Debreceni Egyetem
Informatika Kar

Webalkalmazás fejlesztés .NET környezetben

Témavezető:
Kollár Lajos
egyetemi tanársegéd

Készítette:
Veres István
programtervező
matematikus

Debrecen
2009

Tartalom

1. Bevezetés

1.1 Célkitűzés

Diplomamunkám célja egy konkrét vállalati környezetbe illeszthető rendszeren keresztül bemutatni a webes alkalmazásfejlesztés fortélyait Microsoft .NET 3.5 SP1 keretrendszerben. Erre a célra egy jegy alapú incidens menedzsment rendszer (ún. Issue Tracker) tervezését és fejlesztését választottam, mivel egy ilyen jellegű alkalmazás fejlesztéséhez, illetve problémakör megoldásához a legszélesebb ma használatos webes .NET technológiák halmazát tudom alkalmazni, és ezen keresztül bemutatni.

A kivitelezéshez összegyűjtöttem a rendszer tervezése és fejlesztése során legjobban illeszkedő tervezési mintákat (design patterneket) melyek biztosítják a termék robusztusságát, skálázhatóságát és egyes komponensek cserélhetőségét.

1.2 Az issue tracking rendszerek

A jegyalapú ügyviteli rendszer nem más, mint egy olyan szoftver csomag mely a felhasználó szervezet igényei szerint segédkezik az ügyfél támogatási ügyvitel kezelésében, fenntartásában. Általában a rendszert az adott szervezet telefonos ügyféltámogató szolgálata használja a felmerülő belső és külső ügyfél panaszok, problémák felvételére, kezelésére és megoldására. A legtöbb esetben a csomag tartalmaz egy tudásbázis kezelő modult is, melyben a gyakran felmerülő problémák elhárításának, megoldásának elősegítésére tartalmaz részletes információkat, illetve az ügyvitel szempontjából hasznos egyéb adatokat.

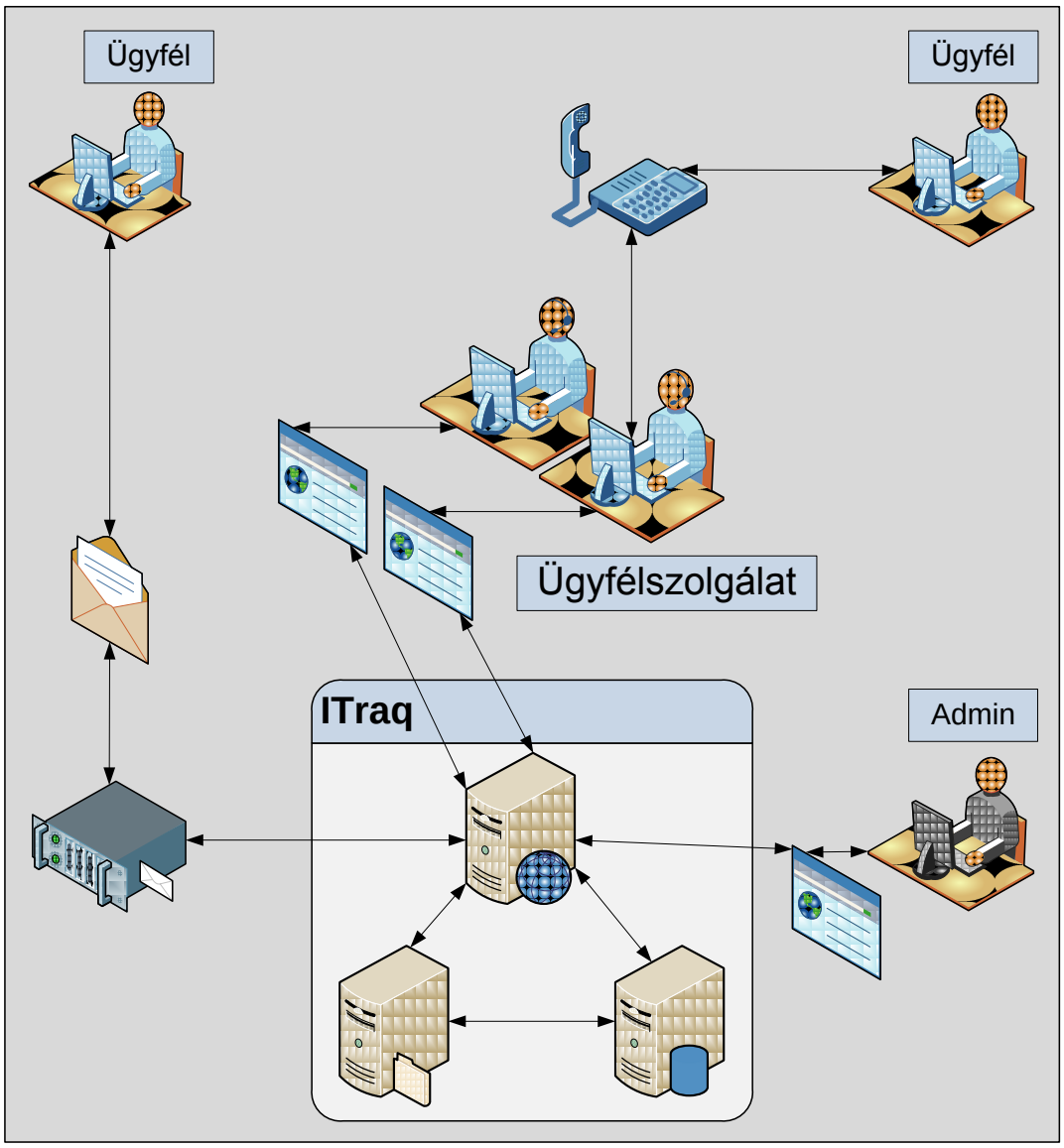
A jegy nem más, mint egy olyan virtuális akta mely tartalmazza magát a felmerült probléma leírását, a probléma szempontjából releváns csatolt állományokat (pl. screenshotok, logfájlok stb.), illetve a probléma feldolgozása, megoldása során az ügyfélszolgálat és az ügyfél, vagy az érdekelt külső szolgáltató és az ügyfélszolgálat között lezajlott kommunikációt szöveges formában.

A jegy kifejezés a telefonos ügyfélszolgálatok korának hajnalából származik, mikor az ügyfélszolgálat dolgozói a telefonon elhangzott problémákat egy kis cetlire jegyezték le az ügyfél adataival együtt majd egy nagy parafa táblára tűzték őket sorba. Ezeket a cetliket nevezték jegyeknek.

2. A tervezés

2.1 Az ITraq

Ebben a fejezetben bővebb betekintést adok a diplomamunka keretein belül elkészült rendszer architektúrájáról, a rendszer folyamatairól, a rendszert alkalmazó környezetről és a kapcsolódó fogalmak definíciójáról. Az 1. ábra betekintés ad az ITraq rendszer illeszkedésére az ügyfélszolgálati folyamatba.



1. ábra. Szereplők és folyamok.

2.1.1 Fogalomszótár

- Probléma: Olyan szituáció, előállt helyzet, melynek megoldásához az ügyfél az ügyfélszolgálathoz folyamodik segítségért.

- Ügyfél: Olyan személy, aki az ügyfélszolgálathoz problémával fordul telefonon, vagy emailen.

- Ágens (agent): Olyan személy, aki az ügyfélszolgálaton dolgozik és az ügyfelekkel és az ügyfelek jegyeivel foglalkozik.

- Csoport: Ágensek egy jól meghatározott halmaza, minden csoportnak van neve, lehetnek a csoportnak jogosultságai melyeket a csoport minden tagja birtokol.

- Jegy: Olyan szöveges és egyéb típusú csatolt adatok jól meghatározott halmaza, melyek a probléma felvétele és megoldási folyamata során álltak elő. Van állapota, tulajdonosa, létrehozója és eredete. Egy jegy aktív amíg az állapota nem lezárt, vagy törölt.

- Jegy eredete: Az az ügyfél, külső eredet esetén annak email címe, belső eredet esetén a felhasználói azonosítója, akinek a problémáját tartalmazza a jegy.

- A jegy állapota: Lehet új, megnyitott, lezárt, elfogadott, elutasított, törölt.

- A jegy tulajdonosa: Az az ágens, aki a jegy megoldásán foglalatoskodik.

- A jegy létrehozója: Ez lehet ágens, vagy ügyfél attól függően, hogy a problémát az ügyfél telefonon, vagy pedig emailen juttatta el az ügyfélszolgálathoz.

- A jegy megfigyelője: Olyan belső felhasználó, akinek olvasási joga van a jegy összes adatára nézvést.

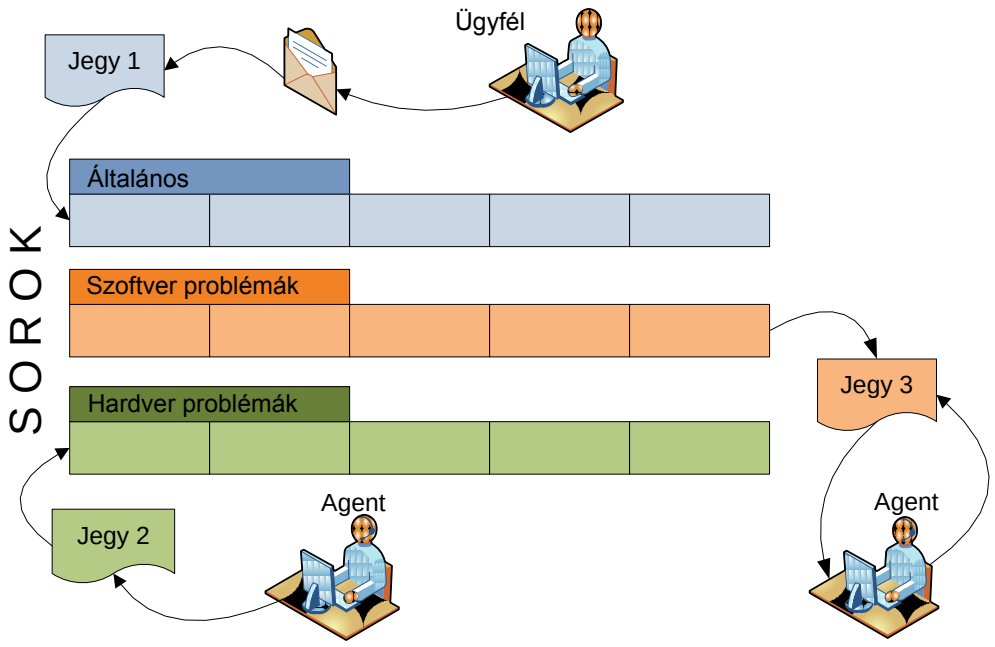
- Jegyasszisztens: Olyan belső felhasználó, vagy külső kontakt (email cím) aki láthatja a jegy minden tartalmát, a törzsszöveget, a csatolt állományokat, megjegyzéseket, és válaszokat, és aki megjegyzéseket tud hozzáadni a jegyhez.

