# **BEVEZETÉS**

Záródolgozatom témáját korábbi munkaköröm ihlette, annak rendkívül sokat tapasztalataiból merítettem. Korábban a vendéglátóiparban tevékenykedtem, egy kisebb panzió alkalmazott recepciósként, mely munka során elengedhetetlen és elkerülhetetlen volt az adminisztrációs célra szánt szoftverek napi szintű használata.

Az teljesen nyilvánvaló valamennyi ember számára, hogy a mai világban már minden létező problémára készült szoftveres megoldás. Magától értetődően léteznek a piacon korszerű, minden igényt kielégítő szálláshelyi információs rendszerek: számlázók, szobanyilvántartó programcsomagok. Ezen szoftverek egy része havi előfizetés keretein belül birtokolhatók, vagy szoftverlicensz alapján egyszeri áron megvásárolható alkalmazások. Irodai, adminisztratív szoftverek rendre nagyobb költségeket jelentenek egy-egy vállalkozás számára, sok kisebb szálláshely pedig nem engedheti meg magának őket, mivel a vállalkozás méretéből adódóan nincs rá égető szükség, vagy a vezetés nem tartja fontosnak a megfelelő, modern szoftver- és abból következő munkakörnyezet megteremtését alkalmazottai számára.

Esetemben az utóbbi helyzet állt fent. A vállalkozás maga éppenséggel nem volt multinacionális vállalat, de kisvállalkozás sem, hogy ne szorult volna rá egy megfelelő, akár kisebb vállalatirányítási információs rendszerre, hogy a napi adminisztráció alkalmával járó adatok (pl. vendégadatok kezelése) és információk (pl. foglalások felvitele adott időpontra) feldolgozása és kezelése ne jelenthessen többletmunkát, és maguk a folyamatok lebonyolítása minél gördülékenyebben történhessen meg.

A fentebb említett szálláshely teljesen átlagos, néhol igencsak „fapados” megoldásokat alkalmazott az adminisztrációs folyamatok bonyolítására és a feladatok megoldására:

* Microsoft Office Excel 2003 táblázatkezelő alkalmazást a foglalások időpontjainak – annak vizuális megjelenítéséből (pl. minden foglalás külön színnel ellátva, megjegyzések hozzáadása cellákhoz, illetve egyéb Excel lehetőségek alkalmazása) fakadó átláthatóság szerinti vezetésére,
* Microsoft Office Word 2003 szövegszerkesztő alkalmazást a foglalások dokumentálását szöveges formában és egyéb, nem vizuális adatok feljegyzése céljából (afféle foglalási napló),
* papír alapú vendég bejelentkező lap vezetése és papír alapon a foglalás részleteit (vendégszám, fizetés módja, napi ár) részletesen tartalmazó ívek vezetése
* Külön megvásárolt számlázóprogram(ok) alkalmazása, amely ugyan tartalmazott beépített vendégadatbázist – megadva a lehetőséget azon törzsvendégek személyes adatainak kezelésére, számlára importálására a későbbiek újra felhasználhatóságának igényét kielégítve –, de alapvetően mindig papír alapú adathalmazból került felvitelre minden újabb vendég adata.

Mondhatni embert próbáló feladat volt megtanulni ezeket a helyi specialitásokat, valamint a későbbiekben dolgozni is benne. Rengeteg más feladattól elvette az értékes munkaidőt és – véleményem szerint – rengeteg felesleges lépést tartalmazott. Tipikus példa erre az adatok többszörös, avagy redundáns eltárolása – papíralapon –, amely még adatok hatékony visszakeresését sem tette lehetővé, hiszen ha szűrni akartunk volna egy adott vendég adott foglalásának időpontjában a szobaszámra és a vendégszámra, a számlázó a személyes adatokon és a számlakiállítás dátumán kívül más hasznos információt nem tartalmazott, így erre az esetre az Excel táblában lévő időpont oszlopban kellett kikeresni a foglalást, esetlegesen a szöveges foglalási naplóhoz folyamodni plusz információhoz.

Ezen tapasztalatok összességéből körvonalazódott ki és született meg a *Virtual Receptionist* ötlete. Egy korszerű, hatékony, egyszerűen használható, integrált komplex „mikro vállalatirányítási rendszer” – kifejezetten kisvállalkozásban működő szálláshelyek számára.

Egy olyan szoftverrendszer, amely ezen folyamatokat oly módon kívánja megvalósítani, hogy általános szálláshelyi komponenseket (vendégadatbázis, számlázás, foglalási napló) próbálja minél egyszerűbb és egységesebb formába önteni, azon műveletekkel járó procedúrákat leegyszerűsíteni, érthetővé tenni, a célnak megfelelő eredményt produkálva. Ezeket mind úgy megvalósítva, hogy a rendszer szolgáltatásait a recepciós, illetve bármely vállalaton belüli informatikai alaptudással rendelkező személy könnyedén használhassa.

