Debreceni Egyetem Informatikai Kar

Agilis adatbázisok

Dr. Juhász István
egyetemi adjunktus

Papp Béla
programtervező informatikus
MSc hallgató

Debrecen, 2011.

Tartalomjegyzék

1. Bevezetés

Napjaink a változásokról és az azokhoz való alkalmazkodásról szólnak. Minden kicserélődik körülöttönk; újabb és újabb eszközök színesítik, gazdagítják és formálják mindennapjainkat. Ebből a sorból a számítástechnika, az informatika és ezáltal a szoftverfejlesztés sem marad ki. Újabb és újabb szoftverek jelennek meg, amik több dologban is eltérnek az elődeiktől! Más felhasználói felülettel rendelkeznek, más - vagy több - operációs rendszeren futtathatóak, és ami nem olyan szembetűnő - de annál fontosabb -, nagyon rövid idő alatt készülnek el! Köszönhető ez annak, hogy a felhasználók „türelmetlenek”, várják az újat és szelektálják a kevésbé használható alkalmazásokat.

Ez az öngerjesztő folyamat arra inspirálja a programozással és informatikával foglalkozó szakembereket, hogy olyan eljárásokat, módszertanokat fejlesszenek ki, amelyekkel gyorsan, olcsón és hatékonyan lehet alkalmazásokat fejleszteni.

Dolgozatomban az egyik eredményt - az agilis programozási módszertan t- mutatom be, kitérve annak az adatbázisokra vonatkozó specialitásaira. Természetesen nem lehet teljes képet festeni, hiszen az agilitás a részletekben és a folyamatban mutatja meg az igazi erejét.

Röviden ismertetem az agilis projektmenedzsment témakörét, ami kiegészíti és támogatja a műszaki vonatkozásokat. Ismertetem a projektvezetés, a projekttagok szerepkörét, valamint bemutatom azt a folyamatot, amit agilis projektmendzsmentnek nevezünk. Külön kitérek azokra a speciális tulajdonságokkal bíró megbeszélésekre, amik ennek a módszertannak a sajátosságait adják.

Természetesen nem lehet egy módszertant úgy bemutatni, hogy csak az előnyeit láttatjuk, figyelmet kell fordítanunk a korlátaira is. Az agilis módszertan kapcsán összehasonlítom az AMDD és a TDD eljárásokat, bemutatom a köztük fennálló lényeges különbségeket.

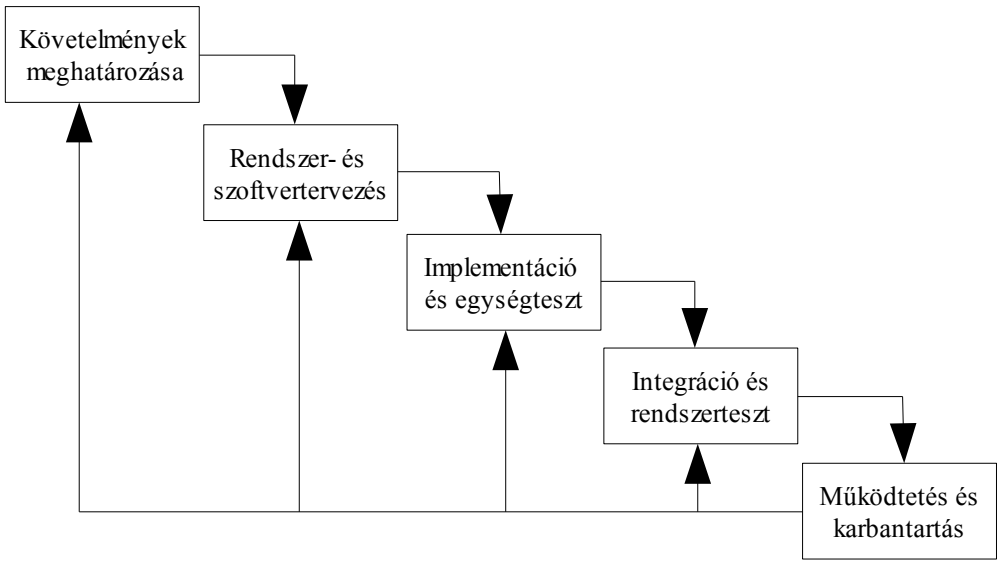
Dolgozatom célja, hogy betekintés adjon az agilis adatbázisok elméletébe, felkeltse a figyelmet egy új, még formálódó módszertan iránt.

2. Agilis szoftverfejlesztés. Az előzmények. [1]

Az agilis szoftverfejlesztési paradigma egy hosszú folyamat eredményeként született meg. A folyamat kezdete a számítógép programozás kezdetével esik egybe. Az első szoftverek megírása - mai szemmel nézve - nélkülözte a tervszerűséget. Nem voltak paradigmák, eljárások, tervezési minták! Adott volt a feladat, aminek a végeredménye egy futó program lett. A fejlesztés a kódolással indult. A számítógépek és a programozás fejlődése felgyorsult, egyre nagyobb programok, szoftverek előállítása vált szükségessé. Erre a robbanásszerű fejlődésre válaszul jelentek meg az első programozási, programtervezési megoldások. Ezek élettartama és használhatósága az idők folyamán igazolódott, vagy egyes esetekben az eljárás „kihalásához” vezetett.
A napjainkban is használatos eljárások rövid ismertetésére és összehasonlítására szükség van ahhoz, hogy az agilis módszertan kialakulásához vezető utat, az igény megfogalmazását megérthessük.