- Megjegyzés: Olyan üzenet melyet az asszisztens(ek), és a tulajdonos tud hozzáadni a jegyhez, és csak a tulajdonos, az asszisztes(ek), és a megfigyelő(k) láthatnak.

- Válasz: Olyan üzenet melyet a tulajdonos, és az eredetként tárolt ügyfélkontakton keresztül az ügyfél tud (emailben, vagy belső eredet esetén a felhasználói felületen) hozzáadni a jegyhez, és minden érintett láthatja.

- Megoldás: Olyan különleges válasz mely tartalmaz két a jegyhez tartozó speciális linket, amelyeket meglátogatva az eredet elfogadhatja, vagy elutasíthatja a megoldást.

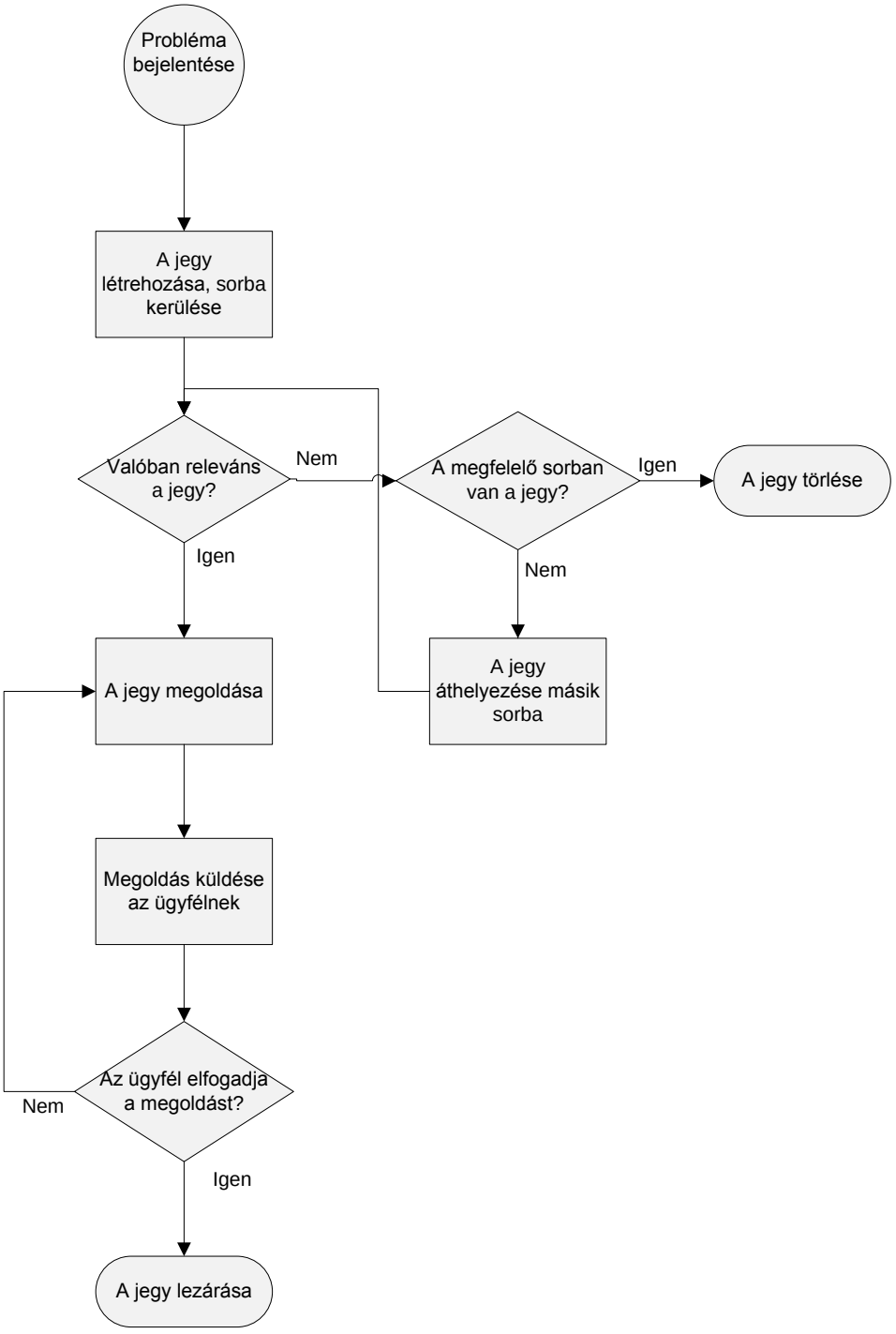
- Sor: A jegyek a rendszerbe kerülésükkor valamilyen rendező elv alapján (a probléma jellege, sürgőssége, az ügyfél kiléte) különböző várakozási sorokba kerülnek, várva hogy ágenshez rendelődjenek.



2. ábra. A sorok szerepe.

2.1.2 A jegy életciklusa

A következő folyamatábra illusztrálja egy jegy életciklusát a rendszerben.



3. ábra. A jegy életciklusa.

2.1.3 A jegy megoldási folyamata

A jegy létrehozása után a következő adatokat tartalmazza:

- A jegy létrehozóját;
- A jegy eredetét;
- A jegy prioritását;
- A törzsszöveget, amely a problémát írja le;
- Adott esetben csatolt állományokat.

A jegy megoldása a rendszer szempontjából a következő módon történik.

1. Egy ágens megnyitja az adott jegyet.
2. megjegyzések és válaszok, esetleg csatolt állományok hozzáadása a jegyhez, az ágens, az asszitens(ek), és az ügyfél által.
3. Megoldás küldése az ügyfélnek.
4. Az ügyfél elfogadja a megoldást és irány az 5-ös pont, vagy elutasítja, ekkor vissza a 2-es pontra.
5. A jegy lezárása.

A jegyhez a külső eredet személy a megoldás során is adhat további csatolt állományokat, mégpedig úgy, hogy egy válasz emailhez csatolja a hozzáadni kívánt állományt és elküldi a rendszernek.

Természetesen a rendszer a lezárás, vagy a logikai törlés után is tárolja a jegyeket tovább, mivel minden jegyekkel kapcsolatos eseményt historikusan tárol.

2.1.4 A jogosultságkezelés

A rendszer jogosultságkezelése két egymással ortogonális osztályozásra bontható. A jogosultságok lehetnek:

- Csoport szintűek;
- Felhasználó szintűek;

Ezek a jogok vonatkozhatnak:

- Sorra;
- Jegyre;
- Csoportra (mint egységre);
- Felhasználóra;
- A rendszerre.

A csoport szintű jogosultságokat a csoportba tartozó összes felhasználó birtokolja, a felhasználó szintűeket pedig egy-egy felhasználó.

A sorra vonatkozó jogosultságokat az adott sorban található összes jegyre birtokolja a felhasználó. De lehetőség van csak egy-egy jegyre is jogokat adni az ágenseknek.

Jegyre, vagy sorra vonatkozó jogok:

- Jegykezelő: létrehozás, megnyitás, lezárás, megjegyzés és válasz hozzáadása a jegyhez;
- Jegy törlése (logikai a historikus jegynyilvántartás miatt);
- Jegy ágenshez rendelése: a tulajdonos beállítása, megváltoztatása;
- Jegy átvállalása: a jog tulajdonosa elveheti, és magához rendelheti a jegyet egy másik ágenstől;
- Megfigyelő: csak olvashatja a jegyet;
- Asszisztens: olvashatja a jegyet és megjegyzést adhat hozzá;
- Jegy adminisztrátor: csoportot, vagy felhasználót rendelhet a jegyhez;
- Sorkezelő: csoportot, vagy felhasználót rendelhet a sorhoz.

Csoportra vonatkozó jogok:

- Felhasználó hozzáadása, eltávolítása.

A rendszerre vonatkozó jogosultságok, adminisztratív jogokat takarnak, amelyek a következők lehetnek:

- Felhasználó adminisztrátor: létrehozás, módosítás, törlés (logikai a historikus jegynyilvántartás miatt);
- Csoport adminisztrátor: létrehozás módosítás, törlés;
- Sor adminisztrátor: létrehozás, módosítás, törlés;

Egy felhasználó nulla, egy, vagy több csoportnak is tagja lehet, illetve csoportnak lehetnek alcsoportjai is, így kialakítható egy gráffal reprezentálható jogosultsági hierarchia, mely a legváltozatosabb igényeket is képes kielégíteni.

Mivel a rendszer megengedő jogosultságkezeléssel van felvértezve ezért egy felhasználó jogosultságainak halmaza, a felhasználói szintű és csoportszintű jogosultságainak uniójaként áll elő.

A rendszer legalább egy felhasználót tartalmaz, aki az úgy nevezett szuperadminisztrátor jogosultságokkal, a teljes rendszerre vonatkozó minden joggal rendelkezik.

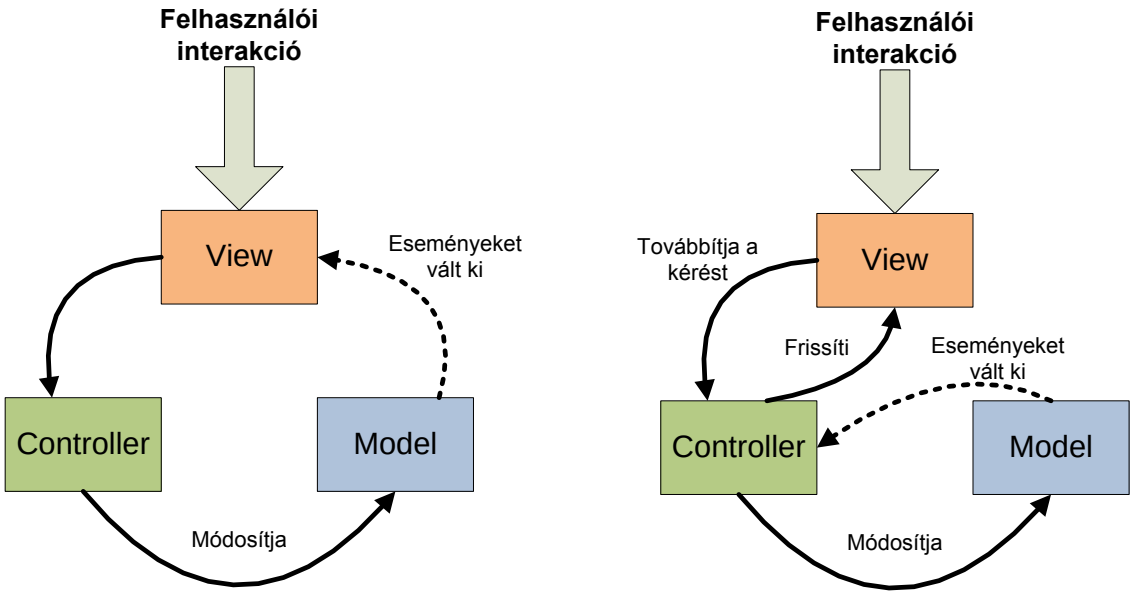
2.2 A felhasznált technológiák és minták

2.2.1 A Model-View-Conroller tervezési minta

Az MVC minta erősíti a rendszer kódjának újrafelhasználhatóságát az által, hogy különválasztja azt a három fő komponenst mely egy felhasználói felülettel rendelkező webalkalmazás (mint például egy egyszerű weboldal) működéséhez és menedzselhetőségéhez szükséges. A Model (modell) tartalmazza az adatot. A View (nézet) megjeleníti azokat és felületet ad a felhasználóknak az adatok manipulálására (a weboldal). A Controller (vezérlő) összeköti a modellt és a View-t, és kezeli az összes közöttük történő interakciót és adat feldolgozási folyamatot a modellben.