Jelen esetben igyekeztem minél általánosabb, absztraktabb funkciókat megvalósítani, de néhol elengedhetetlen volt, hogy egyedi igények szerint szabjam meg egy-egy funkció implementációját. Egy képzelt szálláshely, az Aranyerdő Vendégház képére formálva jött létre a kezdeti verziója – egyfajta prototípus gyanánt – egy webalapú és egy asztali alkalmazásnak, amely egy közös adatbázison keresztül ismerik tudnak közvetetten kommunikálni. Ismerik a konfigurált adatokat és azokkal dolgozhatnak meghatározott feladatokon, meghatározott hatáskörrel, amelyet az alábbi felsorolás taglal:

1. Webalkalmazás: a szálláshely belépése a rendszerbe ezen a konfigurációs felületen történik egy regisztrációval, azonosítással (autentikációval). Szálláshely alapvető tulajdonságainak testre szabása (szálláshely cégadatai: neve, székhelye, adószáma stb.; számlázási tételek beállítása, szobák beállítása: neve, kapacitása, kategóriája stb.) és későbbi konfigurálhatósága felhasználó által, amely alapján a szálláshely működni képes és ismeri önmagát a rendszerben.
2. Asztali alkalmazás: magát a szálláshelyi adminisztrációt, azaz a sok kisebb összeadódó folyamatok kezelését végző felület, amely magába foglal egy egyszerű számlázó, vendégadatbázis és foglalási napló modult.
3. Adatbázis: a szálláshely számára nélkülözhetetlen adatok perzisztens tárolásának céljából, amelyet elér a webes és asztali alkalmazás adat lekérdezés, adat manipulációs műveletek és közvetett kommunikáció céljából.

Ezen az elveken vezérelve a rendszer megtervezése abszolút felhasználó központú tervezést igényelt. Mivel esetemben egy képzelt szálláshely számára készült a rendszer, úgy nagyrészt korábbi tapasztalataimban bízhattam ismételten, illetve azokból a munkatapasztalatokból, amely a fentebb részletezett problémákat vázolta fel, így tudtam mi hasznos és mi nem, illetve mire van szükség. Azonban éltem a lehetőséggel és vendéglátóipari személyes kapcsolataimat felhasználva egy másik, létező szálláshely tulajdonosával végeztem követelmény feltárást specifikáció gyanánt szerepjáték módszer segítségével meg, egy felhasználói interjú elvégzését „projektmenedzser-fejlesztőként” a „megrendelő”-vel. Ezen interjú rengeteg egyéb más ötletet adott a rendszer megtervezéséhez és megvalósításához, lényegében csak hasznos tanácsokkal gazdagodtam. Az interjú kérdéseit és válaszait egy dokumentumban rögzítettem, amely a záródolgozat mellékletében teljes formájában helyet foglal. Adatmodellezéshez környezeti modellt, UML[[1]](#footnote-1) használati-eset diagram vázolja a rendszer funkcionális követelményeit (*Melléklet, 1. ábra*), az adatbázis megtervezéséhez pedig EK-diagram[[2]](#footnote-2) elkészítését alkalmaztam (*Melléklet, 2. ábra*), valamint készültek a grafikus felületeket vázlatosan megjelenítő képernyőtervek (*Melléklet, 3. ábra*).

Minden szükséges plusz információ megszerzése után nekiláttam a komplex szoftver megvalósításához, amely megvalósításának részletes menetét a záródolgozat fejlesztői dokumentációja tartalmazza. Nagyvonalakban a megoldásokról: az adatbázis megvalósításhoz relációs adatmodellt alkalmazva, egy korszerű, kis és közepes alkalmazásokhoz tökéletes, az Oracle által ingyenesen használható MySQL adatbázist valósítottam meg az EK-diagram relációsémává képzésével, melyhez az Apache jóvoltából ingyenesen letölthető XAMPP programcsomag részét képező phpMyAdmin webes RDBMS[[3]](#footnote-3) nyújtott segítségét. A webalkalmazást HTML, CSS, JavaScript és PHP jelölő- illetve szkript nyelvekkel valósítottam meg, külső ingyenes API-t is felhasználva (jQuery, Bootstrap 4, PopperJS stb.). Az asztali alkalmazásom nagyobb funkcionalitását és robosztus tartalmát tekintve többrétegű architektúrában került megvalósításra a Microsoft .NET Frameworkben megtalálható, Windows platformra hagyományos grafikus alkalmazások készítését lehetővé C# magas szintű, tisztán objektum-orientált programozási nyelvvel történő fejlesztéssel. Az alkalmazások fejlesztéséhez további segítséget nyújtott, hogy alkalmazkodva a modern szoftverfejlesztési trendeket követve verziókövető rendszer segítségével fejlesztettem ki. Azon belül is a GitHub alkalmazással webes kezelőfelületével és az asztali grafikus verziókövető menedzser alkalmazással, amely igencsak gördülékennyé és hatékonnyá tette a projektek elkészítését.

Mivel az rendszer webalkalmazása közös W3C[[4]](#footnote-4) szabványok által ajánlott nyelvekben történt, tekinthető cross-platform[[5]](#footnote-5) alkalmazásnak (operációs rendszer tekintetében), amely komponenseinek megjelenítésében csak az internetes böngészők szabhatnak határt; a C# nyelvben íródott asztali alkalmazás Windows platformon történő használatát teszi lehetővé, natív alkalmazás[[6]](#footnote-6) formájában. Így az ajánlott használati szoftverkörnyezet a Microsoft Windows 7, vagy afölött elérhető valamely operációs rendszerek ajánlottak, illetve egyes komponensek működését garantálva a .NET Framework környezet telepítésével. Hardverkörnyezet tekintetében a felhasznált technológiák tekintetében minimum 2 GB operatív tár, illetve valamely 10 évnél nem régebbi processzor használata igényelt, egy alap videókártya típussal.

A továbbiakban a fejlesztői és felhasználói dokumentáció részleteit ismerteti a záródolgozat.

# **1. FEJLESZTŐI DOKUMENTÁCIÓ**

## **1.1 ADATBÁZIS**

A Virtual Receptionist magját az adatbázis adja. A korábbiakban említettem itt tárolódik minden fontos adat, amely az alkalmazások számára nélkülözhetetlenek (pl. szálláshely adatok, szálláshely felhasználói fiókjához tartozó szálláshelyazonosító és jelszó páros, vendégek adatai stb.), valamint ezen keresztül tud közvetetten kommunikálni a web- és asztali alkalmazás.

