kësaj deklarate dëshmojë që ky punim është punë origjinale dhe nuk është botuar më herët në ndonjë universitet apo institucion tjetër akademik. Po ashtu, deklaroj që punimi nuk përmban material të botuar apo shkruar nga ndonjë person tjetër, përveç siç deklarohet në përmbajtjen e këtij teksti.

Po ashtu deklarojë që kamë respektuar në përpikëri rregullat themelore akademike të Universitetit të Mitrovicës “Isa Boletini” për punimin e kësaj teme të diplomës.

_____

Nënshkrimi

ABSTRAKT

Modelet e zhvillimit janë një koncept i përgjithshëm për zgjidhjen e një problemi, të cilat shpesh ngatërrohen me algoritmet sepse që të dyja janë udhëzime për zgjidhjen e problemeve të shpeshta gjatë programimit por që kanë një dallim të qartë mes vete, algoritmet përcaktojnë një listë veprimesh që mund te arrijnë deri të zgjidhja e problemit kurse modelet janë një përshkrim i nivelit më të lartë të një zgjidhjeje, ku mund të shihen rezultatet por metoda e zbatimit varet nga personi që përzgjidh modelin. Të gjitha modelet mund të karakterizohen nga qëllimi i tyre, ato ndahen në tri grupe kryesore:

- Modelet Krijuese (Creational Patterns)
- Modelet Strukturore (Structural Patterns)
- Modelet e Sjelljes (Behavioral Patterns)

Për të i sqaruar më mirë modelet e zhvillimit është krijuar edhe një aplikacion i cili ka për qëllim menaxhimin e punëtoreve të një kompanie dhe përdorë dy modelet më të përdorura. Aplikacioni është i shkruar në gjuhën C#, gjuhë programuese e orientuar në objekte, për ta mundësuar aplikacionin të punojë në Windows është përdorur edhe .Net Framework, Në përgjithësi ky punim ka për synim kuptimin dhe zbatimin e modeleve.

Fjalët kyçe: C#, .Net Framework, Visual Studio, SSMS, Modelet, SOLID, UML.

FALENDERIM

Së pari falënderoj Zotin e madhërishëm për diturin dhe shëndetin që më jepi gjatë këtij udhëtimi tre vjeçare. Një falënderim nga zemra shkon për familjen time që më mbështeten, përkrahen dhe më motivuan vazhdimisht për çdo hap të udhëtimit tim akademik, gjithashtu falënderoj kolegët dhe shokët për ndihmën dhe bashkëpunimin e tyre. Gjithashtu falënderoj edhe mentorin tim të temës së diplomës MSc. Halil Sadiku (PhD Cand) për përkrahjen, motivimin, pozitivitetin dhe mbështetjen e vazhdueshme gjatë punës së temës së diplomës.

PËRMBAJTJA

1. HYRJE 5
2. TEKNOLOGJITË E PËRDORURA 6
2.1 C# 6
2.2 .NET Framework 7
2.3 Visual Studio 8
2.4 SQL Server Management Studio (SSMS) 9
2.4.1 Object Explorer 10
3. MODELET E ZHVILLIMIT NË GJUHËN PROGRAMUESE C# 11
3.1. Klasifikimi i modeleve 11
3.2. Historiku i modeleve të zhvillimit 12
3.3 SOLID Parimet 12
3.4 UML 13
3.5 Creational Patterns 15
3.5.1 Factory Method 16
3.6 Structural Patterns 19
3.6.1 Adapter Pattern 20
3.7 Behavioral Patterns 24
3.7.1 Observer Pattern 24
4. RAST STUDIMI – ZHVILLIMI I APLIKACIONIT 31
4.1 Implementimi i Factory Method 33
4.2 Implementimi i Observer Pattern 38
5. PËRFUNDIMI 42
6. LITERATURA 43
TABELA E FIGURAVE 45

1. HYRJE

Katër autorët Erich Gamma, Richard Helm, Ralph Johnson dhe John Vlissides botuan një libër të quajtur *DESIGN PATTERNS – Elements of Reusable Object-Oriented Software* [1] në vitin 1994, duke iniciuar konceptet themelore të modeleve të zhvillimit në dizajnimin e softuerit, libri i tyre korri aq sukses të madh saqë atyre u dha titulli Gang of Four (GOF).

Modelet e zhvillimit janë zgjidhje tipike për problemet e njëjta që ndodhin vazhdimisht gjatë dizajnit të një softueri. Një model i zhvillimit nuk është një zgjidhje që mund të përdoret menjëherë nëse haset në ndonjë problem por është një përshkrim i detajuar se si mund te zgjidhet një problem, pastaj mund të implementohet në kodin e programit. modelet e zhvillimit ndryshojnë nga shkalla e zbatimit dhe kompleksiteti i tyre. Modelet e kompleksitetit më të ulët i quajmë *idioma* këto modele aplikohen vetëm në një gjuhë programuese, ndërsa modelet me kompleksitet të lartë quajmë modele *arkitektonike* të cilat mund të përdoren në të gjitha gjuhët programuese. Përdorimi i modeleve të zhvillimit në softuerin tonë jo vetëm që i eliminon problemet e vazhdueshme por edhe e bënë më të mirëmbajtur [2].

Pra ky punim diplome trajton modelet e zhvillimit si krijimin, klasifikimin dhe zbatimin e tyre në programim. Me anë të një projekti kemi ilustruar përdorimin e modeleve të zhvillimit në projekte reale.

2. TEKNOLOGJITË E PËRDORURA

Modelet e zhvillimit kanë një zbatim të gjerë në të gjitha gjuhët programuese. Për punimin e projektit me anë të cilit do të ilustrojmë përdorimin e modeleve të zhvillimit të cilin do të shtjellohet më vonë gjatë këtij punimi, janë përdorur teknologjitë të cilat janë duke u mësuar aktualisht në Universitetin e Mitrovicës, ndër të cilat janë:

- C#
- .NET Framework
- Visual Studio
- SQL Server Management Studio

2.1 C#

C# është një gjuhë programimi moderne, e orientuar në Objekte dhe e tipit të sigurt. C# i ka rrënjët në familjen C të gjuhëve dhe do të jetë menjëherë e njohur për programuesit C, C ++, Java dhe JavaScript [3]. C# u zhvillua brenda Microsoft. Shpikësit kryesorë të kësaj gjuhe ishin Anders Hejlsberg, Scott Wiltamuth dhe Peter Golde. Zbatimi i parë i shpërndarë gjerësisht i C# u lëshua nga Microsoft në korrik të vitit 2000, si pjesë e .NET Framework [4]. Figura 1 tregon se si printohet fjala “Hello World” në gjuhën programuese C#.

```
1   using System;
2
    0 references
3   internal class Hello
4   {
        0 references
5       private static void Main(string[] args)
6       {
7           Console.WriteLine("Hello World");
8           Console.ReadKey();
9       }
10  }
```

Figura 1 "Hello, World” në C#

Versioni aktual i C# është 9.0, i cili shton shumë veçori dhe përmirësime, ndër to janë [5]:

- Records - janë një lloj reference që ofron metoda të sintetizuara për të siguruar semantikë vlere për barazinë. Regjistrimet janë të pandryshueshme nga parazgjedhja.

- Init only setters - sigurojnë sintaksë të qëndrueshme për të iniciuar anëtarët e një objekti.
- Top-level statements - largon ceremoninë e panevojshme nga shumë aplikacione.

2.2 .NET Framework

.NET është një platformë falas, ndër-platformë (cross-platform), zhvilluese me burim të hapur për ndërtimin e shumë llojeve të ndryshme të aplikacioneve. me .NET, mund të përdorni shumë gjuhë, redaktorë dhe biblioteka për të ndërtuar për ueb, celular, desktop, lojëra dhe IoT.
Versioni i fundit i .NET është .NET 5 Microsoft e kalojë një gjeneratë që të mos ngatërrohet më versionin e .NET Framework. Zhvilluesit që shkruajnë aplikacione .NET 5 do të kenë qasje në versionin dhe veçoritë më të fundit të C#. .NET 5 bashkë me C# 9 sjellin shumë tipare të reja në gjuhë tiparet të cilat janë diskutuar në faqe 5 kurë është folur rreth C# [6].

.NET Framework është një platformë zhvillimi që përfshin një Common Language Runtime (CLR), e cila menaxhon ekzekutimin e kodit dhe një Base Class Library (BCL), e cila siguron një bibliotekë të pasur të klasave për të ndërtuar aplikacione. Microsoft u përpoq që i gjithë implementimi të funksionojë më së miri në Windows. Që kur .NET Framework 4.5.2, ai ka qenë një komponent zyrtar i sistemit operativ Windows. .NET Framework është instaluar në mbi një miliard kompjuterë kështu që duhet të ndryshojë sa më pak të jetë e mundur për arsye se rregullimet e defekteve në kod mund të shkaktojnë probleme, prandaj përditësohet rrallë [7].

Të gjitha aplikacionet në një kompjuter të shkruar për .NET Framework ndajnë të njëjtin version të CLR dhe bibliotekave të ruajtura në Global Assembly Cache (GAC), gjë që mund të çojë në probleme nëse disa prej tyre kanë nevojë për një version specifik për pajtueshmëri [7].

2.3 Visual Studio

Visual Studio është një vegël krijues që mund të përdoret për të modifikuar, korrigjuar gabimet dhe për të ndërtuar kodin dhe më pas për të publikuar një aplikacion. Visual Studio është një IDE[1] program i pasur me karakteristika që mund të përdoret për shumë aspekte të zhvillimit të softuerit. Sipër dhe poshtë redaktuesin dhe korrigjuesin standard që sigurojnë shumica e IDE-ve. Visual Studio përfshin përpiluesit, mjetet e kompletimit të kodit, krijuesit grafikë dhe shumë veçori të tjera për të lehtësuar procesin e zhvillimit të softuerit [8].