Az MVC mintában, a felhasználónak a nézettel való interakciója, eseményeket vált ki a vezérlőben amelyek frissítik a modellt. Ezután a modell eseményeket vált ki, amelyek frissítik a nézetet. Az egyetlen probléma ezzel az, hogy függés alakul ki a modell és a nézet között. Ezt elkerülendő, úgy érdemes az alkalmazásunkat fejleszteni, hogy minden interakciót, logikát a modell és a nézet között a vezérlő tartalmaz és kezel, így elkerülve a frontend és a

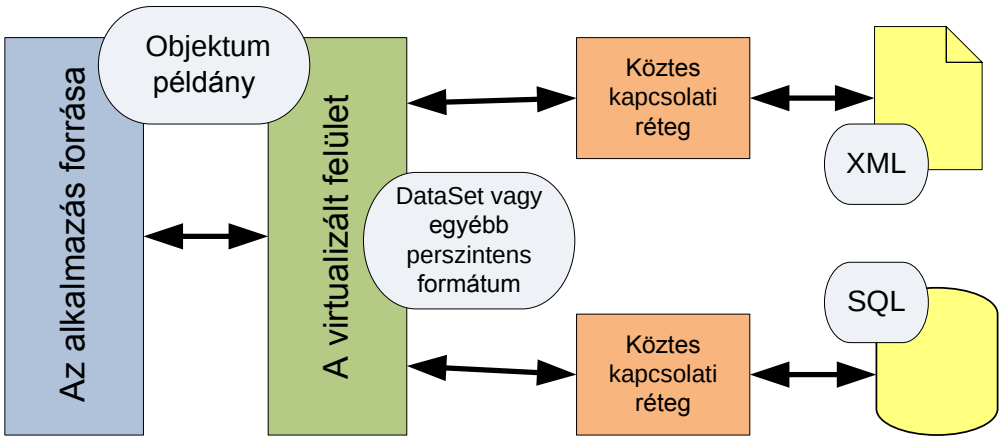
backend szorosabb kapcsolatát, és könnyítve a rendszer tesztelhetőségét. Ez persze növeli az implementáció komplexitását.



4. ábra a Model-View-Controller tervezési minta és a javított változata

2.2.2 A Repository tervezési minta

Ezen minta a modellben történő adatleképzés egy rugalmas és az újrafelhasználhatóságot elősegítő módszertan, mely virtualizálja a perzisztens médiumunk adat entitásait (akár adatbázis, akár egy XML fájl legyen is az). Tesszük ezt a minta mentén úgy, hogy az adatbázis táblák, nézetek, és a lekérdezések eredménytáblái helyett objektumtípusokat, az adat sorok helyett, pedig objektumpéldányokat hozunk létre. Illetve a logikát mely ezen példányokat perzisztálja az adatbázis felé. Így elrejtjük a konkrét adattárolás implementációját az alkalmazástól és csak publikus metóduson keresztül hagyjuk manipulálni azt, mely szintén növeli az újrafelhasználhatóságot és a skálázhatóságot, illetve így az alkalmazás egészét nézve az, a konkrét adatbázistól függetlenné válik. Természetesen ehhez szükségünk van minden egyes tárolási módhoz egy szolgáltató rétegre, mely elvégzi a köztes átalakításokat és perzisztálásokat az osztálypéldányok és a táblák sorai között.



5. ábra a Repository tervezési minta

2.2.3 A pipes and filters tervezési minta

Ezen minta olyan az adatfolyamaink szűrésének egy olyan módját írja elő, melyben a szűrő eljárások atomiak, tetszőlegesen egymás után fűzhetők, és természetesen az újrafelhasználhatóságot szem előtt tartva lettek létrehozva.

Ezen módszertan felépítését és működését egy a való életből vett példa segítségével lehet a legszemléletesebben illusztrálni.

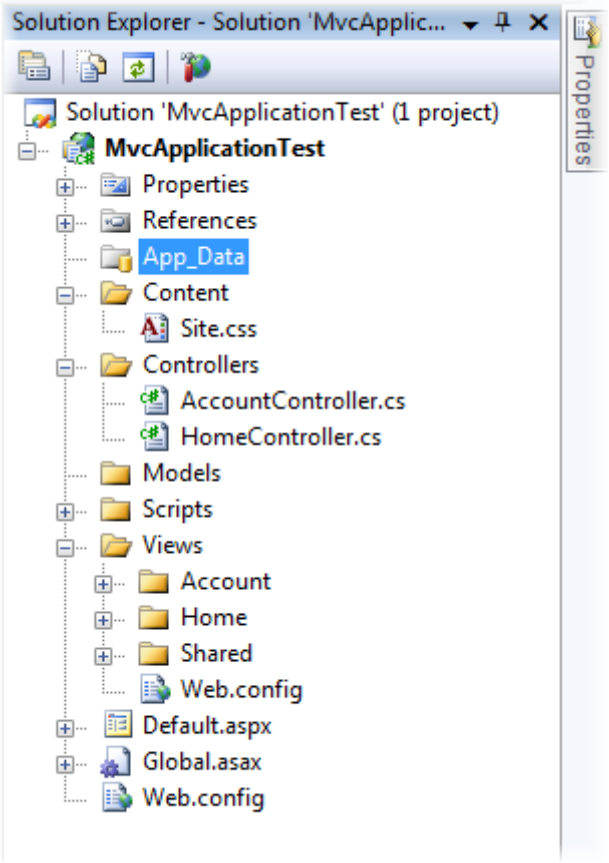
A háztartásunkba csővezetékeken eljutó víz (pipes) legyen az adatfolyamunk, melynek forrása a közeli víztározó (az adatbázis). Attól függően, hogy a vizet milyen felhasználásra szánjuk, ivóvíznek, vagy a kertben locsolásra, különböző szűrők (filters) vannak elhelyezve a csőrendszerben a csapunk (az applikáció) előtt. Ezek a szűrők mindegyike egy-egy különböző fajtájú szennyeződés kiszűrésére alkalmas, így a felhasználás módok függvényében, tetszőleges egymás után fűzhetőek ezeket a csővezeték megfelelő ágain.

2.2.4 ASP.NET MVC 1.0

Olyan eszközrendszer melyet a Microsoft a .NET 3.5-ös keretrendszer első javítócsomagjában adott ki, mely template-ekkel snippetekkel, előre megírt osztálykönyvtárakkal és komplett

metodológiai leírással segíti a Model-View-Controller tervezési minta menti webalkalmazás fejlesztést. A szerver oldali nyelv az ASP.NET, a mögöttes kód nyelve pedig C#, illetve Visual Basic is lehet.

Ha létrehozunk egy ASP.NET MVC Web Application projektek a 6. ábrán látható könyvtár és állomány struktúrát kapjuk eredményül.



6. ábra az ASP.NET MVC projekt könyvtárstruktúrája

Ami első pillantásra szembetűnik, hogy külön könyvtárakat kaptunk a modellek, a nézetek, és a vezérlők tárolására. A *Content* könyvtár a nézet stíluselemeinek tárolására (képek, CSS fájlok stb.) jött létre. Végül, de nem utolsó sorban pedig van egy *Script*s könyvtárunk is mely a nézet kliens oldali szkriptjeit tartalmazza. Mivel a nézet fel van készítve az AJAX-os kontrolok használatára ezért a kezdőállapotban is találunk prototípus szkript fájlokat annak

előkészítéséhez. A létrehozás után akár fordíthatjuk és futtathatjuk is az alkalmazásunkat, hiszen a template tartalmaz egy egyszerű, de futtatható kezdeti példaalkalmazást is.

2.2.5 LINQ to SQL

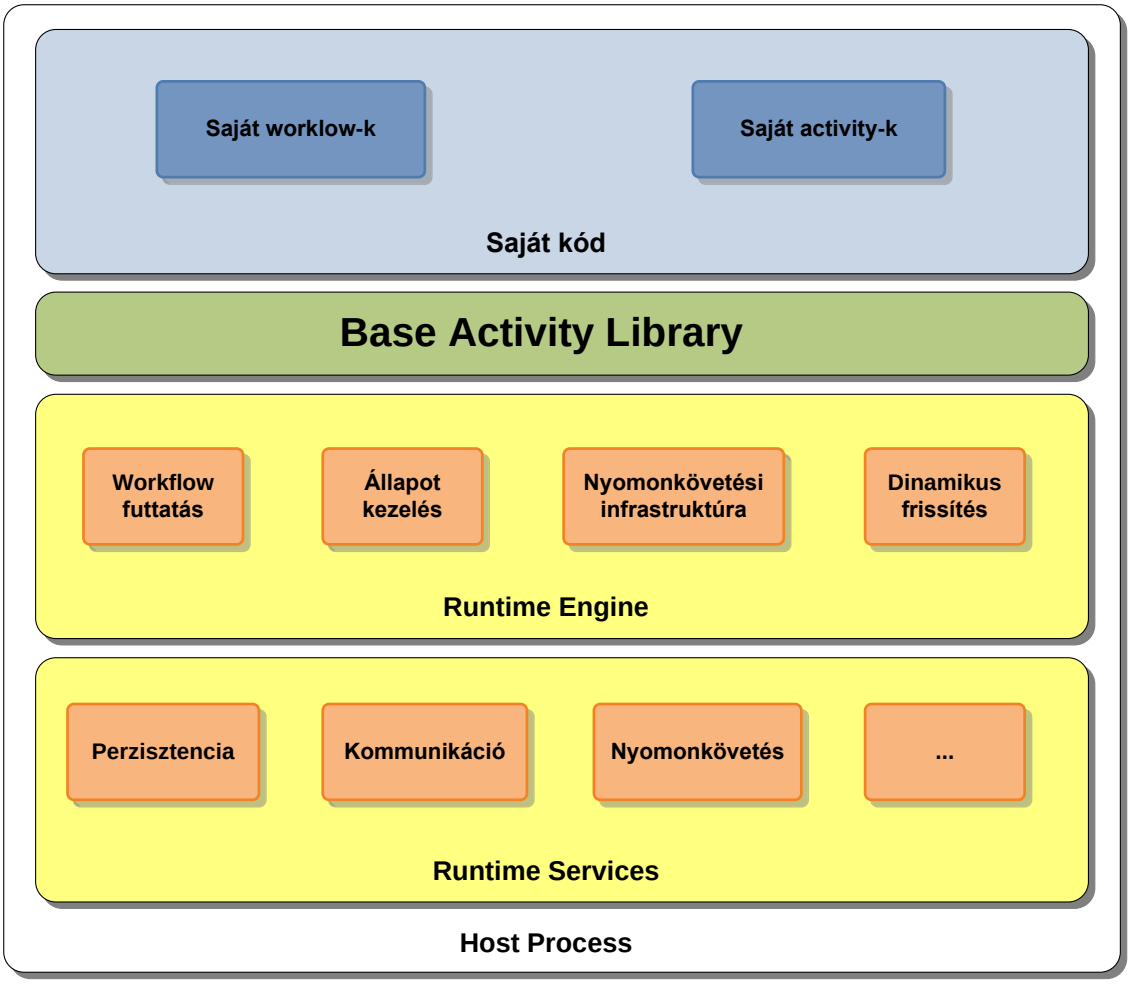
Az ASP.NET MVC ajánlása a modell réteg megvalósítására a LINQ to SQL rendszer mely a 3.5-ös .NET keretrendszerben debütált, és ami nem más, mint egy un. "könnyűsúlyú" ORM (Object-realtional mapping) objektum-relációs leképzést végrehajtó rendszer. Ezen rendszer képes modellezni az adatbázis táblaszerkezetét .NET osztályok segítségével és képes ezeket az 1:1 aranyú leképzés után CLR objektumok formájában kezelni. Majd ezen osztálypéldányokból a LINQ (Language-INtegarted Query) segítségével lekérdezhetjük az adatokat, vagy módosíthatjuk is azokat (insert, update, delete). Az objektumok módosításainak követését és perzisztálását az adatbázisba, a LINQ to SQL csendben a háttérben végzi. E mellet lehetőséget ad az egy táblán alapuló entitás-polimorfizmus megvalósítására is.