2.1 A vízesés modell

A vízesés vagy szoftveréletciklus modell az első publikált modellek közé tartozik. Az 1. ábrán az életciklus modellje látható.



1. ábra: A vízesés modell

A modell szakaszai a következők:

1. *Követelmények elemzése és meghatározása*

   A rendszer felhasználóival történt egyeztetés(ek) után kialakulnak a rendszer határai, megszorításai és céljai, melyeket a későbbiekben pontosítanak és kifejtenek. Ez szolgáltatja a rendszerspecifikációt.

2. *Rendszer- és szoftvertervezés*

   Itt történik a rendszerarchitektúra kialakítása, a szoftver és hardverkövetelmények szétválasztása, a rendszerabsztrakciók és kapcsolatok azonosítása.

3. *Implementáció és egységteszt*

   A programok és programegységek összességeként realizálódik a szoftverterv. Az egységteszt azt ellenőrzi, hogy minden egyes egység megfelel-e a specifikációnak.

4. *Integráció és rendszerteszt*

   Megtörténik a rendszerkomponensek integrációja és teljes rendszerként történő tesztelése. Ellenőrzésre kerül, hogy a rendszer megfelel-e a rendszerkövetelményeknek. A tesztelés után átadható a megrendelőnek.

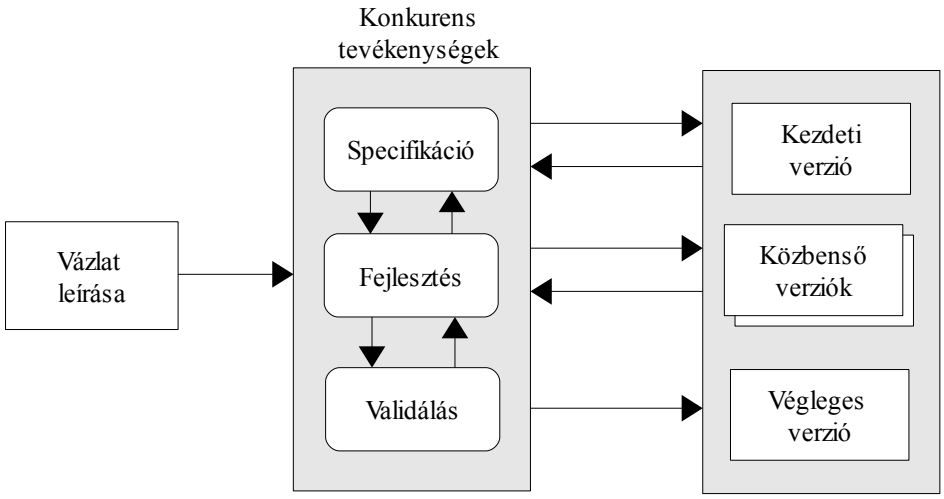
5. *Működtetés és karbantartás*

   Ez a fázis lehet a szoftver életciklusának leghosszabb fázisa. A rendszer telepítése és használatbavétele megtörtént! Az olyan hibák javítása történik, amik a korábbi teszteléseken nem derültek ki, illetve implementációs hiányosságok pótlása és szolgáltatások továbbfejlesztése történik.

Az eddigiekből kiolvasható, hogy a vízesés modell egy statikus megközelítése a szoftverfolyamatnak. Az egyes fázisok végeredménye egy dokumentum, amelynek aláírása jelzi a fázis végét. A fázisok csak a megelőző fázis befejezésekor indulhatnak! A menet közben felmerülő problémák elhárítása és megoldása igen költséges lehet. Napjainkban ezt a modellt csak nagy projektek esetén alkalmazzák.

2.2 Az evolúciós fejlesztés

A modell alapötlete az, hogy gyorsan ki kell fejleszteni egy kezdeti implementációt, amit a megrendelő véleményez, majd ezt továbbfejlesztve sok közbenső verzión keresztül jutunk el a végleges verzióig. A modell a 2. ábrán látható.



2.ábra: Az evolúciós fejlesztés modellje

A modell előnye a jobban megvalósított párhuzamosság és a visszacsatolások miatti rugalmasság.

Az evolúciós modellnek két típusa ismeretes:

1. *Feltáró fejlesztés*

   A megrendelővel történő egyeztetések során feltárásra kerülnek a követelmények, és kialakul a végleges rendszer. A fejlesztést a rendszer ismert részeivel kezdjük, és a fejlesztés során a megrendelő által kért újdonságokat adjuk hozzá a rendszerhez, ezzel közelítve a végleges verziót.