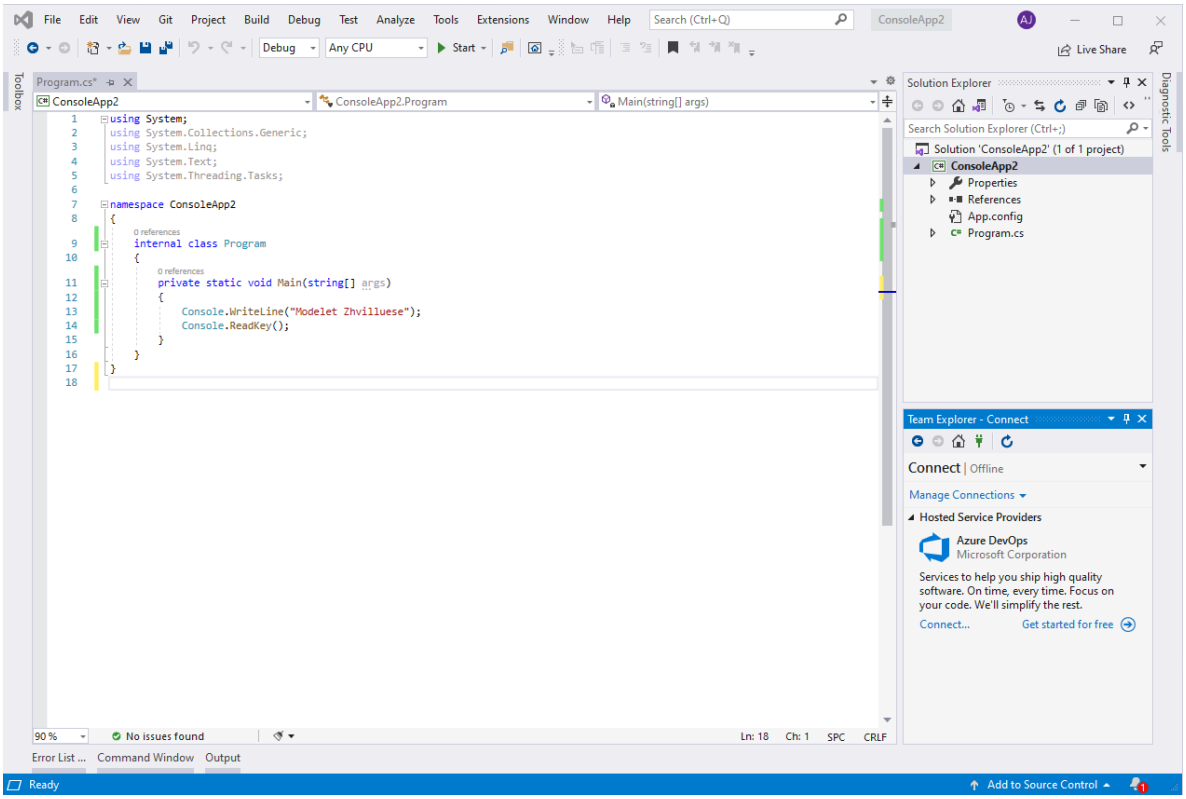

Figura 2 Visual Studio 2019

[1] integrated development environment (IDE), u mundëson programuesve të konsolidojnë aspektet e ndryshme të shkrimit të një programi kompjuterik:
https://www.codecademy.com/articles/what-is-an-ide

Figura 2 tregon pamjen e Visual Sudio 2019 në një projekt të hapur me disa mjete kyçe të cilat do të përdoren [8]:

- Solution Explorer (lart djathtas) ju lejon të shikoni, të lundroni dhe të menaxhoni file tuaj të kodit. Solution Explorer mund të ndihmojë në organizimin e kodit tuaj duke grupuar file dhe projekte.
- Dritarja Edit Window (në qendër), ku do ta kaloni shumicën e kohës, shfaq përmbajtjen e skedarit (file). Kjo është ajo ku ju mund të modifikoni kodin ose të krijoni një interface përdoruesi (UI) siç është një dritare me butona dhe kuti teksti.
- Team Explorer (poshtë djathtas) ju lejon të gjurmoni artikujt e punës dhe të ndani kodin me të tjerët duke përdorur teknologjitë e kontrollit të versionit siç janë Git i cili gjurmon ndryshimet që bën në fajëll, kështu që të kesh një regjistër të asaj që është bërë dhe mund të kthehesh në versione specifike nëse do të të duhej ndonjëherë dhe Team Foundation Version Control (TFVC) i cili ju ndihmon të gjurmoni ndryshimet që bëni në kodin tuaj me kalimin e kohës.

2.4 SQL Server Management Studio (SSMS)

SQL Server Management Studio (SSMS) është një mjedis i integruar për menaxhimin e çdo infrastrukture SQL, nga SQL Server te Azure SQL Database [9] i cili është një shërbim i plotë i të dhënave, që do të thotë që Microsoft operon SQL Server për ju dhe siguron disponueshmërinë dhe performancën e tij [10]. SSMS ofron mjete për konfigurimin, monitorimin dhe administrimin e instancave të SQL Server dhe bazave të të dhënave.
SSMS përdorët për të kërkuar, dizajnuar dhe menaxhuar bazat e të dhënave tuaja dhe depot e të dhënave, kudo që ndodhen - në kompjuterin tuaj lokal ose në Cloud [9] interneti më konkretisht, janë të gjitha gjërat në të cilat mund të qasemi në distancë përmes internetit. Kur diçka është në Cloud, kjo do të thotë se është ruajtur në serverat e Internetit në vend të diskut të kompjuterit tuaj. Një karakteristikë e veçantë e SSMS është edhe **Object Explorer** [11].

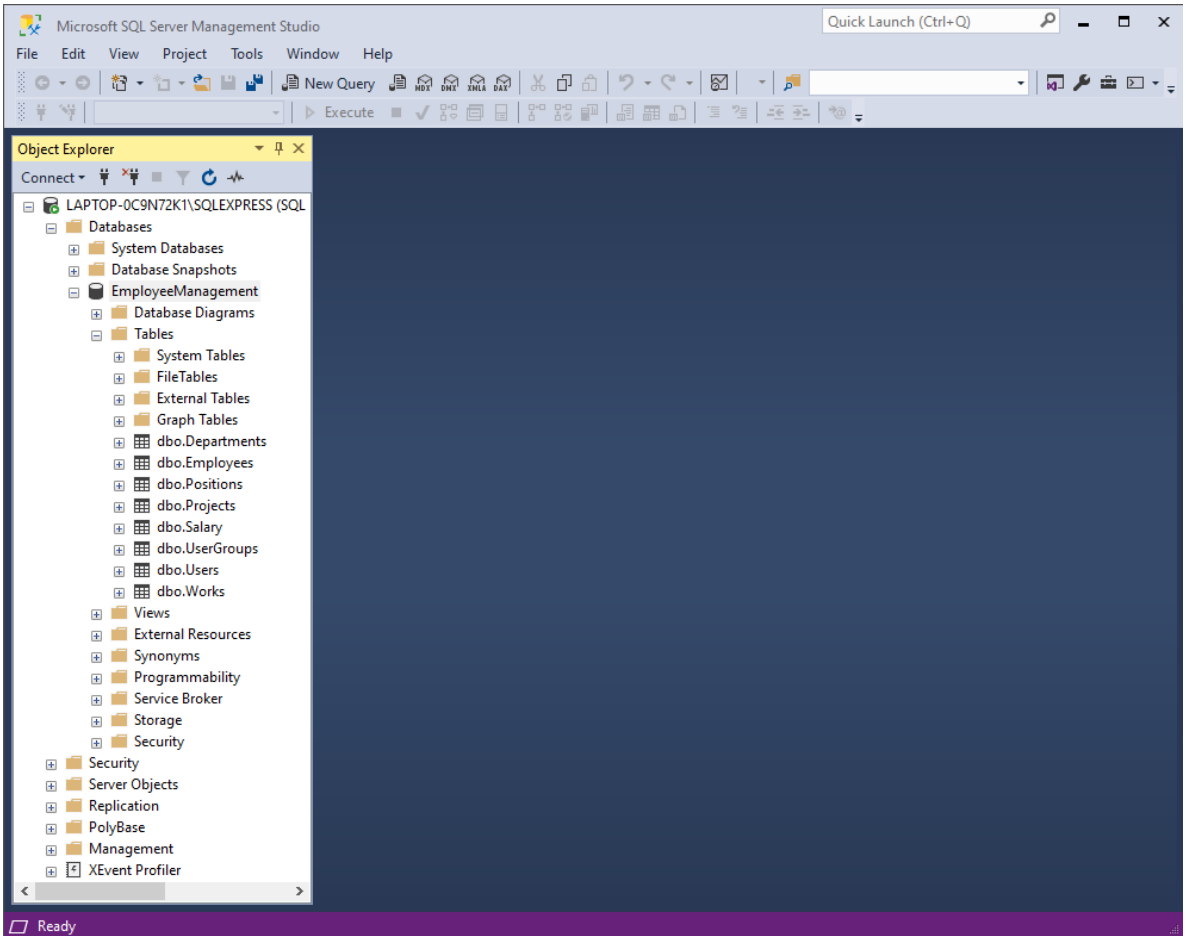

Figura 3 SQL Server Management Studio

2.4.1 Object Explorer

Object Explorer (figura 3 tabela majtas) siguron një interface hierarkike të përdoruesit (UI) për të parë dhe menaxhuar objektet në secilën shembull të SQL Server. Paneli Detajet e Eksploruesit të Objekteve paraqet një pamje tabelare të objekteve të shkallës, dhe aftësinë për të kërkuar objekte specifike. Aftësitë e Object Explorer ndryshojnë pak në varësi të llojit të serverit, por zakonisht përfshijnë tiparet e zhvillimit për bazat e të dhënave dhe tiparet e menaxhimit për të gjitha llojet e serverit [12].

3. MODELET E ZHVILLIMIT NË GJUHËN PROGRAMUESE C#

Modelet e zhvillimit janë zgjidhje tipike për problemet e zakonta gjatë hartimit të një softueri. Ato janë si projeksione të bëra paraprakisht që mund të përshtaten për të zgjidhur një problem të përsëritur të dizajnit në kodin tuaj. Ju nuk mund të gjeni thjesht një model dhe ta kopjoni atë në programin tuaj, mënyrën se si mundeni me funksionet ose libraritë. Modeli nuk është një pjesë specifike e kodit, por një koncept i përgjithshëm për zgjidhjen e një problemi të veçantë. Ju mund të ndiqni detajet e modelit dhe të zbatoni një zgjidhje që i përshtatet programit tuaj. Modelet shpesh ngatërrohen me algoritmet, sepse të dy konceptet përshkruajnë zgjidhje tipike për disa probleme të njohura. Ndërsa një algoritëm gjithmonë përcakton një grup të qartë veprimesh që mund të arrijë një qëllim, një model është një përshkrim më i nivelit të lartë të një zgjidhjeje. Kodi i të njëjtit model që aplikohet në dy programe të ndryshme mund të jetë i ndryshëm. Një analogji me një algoritëm është një recetë gatimi: të dyja kanë të qarta hapat për të arritur një qëllim. Nga ana tjetër, një model është më shumë si një plan: mund të shihet se cili është rezultati dhe tiparet e tij, por rendi i saktë i zbatimit varet nga personi që përdor modelin [2].

3.1. Klasifikimi i modeleve

Modelet e dizajnit ndryshojnë nga kompleksiteti i tyre, niveli i detajeve dhe shkalla e zbatueshmërisë për të gjithë sistemin që po projektohet. Modelet më themelore dhe të nivelit të ulët shpesh quhen **idioma**. Zakonisht ato aplikohen vetëm në një gjuhë të vetme programimi. Modelet më universale dhe të nivelit të lartë janë modelet **arkitektonike**. Zhvilluesit mund t'i zbatojnë këto modele praktikisht në çdo gjuhë. Ndryshe nga modelet e tjera, ato mund të përdorën që të hartoni arkitekturën e një aplikacioni të tërë.

Përveç kësaj, të gjitha modelet mund të kategorizohen nga qëllimi i tyre. Modelet e zhvillimit ndahen në tri grupe [2]:

- **Modelet krijuese (Creational patterns)** sigurojnë mekanizma të krijimit të objekteve që rrisin fleksibilitetin dhe ripërdorimin e kodit ekzistues.
- **Modelet strukturore (Structural patterns)** shpjegojnë grumbullimin e objekteve dhe klasave në struktura më të mëdha, duke i mbajtur strukturat fleksibile dhe efikase.

- **Modelet e sjelljes (Behavioral patterns)** kujdesen për komunikimin efektiv dhe caktimin e përgjegjësive ndërmjet objekteve.