2.2.6 A Repository tervezési minta és a modell illeszkedése

E mellet a Repository tervezési mintát is alkalmaznunk kell a modell létrehozásakor, a fent írtak miatt úgy, hogy a LINQ lekérdezések eredményeit az alkalmazás szempontjából emészthetőbb alakra hozott osztályok példányaiba mentjük, és ezek publikus metódusainak segítségével tesszük módosíthatóvá az entitásokat. Így nem szükséges valódi adatbázis az alkalmazás teszteléséhez, mivel az csak ezekkel az egyszerűsített osztálytípus példányokkal dolgozik, illetve így a modell implementációja egészében bármikor lecserélhető, vagy módosítható, optimalizálható az alkalmazás alatt mivel a Repository használatával az teljesen független a modelltől. Elrejtődik továbbá az alkalmazási réteg elől a perzisztencia kezelése is, így, ezen réteg sokkal tisztább csak az üzleti logikát és a megjelenítést megvalósító része lehet a teljes rendszernek.

2.2.7 Windows Workflow Foundation

A WWF egy olyan keretrendszer mely lehetővé teszi felhasználó orientált és rendszer folyamatok implementálását az alkalmazásunkba, melyet Windows Vistára, XP-re vagy Windows Server 200x-re fejlesztünk. A keretrendszer, mely a .NET 3.5 SP1 keretrendszernek már szerves része, tartalmaz egy processzbeli munkafolyamat motort, debuggert és dizájn eszközöket a Visual Studio számára. A WWF-et használhatjuk olyan egyszerű probléma megoldására, mint például felhasználói felület kontrolok megjelenítése, bemenettől függően, vagy olyan bonyolultabb folyamatok kezelésére, mint az üzleti felhasználásban előforduló rendelés feldolgozás, vagy a leltárkezelés. A WWF munkafolyamat tehát az üzleti logika egy reprezentációját implementálja.



7. ábra a Windows Workflow Runtime felépítése

A Windows Workflow Foundation API teljes támogatást nyújt a Visual Basic .NET, és a C# alatt történő fejlesztéshez, tartalmaz egy specializált munkafolyamat fordítót, eszközöket a munkafolyamaton belüli debugolásra, illetve egy grafikus tervező felületet.

Fontos megemlíteni, hogy mit az a 7. ábrán látszik a workflow runtime nem egy önálló alkalmazás, minden esetben egy folyamatban tárolnunk kell, gyakorlatilag egy .NET-es osztályról van szó. Így ezt megtehetjük bármilyen típusú .NET-es alkalmazással, legyen az konzolalkalmazás, Windows Forms, vagy webalkalmazás, Windows-, vagy webservice, tehát bármilyen CLR AppDomainben befogadható, a lényeg hogy a befogadó folyamat .NET-es legyen.

A befogadással kapcsolatban vannak viszont bizonyos megkötések. Egy folyamat csak egy runtime-ot tartalmazhat, illetve egy AppDomainen belül is csak egy futhat belőle.

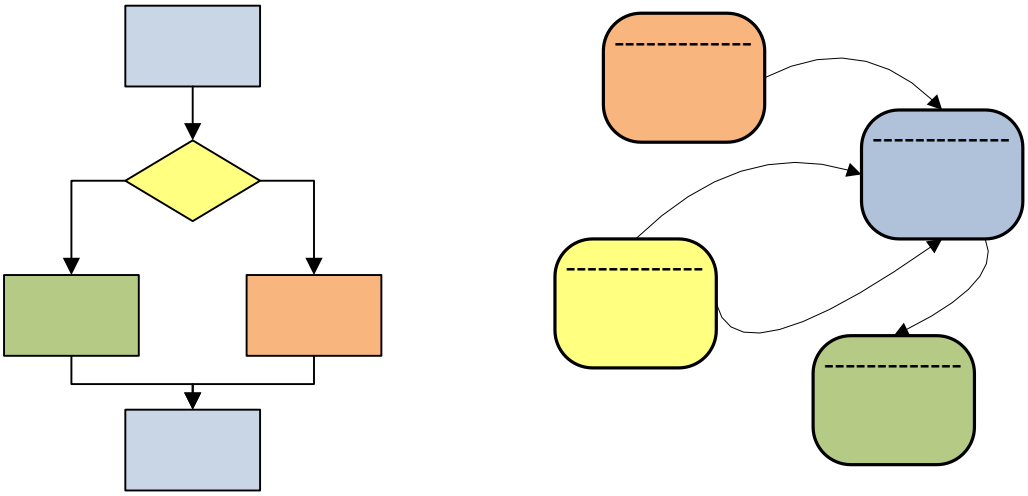
A runtime elsődleges feladatai:

- A munkafolyamat példányokat kezelése.
- A 7. ábrán látható Runtime Services halmazban található szolgáltatásokat felügyelje.
- Illetve bizonyos eseményekkel a befogadó alkalmazást informálja a folyamatok állapotáról.

A Windows Workflow Foundation két munkafolyamat modell típust különböztet meg. Ezek pedig a szekvenciális és az állapotgép modellek.

A szekvenciális munkafolyamatok egymás utáni lépések, úgy nevezett tevékenységek (Activity) sorozatát hajtják végre. Ezekkel kiválóan reprezentálhatók a rendszer folyamatok. Bár, ezen modelltípustól nevéből adódóan egymás utáni lépések végrehajtását várjuk, azért lehetőség van párhuzamos és átlapolódó tevékenység végrehajtásra is benne.

Az állapotgép munkafolyamatokban állapotok vannak és események hatására állapot átmenetek történnek. Ez a fajta modell mely leginkább az állapot diagramhoz hasonlítható, kiválóan alkalmas felhasználó orientált folyamatok leírására.



8. ábra. A szekvenciális és az állapotgép modell.

3. A fejlesztés

3.1 Az előkövetelmények

3.1.1 A fejlesztéshez szükséges eszközök

Mielőtt megkezdenénk a fejlesztést össze kell gyűjtenünk és telepítenünk az ehhez szükséges szoftvereket melyek ebben az esetben a következők.

- Microsoft Windows Vista Business SP1 – az operációs rendszer
- Microsoft Visual Studio 2008 Team Suite – grafikus fejlesztői környezet
- Microsoft SQL 2005 Server Express Edition – az adatbázis kezelő rendszer
- Microsoft SQL Management Studio – grafikus fejlesztői környezet az adatbázishoz
- Microsoft .NET Framework 3.5 SP1 – a futtatáshoz szükséges keretrendszer
- Microsoft IIS 7 – a webszerver alkalmazás mely az ITraq-et tárolni fogja
- Microsoft Expression Web 2 – grafikai tervező szoftver mellyel az alkalmazásunk felhasználói felületének grafikai elemeit CSS stílus fájljait állítom elő

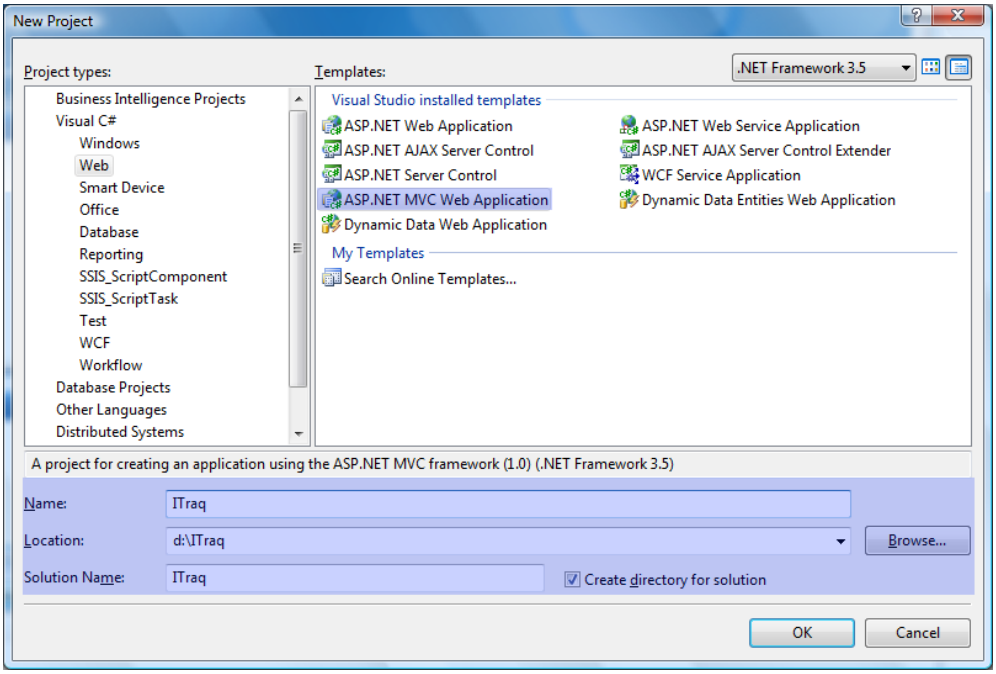
3.1.2 A rendszer futtatásához szükséges szoftverek

Természetesen az elkészült rendszer futtatásának is vannak előkövetelményei, melyek a mi esetünkben a következők.

