Az MVVM és ami mögötte van

Manapság felgyorsult a világ és mély, összetett eszmefuttatások helyett pár gyors szóval szokás elintézni a felmerülő elméleti problémákat.

„Használj design pattern-t, ide jó lesz az MVVM”

Az illetőnek meg legenerál a fejlesztő eszköze egy vázat, benne elemekkel, amiket kitölt tartalommal és készen is vagyunk.

Hiszik Ők!

De az ember történelme tele van azzal, hogy keresi-kutatja a dolgok értelmét, okát és működését.
Ez a cikk is ilyesmikkel kíván foglalkozni. Kikaparni a mozaikszavak mögé rejtett okokat és módszereket, a miérteket és a helyes vagy helytelen válaszokat.

Nézzünk egy tipikus problémát.

Vannak adataink, táblák, rekordok és mezők. Hiszen a legegyszerűbb az adattárolást egy kockás-papírra rajzolt vagy Excel féle táblaként elképzelni. Egy munkalapon sorokban azonos célú, elnevezett oszlopok, ezekben az adattartalom.
Ebbe az adattárba egy programmal adatokat kell felvinnünk, vagy módosítanunk a már benne tároltakat.

Egy tipikus „régimódi” alkalmazás beolvas adatrekordokat majd módosítja ezek tartalmát a perzisztens adattár tartalmának memória reprezentációjában majd visszaírja őket a biztonságos adattárolóra.
Ezen működésre talán nagyszerű példa egy tipikus dBase, FoxBase vagy Clipper alkalmazás (de persze minden nyelven és minden környezetben lehet ilyen kódokat írni, de a fent említett nyelveken csak ilyen alkalmazást lehetett írni).

A programkódban vegyesen találhatóak terminál kezelő kódok (kiírás a képernyőre és adatok beolvasása a billentyűzetről), valamint adatkezelő (adattáblák megnyitása, pozícionálása, mezők olvasása ill. írása) műveletek, meg persze az üzleti logika és a felhasználói interakciók kezelése.
Ha valamilyen adatmódosítás elkészült, akkor azzal nem kell sokat foglalkozni (ha betartjuk az ökölszabályt és minden megnyitott erőforrást, jelen esetben adattáblát, lezárunk magunk után, ha már nem kell).
Kényelmes és könnyen használható technológia, hihetetlenül hatékony tud lenni.
Persze ahogy nő egy rendszer mérete és szaporodnak a problémák előjönnek a gondok.
Több felhasználós környezet, egyszerre módosítanak többen ugyanazokon az adatokon.
Az adatmódosításokat tranzakciókba kell vonni, hogy a belső logikai összefüggések megmaradjanak.
Aztán persze örök probléma, hogy ahogy nő a kód mérete úgy egyre nehezebben karbantartható, hiszen keverten tartalmazza a GUI és az adatkezelés kódját. Ha meg módosul az adatbázis szerkezete, akkor az közvetlen hatással van az egész kódunkra, hiszen a mi programkódunk összekeverten tartalmazza az adatkezelést, felhasználói interakciók kezelését, a GUI-t és az üzleti logikát is. Vagyis a mi kódunknak kell minden adatmező formai/tartalmi helyességét biztosítani, nekünk kell a mezők közötti összefüggéseket ellenőrizni és nekünk kell, a kódunkban a rekordok, sőt táblák közötti kapcsolatokat is ellenőrizni, betartatni. De ezek a feladatok szétszórva a kódban, összekeveredve, sok-sok duplikátummal megfűszerezve találhatóak meg.

Ha ezen ’modell’ néhány hiányosságán túl akarunk lépni, akkor kell egy igazi relációs adatbázis, benne *„táblák, rekordok és mezők”*.
Ez már tranzakcionális, többfelhasználós eszköz alapból. Mindenféle constrain-ek, triggerek és unique indexek is vannak benne, szóval van benne egy üzleti logikai szint is, nem csak tárolja az adatokat,

mint egy irattartó szekrény, de felügyel adatszinten és magasabb logikai szinten is a konzisztens adattartalomra.
Ez a lépés megoldja a problémáink egy jó részét.
De nem mindent, és mi pont azokkal kívánunk a továbbiakban foglalkozni, amelyeket nem old meg.
De még a megoldás sem tökéletes, hiszen az ellenőrzések, konzisztencia vizsgálatok adatbázisba helyezése sem váltja ki a kliens oldali ellenőrzés szükségletét.