3.2. Historiku i modeleve të zhvillimit

Koncepti i modeleve u përshkrua për herë të parë nga Christopher Aleksandri në *A Pattern Language: Towns, Buildings, Construction* [13]. Libri përshkruan një "gjuhë" për hartimin e mjedisit urban. Njësitë e kësaj gjuhe janë modelet. Ato mund të përshkruajnë sa duhet të jenë dritaret e larta, sa nivele një ndërtesë duhet të ketë, sa hapësira të gjelbërta në një lagje supozohet të jenë, e kështu me radhë. Ideja u mor nga katër autorë: Erich Gamma, John Vlissides, Ralph Johnson dhe Richard Helm. Në 1995, ata botuan *Design Patterns: Elements of Reusable Object-Oriented Software* [1], në të cilin ata aplikuan konceptin e modeleve të dizajnit te programimit. Libri përmbante 23 modele që zgjidhnin probleme të ndryshme të dizajnit të orientuar në objekte dhe u bë njëri ndër librat më të shitura shumë shpejt. Për shkak të emrit të saj të gjatë, njerëzit filluan të e quajti atë "the book by the gang of four" i cili shpejt u shkurtua thjesht "the GOF book". Që atëherë, dhjetëra modele të tjera të orientuara në objekte janë Zbuluar. Pattern approach u bë shumë e popullarizuar në fusha të tjera programimi, kështu që shumë modele të tjera tani ekzistojnë edhe jashtë modelit të orientuar në objekte [2].

3.3 SOLID Parimet

Parimet SOLID janë parimet e zhvillimit që na mundësojnë të menaxhojmë shumicën e problemeve të zhvillimit të softuerit. Robert C. Martin i përpiloi këto parime në vitet 1990. Këto parime ofrojnë mënyra për të kaluar nga kodi i bashkuar ngushtë dhe pak kapsulimi në rezultatet e dëshiruara të nevojave reale të shoqëruara lirshëm dhe të kapsuluara siç duhet. SOLID është shkurtesa e mëposhtme [14]:

- S: Single Responsibility Principle (SRP) - "Çdo modul programi duhet të ketë vetëm një arsye për të ndryshuar"
- O: Open closed Principle (OSP) - "Një modul / klasë softuer është i hapur për zgjerim dhe i mbyllur për modifikim"

- L: Liskov substitution Principle (LSP) - "duhet të jesh në gjendje të përdorësh çdo klasë të prejardhur në vend të një klase prind dhe ta kesh atë të sillet në të njëjtën mënyrë pa modifikuar"
- I: Interface Segregation Principle (ISP) - "që klientët të mos detyrohen të zbatojnë interface që nuk përdorin. Në vend të një interface dhjamore, shumë interface të vogla preferohen bazuar në grupe metodash, secila që shërben nga një nën modul."
- D: Dependency Inversion Principle (DIP) - "modulet / klasat e nivelit të lartë nuk duhet të varen nga modulet / klasat e nivelit të ulët. Të dy duhet të varen nga abstraksionet"

3.4 UML

Për të i kuptuar më mirë shembujt në vazhdim duhet të flasim edhe për UML diagramet, UML shkurt për Unified Modeling Language, është një gjuhë e standardizuar modelimi e përbërë nga një grup i integruar diagramesh, i zhvilluar për të ndihmuar zhvilluesit e sistemit dhe softuerit për specifikimin, vizualizimin, ndërtimin dhe dokumentimin e objekteve të sistemeve softuerike.

UML përfaqëson një koleksion të praktikave më të mira inxhinierike që kanë rezultuar të suksesshme në modelimin e sistemeve të mëdha dhe komplekse. UML është një pjesë shumë e rëndësishme e zhvillimit të softuerit të orientuar në objekte dhe procesit të zhvillimit të softuerit. UML përdor kryesisht shënime grafike për të shprehur hartimin e projekteve softuerike. UML kanë struktura të ndryshme por në këtë punim është përdorur diagrami i klasës figura 4. Diagrami i klasës është një teknikë qendrore e modelimit që përshkon pothuajse të gjitha metodat e orientuara në objekte [15].

Klasat janë një përshkrim i një grupi objektesh të gjithë me role të ngjashme në sistem, i cili përbëhet nga *Atributet* përcaktojnë se çfarë "njohin" objektet e klasës dhe *Operatorët* përcaktojë se çfarë "mund të bëjnë" objektet e klasës. Një shënim i klasës përbëhet nga tre pjesë *Emri i klasës* i cili duket në pjesën e parë, *Atributi i klasës* i cili duket në pjesën e dytë zakonisht lloji i atributit tregohet pas dy pikave dhe Operatorët e klasës (Metodat) tregohen në pjesën e tretë ato janë shërbime që ofron klasa, tipi që kthen metoda zakonisht tregohet

pas dy pikave [16], shenja (+) do të thotë që atributi apo metoda është publike kurse shenja (–) nëse janë private, Figura 4.

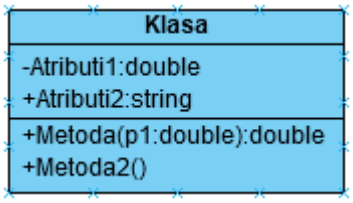

Figura 4 Shënimi i klasës

Ekzistojnë pesë lloje kryesore të marrëdhënieve të klasave (Figura 5) [16]:

1. Simple Association - Një lidhje strukturore midis dy klasave, Një vijë solide që lidh dy klasa.
2. Inheritance or Generalization (Trashegimia) - Përfaqëson një marrëdhënie "është-një", Një klasë abstrakte tregohet me shkronja të pjerrëta. Një vijë e fortë me një shigjetë të zbrazët që drejtohet nga fëmija te klasa prindërore.
3. Aggregation - Një lloj i veçantë i Associacionit. Ajo përfaqëson një marrëdhënie "pjesë e", Një linjë e fortë me një diamant të paplotësuar në fund të associacionit, e lidhur me klasën e përbërësit.
4. Composition (Përbërja) - Një lloj i veçantë i Aggregation ku pjesët shkatërrohen kur e tëra shkatërrohet, Një linjë e fortë me një diamant të mbushur në asociacionin e lidhur me klasën e përbërësit.
5. Dependency (Varësia) - Ekziston midis dy klasave nëse ndryshimet në përkufizimin e njërës mund të shkaktojnë ndryshime në tjetrën (por jo anasjelltas), Një vijë e ndërprerë me një shigjetë të hapur.

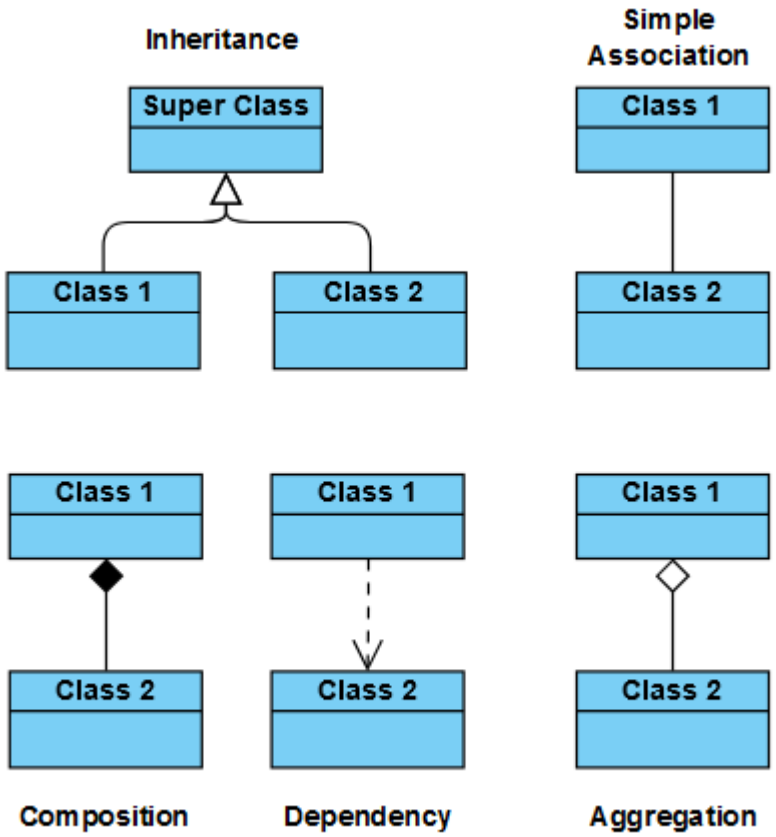

Figura 5 Marrëdhënia e klasave [16]

3.5 Creational Patterns

Creational patterns ose në shqip Modelet krijuese ofrojnë mekanizma të ndryshëm të krijimit të objekteve, të cilat rrisin fleksibilitetin dhe ripërdorimin e kodit ekzistues, ato ndahen në [2]:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

3.5.1 Factory Method

Factory Method është një mënyrë e krijimit të objekteve, por lejimi i nënklasave të vendosin saktësisht se cilën klasë do të instancojnë. Nënklasa të ndryshme mund të implementojnë interface.
Factory Method instancon nënklasën e duhur bazuar në informacionin e furnizuar nga klienti ose të nxjerrë nga gjendja aktuale [17].

Për ta kuptuar më mirë Factory Method është përdorur një shembull i botës reale dhe pastaj është treguar edhe në formë të një programi të shkruar në C#. Konsideroni një dyqan ushqimesh të klasit të lartë në Londër që rezervon avokado gjatë gjithë vitit. Ai mbështetet tek një blerës për të siguruar që avokadot të vijnë rregullisht, pavarësisht nga koha e vitit. Blerësi zgjidhë burimin e avokadove më të mira dhe i furnizon ato në dyqan. Blerësi po operon si një Factory Method, duke kthyer avokadot Keniane, Afrikën e Jugut ose Spanjolle në varësi të periudhës së vitit. Megjithëse prodhimi është i etiketuar, magazinieri nuk është veçanërisht i interesuar për burimin e produkteve [17]. Figura 6 tregon se çfarë mund të ndodhë në periudha të ndryshme të vitit.

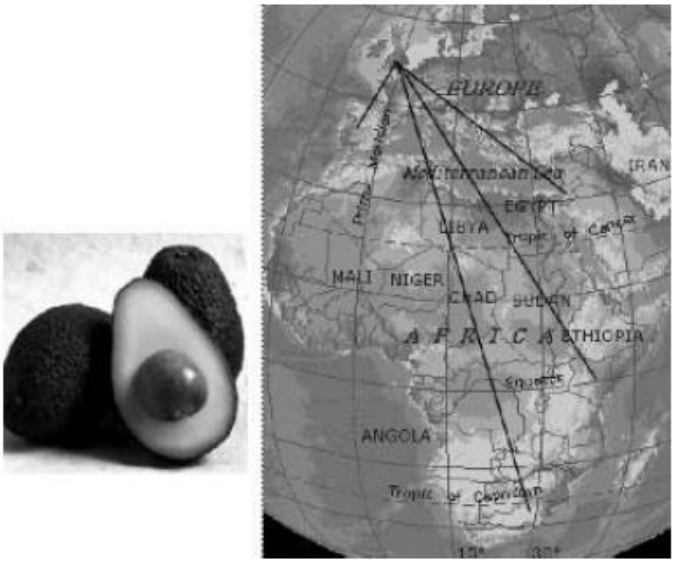

Figura 6 Ilustrimi i Factory Method - Avocado sourcing [17]

Shembulli i cili është përdorur mund të tregohet edhe me anë të një UML diagrami. Klienti deklaron një ndryshore të Produktit por thërret një Factory Method për ta instancuar atë. Kjo shtyn vendimin për atë produkt të veçantë për tu krijuar. Në diagramin UML në Figurën 7, ekzistojnë dy zgjedhje: ProductA dhe ProductB [17].