Az adatbázis mindkét alkalmazásnak szállít szükséges adatot, bizonyos adatait a web- és az asztali manipulálhatja is, de magát az adatbázis sémát egyik sem. Az adatok manipulációját kifejtő hatásköröket a web és asztali alkalmazás alfejezetek részletezik.

Az adatbázis XAMPP alkalmazás segítségével készült el, amely magában foglalja localhost környezetben a phpMyAdmin RDBMS-t. Ez egy egyszerű, grafikus webalkalmazás, amely segítségével SQL[[7]](#footnote-7) utasítások begépelésével vagy grafikusan, menü vezérelt funkciókkal létrehozhatók egyszerű MySQL relációs adatbázisok. Ezen módon történő adatbázis eltárolás egyszerűbbé teszi az alkalmazásfejlesztéshez kifejlesztendő adatbázis használatát a fejlesztési és tesztelési szakaszban, mivel nem kell élő szervert üzemeltetnünk ehhez, hanem az említett localhoston (127.0.0.1[[8]](#footnote-8) IP-címen), helyi hálózaton belül.

Az EK-diagram relációsémává leképzése után én is ebben az RDBMS-ben hoztam létre a Virtual Receptionist adatbázisát, *virtual\_receptionist* néven. Az EK-diagram alapján hoztam létre az adatbázis szerkezeti elemeit is: táblákat, azok mezőit, táblák közötti kapcsolatokat, idegenkulcs megkötéseket, tesztrekordok beszúrását.

Az adatbázis táblák kivétel nélkül magyar nyelvű karakterkódolást, InnoDB tárolómotort tartalmaznak, valamint harmadik normálformáig (3NF) normalizált táblákat tartalmaz, a relációs adatbázisra leginkább jellemző egy-több (1:N) kapcsolatokkal, minél kevesebb redundancia megvalósítása idegenkulcs megkötésekkel az anomáliák és inkonzisztens adatok elkerülése céljából.

Az alábbi táblák alkotják az alkalmazások alapját:

* accomodation (regisztrált szálláshely adatai; cégnév, adószám stb.)
* accomodation\_profile (szálláshely felhasználói fiókjának szálláshelyazonosító-jelszó párosa)
* country (az alkalmazáshoz kapott tábla, amely Föld országainak neveit tartalmazza)
* guest (vendégek adatai; név, lakcím stb.)
* booking (foglalás adatai; érkezés dátuma, távozás dátuma, vendégszám stb.)
* room (szobák adatai; szobaszám, kapacitás stb.)
* billing\_item (szálláshely által beállított saját számlázási tételek; reggeli, idegenforgalmi adó stb.)
* billing\_item\_category (számlázási tételek kategóriáinak adatai, amely az alkalmazáshoz kapott tábla; szállás, ÁFA értéke, mennyiségi egysége stb.)

Ahogyan a *Melléklet, 4. ábrá*ja is mutatja, a legtöbb tábla kapcsolatban áll egymással. Az accomodation és az accomodation\_profile táblák között egy-egy (1:1) kapcsolat áll fent, adott felhasználói fiók szálláshelyazonosító-jelszó párosát csakis egyetlen egy szálláshelyhez köthetjük (egy szálláshelynek egy regisztrációja lehet elven).

A *country* tábla egy alkalmazáshoz kapott tábla (nincs olyan funkció, amely a tábla adatait manipulálni tudná), amelyben országnevek tárolódnak, ezt alkalmazza az asztali alkalmazás vendég felvitel esetén országnév kiválasztással, vagy számlázásnál a számlázási ország űrlapelem kiválasztásával.

A *guest* tábla tárolja a vendégek személyes adatait, idegenkulcsként tárolódik benne az ország tábla egy rekordjának kulcsa, amely a vendég országát hivatott tárolni.

A *room* táblában a szálláshely által beállított szobák tárolódnak, szobanév, szobaszám és maximális férőhely adatok tárolásra ad lehetőséget, valamint egy kapcsolómező, amely a szoba számláhatósági kategóriáját adhatja meg a billing\_item táblából, amely a szoba adott árát tartalmazza.

A *billing\_item* tábla szintén kapcsolatban áll egy másik táblával, a *billing\_item\_category*-val, amely már egy adott kategória ÁFA tartalmát adj meg vagy mennyiségi egységének elnevezését a számlázás folyamatában (nem módosítható tábla).

A *booking* tábla az a tábla, amely a több-több (M:N) kapcsolatot relációsémában leképző kapcsolótábla szerepét tölti be. Ez a tábla idegenkulcs megkötést tartalmaz vendég és szoba azonosítóra, hogy hatékonyan tudjuk eltárolni mikor melyik vendég érkezik, hány fővel melyik szobába érkezési és távozási dátummal kiegészülve.

Ezenkívül ahol szükséges volt, egyedi indexek vannak egyes mezőkre állítva (pl. guest táblában telefonszám vagy e-mail), amely meggátolja, hogy ugyanazon adat többször megismétlődhessen több rekordban (egy vendégnek egyedi e-mail címe van, más nem birtokolhatja más e-mail címét).

Saját konvencióm szerint minden egyes tábla automatikusan növekvő, egyedi értékű „ID” kulcsmezővel rendelkezik, hogy minden egyed előfordulást maradéktalanul be tudjunk azonosítani.