Azzal hogy adatbázisba helyeztük a vizsgálatokat máris nagyot léptünk egy egységes feltétel rendszer felé, hiszen ha a kliens kódban végzünk vizsgálatokat, akkor lehet, hogy ez elszórtan a GUI felület számtalan pontján megtörténik. Ha meg több helyen végezzük ugyanazt a vizsgálatot (pl. a bevitt mezőérték az szám-e, vagy 1 és 854 közé eső szám-e és így tovább, egyre összetettebben), akkor ez a redundancia szükségszerűen elvezet a hibához, miszerint a vizsgálatokat végző kódok közül valamelyik téves lesz és átereszt az adattárolásra nem megfelelő tartalmú adatokat is.
Ezt a központi ellenőrzés, az adatbázisba bevitt üzleti logikával már megfogja, és ha egy „késői” hibaüzenet formájában is, de legalább jelzi. Az adatbázisból érkező hibaüzenet azért „késői”, mert a felhasználó életét nagyban megzavarja, hogy egy olyan adatmódosítására kapja az üzenetet, amelyet korábban végzett (akár percekkel is korábban, egy több lapos űrlap előző lapján) és amely adatmódosítást tartalmazó teljes munkafeladatot befejezettnek gondolta, mert pl. egy *’Kész, az adatok eltárolhatóak’* gombra kattintott.

Vagyis logikusan következik a következő lépés, az adatmezők ellenőrzése kliens oldalon is szükséges marad, de ezeket a kliensen egy központosított kódban, egyszer kell implementálnunk.
Az persze logikus, hogy ennek a kódnak valahol az adatbázis adatainak a memóriába átolvasott reprezentációjának a közelében kell lennie.
Részint mert sok esetben az adatok ismerete szükséges az ellenőrzés végrehajtásához, mivel az adatmezők, adatrekordok között összefüggések, megkötések lehetnek, részint pedig azért, mert ha az OOP gondolatmenet mélyen gyökerezik az emberben, akkor ott keresi egy adatmező ellenőrzési kódját, ahol az adatmezőt is tárolja. Tehát pl. ha egy ’ÁFA kódok’ adattáblában tárolunk ’adócsoport kód’ és ’adó százalék’ mezőket, akkor logikusan a mezők mellett (azonos osztályban) tartjuk az ezeket ellenőrző programkódot is. (Már csak azért is, mivel fejlett környezetekben nem a GUI-hoz tartozó programkód feladata, hogy előzetesen ellenőrizze egy mező formai és tartalmi helyességét és csak a megfelelő értéket írja be az azt tároló objektumba, hanem maga az adattároló objektum property-je ellenőrzi a tartalom helyességét és megszakítással „honorálja” a hibás adatbeírási kísérletet. Ez így biztonságos és databinding megoldások használatával automatikus, vagyis nem téveszthető.)

A másik problémánk, hogy ha közvetlenül az adatbázis rekord szerkezetével megegyező adatokat tartalmazza az adatbázis memóriába átemelt reprezentációja (erre példa a Delphi TDataSet, vagy az ADO.NET), akkor mindjárt két problémát is a nyakunkba veszünk. Az adatok tárolása, reprezentációja nagyon merev, kötött az sql adatbázis kezelő adattárolási, reprezentációs megoldásaihoz, amelyek természetesen idegenül hatnak egy modern programnyelv eszközeit ismerve. Másrészt, ha az adatbázisban bármilyen shema változás történik, akkor annak szükségszerű hatása a kliens program szükséges részeinek aktualizálása, ami kihat a teljes felhasználói felületre, és az azokkal kapcsolatos kódokra. Pedig létezhetnek esetek, amikor az adatbázisban levő üzleti logika változása és az ezzel járó adatbázis shema változása nem érinti a kliens program működési logikáját, vagyis csak kisebb mértékű és elkülönülten lekezelhető programváltoztatás is elegendő lenne.