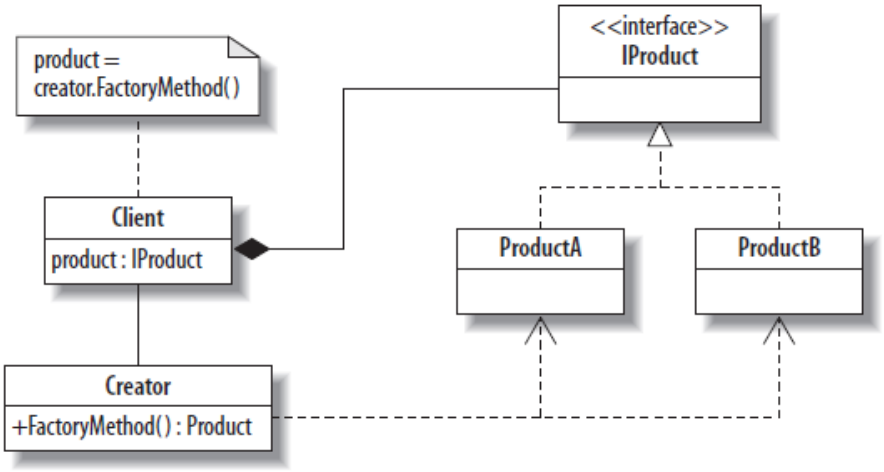

Figura 7 Factory Method UML diagram [17]

Përbërësit e diagramit UML që tregohet në figurën 7 janë:

- IProduct : Interface për produktet
- Product A dhe Product B : Klasat që implementojnë Produktin
- Creator: Siguron Metodën e Fabrikës
- Factory Method: Vendos se cilën klasë do ta instancoj

Dizajni i këtij modeli mundëson që vendimmarrja se cili produkt të instancohet të trajtohet në një vend. Nëse klienti do të dinte për të gjitha opsionet, vendimet do të shpërndaheshin në kodin e tij. Siç është, klienti duhet të merret vetëm me marrjen e produkteve, madje nuk duhet të lidhet në të gjitha nënklasat e ndryshme të produkteve. Sigurisht, klienti mund të ketë më shumë se një krijues, për lloje të ndryshme të produkteve [17].

Një program i thjeshtë që zbaton shembullin e furnizuesit të avokados është treguar në Figurën 6. Secili prej vendeve prodhuese të avokados ka një klasë, dhe të gjithë ata zbatojnë IProduct në mënyrë që të mund të transportojnë avokado te blerësi. Në varësi të muajit, Factory Metoda (linjat 37 – 48) do të zgjedhë të furnizojë ose ProductA (avokado nga Afrika

e Jugut) ose ProductB (avokado nga Spanja). Ekziston edhe një opsion i tretë, një DefaultProduct, i cili zgjidhet kur avokadot nuk janë të disponueshme (në këtë rast, në muajin 3). Linja e rëndësishme është linja 53, e cila tregon që një produkt krijohet nga një fabrikë (blerësi) pa e ditur klienti klasën e produktit, në figurën 18 është treguar kodi i Factory Method [17].

```
1  using System;
2  using System.Collections;
3
   0 references
4  internal class FactoryPattern
5  {
6      // Factory Method Pattern Judith Bishop 2006
7
       5 references
8      internal interface IProduct
9      {
           4 references
10         string ShipFrom();
11     }
12
       0 references
13     internal class ProductA : IProduct
14     {
           4 references
15         public String ShipFrom()
16         {
17             return " from South Africa";
18         }
19     }
20
       0 references
21     internal class ProductB : IProduct
22     {
           4 references
23         public String ShipFrom()
24         {
25             return "from Spain";
26         }
27     }
28
       0 references
29     internal class DefaultProduct : IProduct
30     {
           4 references
31         public String ShipFrom()
32         {
33             return "not available";
34         }
35     }
36
       1 reference
37     internal class Creator
38     {
           1 reference
39         public IProduct FactoryMethod(int month)
40         {
41             if (month >= 4 && month <= 11)
42                 return new ProductA();
43             else
44               if (month == 1 || month == 2 || month == 12)
45                 return new ProductB();
46             else return new DefaultProduct();
47         }
48     }
49
       0 references
50     private static void Main()
51     {
52         Creator c = new Creator();
53         IProduct product;
54
55         for (int i = 1; i <= 12; i++)
56         {
57             product = c.FactoryMethod(i);
58             Console.WriteLine("Avocados " + product.ShipFrom());
59         }
60         Console.ReadKey();
61     }
62 }
```

Figura 8 Factory Method kodi shembull [17]

Figura 9 tregonë daljen pasi është ekzekutuar shembulli i Factory Method.

```
Avocados from Spain
Avocados from Spain
Avocados not available
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados from Spain
```

Figura 9 Dalja e kodit të Factory Method [17]

Factory Method është një model i lehtë që arrin pavarësinë nga klasat specifike të aplikimit. Klienti programon në Interface (në këtë rast, IProduct) dhe lejon që modeli të zgjidhë pjesën tjetër. Një avantazh i veçantë i Factory Method është se ai mund të lidhë hierarkitë e klasave paralele. Nëse secila hierarki përmban një Factory Method, ajo mund të përdoret për të krijuar shembuj të nënklasave në një mënyrë të ndjeshme [17].

3.6 Structural Patterns

Structural Patterns në shqip Modelet Strukturore shpjegojnë sesi të grumbullohen objektet dhe klasat në struktura më të mëdha, duke e mbajtur kështu strukturën më fleksibile dhe efikase. Modelet Strukturore në C# janë [2]:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

3.6.1 Adapter Pattern

Adapter Pattern shndërron një interface klase në një interface tjetër që klientët presin. Adapter Pattern lejon që klasat të punojnë së bashku edhe pse ato kanë interface të pa pajtueshëm [18].

Supozoni se keni një aplikacion që mund të klasifikohet gjerësisht në dy pjesë: Interface i përdoruesit (UI ose Front-End) dhe baza e të dhënave (Back-End). Përmes interface-it të përdoruesit, klientët mund të kalojnë disa lloje specifike të të dhënave ose objekteve. Baza e të dhënave tuaja është në përputhje me ato objekte dhe mund t'i ruajë ato pa probleme. Gjatë një periudhe kohore, mund të mendoni se keni nevojë të përditësoni softuerin tuaj për të bërë të lumtur klientët tuaj. Kështu që, mund të dëshironi të lejoni që një lloj tjetër objekti gjithashtu të kalojë përmes interfacit të përdoruesit. Por në këtë rast, problemi i parë do të vijë nga baza e juaj e të dhënave sepse nuk mund të ruajë këto lloje të reja të objekteve. Në situatë të tilla, mund të përdorni një përshtatës (adapter) që do të kujdeset për përshtatshmërinë e këtyre objekteve të reja në një formë të pajtueshme që baza juaj e të dhënave të vjetra mund të pranojë [18].

Në një shembull tjetër, mund të llogaritni sipërfaqen e një drejtkëndëshi lehtë. Në klasën Calculator (Llogaritësi), duhet të siguroni një drejtkëndësh në metodën GetArea() për të marrë sipërfaqen e drejtkëndëshit.
Tani është supozuar se doni të llogarisni sipërfaqen e një trekëndëshi. Por kufizimi juaj është që ju doni të merrni zonën e saj përmes metodës GetArea() të klasës Calculator (Llogaritësi). Si mund ta arrini atë? Për t'u marrë me këtë problem, ju bëni një CalculatorAdapter për një trekëndësh dhe kaloni një trekëndësh në metodën e tij GetArea(). Nga ana tjetër, metoda do t'i trajtojë këto trekëndësha si drejtkëndësha për të marrë zonat e tyre nga metoda GetArea () e klasës Calculator (Llogaritësi). Figura 10 tregon diagramin e klasës [18].

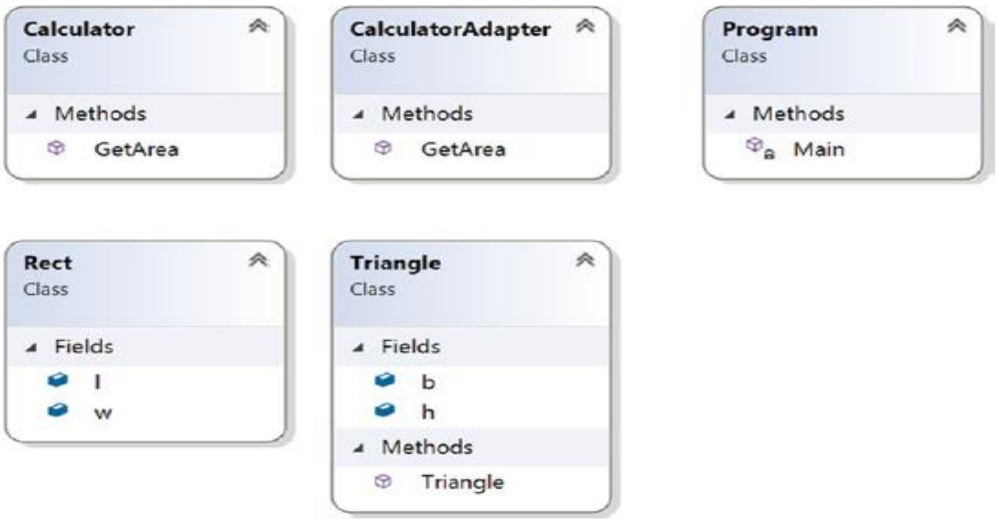

Figura 10 Diagrami i klasës te Adapter pattern [18]

Figurat 11 dhe Figura 12 tregojnë implementimin e Adapter Pattern në një program. Interface RectInterface ka një metodë dhe një funksion (linja 5 – 10), klasa Rect trashëgon interface RectInterface dhe është e obliguar të implementojë metodën dhe funksionin e sajë, gjithashtu në klasën Rect janë deklaruar dy variabla double që nevojiten për të llogaritur sipërfaqen e drejtkëndëshit (linja 12 – 32).

Interface tjetër i quajtur TriInterface ka një metodë dhe një funksion i cili përdorët për ta llogaritur sipërfaqen e trekëndëshit (linja 34 – 40), klasa Triangle trashëgon TriInterface dhe është e obliguar që ta implementoj metodën dhe funksionin e sajë, gjithashtu edhe në klasën Triangle janë deklaruar dy variabla double që do të përdorën për ta llogaritur sipërfaqen e trekëndëshit (linja 41 – 61).