- Microsoft Windows Vista Business SP1, vagy Windows Server 2008 – operációs rendszer
- Microsoft .NET Framework 3.5 SP1 – a futtatáshoz szükséges keretrendszer
- Microsoft SQL Server 2005 Express Edition, vagy újabb – az adatbázisunk kezeléséhez
- Microsoft IIS 6, vagy újabb – a webalkalmazásunk tárolására

3.2 A projekt kezdeti kialakítása

Miután minden a fejlesztéshez szükséges minden szoftvert telepítettünk és kialakítottuk a számunkra kényelmes munkakörnyezetet, nincs más hátra, mint hogy létrehozzuk a fejlesztői projektünket Visual Studioban.



8. ábra új projekt létrehozása

A projektünk létrehozása után mely során az ASP.NET MVC 1.0-ról szóló fejezetben leírtak szerint létrejött a könyvtárstruktúra. Az előkészítés utolsó lépéseként létrehozzuk az adatbázis fájlokat, melyek adatbázis sémáinkat reprezentálják. Ebből kettőre lesz szükségünk, az egyik az *ITraqDB.mdf* nevű mely a rendszerünk tábláit fogja tartalmazni, a másik pedig az *aspnetdb.mdf* amely az ASP.NET 3.5 Membership/Role Management számára szükséges adatokat. Mindkettőt létrehozzuk egyszerűen a projektünk App_Data könyvtárába. De itt még az előkészítés nem ér véget, a második adatbázis tábláit le kell generáltatnuk a .NET keretrendszerrel automatikusan az aspnet_regsql power commanddal. Ha az adatbázis táblái létrejöttek, a projektünk elmentésével el is készültünk a kezdeti kialakítással.

3.3 A modell létrehozása

3.3.1 Az adatbázis táblák

Az *ITraqDB* adatbázisban létrehozzuk a modell tárolására szükséges relációs táblaszerkezetet, melyet előre gondosan megterveztünk. Optimalizáció gyanánt normalizálást, a táblaszerkezet normál formára hozását alkalmazzuk.

A normalizálás az adatbázisban található adatok rendszerezését jelenti. Táblákat hozhatunk létre, és azok között kapcsolatokat létesíthetünk szabályok szerint. A szabályok célja az adatok védelme és az adatok rugalmasabbá tétele (például a redundanciák és az inkonzisztens függőségek kiküszöbölése).

A redundáns adatok feleslegesen foglalják a lemezterületet, és karbantartási problémákat okoznak. Ha a több helyen megtalálható adatot módosítani kell, a módosítást minden helyen pontosan ugyanúgy kell elvégezni.

Mi az „inkonzisztens függőség”? Ha a felhasználó egy vevő címére kíváncsi, azt magától értetődően a Vevők táblában keresi, de például a vevőhöz tartozó alkalmazott bérét nem lenne logikus ebben a táblában tárolni. Az alkalmazott bére az alkalmazotthoz kapcsolódik (attól függ), ezért azt az Alkalmazottak táblában kell tárolni. Az inkonzisztens függőségek hatására előfordulhat, hogy az adatokhoz nehezen lehet hozzáférni, mert az adatok elérési útja hiányos vagy hibás lehet.

Az adatbázisok normalizálásának van néhány szabálya. A szabályok neve „normálforma”. Ha az adatbázisra teljesül az első szabály, „első normálformában” van. Ha az adatbázis megfelel az első három szabálynak, „harmadik normálformában” van. Bár a normalizálás további szintjei is lehetségesek, a legtöbb alkalmazás esetében a harmadik normálforma a szükséges legmagasabb szint.

A valós problémák számos esetben nem feleltethetők meg tökéletesen a szabályoknak és specifikációknak. A normalizáláshoz általában új táblák létrehozása szükséges, és egyes ügyfelek ezt fáradságosnak találják. Ha úgy döntünk, hogy a normalizálás első három szabálya közül valamelyiknek nem tesz eleget, meg kell bizonyosodnunk arról, hogy az alkalmazás kiküszöböli az esetlegesen felmerülő problémákat (például az adatok redundanciáját és az inkonzisztens függőségeket).

Első normálforma

- Szüntessük meg az egyes táblákban az ismétlődő csoportokat.
- Az egymáshoz kapcsolódó adatok minden halmazához hozzunk létre külön táblát.
- Az egymáshoz kapcsolódó adatok minden halmazát azonosítsuk elsődleges kulccsal.

Egy táblán belül ne használjunk több mezőt hasonló adatok tárolására.

Második normálforma

- Hozzunk létre külön táblákat a több rekordra vonatkozó értékcsoportokhoz.
- A táblákat külső kulccsal kapcsoljuk össze.

A rekordok csak a tábla elsődleges kulcsától függhetnek (ha szükséges, ez lehet összetett kulcs). Vegyük például egy vevő címét egy számlázási rendszerben. A címre szükség van a Vevők táblában, de a Rendelések, a Szállítás, a Számlák, a Követelések és az Inkasszó táblában is. Ahelyett, hogy a vevő címét minden táblában külön bejegyzésként elhelyeznénk, tároljuk azt egyetlen helyen, a Vevők táblában vagy egy különálló Címek táblában.

Harmadik normálforma

- Szüntessük meg a kulcstól nem függő mezőket.

A rekordok azon értékei, amelyek nem részei a rekord kulcsának, nem tartoznak a táblához. Általánosan elmondható, hogy ha egy mezőcsoport tartalma a tábla egynél több rekordjára vonatkozhat, a mezőcsoportot egy külön táblába kell helyezni.

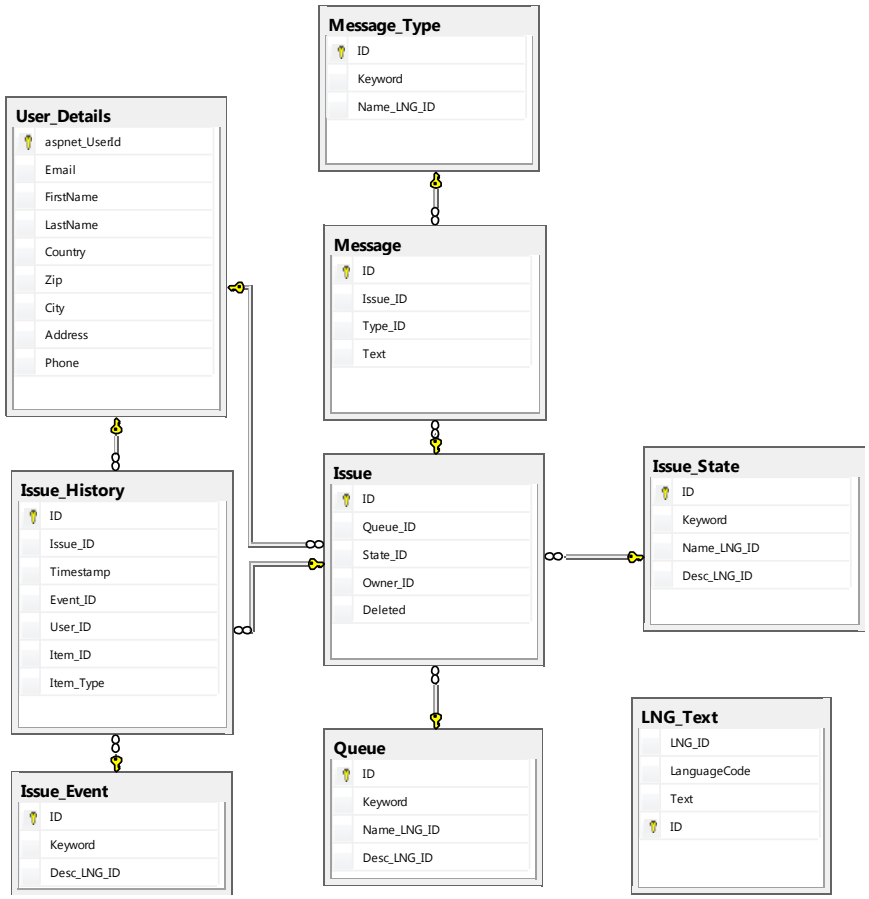
Bár a harmadik normálforma használata elméletileg mindig szükséges, a gyakorlatban nem minden esetben célravezető. Ha a Vevők táblában például meg szeretnénk szüntetni a mezők közötti összes lehetséges függőséget, külön táblát kell létrehoznia a városok, az irányítószámok, az értékesítési képviselők, a vevőosztályok és minden, a rekordok között esetlegesen ismétlődő tényező számára. A normalizálás elméletben minden esetben hasznos. A sok kis tábla kezelése azonban csökkentheti a teljesítményt.

Ezért célszerűbb a harmadik normálformát csak olyan esetben alkalmazni, ha az adatok gyakran változnak.

További normálformák

A negyedik normálforma – más néven Boyce-Codd normálforma (BCNF) – és az ötödik normálforma létezik ugyan, de a gyakorlatban ritkán használják. Ezen szabályok figyelmen kívül hagyásával az adatbázis felépítése nem tökéletes, de ez a felhasználhatóságára nincs hatással.

Így született meg az alábbi táblaszerkezet, mely megfelelően normalizált az adataink hatékony tárolásához.



9. ábra. Az adatbázis táblaszerkezete.

A táblák létrehozása, a kapcsolatok kialakítása és a megszorítások hozzáadása után feltöltjük a megfelelő táblákat alapadatokkal majd elmentjük az adatbázist. Ha az adatbázisunk teljesen kész, szerkezetileg már nem akarunk módosítani rajta, az SQL Management Studio segítségével telepítő szkriptet készítünk az adatbázisról, ami nem tesz mást, mint T-SQL nyelven létrehoz egy szkriptfájlt, melynek futtatásával elő tudjuk állítani az adatbázis jelenlegi úgymond kezdeti állapotát.

3.3.2 ORM készítése LINQ to SQL-el

A modell enkapszulációjának első lépéseként a kész adatbázist leképezzük LINQ to SQL osztályokra, így a tábláink sorait az őket reprezentáló osztályokon keresztül érhetjük majd el.

Ehhez nem kell mást tenni mint a *Model* könyvtáron jobb egérgombbal kattintani az *Add New Item->Data->Linq to SQL Classes* elemre majd elnevezni, majd *OK* gomb.

Egy grafikus tervezői felülethez jutottunk, ahová a *Server Explorerből* be tudjuk húzni a leképezni kívánt adatbázis tábláinkat. Itt az *ITraqDB* adatbázis összes tábláját hozzáadjuk, majd mentünk. Ekkor a rendszer létrehozza a leképzéshez szükséges fájlokat. A tervező ablakban láthatjuk, hogy az adatbázis tábláink mellet a rendszer a köztük lévő kapcsolatokat és megszorításokat is észlelte és leképezte, így azok meg is jelennek a diagramon.