Ezen okok miatt logikus, hogy az adatbázisból kapott adatok tárolása a kliens programban a programnyelvi lehetőségeket legjobban kihasználó módon, jelen esetben objektumokban (osztályokban), és az objektumokból szervezett tárolókban történjen.
Ha pedig az adatbázis adatmezők a kliens program objektumainak mezőibe másolódnak, akkor logikusan az ezen mezőket kezelő minden függvény (pl. tartalom ellenőrző függvények) is ezen osztályok elemei legyenek, centralizáltan, egybegyűjtve, rendszerezve.

Ha betükódokkal akarnánk bűvészkedni, ezt a patternt MV-nek hívhatnánk. Model-View, két rétege a programnak. A modellben az adatkapcsolati réteg és a teljes kliens oldali üzleti logika.
A View tartalmazza az adatmegjelenítés, kezelés, manipulálás kódjait, de semmit nem tud igazán az adatbázisról és igen keveset az üzleti logikáról. Csak a modell számára publikált interfacejával tart kapcsolatot, annyit ismer.

A Model számára megfelelő megközelítést adnak az ORM megoldások, pl. az Entity Framework vagy a Hibernate.
Ezek már nem csak az adatelérést és az adatok memória objektumba másolását segítik, de lehetővé tesznek „mapping” műveletet is, vagyis hogy az adatbázis shema és a kliens programban használt objektum modell akár radikálisan eltérjen.
A lehetőség adott, de mégis, ennek erőteljes kihasználása sok probléma forrása lehet, további gondok keletkeznek ennek használatából, ha nagyon erőteljesen a kliens oldali logikára akarjuk kényszeríteni az objektum modellt és ez nagyon eltér az adatbázis szervezésétől.

Sőt, további probléma lehet, hogy egy azon kliens programban is akár az adatbázis shema szervezésétől, logikájától többféleképpen eltérő és egymástól is különböző szemléletben lehet szükségünk az adat objektumokra. Ebben az esetben az ORM mappingje sem ad segítséget, hiszen az is csak egyféle „eltérítést” tesz lehetővé az adatok objektum reprezentációjára.

Most egy pici kitérő következik az adatfelvitel nyugtázására, majd visszatérünk ehhez a fő sodorhoz, hogy összefogva a két szálat, rátérjünk az MVVM „mágiára”.

Néhány esettől eltekintve (amelyek engem feszélyeznek, mivel nem értem a szituációt és keresem a ’teendőmet’) az adatfelviteli felületen valamilyen nyugtázás és elutasítás gomb is megtalálható.
Ez megfelel az adatbázis műveletek commit/rollback funkcionalitásának.
Az adatfelvitel folyamán (ritka kivételektől eltekintve) egymással összefüggő adatmezők felvitele ill. módosítása történik. Mivel ezek a mezők egymással függőségben vannak, az adatbázisban történő módosításaikat is egyszerre szükséges végrehajtani.
Persze ’régen’ az volt a szokás, hogy az adatmódosítások mehettek azonnal az adatbázisba és legfeljebb nem kerül megerősítésre (commit), ami által a félig felvitt adattartalom visszaforgatható a felírás előtti állapotba (rollback), ami meg még felírásra sem került, az egyszerűen elvész.
Ez a működési mód túlhaladott lett, amikor az adatelérési komponensek az állandó kapcsolat típusról átalakultak egy kapcsolat nélküli üzemmódra. Vagyis valamiféle bufferban kell tárolnunk az adatainkat. Erre a felhasználói felület komponenseinek mezői nem alkalmasak, mivel egyrészt szétszórtak és rendezetlenek, vagyis pont a megkívánt funkcionalitásnak és rétegekbe szervezésnek mondana ellent, másrészt bonyolult szervezettségű és nagy mennyiségű adatokat is szükséges lehet tárolni (grid, tree), ami miatt adatszerkezetekbe helyezett objektumok az ideális tároló eszközök (hiszen a gyakorlati megvalósításokban a GUI elemek csak a tárolókban szereplő adatok egy részét, az éppen láthatóakat jelenítik csak meg, időről-időre frissítve a megjelenített adatok körét).
A kapcsolat nélküli adatelérés miatt az összes adatmező módosításnak a cliens oldalon kell maradnia és csak a jóváhagyáskor kerülnek ezek, csomagban-egyszerre felvitelre az adatbázis perzisztens tárolóra, pl, adatbázis szerverre.
Ugyan a programokban az adatbázis kliens oldali reprezentációja is lehetne az a puffer, ami tárolja a folyamatosan módosuló adatokat, de ez nem célszerű mivel az adatok módosításának visszavonása így nagy nehézségekbe ütközne, másrészt lehetetlen is, ha nem csak modális ablakokkal illetve szálak nélkül kívánunk dolgozni az alkalmazásban. Nem modális ablakok esetén illetve szálak használatával ugyanis a függetlenül futó más-más kódok (egy másik nem modális ablak, amire átválthatunk vagy egy függetlenül futó másik szál) nem végleges, de megváltozott adatokat is látnának (mint a relációs adatbázisokban a dirty módosítás státusz, amely persze már évtizedek óta nem ’divatos’ egyetlen valódi és komoly adatbázisban sem).
Vagyis célszerű a felhasználói felület adatait saját tároló objektumokban és tárolókban tartani, amelyek csupán másolatai az adatbázis memóriabeli reprezentációjának. Ezekhez a közbülső buffer objektumokhoz érdemes kapcsolni (binding) az adatmegjelenítést és az adatvisszaírást.
Ezzel az érvénytelenítés (adatmódosítások eldobása) nagyon egyszerűvé válik. Egyszerűen nem mentjük el a buffer tartalmát a helyére. Ha meg jóváhagyja a felhasználó a módosításokat, akkor ezt a buffert lehet áttölteni az adatbázis memória reprezentációjába, hogy onnan azonnal (azonnali committel) bekerüljön a permanens tárolóba vagy az alkalmazás valamilyen más logika alapján később tárolja el azt.