## **1.2 WEBALKALMZÁS**

A webalkalmazás a Virtual Receptionist konfigurációs felülete. Ezen a felületen állíthatók be a szálláshely „specialitásai”.

Bootstrap 4 reszponzív felülettel rendelkezik és ő is közvetlenül csatlakozik az adatbázishoz (ez teszi lehetővé asztali és webalkalmazás között a kommunikációt).

PHP nyelven íródott, nincs architektúrába rendezve és natív PHP nyelven íródott keretrendszer nélkül, mivel kicsi az alkalmazás és alapvető funkciókat lát el (PHP nyelv erre tökéletes), és programtervezési minták sincsenek benne alkalmazva.

Számlázási tételek és szobamenedzsment feladatokat lát el, amellyel az asztali alkalmazás képes adminisztrálni.

alkalmazott library-k, API-k:

AJAX, Bootstrap 4 CSS és JS, jQuery

NINCS FRAMEWORK, ARCHITEKTÚRA,   
  
ADATBÁZIS CSATLAKOZÁS WEBEN

connect.php kapcsolatleíró hozza létre, globálisan felhasználható a projekt bármely állományában.

## **1.3 ASZTALI ALKALMAZÁS**

Az asztali alkalmazás Visual Studio 2017 Community Edition IDE[[9]](#footnote-9)-ben került kifejlesztésre, *WinForms* beépített grafikus osztálykönyvtár felhasználásával .NET 4.6.1 keretrendszer verziójában, C# programozási nyelven.

Az asztali alkalmazás a Virtual Receptionist adminisztrációs felületét adja. A korábban, webalkalmazásban is használható szálláshelyazonosító-jelszó párossal lehet belépni az alkalmazásba, amely alapján adatbázisból beazonosítja az adott szálláshelyet a rendszerben.

A korábban, webalkalmazásban bekonfigurált adatokkal tud az alkalmazás dolgozni, amely korábban is említett három fő modulból épül fel:

* Foglalási napló
* Vendégadatbázis
* Számlázó

A foglalási napló modulban van lehetőség foglalások rögzítésére, módosítására vagy törlésére. Minden foglalás felvitel esetén rögzítésre kerül az adatbázis *booking* kapcsolótáblájában a vendég minden adata, így azok későbbiekben újra felhasználhatóak, megspórolva az újbóli vendégadat űrlapmezők kitöltését.

A vendégadatbázis modulban a meglévő vendégek adatait tarthatjuk karban egy adattábla segítségével, amelyen szintén végezhetők adatmanipulációs műveletek a *guest* adatbázistáblájában.

A számlázó modulban a korábban felvitt, új foglalásokat számlázhatjuk ki. Minden felvitt foglalás alapértelmezetten nem számlázottként kerül be az adatbázisba, amelyet egy logikai típusú „Paid” mező tárol, 0 értékkel inicializálva. A számlázóban a webalkalmazásban beállított számlázási tételek közül válogathatunk és vihetjük fel egy adott foglalásra, amelyek megjelennek, mint nem számlázott foglalás. A számlanyomtatás funkcióval a felvitt rekordokból kész számlát kapunk, miközben a „Paid” mező az adatbázis adott foglalás rekordjában 1-es értékre változik, így később nem jelenik meg a számlázóban az adott foglalás nem számlázottként.

Ezen robosztus alkalmazás komplexitása rendszertervezés szintjén már nem engedhette meg azt a luxust, hogy nem alkalmazunk benne többrétegű architektúrát, ne alkalmazzunk programtervezési mintákat, illetve objektum-orientált programozási paradigmát. Itt szükségessé válik a későbbi nagyobb továbbfejlesztések lehetőségének megteremtése céljából izolálni a grafikus felhasználói felületet, az alkalmazás interakciós logikáját, üzleti logikáját és perzisztens programszintű adattárolási metódusait. Ezekből kifolyólag az asztali alkalmazást többrétegű architektúrában terveztem és valósítottam meg. Ennek alapját a szoftverfejlesztési trendekben közismert *MVC*[[10]](#footnote-10) architektúra ihlette. Maga az alkalmazás nem tartja az MVC egyedi szabályait, inkább hagyományos többrétegű architektúrában készült – minden réteg hierarchikus és csak a mellette lévő réteggel kommunikálhat közvetlenül –, alkotói szabadságot adva a rétegek saját működésbeli definíciójának meghatározását, azonban felhasznál az MVC komponenseinek elnevezéséből.

Maga a projekt mappaszerkezete és névterek elnevezési konvenciója is így lett kialakítva. A rétegek rendszerterveit a *Melléklet 5., 6. és 7. ábrá*ja tartalmazza. A továbbiakban rétegről rétegre fejtem ki az alkalmazás működését és funkcióit.

### **1.3.1 REPOSITORY (ADATTÁR)**

Az alkalmazás legfelsőbb rétege a Repository, vagy adattár.

Ezen réteg tartalmazza az alkalmazás üzleti logikáit és perzisztens adattároló rétegként szolgál adatbázis és alkalmazás között az általam preferált adatszerkezetek (nyelvi szinten beépített generikus listák) formájában, több kisebb származtatott, specializált adattár osztályban. Ezenkívül több más komponenst is magába foglal ez a réteg:

* Egy saját fejlesztésű adatbázis csatlakozást segítő adat lekérdezést és adatmanipulációs metódusokat definiáló API-t, az Oracle által .NET környezetre implementált MySQLConnector ADO.NET osztálykönyvtár legfontosabb metódusait felhasználva (Database.cs, Singleton programtervezési mintában kifejlesztve)
* Az adatbázis típusának csatlakozására vonatkozó saját, specializált kivétel (InvalidConnectionTypeException.cs)
* Az adatbázis tábláit leképző modell osztályokat, amelyek az adatbázis egyedek (táblák) object-relational mapping[[11]](#footnote-11) objektumai, amelyekből adatbázisból lekérdezéskor példányok készülnek, hogy azokat a lista adatszerkezetek eltárolhassák (pl. Guest, Booking, Room stb.)
* Valamint tartalmaz egy DAO (Data-Access Object) programtervezési mintában megírt generikus interfészt (IGenericDAO.cs), amelyben deklarálva vannak az alapvető adatmanipulációs metódusok (törlés, módosítás, beszúrás).