Klasa TriangleAdapter trashëgon interface RectInterface dhe është e obliguar të implemetojë metodën dhe funksionin e sajë, në atë klasë është krijuar një objekt i klasës Triangle me anë të cilës do të mundë të qasen përbërësit e klasës Triangle (linja 67 – 85) dhe në fund në main është krijuar një objekt i një Rect dhe është mbushur me vlera, pastaj është thirrur funksioni i cili bënë llogaritjen e sipërfaqes dhe që shfaqë në ekran vlerën e llogaritur, e njëjta mënyrë shkon edhe për klasën Triangle por ne linjën 99 shihet që kalon nga objekti i tipit Rect në tipin Triangle (linja 87–114).

```
using System;

namespace AdapterPattern_Modified
{
    4 references
    internal interface RectInterface
    {
        3 references
        void AboutRectangle();

        4 references
        double CalculateAreaOfRectangle();
    }

    2 references
    internal class Rect : RectInterface
    {
        public double Length;
        public double Width;

        1 reference
        public Rect(double l, double w)
        {
            this.Length = l;
            this.Width = w;
        }

        4 references
        public double CalculateAreaOfRectangle()
        {
            return Length * Width;
        }

        3 references
        public void AboutRectangle()
        {
            Console.WriteLine("Actually, I am a Rectangle");
        }
    }

    1 reference
    internal interface TriInterface
    {
        2 references
        void AboutTriangle();

        3 references
        double CalculateAreaOfTriangle();
    }

    4 references
    internal class Triangle : TriInterface
    {
        public double BaseLength;//base
        public double Height;//height

        1 reference
        public Triangle(double b, double h)
        {
            this.BaseLength = b;
            this.Height = h;
        }

        3 references
        public double CalculateAreaOfTriangle()
        {
            return 0.5 * BaseLength * Height;
        }

        2 references
        public void AboutTriangle()
        {
            Console.WriteLine("Actually, I am a Triangle");
        }
    }
```

Figura 11 Zbatimi i Adapter Pattern [18]

```
62
63      /*TriangleAdapter is implementing RectInterface.
64      So, it needs to implement all the methods defined
65      in the target interface.*/
66
        1 reference
67      internal class TriangleAdapter : RectInterface
68      {
69          private Triangle triangle;
70
            1 reference
71          public TriangleAdapter(Triangle t)
72          {
73              this.triangle = t;
74          }
75
            3 references
76          public void AboutRectangle()
77          {
78              triangle.AboutTriangle();
79          }
80
            4 references
81          public double CalculateAreaOfRectangle()
82          {
83              return triangle.CalculateAreaOfTriangle();
84          }
85      }
86
        0 references
87      internal class Program
88      {
            0 references
89          private static void Main(string[] args)
90          {
91              Console.WriteLine("***Adapter Pattern Modified Demo***\n");
92              //CalculatorAdapter cal = new CalculatorAdapter();
93              Rect r = new Rect(20, 10);
94              Console.WriteLine("Area of Rectangle is :{0} Square unit",
95              r.CalculateAreaOfRectangle());
96              Triangle t = new Triangle(20, 10);
97              Console.WriteLine("Area of Triangle is :{0} Square unit",
98              t.CalculateAreaOfTriangle());
99              RectInterface adapter = new TriangleAdapter(t);
100             //Passing a Triangle instead of a Rectangle
101             Console.WriteLine("Area of Triangle using the triangle adapter is :{ 0} Square unit", GetArea(adapter));
102             Console.ReadKey();
103         }
104
105         /*GetArea(RectInterface r) method does not know that through
106         TriangleAdapter, it is getting a Triangle instead of a Rectangle*/
107
            1 reference
108         private static double GetArea(RectInterface r)
109         {
110             r.AboutRectangle();
111             return r.CalculateAreaOfRectangle();
112         }
113     }
114 }
```

Figura 12 Zbatimi i Adapter Pattern [18]

Figura 13 tregon daljen pasi është ekzekutuar kodin i shembullit të Adapter Pattern.

```
***Adapter Pattern Modified Demo***

Area of Rectangle is :200 Square unit
Area of Triangle is :100 Square unit
Actually, I am a Triangle
```

Figura 13 Të dhënat dalëse të Adapter Pattern [18]

3.7 Behavioral Patterns

Behavioral Patterns në shqip Modelet e Sjelljes kanë të bëjnë me algoritmet dhe caktimin e përgjegjësive ndërmjet objekteve. Modelet e sjelljes në C# janë [2]:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

3.7.1 Observer Pattern

Observer Pattern përcakton një varësi një në shumë midis objekteve në mënyrë që kur një objekt të ndryshojë gjendjen, të gjithë bartësit e saj njoftohen dhe përditësohen automatikisht [1].

Shumë vegla grafike të ndërfaqes së përdoruesit (UI) ndajnë aspektet prezantuese të ndërfaqes së përdoruesit nga të dhënat themelore të aplikacionit. Klasat që përcaktojnë të dhënat e aplikacioneve dhe prezantimet mund të ripërdoren në mënyrë të pavarur. Ata gjithashtu mund të punojnë së bashku. Si një objekt spreadsheet ashtu edhe një objekt bar chart mund të përshkruajnë informacionin në të njëjtin objekt të të dhënave të aplikacionit duke përdorur prezantime të ndryshme. Tabela dhe grafiku i shtyllave nuk dinë për njëri-tjetrin, duke lejuar kështu të ripërdoren vetëm ato që nevojiten. Por ata sillen sikur dinë. Kur përdoruesi ndryshon informacionin në spreadsheet, diagrami me shirita pasqyron menjëherë ndryshimet dhe anasjelltas [1].

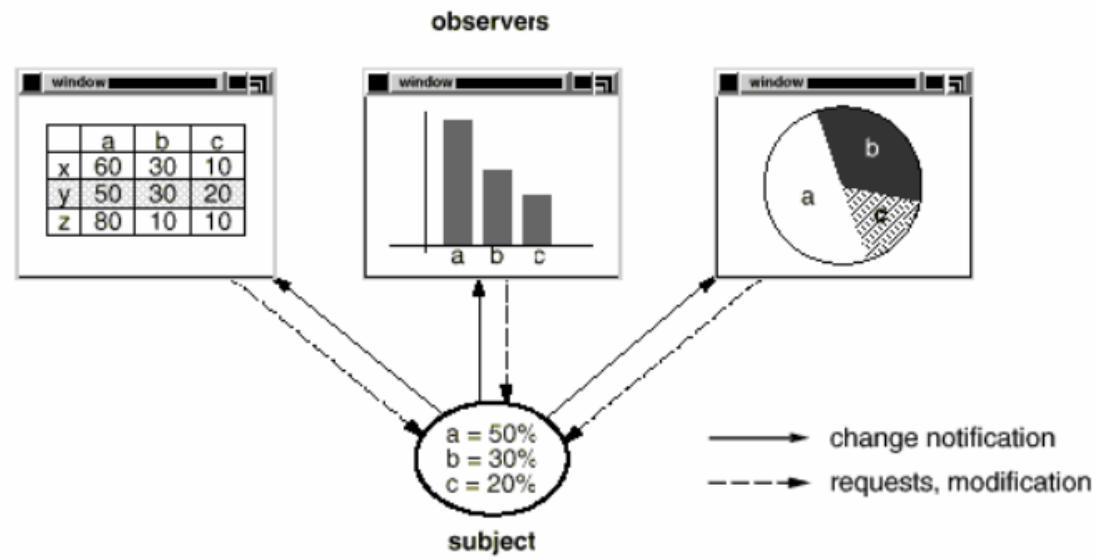

Figura 14 Shembulli Observer [1]

Kjo sjellje nënkupton që spreadsheet dhe diagrami i shiritave janë të varur nga objekti i të dhënave dhe për këtë arsye duhet të njoftohen për çdo ndryshim në gjendjen e tij. Dhe nuk ka asnjë arsye për të kufizuar numrin e objekteve të varura në dy, mund të ketë ndonjë numër të ndërfaqeve të ndryshme të përdoruesit për të njëjtat të dhëna [1].

Observer Pattern përshkruan se si të vendosen këto marrëdhënie. Objektet kryesore në këtë model janë **subject** dhe **observers** (vëzhgues). Një subjekt mund të ketë ndonjë numër të vëzhguesve të varur. Të gjithë vëzhguesit njoftohen sa herë që subjekti pëson një ndryshim në gjendje. Si përgjigje, secili vëzhgues do të pyesë subjektin për të sinkronizuar gjendjen e tij me gjendjen e subjektit [1].

Ky lloj ndërveprimi njihet gjithashtu si **publish-subscribe**. Subjekti është botuesi i njoftimeve. Ai dërgon këto njoftime pa pasur nevojë të dijë se cilët janë vëzhguesit e tij. Çdo numër vëzhguesish mund të pajtohen për të marrë njoftime [1].

Përdorni modelin e vëzhguesit në ndonjë nga situatat e mëposhtme [1]:

- Kur një abstraksion ka dy aspekte, njëri varet nga tjetri. Kapsulimi i këtyre aspekteve në objekte të veçanta ju lejon të ndryshoni dhe t'i ripërdorni ato në mënyrë të pavarur.
- Kur një ndryshim në një objekt kërkon ndryshimin e të tjerëve, dhe ju nuk e dini se sa objekte duhet të ndryshohen.
- Kur një objekt duhet të jetë në gjendje të njoftojë objekte të tjera pa bërë supozime se kush janë këto objekte. Me fjalë të tjera, ju nuk doni që këto objekte të bashkohen fort.

Figura 15 tregon UML diagramin e Observer Pattern.

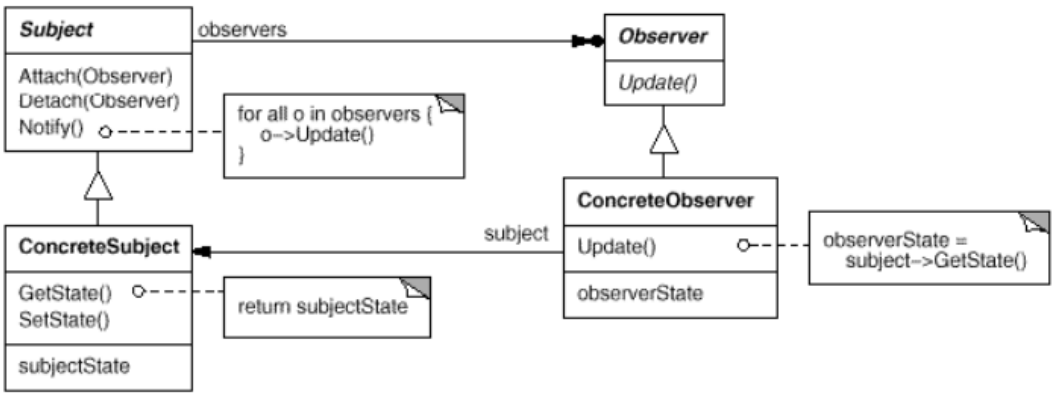

Figura 15 UML diagrami i Observer Pattern [1]

Përbërësit e strukturës në figurën 15 dhe roli i tyre janë shtjelluar në vijim:

- Subject
  - njeh vëzhguesit e saj. Çdo numër i objekteve të Vëzhguesit mund të vëzhgojë një subjekt.
  - siguron një interface për bashkëngjitjen dhe shkëputjen e objekteve të Vëzhguesit.
- Observer
  - përcakton një interface të përditësimit për objektet që duhet të njoftohen për ndryshimet në një Subject.
- ConcreteSubject

- ruan gjendjen me interes për objektet ConcreteObserver.
- dërgon një njoftim vëzhguesve të saj kur gjendja e tij ndryshon.