Ez a közbenső buffer nem csak az alkalmazáson belüli tranzakcionális és csoportos adatmozgatás miatt célszerű, de pl. könnyedén lehetővé teszi a módosított adatok helyett az ’eredeti’ (memória reprezentációban=Model) tárolt adatokra visszatérést (pl. az adatokba véletlen begépelés esetén), hiszen a módosítások tevőleges véglegesítéséig rendelkezésünkre áll a ’módosításmentes’ eredeti adat is a Model-ben. Ezzel a trükkel sokat segíthetünk az ORM rendszerek egyik nagy gondján az adatszennyeződésen, vagyis hogy a memóriában szereplő adatbázis reprezentáció tartalma eltér az adatbázis eredeti tartamától. (Ez a gond, persze sok más okból is előfordulhat, de már az eggyel kevesebb ok is valami.)

Most visszatérve az eredeti témánkhoz.

A MVVM mozaikszó a Model – View – ViewModel fogalmak összekapcsolásával keletkezett.
Ez egy 'architectural pattern' a modern esemény vezérelt, a felületet valamilyen markup nyelvvel leíró alkalmazások számára. Vagyis a korábban részletezett paradigmákra jó megoldást adó megoldási módszertan.

A 'Model' maga az adatbázis tartalma, amellyel összetartozik az alkalmazásunk, és ez a modell nagyon változó megvalósítású lehet. A legegyszerűbb módon az adatbázis tábláinak reprezentációjától (Delphi TDataSet, Ado.Net) a DLinQ féle félig ORM megközelítésen át a teljes ORM logikájú elvonatkoztatott adattárolásig (EF, Hibernate), vagyis a technikai megoldása sokféle módon elképzelhető. A Model értelem szerűen az adatok az egész alkalmazásra vonatkozó reprezentációja, üzleti logikája és áramlásának vezérlője.
A megvalósítás jelen írás szempontjából lényegtelen, de természetesen megvan a saját természetes fejlődése és az egyre összetettebb technológiák előnyei (meg persze a járulékos hátrányok is), amely téma egy külön, önálló írást igényel.

A 'View' a GUI felület, egy önálló interaktív alkalmazási egység (ritkább esetben valamilyen önállóan működő, és eredményt szülő, illetve eseményvezérelten működő kódrész).
Tipikus esetben egy megjelenítési és/vagy adatbeviteli felület.