Mint a korábbiakban említettem, egy saját kifejlesztett adatbázis elérést tartalmazó osztály végzi a Repository réteg adatbázissal történő kommunikációt.

Ezen osztály olyan metódusokat tartalmaz, amelyek megnyitják majd lezárják az adatbázis csatlakozást (erőforrás-felszabadítás céljából), valamint a CRUD[[12]](#footnote-12) metódusok implementációját a megfelelő MySQLConnector metódusok alkalmazásával, hogy a MySQL adatbázishoz eljussanak.

Az első adatbázis kapcsolat megteremtése az autentikáció pillanatában indul el és ekkor állítja be a Database osztály az adatbázis elérésnek útvonalát, amely futás idejű konfigurálást tesz lehetővé. Egy SetConnection metódus teszi ezt lehetővé, hogy iskolai vagy otthoni adatbázis útvonalat akarunk-e elérni, a kapcsolódás típusát paraméterátadáson keresztül kapja a metódus a grafikus felületről.

Otthoni és iskolai típus esetén egy nyelvi szinten beépített statikus osztály, a ConfigurationManager olvassa ki az előzőleg App.config XML állományba statikusan megadott adatbázis elérési útvonalat. A ConfigurationManager egy metódusa az XML-ben megadott “local” és “remote” attribútumokból választhat, hogy a helyi vagy távoli kiszolgálóhoz csatlakozik. Ezt a SetConnection elnevezésű metódus végzi feltétel vizsgálattal, attól függően, hogy ‘iskolai’ vagy ‘otthoni’ típust kap karakterláncként paraméterben. Ha nem kap semmit, akkor az említett saját kivétel képződik.

A fentebb említett programtervezési minták kifejtéséről külön alfejezetben értekezem.

### **1.3.2 CONTROLLER (VEZÉRLŐ)**

Az alkalmazás köztes rétege a vezérlő. Ez a réteg teremti meg az üzleti logika megjelenítését a legfelső réteg, a Repository és a legalsóbb réteg, a View (nézet) között. Így a View réteg csak a Controlleren keresztül jeleníthet meg adatot az adattárból, vagy kaphat vissza üzleti logika által számított adatot (pl. számlázási tétel által számított kedvezmény összegét). A vezérlő felelőssége a megfelelő nézetre konvertálás is. Ezalatt arra gondolok, ha szükségünk van egy legördülő listában az országok neveinek megjelenítése, akkor a vezérlő elkéri az adattárból az Country objektumokat tartalmazó listát és egy egyszerű szűrés által egy karakterláncokat tartalmazó listával tér vissza a View számára.

A rétegben több, modulokra osztott vezérlő található, melyek egy ős Controller osztályból öröklődnek, amelyben csak a Repository réteg elérésének inicializációja található; illetőleg minden specializált vezérlő a neki szükséges gyermek adattár osztály egy példányán érhet el üzleti logikát és adatokat.

A vezérlő szerepkörei tételesen:

* valamely eseményre történő mechanizmus elindítása
* felhasználó által (input) adat validációja a nézet valamely grafikus komponenséből
* itt helyezkedik el minden validátor metódus
* saját, specializált kivételek
* általánosan, több modul által felhasznált metódusok továbbítása a View rétegbe, megjelenítés céljából a Repositoryból
* megfelelő nézet kialakítása (konvertálás)

### **1.3.3. VIEW (NÉZET)**

Az alkalmazás legalsó rétege a nézet, azaz a grafikus felhasználói felület (GUI). A felhasználó ezeken keresztül tud kapcsolatot teremteni az alkalmazással.

Fontos szerepet játszott a megtervezésük során az ergonómia, mivel a felhasználó ezen keresztül látja az alkalmazás működését, ez teremt vele kapcsolatot. Az alkalmazás minden ablaka kerüli a kirívó színeket, egységes kinézetet biztosít, jól látható pontokon helyezkednek el a gombok, szövegmezők, táblázatok és egyéb GUI komponensek.

Maga a nézetek, úgynevezett *code-behind[[13]](#footnote-13)* részében maga a C# programozási nyelv színtisztán objektum-orientált mivolta okán egy osztály található, amely öröklődik a WinForms keretrendszer ős Form osztályából, amely egy ablakot hoz létre ezáltal. Ebből adódóan egy gyermekosztályként fogható fel minden ablak, azonkívül töredék osztályként, mivel *partial* módosító kulcsszóval van ellátva (ezzel lehetővé válik egy logikailag összetartozó osztály tárolása több, szuverén fájlban). Az ablak konstruktorában található egy InitializeComponent elnevezésű metódus, ez az a metódus, amely az ablakra ráhúzott grafikus vezérlőket létrehozza és megjeleníti, és ez logikailag ugyanezen osztályban létező kód, azonban a jól olvashatóság miatt egy másik fájlban találhatók meg.