- ConcreteObserver
  - mban një referencë për një objekt ConcreteSubject.
  - dyqanet deklarojnë se duhet të qëndrojnë në përputhje me ato të subjektit.
  - zbaton interface e përditësimit të Vëzhguesit për të mbajtur gjendjen e saj në përputhje me atë të subjektit.

Figurat në vazhdim 16, 17, 18 tregojnë implementimin e Observer Pattern në një program, Çdo herë që është përdorur Observer Pattern janë deklaruar dy interface, Interface i parë më emër IObserver deklaron një metodë Update e cila pranon përditësimet nga Subjekti (linja 7 - 11), interface i radhës është ISubject i cili ka tri metoda Attach i cili bashkëngjitë një Vëzhgues në subject, Detach i cili largon Vëzhgues nga subjecti dhe metodën Notify e cila njofton të gjithë Vëzhguesit për një event (linja 13 – 23), pastaj është deklaruar një klasë më emrin Subject e cila trashëgon ISubject dhe është e obliguar të implementojë metodat e ISubject.

Klasa Subject përmban një gjendje të rëndësishme dhe i njofton vëzhguesit kurë ndryshon gjendja e sajë, Në klasën Subject është deklaruar një properti state (gjendje) dhe një listë e cila përmban të gjithë vëzhguesit të cilët do të instancohën, Klasa Subject zakonisht brenda sajë përmban një ose më shumë metoda të cilat përmbajnë logjikë të rëndësishme (linja 27 – 75).

Mundë të deklarohen disa vëzhgues të cilët reagojnë pas një njoftimi i cili vije nga Subject por në këtë shembull kemi vetë dy (linja 79 – 99).

Në metodën main shihet kodi i klientit në të cilin përdorën të gjitha metodat e klasave dhe interface (linja 101 – 122).

```
1   using System;
2   using System.Collections.Generic;
3   using System.Threading;
4
5   namespace RefactoringGuru.DesignPatterns.Observer.Conceptual
6   {
        8 references
7       public interface IObserver
8       {
9           // Receive update from subject
                3 references
10          void Update(ISubject subject);
11      }
12
            4 references
13      public interface ISubject
14      {
15          // Attach an observer to the subject.
                3 references
16          void Attach(IObserver observer);
17
18          // Detach an observer from the subject.
                2 references
19          void Detach(IObserver observer);
20
21          // Notify all observers about an event.
                2 references
22          void Notify();
23      }
24
25      // The Subject owns some important state and notifies observers when the
26      // state changes.
            3 references
27      public class Subject : ISubject
28      {
29          // For the sake of simplicity, the Subject's state, essential to all
30          // subscribers, is stored in this variable.
                5 references
31          public int State { get; set; } = -0;
32
33          // List of subscribers. In real life, the list of subscribers can be
34          // stored more comprehensively (categorized by event type, etc.).
35          private List<IObserver> _observers = new List<IObserver>();
36
37          // The subscription management methods.
```

Figura 16 Implementimi i Observer Pattern [19]

```
38          public void Attach(IObserver observer)
39          {
40              Console.WriteLine("Subject: Attached an observer.");
41              this._observers.Add(observer);
42          }
43
            2 references
44          public void Detach(IObserver observer)
45          {
46              this._observers.Remove(observer);
47              Console.WriteLine("Subject: Detached an observer.");
48          }
49
50          // Trigger an update in each subscriber.
            2 references
51          public void Notify()
52          {
53              Console.WriteLine("Subject: Notifying observers...");
54
55              foreach (var observer in _observers)
56              {
57                  observer.Update(this);
58              }
59          }
60
61          // Usually, the subscription logic is only a fraction of what a Subject
62          // can really do. Subjects commonly hold some important business logic,
63          // that triggers a notification method whenever something important is
64          // about to happen (or after it).
            3 references
65          public void SomeBusinessLogic()
66          {
67              Console.WriteLine("\nSubject: I'm doing something important.");
68              this.State = new Random().Next(0, 10);
69
70              Thread.Sleep(15);
71
72              Console.WriteLine("Subject: My state has just changed to: " + this.State);
73              this.Notify();
74          }
75      }
76
77      // Concrete Observers react to the updates issued by the Subject they had
78      // been attached to.
        0 references
79      internal class ConcreteObserverA : IObserver
80      {
            3 references
81          public void Update(ISubject subject)
82          {
83              if ((subject as Subject).State < 3)
84              {
85                  Console.WriteLine("ConcreteObserverA: Reacted to the event.");
86              }
87          }
88      }
89
        0 references
90      internal class ConcreteObserverB : IObserver
91      {
            3 references
92          public void Update(ISubject subject)
93          {
94              if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
95              {
96                  Console.WriteLine("ConcreteObserverB: Reacted to the event.");
97              }
98          }
99      }
```

Figura 17 Implementimi i Observer Pattern [19]

```
0 references
internal class Program
{
    0 references
    private static void Main(string[] args)
    {
        // The client code.
        var subject = new Subject();
        var observerA = new ConcreteObserverA();
        subject.Attach(observerA);

        var observerB = new ConcreteObserverB();
        subject.Attach(observerB);

        subject.SomeBusinessLogic();
        subject.SomeBusinessLogic();

        subject.Detach(observerB);

        subject.SomeBusinessLogic();
        Console.ReadKey();
    }
}
}
```

Figura 18 Implementimi i Observer Pattern [19]

Figura 19 tregon daljen pasi është ekzekutuar kodi i shembullit të Observer Pattern.

```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 8
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 1
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
```

Figura 19 Të dhënat dalëse të Observer Pattern [19]

4. RAST STUDIMI – ZHVILLIMI I APLIKACIONIT

Pjesë e këtij punimi është edhe aplikacioni që është zhvilluar si pjesë implementuese e dy modeleve kryesore. Pra, aplikacioni implementon dy nga modelet e zhvillimit siç janë Factory Pattern dhe Observer Pattern dhe janë përzgjedhur për arsye se janë modelet më të përdorura në çdo gjuhë programimi.

Aplikacioni quhet "Employee Management" dhe qëllimi i këtij aplikacioni është kryerja e të gjitha nevojave më rëndësi për menaxhimin e punëtoreve. Aplikacioni përmban një login më anë të cilit bëhet qasja në aplikacion siç tregohet në figurën 20.

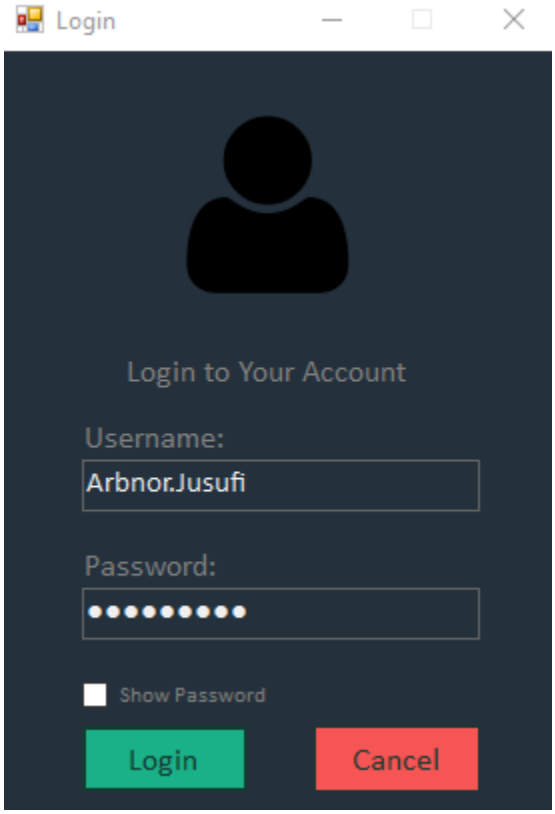

Figura 20 Login Form

Aplikacioni bënë dallimin nëse personi që është qasur është një Administrator apo një Menaxher, në varësi të përdoruesit do të kenë më shumë ose më pak kontroll rreth aplikacionit. Pasi të jemi qasur në aplikacion do të hapetë forma kryesore siç tregohet në figurën 21.

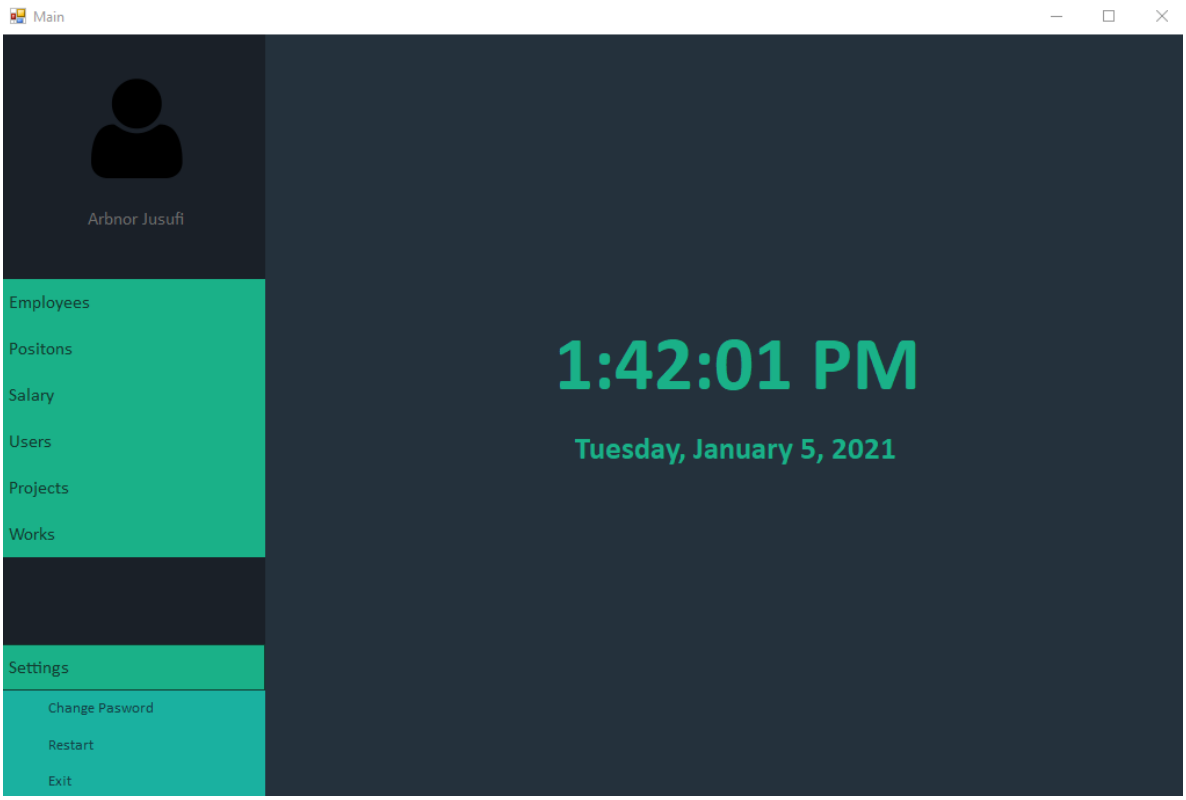

Figura 21 Main From

Të dy përdoruesit mundë të regjistrojnë një punëtor, të bëjnë përditësime apo edhe ta fshijnë atë, por vetëm administratori ka mundësinë të shtojë apo ta fshijë një përdorues.