A 'ViewModel' tulajdonképpen egy 'nézete' az adatbázis tartalomnak, pontosabban ami abból a Model-en, mint önálló rétegen át látszódik. Tulajdonképpen tartalmilag, mennyiségileg szűrt és a View-hoz igazított szerkezetű valamint gyakran konvertált adattartalom, csak azokat az adatokat tartalmazza és csak abban a szerkezetben, amiben a View GUI felületnek a működésére szüksége van. Ez egy olyan közös metszet, ami csak a GUI igényeinek megfelelő adatokat tartalmazza, és a Model-ből csak annak egy részhalmazát mutatja csupán, miközben nem teremt akadályt arra sem, hogy transzformálja az adatait, vagyis ne teljesen azonos adathalmazt tároljon az eredeti Model-el, hanem annak akár szerkezetileg, formailag, vagy tartalmában módosított részét.

Így a View igényeinek megfelelő adattartalom jelenik meg, miközben a konzisztencia megmarad a Model adattartalmával, hiszen mind a kiolvasás, mind a visszaírás egy egykapus rendszerben történik, amely gondoskodhat a transzferben a konverziókról is.
A View nem látja a Model-t, csupán a számára mutatott ViewModel rétegen át kommunikál vele. Eközben a ViewModel képes önálló bufferként is tárolni az adatokat a két szélső réteg között vagy csak egy interface, amin keresztül közvetlenül áramlanak az adatok, és csak konvertálásokat, adatszűréseket és megfelelő adatszerkezet kialakítást végez.

A működés a következő:

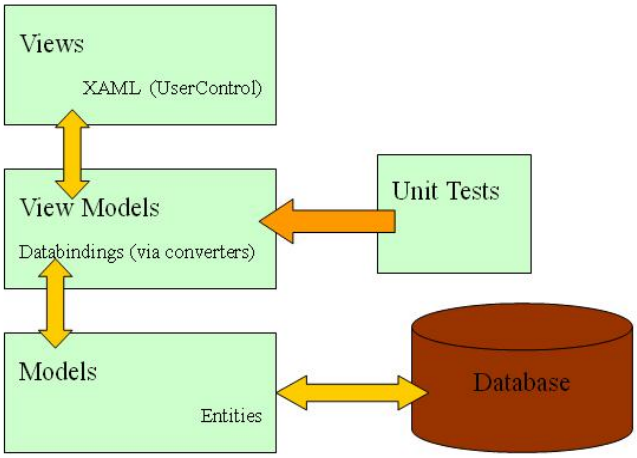
Ha elindítunk egy új View-t megvalósító feladatot, akkor ahhoz létre kell hoznunk a szükséges ViewModel-t is és átadni a View-nek. Természetesen a View kódja tartalmazhat olyan dinamikus adatelérést, amikor a ViewModel mint egy közvetítő, egy adott interface-n keresztül dinamikusan építi fel a kért adatobjektumokat a Model-ből, annak objektumait és eljárásait használva.

A View a kapott ViewModel objektumokkal kommunikál, onnan szerzi az adatokat és oda tárolja el a felhasználói interakcióból nyert adatokat.

A View bezárásakor a ViewModel adatai, ha nem közvetlenül mozgatta át a módosításokat, hanem valamiféle közbenső buffert képzett, akkor mintegy tranzakcionálisan, egységbefoglaltan és atomian kezelve kerülnek át a Model-be. Vagyis a ViewModel el is szeparálhatja az adatokat, egyfajta bufferként funkcionálhat, vagyis ha a View interakció *'Kész, felvihető'* művelettel zárul, akkor a felvitt

vagy módosított adatok tovább áramlanak a Model-be, ha pedig *'Mégsem'*-el zárul a művelet, akkor eldobásra kerülnek a ViewModel adatai.