3.3.3 A Repository Pattern megvalósítása

Első lépésként létrehozzuk azokat az osztálytípusokat melyek példányaival, mint adategységekkel szeretnénk, hogy az alkalmazásunk dolgozzon. Lássuk egy konkrét példán keresztül, hogyan is fog ez kinézni.

Az *Issue* osztály definíciója, melynek egy példánya egy rendszerbeli jegyet reprezentál, a következőképp néz ki.

```
public class Product {

        public long ID { get; set; }
        public IssueState State { get; set; }
        public ITraqUser Owner { get; set; }
        public string Origin { get; set; }
        public IList<Message> Messages { get; set; }
        public IList<Attachment> Attachments { get; set; }
        public IList<HistoryElement> History { get; set; }

        public Product() {
            this.Messages = new LazyList<Message>();
            this.Attachments = new LazyList<Attachment>();
            this.History = new LazyList<HistoryElement>();
        }

        public Product(string origin) {

            this.Messages = new LazyList<Message>();
            this.Attachments = new LazyList<Attachment>();
            this.History = new LazyList<HistoryElement>();

            this.Origin = origin;

        }
    }
```

Az osztálydefiníció igen egyszerű, az adattagokat tároló publikus tulajdonságokból és az osztály konstruktoraiból áll. Láthatjuk, hogy míg a *Messages*, *Attachments* és *History*

tulajdonságok deklarációjánál *IList* típust használtam, addig a konstruktorokban már *LazyList* típusú értéket adok nekik. Ez a *LazyList* osztály egy a lusta értékadást megvalósító megvalósítása az *IList* interfésznek. Ennek használatával a modell csak akkor kérdezi le az adatbázisból ezen listák tartalmát, ha az alkalmazásunk el akarja azt érni, ezzel további erőforrásokat takaríthatunk meg olyan esetekben, mikor nincs szükség az adott objektum példány adott tulajdonságaira.

Az osztályok létrehozása után, definiálunk egy interfészt *IRepository* néven. Melyben az összes adategység osztályhoz tartozó lekérdező és beállító metódusok prototípusát felvesszük.

```
public interface IRepository {

        IQueryable<Issue> GetIssues();
        void SetIssue(Issue is);
        .
        .
        .
}
```

Erre azért van szükség, mert ezen interfész megvalósításának segítségével bármilyen adatbázishoz írhatunk saját repository osztályt, így függetlenítve a rendszerünket az adatbázis kezelő rendszertől. Látható, hogy a lekérdező metódusok egy *IQueryable* generikus típussal térnek vissza melynek hatalmas előnye, hogy a tartalma LINQ utasítások segítségével lekérdezhető.

Következő lépésként az *SqlRepository* osztállyal megvalósítjuk az előbb vázolt interfészt és ezzel kész is a minta implementációja.

3.3.4 Adatszűrés a Pipes and Filters tervezési minta alkalmazásával

A modellünk egyik nagy problémája az implementáció jelenlegi szakaszában, hogy nem tartalmaz semmilyen szűrést. Az *SqlRepository* osztály get metódusainak az eredményhalmaza az adatbázis teljes idevonatkozó tartalma lesz. Ez így a legtöbb esetben erőforrás pazarló és felesleges. Ennek kiküszöbölése érdekében alkalmazzuk a Pipes and Filters tervezési mintát a modellünkben a következő módon.

Létrehozunk egy statikus *Filters* osztályt melyben megírjuk a fejlesztés további szakaszaiban felhasznált szűrő metódusokat, melyek magját minden esetben egy-egy LINQ lekérdezés adja és természetesen ezek is egy *IQueryable* generikussal térnek vissza.
Lássunk példaképp egy ilyen szűrő metódust.

```
public static IQueryable<Issue> WithIssueID(this IQueryable<Issue> qry,
long ID) {

         return from i in qry
         where i.ID == ID
         select i;
}
```

Nagy előnye ezen szűrő metódusoknak, hogy hívásaik hozzáfűzhetőek a repository osztályunk metódusainak hívásához, így tovább egyszerűsítik a kódot is.

```
Issue i =_repository.GetIssues().WithIssueID(id).Single();
```

Ráadásul nem csak a hívások összefűzhetőek, a LINQ to SQL az adatbázisból történő lekérdezést úgy alkotja meg, hogy a különböző szűrőfeltételeket is hozzáadja az adatbázisbeli lekérdezéshez. Így, csak egyszer történik lekérdezés az adatbázisból, és az is a teljes komplex szűrőfeltételeinkkel együtt.

3.3.5 Szolgáltató osztály létrehozása

Ahhoz, hogy a modell enkapszulációja teljes legyen, létrehozunk egy publikus szolgáltató osztályt, melyben olyan metódusok kapnak helyet, melyek az alkalmazás többi részének igényei szerint nyernek ki adatot a modellből. A használatba vételéhez az applikáció megfelelő részeiben csak példányosítani kell ezt az osztályt, a konstruktorának paramétereként átadni egy repository példányt. Íme, az osztályunk kódjának egy szignifikáns része.

```
public class Service {

        IRepository _repository = null;

        public Service(IRepository repository) {
            _repository = repository;
            if (_repository == null)
                throw new InvalidOperationException("Repository cannot be
null");
        }


        public Issue GetIssue(long id) {

            Issue result = _repository.GetIssues()
                .WithIssueID(id)
                .SingleOrDefault();


            return result;
        }

        . . .
```

Ezzel el is készültünk a modellünk megvalósításával, mely készen áll az alkalmazás többi részével, a kontrollerekkel, és nézettekkel való együttműködésre, azok kiszolgálására.

3.4 A vezérlők kialakítása

3.4.1 Az aktív jegyek életciklusának kezelése WWF-el

A rendszerünkben aktívnak nevezzünk minden olyan jegyet, mely nem lezárt, vagy törölt állapotban van.

Nyílván való követelmény a rendszerünkkel szemben, hogy a jegyek életciklusát nyomon tudjuk vele követni. Megtehetjük ezt úgy, hogy a munkafolyamat logikáját beégetjük, belekódoljuk, az ASP.NET MVC vezérlőosztályokba, vagy létrehozhatunk hozzá egy WWF munkafolyamatot. Az első módszer azért nem túl szerencsés, mivel beégetett kódról van szó igen nagyfokú flexibilitást vesztünk vele, a munkafolyamat módosításának egyszerűsége terén. Így kézenfekvő a Windows Workflow használata.

A követelmények tekintetében egy állapotgép munkafolyamat létrehozása tűnik számunkra megfelelőnek azon egyszerű okból kifolyólag, hogy az aktív jegy definiálása a jegy állapotain keresztül történt, és egy jegy életciklusának az alapvető mérföldkövei, az állapotának változtatása során jönnek létre.

Létrehozunk tehát egy *State Machine Workflow Library* típusú projektet *IssueFlowLibrary* néven, majd ebben a projektben létrehozzuk az *ActiveIssueStateMachine* nevű munkafolyamatunkat. Ezután a grafikus tervezői felület segítségével hozzáadjuk a köztes állapotokat és a végállapotot, az állapotgépünkhöz, ami a kezdő állapotot a létrehozás utáni alapállapotában már tartalmazza. Így a munkafolyamatunk állapothalmaza a következőket fogja tartalmazni.

- InitialIssueState – jegy temporális iniciális állapotban van.
- NewIssueState – azaz a jegy új állapotú.
- OpenIssueState – a jegy megnyitott.
- AcceptedSolutionIssueState – a jegy megoldását az ügyfél elfogadta.
- RejectedSolutionIssueState – a jegy megoldását az ügyfél elutasította.
- ClosedIssueState – a jegy lezárt állapotban van.
- DeletedIssueState – a jegy törölt állapotú.
- FinalIssueState – a jegy a hozzá tartozó munkafolyamat lezárást megelőző állapotban van.

Ezek után léterhozzuk az állapotokon belül szükséges tevékenységeket, majd kialakítjuk az állapotok közötti, kapcsolatokat, és átmeneteket esemény vezérelt tevékenységek segítségével. A tevékenység (Activity) a WWF munkafolyamatok alap, építő eleme.

Futási időben egy-egy állapotgép munkafolyamat fog tartozni, és futni minden egyes aktív jegyhez. Ezek a munkafolyamatok könnyen elérhetőek, hiszen mindegyik rendelkezik globális egyedi azonosítóval (GUID).

Mivel a folyamatokon keresztül pontos információt tudunk szerezni a jegyek állapotáról, ez alkalmassá teszi a rendszert esetleges későbbi SLA-hoz (Servive Level Agreement – az

elégséges szolgáltatásról szóló megállapodás) kapcsolódó mérési, és felügyeleti eszközöket megvalósító komponensek gyors, és egyszerű fejlesztését, integrálását a rendszerbe.

3.4.2 A vezérlő osztályok létrehozása

ASP.NET MVC-ben a vezérlő osztályok tartalmazzák az alkalmazás logikáját, összekapcsolják a modellt és a nézeteket. Ezen osztályok példányai nyerik ki a modellből az adatokat, melyeken végrehajtják a megfelelő átalakításokat, és csak az adott szituációban megjeleníteni kívánt adathalmazt továbbítják a nézeteknek.

Minden ASP.NET MVC osztály vezérlőosztály mely publikus láthatósággal rendelkezik, illetve a nevének utótagja a *„Controller”* szó. *Pl.: QueueController, TicketController, stb.* Praktikusságból és a projekt átláthatóságát fenntartása érdekében ezeket az osztályainkat a projekt létrehozásakor automatikusan létrejött *Controllers* könyvtárban helyezzük el.

A vezérlő osztályok, vezérlő tevékenységekből épülnek fel, melyek nem mások, mint olyan publikus metódusok, melyeket URL-ből tudunk hívni. Egy ilyen hívás a következőképpen néz ki.

http://azenitraqem.com/Sorok/JegyReszletek/Altalanos/10

Ez az URL meghívja a *„Sorok”* vezérlő *„JegyReszletek”* tevékenységét és annak átadja az *„Altalanos”* és a *„10”* paramétereket. Aminek eredményeképpen az alkalmazás megjeleníti az Általános sorban tízedikként várakozó jegy részleteit.

Vigyáznunk kell azonban ezekkel a tevékenységek fejlesztésénél, mert ezeket, az alkalmazás felhasználói később bármikor bárhonnan hívhatják, mely adott esetben adatbiztonsági problémákat vethet fel.

A mi alkalmazásunk logikai felépítéséből adódóan négy különböző vezérlőosztály hozunk létre melyek a következőek.

- HomeController: Ez a vezérlő tartalmazza az authentikációval kapcsolatos tevékenységeket, illetve a bejelentkezés után megjelenő főoldal által megjelenített adatok összegyűjtését és a megfelelő formában való átadását a megfelelő nézetek számára.