Mivel többrétegű alkalmazást fejlesztettem, így a code-behind fájlokban csak megjelenítési kódot írtam, de logikát nem; ilyen kód lehet egy törlés esemény hatására egy táblázat egy rekordjának eltüntetése. A logikai kód már a következő rétegben, a Controllerben, azaz a vezérlőben helyezkedik el. Valamely felhasználói interakció (pl. gombnyomás) azonnal abba a rétegbe kerül, hogy ott valamely logika, vagy adathalmaz kerüljön visszatérésre, hogy azt a nézet megjeleníthesse. Így a nézetek eseményeket leíró metódusaiban a megjelenítési logikán kívül egy-egy megfelelő vezérlőmetódus hívás történhet (ez nagyban elősegíti a kód újra felhasználást).

## **1.4 ALKALMAZOTT PROGRAMTERVEZÉSI MINTÁK**

Korábban már említettem, hogy felhasználtam úgynevezett “design pattern”-eket. Kettő ilyen ismert mintát implementáltam:

* Singleton (egyke programtervezési minta),
* DAO (Data Access Object, adathozzáférési objektum programtervezési minta)

Az adatbázis kapcsolódást és adatbázisokban adat lekérdezést és adatmanipulációs metódusokat szolgáló osztályom, a *Database* és a szálláshely egyedet leíró modell osztály, az *Accomodation* osztályom ezen tervezési minta révén fejlesztettem ki.

A Singleton minta lényege, hogy az egész programfutás során egyetlen egy darab példány létezhet egy adott osztályból. Ennek megoldására a statikus változók jelentenek megoldást, amely a futás idején mindvégig élnek a memóriában és nem vesztik el értéküket. Az objektum-orientált nyelvekben a globális változók létezését próbálja megteremteni a fejlesztőnek egy statikus változó.

Ahhoz, hogy egy darab objektum jöhessen létre, egy az osztály típusával megegyező típusú statikus változót kell a mezők között deklarálni, az osztály konstruktorát *private* láthatósági módosítóval kell ellátni (kívülről ne lehessen példányosítani *n* számú objektumot) és a példányosított objektummal egy statikus getter metódus fogja visszatérni.

A getterben egy feltétel vizsgálat ellenőrzi, hogy a statikus mező inicializálatlan, null értékű-e még. Első alkalommal a getter minden esetben visszatér a privát elérésű konstruktor hívás utáni példányosításával a példánnyal (belülről jön létre), amely a programfutás végéig élő objektum lesz.

Amennyiben később is szükség lenne példányra az osztályból, hogy igénybe vegyük egy metódusát, úgy a továbbiakban is a getterhez kell fordulni, hogy a már élő objektummal visszatérjen, azáltal, hogy a feltétel vizsgálat kiértékelése során már nem lép be az igaz ágra.

Ugyanezen mintával építettem fel a szálláshely osztályt, amely autentikációs feladatokat végez (az alkalmazásban egy szálláshely él, így egy példány lehet belőle elv).

A másik programtervezési minta a DAO. Az adathozzáférési objektum lényege, hogy egy absztrakt interfészt biztosít adatbázis adat lekérdezés vagy adatmanipulációs mechanizmusokhoz.

Általános elvárás, hogy minden modell osztályra legyen DAO interfész, hogy azt a modell osztályra örököltetve implementálható legyen. Jelen projektben nem a modell osztályok metódusai lesznek a CRUD műveletek implementációjának helye, hanem a specializált adattárak, mint üzleti logika metódusok tára. Ahhoz, hogy ne kelljen minden modellre megírni az interfészt, így egy generikus DAO-t dolgoztam ki.

Az általánosságát az adja, hogy bármely típusra működhetnek az implementációk. Ezen interfésznek generikus paraméterének szüksége van egy osztálytípusra, amely paraméterrel létrehozhatja metódusait, a típus a megadott modell típusa lesz (pl. IGenericDAO<Guest>).

Az interfész blokk implementációjában pedig helyet kapnak azon SQL utasításokat, amelyek eljuttatják az új, törlendő vagy módosítandó objektumot a fizikai adatbázisba.

## **1.5 EGYSÉGTESZTEK**

A fejlesztés során, mint általában készülnek apró, bizonyos metódusok viselkedését vizsgáló, izolált egységtesztek. Ez jelen projekt esetében sem történt másképp.

Az egységtesztek általában annyira apró területet fednek le egy robosztus projekt fejlesztése során, hogy sokszor a fejlesztők elhalasztják ezen tesztmetódusok kifejlesztését, főleg, ha nem teszt-vezérelt fejlesztési metodikában dolgoznak ki egy-egy adott problémakört és rögtön integrációs teszt, majd interfész tesztelésre továbbadjanak egy rossz eredményt produkáló kódrészletet. Ezt kerülendő nem árt írni egységteszteket, hogy valóban megfelelően reagál bizonyos felhasználói ingerekre egy adott metódus a továbbiakban és ne futás idejű tesztelés közben kelljen kitapasztalni a szemantikus működést.

Az alkalmazás fejlesztése során éltem a tanulmányaim során megszerzett egységtesztelési lehetőségekkel és megszerzett tudással és készítettem is teszteket a létező legtöbb tesztesetet szimulálva, hogy minden hiba kiszűrhető lehessen vele.

Mivel egy többrétegű alkalmazás legnagyobb előnye, hogy az üzleti logika, vagy bármely más logika egyébként is izoláltan helyezkedik el, rendkívül elszeparált környezetben a grafikus felület és megjelenítési kódoktól, így könnyen egységtesztelhető is szimulált tesztadatokkal.