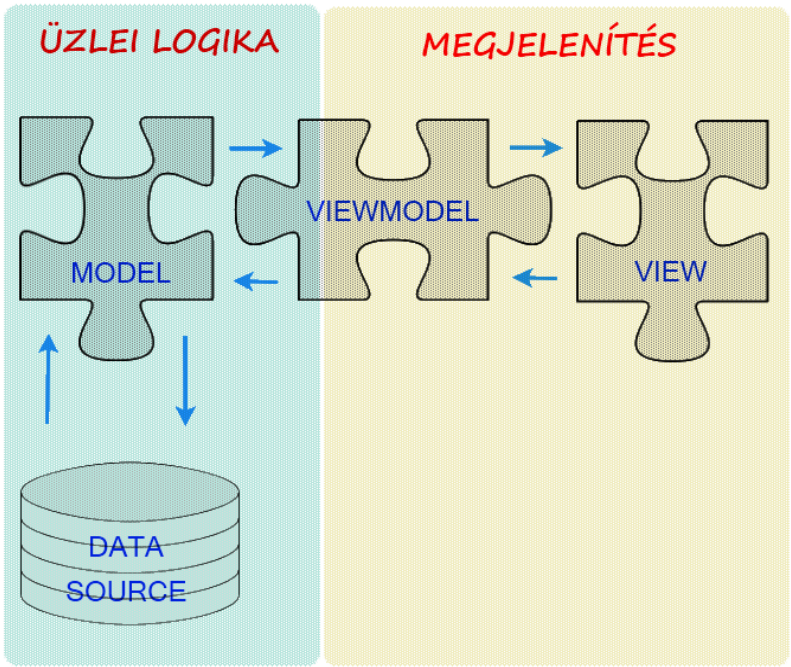
Az MVVM-nek is, mint minden technológiának, nem csak előnyei, hanem hátrányai is vannak természetesen.
Ezzel a rétegezéssel egyre távolabb kerül a kliens programkód és az adatbázis tartalma, így egyrészt jelentősen több munka felépíteni a rétegeket és gondoskodni az adatáramlásról a rétegek között, másrészt ebből is következően megnő a hiba lehetőségek száma (más jellegűeké meg csökken) és az „egész" áttekintése, a fejlesztő rálátása is csökken, ami frusztráló, hiszen vakon kell használni egy részletét az egész problematikának. Ha meg a tervezéskor még nem átlátott bővítése szükséges a ViewModel-nek, akkor ez esetleg nem csak egy gyors lépés (mint a korábbi technológiák ad-hoc adatelérése), hanem több réteget is érintő újratervezés illetve ezzel a bővítéssel vagy változtatással is módosított kódok generálása, ha olyan eszközünk van.



Íme ahogy a WPF vagy Silverlight illetve Win8 (Metro)/WinPhone alkalmazásoknál az MVVM működését ábrázolni szokták.

Az adatbázis a Model-hez van kötve, ami ugye az adatbázis programbeli reprezentációját jelenti.
A Model és a ViewModel között oda-vissza adatáramlás zajlik.
Ugyanígy a View és a ViewModel között is oda-vissza adatáramlás zajlik.

Én egy másik ábrát is alkottam a könnyű és gyors megértéshez.
(Ugye a nagy igazság: egyetlen kép többet mesél, mint ezernyi szó, itt is megáll.)



Ez az ábra főként a ViewModel-nek az ***üzleti logika*** és a ***megjelenítés*** rétegek közötti elhelyezéséről szól.
A View és a Model szerepe nagyon egyértelmű a szokásos ábránál is, de a ViewModel elhelyezkedése az alkalmazás ezen két végpontja között nem annyira egyértelmű.

Mint az ábrán látható, a ViewModel főként ill. nagyrészt a megjelenítési réteghez kapcsolódik és ily' módon a kódszerkezete a View szerkezetéhez kapcsolódik erősebben.
Gyakorlatilag minden View-hoz saját önálló ViewModel kapcsolódik. [Persze lehetnek kivételek, ha ugyanazon adatkört ugyanolyan adatmegjelenítési tartalommal, de másképp, más formában ill. más szemléletben kívánunk megjeleníteni. Ekkor ezen két különböző View osztozhat a közös egyetlen ViewModel-en.]
Ezzel szemben átlagos esetben egyetlen nagy Model-t használunk, amelyik kiszolgálja adattartalommal az alkalmazás összes ViewModel-jét.
[Természetesen speciális esetben egy alkalmazásban is lehet több Model-ünk, mert vagy több elkülönülő adatbázishoz kapcsolódik az alkalmazásunk vagy egy adatbázist valamilyen okból több szemléletben is el akarunk érni. Pl. az adatbázis más-más (egymást azért átfedhető) részeit önálló modellekbe szervezzük.]