- IssueController: A jegyek részletes megjelenítéséhez, létrehozásához, módosításához és törléséhez (CRUD: CReate, Update, Delete) szükséges tevékenységeket, illetve az ehhez szükséges adatok összegyűjtéséhez szükséges segédmetódusokat ebben a vezérlőben tároljuk. Ezen felül ebben a vezérlővel kapcsolódunk az aktív jegyeink életciklusát reprezentáló WWF munkafolyamatokhoz is.

- QueueController: A sorok megjelenítéséhez és kezeléséhez szükséges tevékenységeket tároljuk itt.

- AdminController: Minden adminisztratív szituáció kezeléséhez szükséges tevékenység, adat transzformáció ebben a vezérlőben kap helyet, legyen az rendszer, sor, jegy, vagy csoport szintű. Ebben a vezérlőben találjuk a felhasználók menedzseléséhez szükséges tevékenységeket is.

Lássuk egy vezérlő kódszintű felépítését is.

```
public class IssueController {

    IRepository rep = new SqlRepsoitory();
    Service serv = new Service(rep);

    [Authorize]
    public ActionResult Index()
    {
      Ilist result = null;
      result = serv.GetIssuesFromQueue(
               serv.GetDefaultQueueIdForUser(User.Identity.Name))
               .ToList();
      return View(Index,"result");
    }

    [AcceptVerbs(HttpVerbs.Post), Authorize]
    public ActionResult Create(Issue issueToCreate)
    {
      . . .
    }   . . .
```

Mint azt az Index tevékenység-metódusnál láthatjuk a visszatérési értéke *ActionResult* típusú mely a következő eredményosztályok őse:

- ViewResult: Egy HTML dokumentumot reprezentál.
- EmptyResult: Üres eredményosztály.
- RedirectResult: Egy másik URL-re való átirányítást valósít meg.
- JsonResult: JavaScript Object Notation eredményt reprezentál melyet AJAX-os alkalmazásokban használhatunk.
- JavaScriptResult: Egy JavaScript szkriptet tartalmaz.
- ContentResult: Egyszerű szöveges válasz.
- FileContentResult: Egy letölthető bináris tartalmú fájlt reprezentál.
- FilePathResult: Egy letötölthető fájlt az elérési útjával együtt tartalmazó eredmény.
- FileStreamResult: Letölthető fájl folyamot tartalmaz.

A mi esetünkben a visszatérési érték, konkrétan egy *ViewResult* típusú lesz, amit a *Views* mappa *Issue* almappájának *Index.aspx* nézetét a *result IList* típusú paraméterrel meghívva kapunk vissza abban az esetben, ha a felhasználó, aki a kontroller tevékenységet hívta, be van jelentkezve a rendszerbe, mely megkötést a *[Authorize]* attribútummal helyeztünk el a tevékenységen.

3.5 A nézetek implementálása

3.5.1 A Mesterlap kialakítása

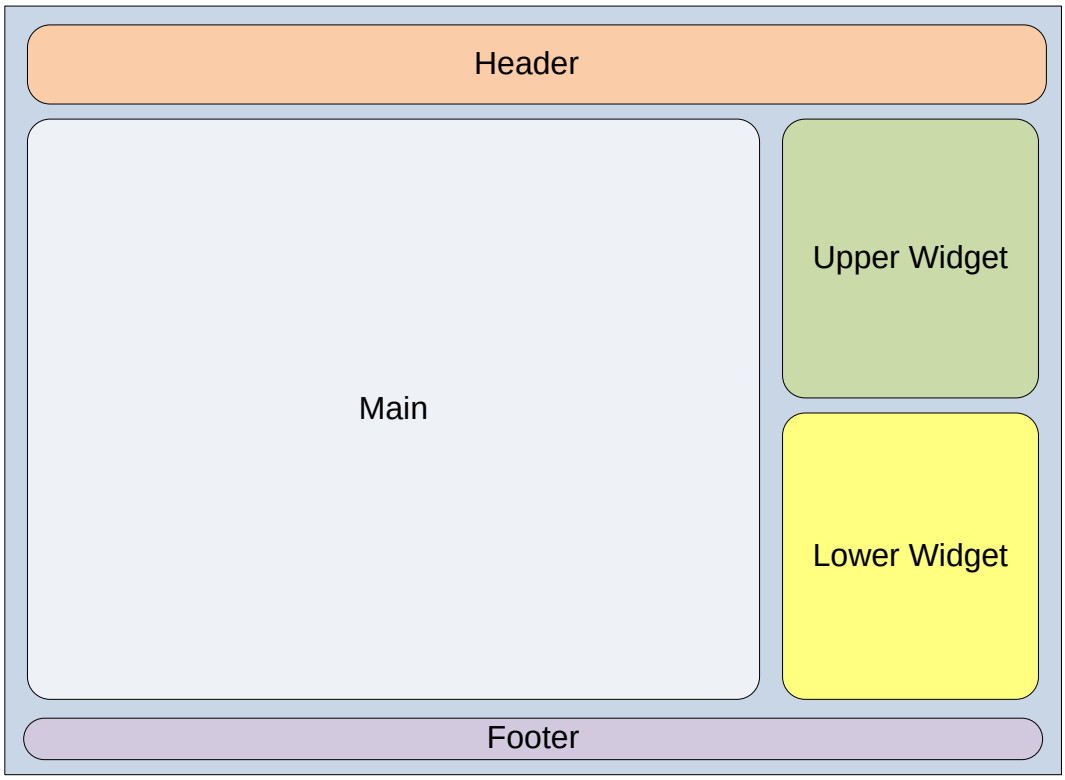
Az alkalmazásunk külső megjelenését a nézetlapok adják. Ahhoz, hogy a felhasználói felület egységes megjelenésű legyen, és a nézetek ne tartalmazzanak a megjelenítésre vonatkozó redundáns kód részeket, mesterlapot (Master page) alkalmazunk.

A mesterlap nem más, mint egy ASP.NET oldal mely olyan tartalom-helyőrző (Content placeholder) vezérlőket tartalmaz, melyekben a nézeteink a saját nézet részeiket illesztik

hívásuk során. A mesterlapot CSS (cascade style-sheet) stíluslapokkal tudjuk formázni, így igen vagy flexibilitást kapunk az egységes, mégis változatos felhasználói élmény kialakítása érdekében.

A felhasználói felület elrendezésének, színvilágának megtervezésekor természetesen figyelembe kell vennünk a felhasználói célcsoport igényeit és az ergonómia tudományának ide vonatkozó szabályait, előírásait, melyeket a téma összetett mivoltából kifolyólag a diplomamunkámban nem részletezek.

A mesterlapunkat, mely *Site.Master* néven a */Views/Shared* könyvtárban automatikusan létrejött a projekt létrehozása során a 10.ábrán látható képi beosztás szerint alakítjuk ki.



10. ábra. A mesteroldal elrendezése.

Látható, hogy a felületet öt különálló részre osztottuk a tartalom-helyőrzők segítségével. Ezeket a következő célokkal hoztuk létre.

- HeaderContentPlaceHolder: Avagy fejléc. A főmenü és az ehhez kapcsolódó navigációs információk megjelenítését végző nézetek helye.

- MainContentPlaceHolder: Főtartalom. Az adott tartalomhoz kapcsolódó almenüt, illetve a megjelenítendő tartalom részleteit tartalmazó nézetek helye.

- LowerWidget-, és UpperWidgetContentPlaceHolder: A főtartalomhoz kapcsolódó gyorshivatkozásokat, egyéb kiegészítő adatokat megjelenítő nézetek helye.

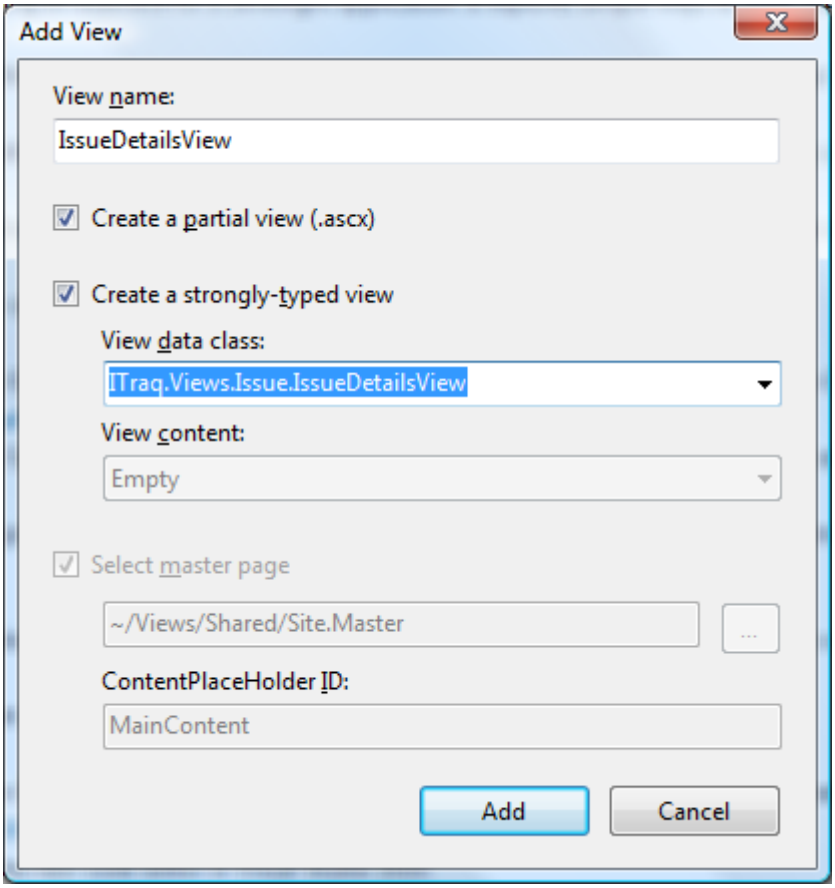
- FooterContentPlaceHolder: Státuszjelentések, és egyéb rövid statikus információk megjelenítésére.

3.5.2 A nézet lapok létrehozása

A nézeteket úgy rendszerezzük, hogy a *Views* könyvtárban a megfelelő vezérlőhöz tartozó nézeteket a vezérlő nevének előtagjával elnevezett alkönyvtárakba hozzuk létre. Pl.: az *IssueControllerhez* tartozó nézetek tárolásához létrehozzuk a */Views/Issue* könyvtárat.

Külön nézeteket hozunk létre a vezérlők minden tevékenységéhez. Az így létrehozott nézetek nem mások, mint ASP.NET lapok, melyek tartalom vezérlőket tartalmaznak (Content control) ezen vezérlők tartalma fog megjelenni a mester lap megfelelő tartalom-helyőrzőiben. Hogy melyik melyikben, azt a *ContentPlaceHolderID* tulajdonságuk értéke adja meg, melybe a megfelelő tartalom-helyőrző azonosítóját kell, hogy megadjuk.