Pasi të regjistrohet një punëtor pa dyshim nevojitet edhe të ju caktohet ndonjë pagë të cilën po ashtu mund ta bëjmë më anë të këtij aplikacioni pasi të është regjistruar punëtori dhe paga e tij, ajo ruhet në bazën e të dhënave në mënyrë që të kemi një pasqyre të punëtoreve dhe pagave të tyre, prej nga mund të llogarisim pagën bruto në neto dhe anasjelltas. Përdoruesi ka mundësin të regjistroj edhe projektet të cilat gjithashtu do të ruhen në bazën e të dhënave të kompanisë dhe ti caktohet ndonjë menaxher nga punëtoret e ruajtur në bazën e të dhënave dhe të ju regjistrohet se sa kanë punuar punëtoret për ti përfunduar ato projekte. Projektit mund të ju shtohet data e fillimit, data e mbarimit dhe ndonjë koment më rëndësi qe mundë të ju hyjë në punë punëtoreve. Pra, me anë të këtij projekti mundë të bëhet:

- Regjistrimi i një Përdoruesit
- Regjistrimi i një Punëtori
- Editimi i një Punëtori
- Fshirja e një Punëtori

- Regjistrimi i pagës së një Punëtori
- Kalkulimi i pagës së Punëtorit
- Menaxhimi i Projekteve

4.1 Implementimi i Factory Method

Në këtë projekt Factory Method është përdorur për ta bërë kodin më fleksibil. Figura 22 tregon strukturën e projektit.

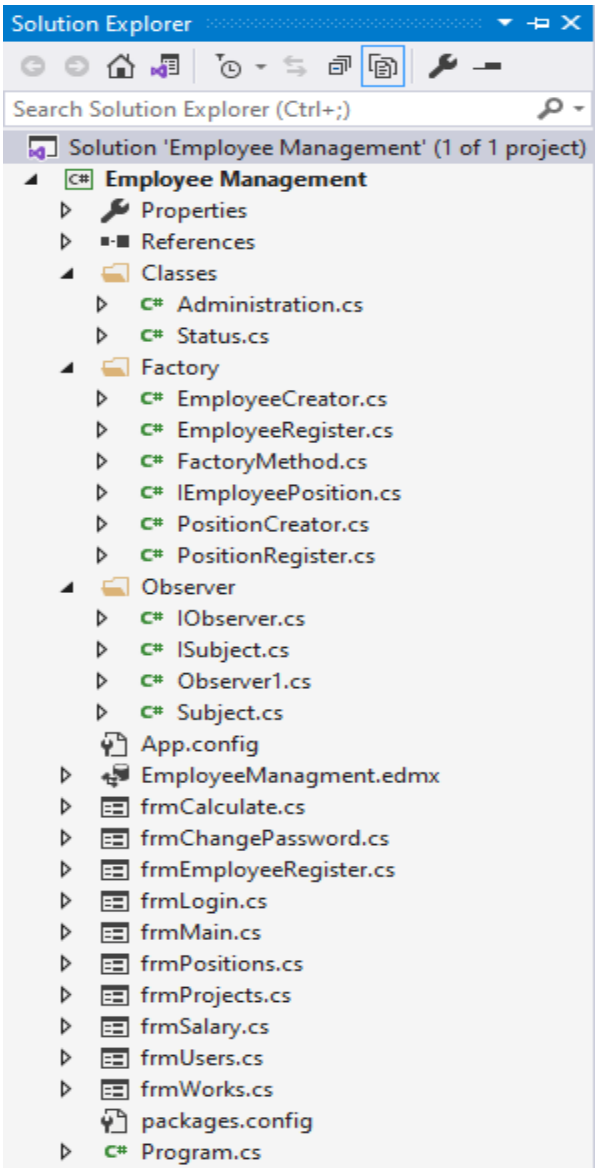

Figura 22 Pamja e Solution Explorer

Klasa **FactoryMethod** deklaron një metodë abstrakte që kthen një IEmployePosition. Figura 23 tregon klasën **FactoryMethod**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Salary_Managment.Factory
{
    11 references
    public abstract class FactoryMethod
    {
        11 references
        public abstract IEmployeePosition Create();
    }
}
```

Figura 23 Klasa FactoryMethod

Interface **IEmployeePosition** deklaron të gjitha operacionet që çdo klasë që e trashëgon duhet të implementojnë. Figura 24 tregon Interface **IEmployeePosition**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Salary_Managment.Factory
{
    //IEmployeesPosition
    5 references
    public interface IEmployeePosition
    {
        4 references
        object GetCMBList();

        4 references
        object GetDGVList(string searchtext);

        4 references
        object EditSaveDGVList();

        4 references
        object DGVRowEnter();

        3 references
        object DeleteRecord();
    }
}
```

Figura 24 Interface IEmployeePosition

Klasa **EmployeeRegister** siguron implementimin e Interface **IEmployeePosition**. Figura 25 dhe 26 tregojnë klasën **EmployeeRegister**.

```
using System;
using System.Linq;

namespace Salary_Managment.Factory
{
    3 references
    public class EmployeeRegister : IEmployeePosition
    {
        2 references
        public bool IsEditting { get; set; }
        6 references
        public int SelectedEmployeeId { get; set; }
        5 references
        public string firstname { get; set; }
        5 references
        public string lastname { get; set; }
        5 references
        public int departmentId { get; set; }
        5 references
        public string email { get; set; }
        5 references
        public DateTime dtofbirth { get; set; }
        5 references
        public string mobile { get; set; }
        2 references
        public bool genderF { get; set; }
        5 references
        public bool genderM { get; set; }
        5 references
        public DateTime dtpRegisteredDate { get; set; }
        5 references
        public string address { get; set; }

        4 references
        public object GetDGVList(string searchtext)
        {
            using (var context = new EmployeeManagementEntities())
            {
                var user = (from e in context.Employees.AsNoTracking()
                            where (searchtext != null ? (
                            e.FirstName.ToLower().StartsWith(searchtext) ||
                            e.LastName.ToLower().StartsWith(searchtext) ||
                            e.Department.Name.ToLower().StartsWith(searchtext)) : true)
                            orderby e.Id descending
                            select new
                            {
                                e.Id,
                                e.FirstName,
                                e.LastName,
                                e.Department.Name,
                                e.DateofBirth,
                                e.Email,
                                e.Phone,
                                e.Gender,
                                e.RegisteredDate,
                                e.Address
                            }).ToList();
                return user;
            }
        }

        4 references
        public object GetCMBList()
        {
            using (var context = new EmployeeManagementEntities())
            {
                var department = (from d in context.Departments.AsNoTracking()
                                  select new
                                  {
                                      d.Id,
                                      Name = d.Name
                                  }).ToList();
                return department;
            }
        }

```

Figura 25 Klasa EmployeeRegister

```
62          public object EditSaveDGVList()
63          {
64              using (var context = new EmployeeManagementEntities())
65              {
66                  Employee emp;
67                  if (IsEditting)
68                  {
69                      emp = context.Employees.Where(t => t.Id == SelectedEmployeeId).FirstOrDefault();
70                      emp.FirstName = firstname;
71                      emp.LastName = lastname;
72                      emp.DepartmentId = departmentId;
73                      emp.Email = email;
74                      emp.DateofBirth = dtofbirth;
75                      emp.Phone = int.Parse(mobile);
76                      emp.Gender = (genderM ? "M" : "F");
77                      emp.RegisteredDate = dtpRegisteredDate;
78                      emp.Address = address;
79                  }
80                  else
81                  {
82                      emp = new Employee();
83                      emp.FirstName = firstname;
84                      emp.LastName = lastname;
85                      emp.DepartmentId = departmentId;
86                      emp.Email = email;
87                      emp.DateofBirth = dtofbirth;
88                      emp.Phone = int.Parse(mobile);
89                      emp.Gender = (genderM ? "M" : "F");
90                      emp.RegisteredDate = dtpRegisteredDate;
91                      emp.Address = address;
92                      context.Employees.Add(emp);
93                  }
94                  context.SaveChanges();
95                  return emp;
96              }
97          }
98
            4 references
99          public object DGVRowEnter()
100         {
101             using (var context = new EmployeeManagementEntities())
102             {
103                 var emp = context.Employees.Where(t => t.Id == SelectedEmployeeId).FirstOrDefault();
104                 firstname = emp.FirstName;
105                 lastname = emp.LastName;
106                 email = emp.Email;
107                 mobile = emp.Phone.ToString();
108                 genderM = (emp.Gender.Contains("M") ? true : false);
109                 genderF = (emp.Gender.Contains("F") ? true : false);
110                 dtofbirth = emp.DateofBirth;
111                 dtpRegisteredDate = emp.RegisteredDate;
112                 address = emp.Address;
113                 departmentId = emp.DepartmentId;
114                 return emp;
115             }
116         }
117
            3 references
118         public object DeleteRecord()
119         {
120             using (var context = new EmployeeManagementEntities())
121             {
122                 var emp = context.Employees.Where(t => t.Id == SelectedEmployeeId).FirstOrDefault();
123                 context.Employees.Remove(emp);
124                 context.SaveChanges();
125                 return emp;
126             }
127         }
128     }
129 }
```

Figura 26 Klasa EmployeeRegister

Klasa **EmployeeCreator** bën overwrite metodën e **FactoryMethod** në mënyrë që të ndryshojë llojin e produktit që kthen. Figura 27 tregon klasën **EmployeeCreator**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Salary_Managment.Factory
{
    0 references
    public class EmployeeCreator : FactoryMethod
    {
        11 references
        public override IEmployeePosition Create()
        {
            return new EmployeeRegister();
        }
    }
}
```

Figura 27 Klasa EmployeeCreator

Në figurën 28 mund të shihet implementimi i FactoryMethod në eventin **btnSave_Click** të klasës **frmEmployeeRegister**.

Siç shihet në figurë pasi të klikohet butoni **btnSave** në formën **frmEmployeeRegister** do të ekzekutohet kodi në figurë. Nëse kthehet vlera **True** pas ekzekutimit të kushtit nga funksioni **AreValid()** atëherë do të ekzekutohet kodi brenda tij. Fillimisht krijohet një objekt më emrin **EmployeeLists** të tipit **EmployeeCreator**. Pastaj krijohet një objekt me emrin **emp** i tipit **EmployeeRegjister**, **EmployeeLists.Create(),** ekzekuton metodën **Create()** e cila është bërë overwrite në klasën **EmployeeCreator** që kthen një objekt të tipit **EmployeeRegister**. Objekti **emp** duhet të kastohet në klasën **EmployeeRegjister** dhe të ruhet në një variabël më emrin **employee** e cila do të përdoret për tu qasur në propertit e klasës **EmployeeRegister** dhe ti mbush ato me të dhëna siç shihet në Figurën 28. Krejt në fund ekzekutohet metoda **EditSaveDGVList()** me anë të objektit **emp** më kodin tjetër vijues.