Két tesztosztály készítettem, egyikben a *TDD*[[14]](#footnote-14) elve szerint fejlesztettem ki egy a számlázó modul által használt függvényt: egy egységárból kedvezményes árat számoló metódust; a másik tesztosztályban egy már korábban általam kifejlesztésre került elemi algoritmusokból álló, statikus függvényeket tartalmazó statikus függvénygyűjtemény (Controller réteg InputValidation.cs) metódusainak integrálása után felhasználói adatbevitelt ellenőrző validációs metódusok tesztelése történt meg. A tesztesetek között inkább a legkritikusabb inputokra történő érzékenység tesztelése valósult meg (ilyen lehet egy név, amely szám nem lehet, vagy megfelelő formátumú e-mail cím ellenőrzése).

Ezen validációs metódusok kifejlesztése tökéletesen sikerült a TDD elv szigorú követésével. Minden esetben egy bukott tesztet készítettem, mely magával vonta az üzemi kód refaktorálását. A validációs metódusok minden egyes inputra elkészültek (név, születési dátum formátum, e-mail cím stb.) és mindegyik a saját kivételét dobta a programkódban, hogy egy-egy rossz adatbevitel esetén a lehető legpontosabb hibaüzenetet kapja a felhasználó, amikor az alkalmazást használja. Maguk a validációs metódusok a legtöbb alap függvényt (pl. IsEmpty metódus, amely egy input üres tényét állapítja meg) felhasználva, feltétel vizsgálatban, automatizáltan lefutásra kerültek.

Egyik ilyen érdekes teszteset, amikor egy e-mail cím helyes formátumát ellenőriztem *Regex*[[15]](#footnote-15) segítségével, vagy egy olyan metódus, amely tesztpéldányokkal ellenőrzi az adattár egy listájában lévő foglalási adatok közül, hogy az érkezési dátum a foglalási naplóban egy naptár GUI vezérlő segítségével nem távolabbi dátum volt-e, mint a távozás dátuma; vagy olyan tesztpéldányokat alkalmazó metódus, ahol azt ellenőrzi a validációs metódus, hogy adott napon nincs-e már egy adott szoba kiadva, meggátolva a rendszerben a túlfoglalás lehetőségét.

Ennek a gyakorlati megvalósítását úgy kell elképzelni, hogy beérkezik a függvény paraméterébe a szobaszám és az érkezése dátuma a grafikus felületről, ahol egy tesztobjektum minden esetben

Az egységtesztelés befejezése után nem maradt olyan tesztfüggvény, amely megbukott volna a további teszteseteken, összesen 37 teszteset készült a legkülönfélébb validációs és üzleti logika metódusok közül válogatva.

# **2. FELHASZNÁLÓI DOKUMENTÁCIÓ**

## **2.1 TELEPÍTÉSI ÚTMUTATÓ**

Hhhjhzjzjbnfbkfnbkfnbkfb nfkbnfkbnfbk fn b kfbnfkbnfkbf nbkfnbkfnbk fbnfkbgf..

JFjfeofjefoefoejfefofjefoefjeofjfeof ejfeofjeoe jf eofjeejfo ejf oej of fejo jefoe fjo efjo efojo jefo jef joef eoj ojefoj efjoe foj efoje fj oef ejofjo feoj f jeof.

# **ÖSSZEGZÉS**

Záródolgozatom megvalósításának végéhez érve rengeteg konklúziót tudok levonni. Elsősorban a kezdeti terveket átnézve viszonylag sok dolgot sikerült a megadott időn belül megvalósítanom, amit nem gondoltam volna, hogy sikerül is.

Itt leginkább az asztali alkalmazásra gondolok, amelynél rengeteg elvárást támasztottam magammal szemben; de az igen nagyra sikerült és rengeteg apró komponens tartja össze ezt az építményt mégis hatékonyan továbbfejleszthető a későbbiekben. A többrétegű architektúra és az objektum-orientált programozási paradigma alkalmazása rengeteg gondot levett a vállamról, mondhatni játék volt a fejlesztés. Teljességében sikerült elérnem a mikro vállalatirányítási rendszer alapszintű kifejlesztését, amely tesztkörnyezetben a megfelelő, elvárt eredményeket és szemantikus működést váltja ki.

A webalkalmazáson a későbbiekben rengeteg jelenleg kiaknázatlan lehetőséget lehet megvalósítani, elsősorban objektum-orientált felépítésre megtervezni és itt is alkalmazni többrétegű architektúrát, lehetőleg az MVC architektúrát, hogy rugalmasan lehessen bele további funkciókat implementálni és kövesse az elvárt szakmai trendeket.

A legnagyobb kihívást a koncepcionális kérdések okozták. A túl nagy absztrakciós ambíció itt-ott megállította napokra a fejlesztést és komplett újra tervezést igényelt egy-két apró részen. Leginkább az, hogy milyen adatot lenne még érdemes eltárolni egy valós, napi szintű éles munkafolyamat általi felhasználás során és mit nem – külön céges vendégtáblázat, amely cég adatokat hozzáköthetők lehetnének a személyes vendégadatokhoz, hogy tisztább képet kapjunk ki céges és ki magán vendég –, anélkül, hogy a teljes adatbázis séma kapcsolatait meg kelljen változtatni, vagy nagyobb, olykor csapongóbb refaktort végezzek rajta ismételten. Kisebb, nem égbekiáltóan szükséges funkciók létrehozása elmaradt, azonban van létfontosságú elem is bennük, amelyek nem készültek el, de későbbiekben mindenképpen beépítésre kerülnek.