A ViewModel csak olyan adatokat tartalmaz és olyan szerkezetben, ami a hozzá kapcsolódó View számára szükséges. Ezért technikailag igen erősen kötődik a View réteghez.
Az én ábrámon azért nyúlik át kissé a Model „térfelére", mivel a ViewModel egy specializált, leszűkített-átszervezett-átstrukturált, néhol az adattartalmat átkódoló adatbázis modell. Csak nem közvetlenül az adatbázishoz kapcsolódik, nem kötelezően annak szerkezetéből veszi az adatszerkezetét, hanem a virtualizált Model-t használja. De ugyancsak valamilyen üzleti logikához illeszkedő, csak a konkrét használathoz még jobban idomuló, a konkrét megjelenítési, adatbeviteli igényekhez jobban illeszkedő üzleti logikát, üzleti adatmodellt valósít meg. Ez ugyan nem az egész adatbázis üzleti modellje, hanem a már megalkotott üzleti modell egy igen kicsiny szeletének áttervezése, átszervezése, de ez is egyfajta másodszintű üzleti modellt tartalmaz, az eredeti üzleti modellre ráültetve.

Egy apró kiegészítés (eltévelyedés) a témához.
Egy másik 'varázs-szó' az MVC, ez főként weblapok esetén használt pattern.
Jelentése Model-View-Conroler.
A View itt nagyjából hasonló jelentéssel bír, de tipikusan az MVC féle VIEW az HTML kód, míg az MVVM féle VIEW inkább a WPF, Silverlight vagy mobiltelefonok megjelenítési rétege. Persze lehet HTML is vagy akár WinForm. Csak jó programkód kérdése.
A Controler ebben az esetben egy vezérlő, irányító kód, amely a felhasználói interakciók / url kérések következtében automatikusan választódik ki és indul el. Ez választ VIEW-t, aminek átadja a megfelelő adat reprezentációjú és előkészített (pl. adatokkal előtöltött) Model-t.
Ez a Model azonban (bár a leírások tipikusan erre nem térnek ki) valójában a gyakorlatban célszerűen két rétegre bontható illetve úgy is használják (a jobbak). Az MVC Model-je ilyen értelemben megfelel a ViewModel feladatnak, hiszen az adott View-hoz illeszkedik és jó esetben ez a Model is egy elvonatkoztatása a valós adatbázist lemodellező memória reprezentációnak, mégpedig a VIEW-hoz illeszkedő szerkezettel és tartalommal. Ebben az esetben az MVC Model-je tartalmazza az MVVM-ben látott mindkét réteget, de értelem szerűen a jó programozó ezt is egyértelműen kettéválasztja, mint ahogy az MVVM Model és ViewModel megbontásánál látható.

Ha a fenti szöveget vagy valamely részét weblapon kívánja elhelyezni html formátumban, kérem az alábbi licence információkkal is bővítse a weblapot:

```
<a rel="license" href="http://creativecommons.org/licenses/by-nc-
sa/3.0/deed.hu"><img alt="Creative Commons Licenc" style="border-width:0"
src="http://i.creativecommons.org/l/by-nc-sa/3.0/88x31.png" /></a><br /><a
xmlns:cc="http://creativecommons.org/ns#" href="http://www.emel.hu/doc"
property="cc:attributionName" rel="cc:attributionURL">Moravecz László alias
eMeL</a> <span xmlns:dct="http://purl.org/dc/terms/" property="dct:title">Az MVVM
és ami mögötte van</span> című műve <a rel="license"
href="http://creativecommons.org/licenses/by-nc-sa/3.0/deed.hu">Creative Commons
Nevezd meg! - Ne add el! - Így add tovább! 3.0 Unported Licenc</a> alatt
van.<br />Based on a work at <a xmlns:dct="http://purl.org/dc/terms/" href="
http://www.emel.hu/doc" rel="dct:source">www.emel.hu/doc</a>.<br />Permissions
beyond the scope of this license may be available at <a
xmlns:cc="http://creativecommons.org/ns#" href="http://www.emel.hu/doc"
rel="cc:morePermissions">www.emel.hu/doc</a>
```