```
105         private void btnSave_Click(object sender, EventArgs e)
106         {
107             if (AreValid())
108             {
109                 FactoryMethod EmployeeLists = new EmployeeCreator();
110                 var emp = EmployeeLists.Create();
111                 var employee = (EmployeeRegister)emp;
112                 employee.IsEditting = IsEditting;
113                 employee.SelectedEmployeeId = SelectedEmployeeId;
114                 employee.firstname = txtFirstName.Text;
115                 employee.lastname = txtLastName.Text;
116                 employee.departmentId = int.Parse(cbDepartment.SelectedValue.ToString());
117                 employee.email = txtEmail.Text;
118                 employee.dtpRegisteredDate = dtpRegisteredDate.Value;
119                 employee.mobile = txtmoblie.Text;
120                 employee.genderM = rbMale.Checked;
121                 employee.dtofbirth = dtofbirth.Value;
122                 employee.address = txtAddress.Text;
123                 emp.EditSaveDGVList();
124                 ClearFields();
125                 FillGrid();
126                 errorProvider1.Clear();
127                 MessageBox.Show("Employee saved Successfully");
128             }
129         }
```

Figura 28 Ekzekutimi i Factory Pattern në eventin btnSave_Click

4.2 Implementimi i Observer Pattern

Në këtë projekt modeli Observer është përdorur që të njoftojë një formë për ndryshimet që ndodhin në një formë tjetër. Siç shihet edhe nga figura 22 janë përdorur dy interface dhe dy klasa.

Për çdo vëzhgues duhet të krijohet një interface me emrin ISubject që ka tri metoda kryesore:

- Attach – përdorët për të bashkangjitur një vëzhgues.
- Remove – përdorët për të fshirë një vëzhgues.
- Notify – përdorët për të njoftuar vëzhgues për ndryshimet që janë bërë.

Figura 29 tregon interface **ISubject**.

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace Salary_Managment.Observer
8   {
        2 references
9       public interface ISubject
10      {
            2 references
11          void Attach(IObserver o);
12
            1 reference
13          void Remove(IObserver o);
14
            2 references
15          void Notify(string employeeName, double employeeSalary);
16
17      }
18  }
```

Figura 29 Interface ISubject

Po ashtu për çdo vëzhgues duhet të krijohet edhe një interface me emrin **IObserver** që ka një metodë update që bënë përditësimin e të dhënave figura 30.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Salary_Managment.Observer
{
    8 references
    public interface IObserver
    {
        2 references
        void Update(string employeeName, double employeeSalary);
    }
}
```

Figura 30 Interface IObserver

Klasa **Subject** implementon të gjitha metodat e interface që trashëgon në këtë rast interface **ISubject** po ashtu në këtë klasë është deklaruar edhe një listë që i mbanë të gjithë vëzhguesit e krijuar të cilët do të njoftohen për ndonjë ndryshim eventual figura 31.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Salary_Managment.Observer
{
    1 reference
    public class Subject : ISubject
    {
        private IList<IObserver> observers;

        1 reference
        public Subject()
        {
            observers = new List<IObserver>();
        }

        2 references
        public void Attach(IObserver o)
        {
            observers.Add(o);
        }

        1 reference
        public void Remove(IObserver o)
        {
            observers.Remove(o);
        }

        2 references
        public void Notify(string employeeName, double employeeSalary)
        {
            foreach (IObserver item in observers)
            {
                item.Update(employeeName, employeeSalary);
            }
        }
    }
}
```

Figura 31 Klasa Subject

Edhe klasa **Observer1** që shihet në figurën 32, implementon të gjitha metodat e interface që trashëgon. Në këtë rast interface **IObserver,** metoda update pranon dy parametra njërin të tipit string më emër **employeName** dhe një parametër me emër **employeSalary** të tipit double. Këta dy parametra siç shihet edhe në foto do të përdorën për të përditësuar vlerat e formës **frmCalculate**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Salary_Managment.Observer
{
    0 references
    public class Observer1 : IObserver
    {
        2 references
        public void Update(string employeeName, double employeeSalary)
        {
            using (var cal = new frmCalculate())
            {
                cal.txtEmployee.Text = employeeName;
                cal.txtSalary.Text = employeeSalary.ToString();
                cal.txtSalary_withEuroSign.Text = employeeSalary.ToString("C");
                cal.ShowDialog();
            }
        }
    }
}
```

Figura 32 Klasa Observer1

Figura në vazhdim tregon se si implementohet subjekti (figura 33). Së pari duhet të krijohet një objekt të interface ISubject pastaj atë objekt në klasën e formës të cilën dëshironi ta bëni subjekt. Në konstruktorin e asaj klase inicializohet objekti me tip të klasës Subject pastaj po në atë vend thirret metoda Attach dhe e krijohet një vëzhgues i ri. Figura 33 tregon implementimin e subjektit.

```
18  namespace Salary_Managment
19  {
        2 references
20      public partial class frmSalary : Form
21      {
22          private static bool IsEditting;
23          private static int SelectedRow;
24          private static int SelectedEmp;
25          private static int PositionId;
26          private static double nrofdays;
27          private static double salaryperday;
28          private static double calsalary;
29          private ISubject _subject;
30
            1 reference
31          public frmSalary()
32          {
33              InitializeComponent();
34              _subject = new Subject();
35              _subject.Attach(new Observer1());
36          }
```

Figura 33 Implementimi i subjectit në konstruktorin e klasës frmSalary

Dhe në fund në eventin btnCalculate_Click, të formës frmSalary thirret metoda Notify e cila pranon dy parametra më vlera që do të përdoren për të përditësuar të dhënat në formën frmCalculate. Figura 34 tregon implementimin e metodës Notify në eventin btnCalculate_Click.

```
263         private void btnCalculate_Click(object sender, EventArgs e)
264         {
265             nrofdays = double.Parse(txtNrOfDays.Text);
266             salaryperday = double.Parse(txtSalary.Text);
267             calsalary = nrofdays * salaryperday;
268
269             _subject.Notify(cbEmployee.Text, calsalary);
270
271         }
```

Figura 34 Implementimi i metodës Notify në eventin btnCalculate_Click

5. PËRFUNDIMI

Qëllimi i këtij punimi ka qenë që të shtjellohen konceptet e modeleve të zhvillimit dhe mënyra e funksionimit të tyre në dizajnimin e një softueri. Është sqaruar roli i një modeli që është i nevojshëm për të vendosur se për cilat probleme të përdoret.

Me anë të figurave është paraqitur struktura e modeleve me anë të cilave është pasqyruar një strukturim i kodit aktual me të cilin zgjidhen problemet gjatë kodimit. Po ashtu, me anë të figurave është treguar edhe një shembull pa të cilën do të ishte e pa mundur kuptimi i plotë i modeleve të zhvillimit i cili përdori Factory Pattern që në bazë të muajve të vitit të merrej produkti pa pasur nevojë të dihet nga cila klasë është marrur produkti.

Implementimi i modeleve të zhvillimit është bërë në gjuhën programuese C#, në .Net Framework e cila mundëson krijimin e çdo lloj aplikacioni në platforma të ndryshme. Qëllimi i Modeleve të zhvillimit ishte që më anë të përdorimit të një ose disa modeleve të zgjidhen problemet e shpeshta që ndodhin në kod.

Në të ardhmen është menduar që desktop aplikacioni i përdorur në këtë punim të avancohet në ueb aplikacion.

6. LITERATURA

[1] “Design Patterns: Elements of Reusable Object-Oriented Software” Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994.

[2] ”Dive Into DESIGN PATTERNS” Alexander Shvets, 2019.

[3] Microsoft, “A tour of the C# language”, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/, qasur më 17.12.2020

[4] “C# Language Specification” (5th ed.) Ecma International, 2017.

[5] Microsoft, “What`s new in C# 9.0”, https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#top-level-tatements, qasur më 17.12.2020.

[6] Microsoft, “What is new in .NET 5”, https://docs.microsoft.com/en-us/dotnet/core/dotnet-five, qasur më 15.01.2021.

[7] “C# 8.0 and .NET Core 3.0 Modern Cross-Platform Development” (4th ed.) Mark J. Price, 2019.

[8] Microsoft, “Welcome to the Visual Studio IDE”, https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-019, qasur më 18.12.2020.

[9] Microsoft, https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15, qasur më 18.12.2020.

[10] Microsoft, https://azure.microsoft.com/en-us/services/sql-database/, qasur më 18.12.2020.

[11] “What is the Cloud”, https://edu.gcfglobal.org/en/computerbasics/understanding-the-cloud/1/, qasur më 18.12.2020.

[12] Microsoft, “Object Explorer”, https://docs.microsoft.com/en-us/sql/ssms/object/object-explorer?view=sql-server-ver15, qasur më 18.12.2020.

[13] “A Pattern Language: Towns, Buildings, Construction” Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King Shlomo Angel, 1977.

[14] C# Corner, “SOLID Principles in C#”, https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp, qasur më 23.12.2020.

[15] Visual Paradigm, “What is Unified Modeling Language (UML) ?”, https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/, qasur më 14.01.2021.

[16] Visual Paradigm, “What is Class Diagram”, https://www.visual-

paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/, qasur më 15.01.2021.

[17] ”C# 3.0 Design Patterns” Judith Bishop, 2007.

[18] ”Design Patterns in C# A Hands-on Guide with Real-World Examples” Vaskaran Sarcar Foreword by Priya Shimanthoor, 2018.

[19] Refactoring Guru, https://refactoring.guru/design-patterns/observer/csharp/example#example-0, qasur më 02.01.2021.

TABELA E FIGURAVE

Figura 1 "Hello, World' në C# 6
Figura 2 Visual Studio 2019 8
Figura 3 SQL Server Management Studio 10
Figura 4 Shënimi i klasës 14
Figura 5 Marrëdhënia e klasave [16] 15
Figura 6 Ilustrimi i Factory Method - Avocado sourcing [17] 16
Figura 7 Factory Method UML diagram [17] 17
Figura 8 Factory Method kodi shembull [17] 18
Figura 9 Dalja e kodit të Factory Method [17] 19
Figura 10 Diagrami i klasës te Adapter pattern [18] 21
Figura 11 Zbatimi i Adapter Pattern [18] 22
Figura 12 Zbatimi i Adapter Pattern [18] 23
Figura 13 Të dhënat dalëse të Adapter Pattern [18] 23
Figura 14 Shembulli Observer [1] 25
Figura 15 UML diagrami i Observer Pattern [1] 26
Figura 16 Implementimi i Observer Pattern [19] 28
Figura 17 Implementimi i Observer Pattern [19] 29
Figura 18 Implementimi i Observer Pattern [19] 30
Figura 19 Të dhënat dalëse të Observer Pattern [19] 30
Figura 20 Login Form 31
Figura 21 Main From 32
Figura 22 Pamja e Solution Explorer 33
Figura 23 Klasa FactoryMethod 34
Figura 24 Interface IEmployeePosition 34
Figura 25 Klasa EmployeeRegister 35
Figura 26 Klasa EmployeeRegister 36
Figura 27 Klasa EmployeeCreator 37
Figura 28 Ekzekutimi i Factory Pattern në eventin btnSave_Click 38
Figura 29 Interface ISubject 38
Figura 30 Interface IObserver 39
Figura 31 Klasa Subject 39
Figura 32 Klasa Observer1 40
Figura 33 Implementimi i subjectit në konstruktorin e klasës frmSalary 41
Figura 34 Implementimi i metodës Notify në eventin btnCalculate_Click 41