A továbbiakban tételes listát készítettem a konkrét továbbfejlesztési lehetőségekről minden komponense vetítve:

Webalkalmazás:

* jelszavak hash-elt eltárolása adatbázisban (biztonsági okok)
* regisztrációs űrlap elkészítése, mivel jelenleg előregisztrált a szálláshely a rendszerben
* jelszócsere lehetőség kifejlesztése az adott szálláshely számára
* jQuery alapú intelligens adattábla megvalósítása Ajax technológiával a DatTables API-val (minőségi és rövidebb kód megvalósítása)
* A minőségi és rövidebb, átláthatóbb kód elvén tovább lépve az alkalmazás áttervezése MVC architektúrára, valamely PHP framework alkalmazása (SlimPHP mikro alkalmazásokhoz a kifejezetteb ajánlott technológia), és ezen technológiák alkalmazása megköveteli az objektum-orientált szerveroldali webfejlesztést teljeskörű elsajátítását
* az olykor puritán felhasználói felület javítása gazdagabb tartalommal, viszont ugyanolyan egységes megjelenést kölcsönözve

Asztali alkalmazás:

* hashelt jelszó dekódolása autentikáció esetén az adatbázisból
* saját, helyi adatok eltárolása a program részeként (SQLite vagy Mozilla Firebird adatbázis valamely alkalmazásával), mivel helyi specialitásokat felesleges egy központi, csak autentikációs feladatokat és előre adott táblák adatainak szolgáltatásán kívül
* Foglalási napló modulban vendég azonosítóval történő foglalás módosítás, hogy ne adatintegritás megsérülését azonos nevű vendégek esetén
* Végszámla generálása a számlázó modulban és előzetes előszámla megjelenítése nyomtatás előtt, vagy az alkalmazott C# .NET komponenskönyvtár egy vezérlőjének segítségével direkt nyomtató portra vezetett fizikai nyomtatás megvalósítása
* Vendégadat importálás a foglalási naplóba, a felesleges űrlap kitöltés idejének megspórolása a munkafolyamatból
* A komplett alkalmazás integrálása natív alkalmazásból platformfüggetlen alkalmazásra (pl. Java nyelv alkalmazása), vagy .NET platformon korszerű grafikus keretrendszerrel a felhasználói felület újra tervezését (pl. WPF keretrendszerre)

Adatbázis:

* Relációséma átalakítása, hogy redundancia nélkül lehessen céges vendég adatokat eltárolni személyes vendégadatok mellett és ne csak a számlázóban lehessen cégnevet megadni, hanem automatikus működésre bírjuk itt is a rendszert
* Minden helyi adat migrálása és később helyi adatbázisban történő felvétele (SQLite stb.)
* Korszerű és biztonságosabb felhasználói profiladatok tárolása

Mindent érintően vagy webalkalmazásba vagy asztali alkalmazásba az összes, jelenleg osztott funkció integrálása.

Összességében úgy gondolom, a későbbi tudások megszerzésével egyre jobbra bővíthető lesz ez a rendszer és megállja magát a mai szoftverfejlesztés világában.

# **MELLÉKLET**

A záródolgozat mellékletét képzik ezenfelül egy CD lemez az alábbi tartalommal:

1. A záródolgozat szövege egy-egy példányban .docx és .pdf kiterjesztésű állományként
2. A gyakorlati munkák forráskódjai
3. A gyakorlati munkák futtatható állapotú állományai

1. Unified Modelling Language: általános célú modellező nyelv [↑](#footnote-ref-1)
2. Egyed-kapcsolat diagram: adatbázisok logikai megtervezését segítő grafikus leíró eszköz [↑](#footnote-ref-2)
3. Relational Database Management System: relációs adatbázis-kezelő rendszer [↑](#footnote-ref-3)
4. World Wide Web Consortsium: világhálóra történő szabad szoftver ajánlásokat készítő szervezet [↑](#footnote-ref-4)
5. Platformfüggetlenség: bármely operációs rendszerrel rendelkező számítógépen, ugyanolyan formában futtatható alkalmazás [↑](#footnote-ref-5)
6. Natív alkalmazás: egy platform saját nyelvében íródott, és képes annak a platformnak minden lehetőségét kihasználni [↑](#footnote-ref-6)
7. Structured Query Language: strukturált lekérdezőnyelv, a relációs adatbázisok által használt 4. generációs nyelv [↑](#footnote-ref-7)
8. Localhost: hálózati kártya visszacsatolási címe, helyi hálózaton történő alkalmazásfejlesztés hálózati csatlakozás meglétének tesztelése [↑](#footnote-ref-8)
9. Integrated Developement Enviroment: integrált fejlesztőkörnyezet, valamely alkalmazásfejlesztés során alkalmazott felület a kódolás folyamatának megteremtése érdekében [↑](#footnote-ref-9)
10. Model-View-Controller: Modell-nézet-vezérlő architektúra, leginkább webalkalmazások esetében, ahol jól elkülönített rétegben kerül implementációra a grafikus felhasználói felület, az üzleti logika, interakciós logika, adattárolás programszinten [↑](#footnote-ref-10)
11. Objektum-relációs leképzés: adattáblából konkrét forráskódszintű egyedek, afféle virtuális objektum adatbázisként írható körül [↑](#footnote-ref-11)
12. Négy alapvető adatbázis műveletre alkalmazott mozaikszó: create, read, update, delete [↑](#footnote-ref-12)
13. Háttérkód: valamely grafikus ablak mögött elhelyezkedő fájl, amely GUI komponens inicializáló kódot tartalmaz [↑](#footnote-ref-13)
14. Test-Driven Developement: Teszt-vezérelt fejlesztés [↑](#footnote-ref-14)
15. Regular Expression: Egy adott minta alapján történő bemenet összehasonlítás [↑](#footnote-ref-15)