Lehetőség van résznézetek (Partial View) létrehozására, melyek remekül használhatók olyan esetekben, mikor a különböző nézetek egy része ugyan olyan vezérlőket tartalmaz. Ekkor létrehozunk egy résznézetet mely a közös megegyező részt tartalmazza, majd mindegyik nézetlapon a megfelelő helyen elhelyezzük a *<% RenderPartial(„ReszNezetNev”); %>* szerveroldali hívást, amivel beillesztjük a résznézetet az adott nézet lapba. Ez a módszer olyan esetekben is alkalmazható, és a nézeteink kialakítása során alkalmazzuk is, mikor egy nézet lap igen komplikált és ésszerű annak alrészekre bontása, növelvén a kód átláthatóságát.



11. ábra. Résznézet hozzáadása

A mi esetünkben az újrafelhasználhatósági, és átláthatósági megfontolások mentén, a nézetek különböző tartalom vezérlőit külön-külön résznézetekbe hozzuk létre, Így a lehető legkisebb logikai egységekbe szervezve azokat.

Létre kell hozzunk még olyan nézeteket is melyek különböző kivételek kiváltódása, vagy egyéb probléma esetén jelennek meg. Pl.: *NotValidUser.aspx, IssueNotFound.aspx,* stb.

Lássunk tehát egy szemléletes példát a nézetlapra és a résznézetre is.

A *DinnerForm* részнézet:

```
   <% using (Html.BeginForm()) { %>

    <fieldset>
        <p>
            <label for="Title">Dinner Title:</label>
            <%= Html.TextBox("Title", Model.Dinner.Title) %>
        </p>
        <p>
            <label for="EventDate">Event Date:</label>
            <%= Html.TextBox("EventDate", Model.Dinner.EventDate) %>
        </p>
        <p>
            <label for="Description">Description:</label>
            <%= Html.TextArea("Description", Model.Dinner.Description) %>
        </p>
        <p>
            <label for="Address">Address:</label>
            <%= Html.TextBox("Address", Model.Dinner.Address) %>
        </p>
        <p>
            <label for="Country">Country:</label>
            <%= Html.DropDownList("Country", Model.Countries)
%>
        </p>
        <p>
            <label for="ContactPhone">Contact Phone #:</label>
            <%= Html.TextBox("ContactPhone", Model.Dinner.ContactPhone) %>
        </p>

        <p>
            <input type="submit" value="Save"/>
        </p>
    </fieldset>

   <% } %>
```

A *Create.aspx* nézetlap:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Host a Dinner
</asp:Content>

<asp:Content ID="Create" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Host a Dinner</h2>

    <% Html.RenderPartial("DinnerForm"); %>

</asp:Content>
```

Az *Edit.aspx* nézetlap:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Edit: <%=Html.Encode(Model.Dinner.Title) %>
</asp:Content>

<asp:Content ID="Edit" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Dinner</h2>

    <% Html.RenderPartial("DinnerForm"); %>

</asp:Content>
```

3.5.3 HTML segédek használata a nézetekben

A HTML segédek (HTML Helper) nem mások, mint olyan metódusok (sztring visszatérési értékkel), melyekkel megkönnyíthetjük a nézetek fejlesztését úgy, hogy egy-egy HTML elem helyett az azt előállító segédet hívjuk meg egy szkript betétben. Példa gyanánt lássuk egy bejelentkezési űrlap nézetkódját, segédekkel és nélkülük, látva a kód olvashatóság és kompaktságbeli különbségét.

A példa segédek használatával:

```
.
.
.

<body>
    <div>

    <% using (Html.BeginForm())
       { %>

        <%= Html.Label("UserName"," UserName") %>
        <br />
        <%= Html.TextBox("UserName") %>
        <br /><br />
        <%= Html.Label("Password","Password") %>
        <br />
        <%= Html.Password("Password") %>
        <br /><br />
        <input type="submit" value="Log in" />

    <% } %>

    </div>
</body> . . .
```

A segédek használata nélküli változat:

```
.
.
.

<body>
    <div>

    <form method="post" action="/Home/Login">

    <label for="userName">User Name:</label>
    <br />
    <input name="userName" />

    <br /><br />

    <label for="password">Password:</label>
    <br />
    <input name="password" type="password" />

    <br /><br />
    <input type="submit" value="Log In" />

    </form>

    </div>
</body>
. . .
```

Természetesen az alap segédeken felül létrehozhatunk saját HTML segédeket is, melyekkel gyakran használt bonyolultabb HTML kód szerkezeteket is előállíthatunk. Ehhez nem is kell mást tennünk, mint létrehozni egy publikus statikus osztályt, amelyben létrehozzuk a segédünket, egy sztring visszatérési értékű bővítmény metódus formájában.

A bővítmény metódus egy olyan statikus metódus mely publikus láthatóságú, és legalább egy paramétere van ami a bővítendő osztály típusával rendelkezik.

Példaként lássunk a <label> HTML lapelemet előállító segéd bővítmény metódust.

```
public static class LabelExtensions
{
          public static string Label(this HtmlHelper helper,
                                     string target,
                                     string text)
          {
               return String.Format("<label for='{0}'>{1}</label>",
                                    target,
                                    text);
          }
}
```

3.6 A rendszer telepítése

Miután a rendszerünk elkészült, ahhoz hogy az ügyfél infrastruktúráján hadrendbe állíthassuk, publikálnunk kell a webes alkalmazásunkat a célrendszerre. Mielőtt ezt megtennénk érdemes az *App_Data* könyvtárban található adatbázisainkat az elején létrehozott adatbázis szkriptek segítségével újra létrehozni, hogy publikáláskor egy alapállapotú rendszer kerüljön a megrendelőhöz.

Fontos megjegyeznünk azt, hogy egy ASP.NET MVC webalkalmazás a Microsoft IIS 7.0 verziójú web szerverre a legegyszerűbb telepíteni abban az esetben, ha az integrált módban fut. Régebbi verziójú, vagy klasszikus módban futó IIS esetén a szerver beállításainak jelentős átalakítására van szükség, így erősen ajánlott az integrált mód használata.

Összefoglalás

A termék kifejlesztése során akkurátusan végrehajtottuk a szükséges lépéseket annak érdekében, hogy a megrendelő egy robusztus, könnyen skálázható, jó minőségű, gyors az igényeinek teljesen megfelelő alkalmazást kapjon a kezébe.

Első körben felmértük a felhasználó igényeit a kifejlesztendő alkalmazással, majd közösen lefektettük az alkalmazás alapfogalmait, a részletes funkció listát, és a vele végrehajtandó pontos munkafolyamatokat.

Ezután felkutattuk és kiválasztottuk, az igények figyelembevételével, a leendő alkalmazásunkra legjobban illeszkedő tervezési mintákat és .NET-es technológiákat, majd ezeket felhasználva, elkészítettük a rendszer architektúrális tervét, majd elkezdtük a fejlesztést.

A kiválasztott tervezési minták a Model-View-Controller, a Repostory, és a Pipes and Filters minták lettek.

A felhasználandó technológiák pedig az ASP.NET MVC, MS SQL 2005, Windows Workflow Foundation, és a LINQ to SQL lettek.

A fejlesztés első lépéseként a modell megalkotását tűztük ki célul melynek során először létrehoztuk a szükséges normalizált táblaszerkezetű adatbázisainkat, majd ORM-et készítettünk a LINQ to SQL segítségével. Ezek után implementáltuk a Repository tervezési mintában foglaltakat, majd alkalmaztuk a Pipes and Filters módszereit a hatékony adatszűrés elérése érdekében.

Miután a modellünk elkészült, létrehoztuk az aktív jegyek életciklusát követő és kezelő állapotgép munkafolyamatunkat a Windows Workflow Foundation segítségével. Ezek után létrehoztuk a vezérlőinket, melyek az alkalmazásunk üzleti logikáját implementálják. Itt és a

nézetek fejlesztése során implementáltuk a rendszer biztonsági megoldásait, melyek segítségével egy tényleg vállalati környezetbe illeszthető megoldássá válik a rendszerünk. Sajnos ezen megoldásokról a problémakör komplexitása miatt ezen diplomamunka keretein belül nincs módomban beszámolni.

Ezek elkészültével kialakítottuk az alkalmazásunk felhasználói felületének egységes megjelenését mester oldal létrehozásával. Ezek után létrehoztuk a nézeteinket, melyek a rendszer és a felhasználó közötti interakcióit hivatottak szolgálni.

Miután elkészült a rendszerünk, leteszteltük, és a megrendelő is elfogadta, az alkalmazásunkat telepítési alapállapotra hoztuk majd publikáltuk a megfelelően beállított megrendelői webszerverre.

A folyamat ezen a pontján nincs is más feladatunk hátra, mint a végfelhasználók, a rendszer használatára való oktatása, illetve az ügyféllel kötött szerződéstől függően támogató tevékenység szolgáltatása (javításcsomagok készítése, esetleges hibaelhárítás, stb.).

Irodalomjegyzék

- Judith Bishop: C# 3.0 Design Patterns (O'Reilly)
- Paul Turley, Dan Wood: Beginning T-SQL with Microsoft SQL Server 2005 and 2008 (Wrox)
- Omar AL Zabir: Building a Web 2.0 Portal with ASP.NET 3.5 (O'Reilly)
- K. Scott Allen: Programming Windows Workfow Foundation: Practical WF Techniques and Examples using XAML and C# (PACKT)
- Rob Conery, Scott Hanselman, Phil Haack, Scott Guthrie: Professional ASP.NET MVC 1.0 (Wrox)
- Molly E. Holzschlag, Dave Shea: The Zen of CSS Design: Visual Enlightment for the Web
- http://msdn.microsoft.com – Microsoft Developer Network
- http://msdn.microsoft.com/en-us/library/dd394709.aspx - ASP.NET MVC 1.0 @ MSDN
- http://www.asp.net/mvc
- http://msdn.microsoft.com/hu-hu/practices/default(en-us).aspx – Patterns and Practicles @ MSDN
- http://blog.wekeroad.com/ - Rob Connery blogja

Köszönetnyilvánítás

Szeretnék köszönetet mondani mindazoknak, akik segítették eme diplomamunka létrejöttét külön megköszönve

Kollár Lajosnak, témavezetőmnek, tanáromnak a hasznos tanácsokért, a kitartó segítőkészségéért, a szakmai és emberi bíztatásért, nem csak a diplomamunka készítése, hanem az egyetemi évek során is.

Juhász István Tanár Úrnak, aki bebizonyította nekem, hogy képes vagyok idáig eljutni.

Juhász József Csabának, aki bevezetett a vállalati alkalmazások fejlesztésének világába, mentorált, és inspirált mind szakmailag, mind emberileg.

Végül, de nem utolsó sorban a szeretteimnek, akik mindvégig mellettem voltak.