2. *Eldobható prototípus készítése*

   Ebben az esetben nagy figyelmet kell fordítani arra, hogy minél jobban megértsük a felhasználó igényeit, és azokra alapozva definiáljuk a követelményeket. A prototípus próbálgatásos módszerrel jön létre, és az ügyfél által adott követelmények kevésbé érthető részeire kell koncentrálnia.

A modell rugalmasságából adódóan ez egy hatékonyabb módszer, mint a vízesés modell.
A specifikációban történő változásokat hamarabb és hatékonyabban tudja lekezelni, mint a vízesés modell. A felhasználói igényeket gyorsabban tudja lekövetni, és gyorsabban szolgáltat eredményt is. A használata során az alábbi problémákkal találkozhatunk:

1. *A folyamat nem látható*

   A fejlődés méréséhez gyorsan leszállítható verziókra van szükség, ami azt eredményezi, hogy az összes verzióhoz tartozó dokumentáció előállítása költséges.
2. *A rendszerek gyakran szegényesen strukturáltak*

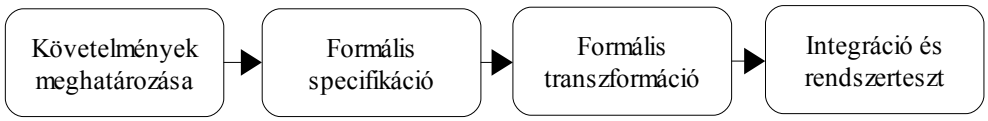
   A gyakori változtatások hatására leromlik a szoftver strukturáltsága, nehezen olvasható kód keletkezik, a változtatások összevonása nehézkessé válik és költséges.
3. *Speciális eszközökre és technikákra van szükség*

   A gyors fejlesztéshez szükséges eszközök inkompatibilitáshoz vezethetnek egyéb eszközökkel vagy technikákkal, és speciális ismereteket igényelnek.

Ezzel a módszerrel közepes méretű rendszerek fejlesztése lehetséges. Nagyobb rendszerek esetében az evolúciós módszer korlátai miatt már problémákba ütközhetünk. Ebben az esetben a vízesés- és evolúciós modell együttes alkalmazása hozza meg az eredményt.

2.3 Formális rendszerfejlesztés

A formális rendszerfejlesztés olyan megközelítés, ami kapcsolódik a vízesés modellhez, de az alapja a rendszerspecifikáció futtatható programmá történő transzformálása matematikai eszközökkel. A folyamatot a 3. ábra illusztrálja.



3. ábra: A formális rendszerfejlesztés folyamata

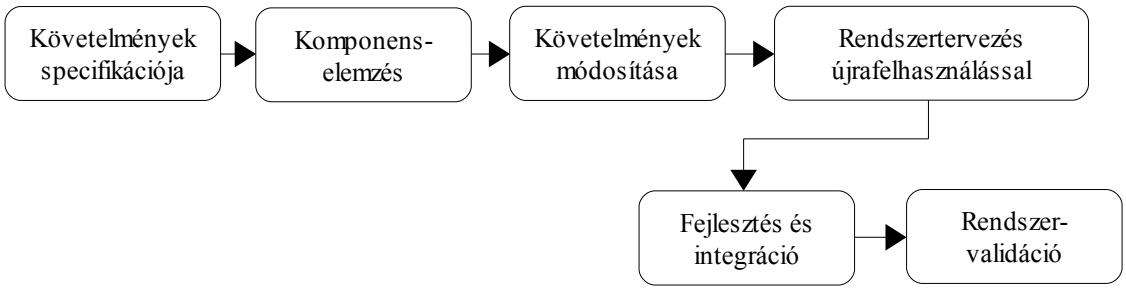
A formális és vízesés modell közötti különbségek:

1. A szoftverkövetelmények specifikációit részletes, matematikai jelölésekkel leírt formális specifikációként kell megadni.
2. A tervezés, implementáció és az egységteszt folyamatokat egy transzformációs folyamat helyettesíti, ami további transzformációkon keresztül finomítja a formális definíciót, amíg az programmá nem válik.

A módszer tehát egy jól definiált matematikai utat használ a specifikáció és a futtatható program közötti út bejárásához. A matematikai levezetés biztosítja, hogy gyakorlatilag nincs szükség az elkészült program helyességének ellenőrzésére, hiszen a transzformáció során végrehajtott átalakításokkal az nem sérülhet. Ennek következtében ez a módszer nagy biztonsági igényeket támasztó projektekben alkalmazható. További tulajdonsága, hogy a rendszer előállítása során felmerülő költségekben nem jelentkezik nagy eltérés más megközelítésekhez képest.

2.4 Újrafelhasználás-orientált fejlesztés

A szoftverfolyamatok többségében megtalálható valamilyen szintű újrafelhasználás. A gyakorlatban ez azt jelenti, hogy a projekten dolgozók ismernek olyan rendszert, tervezést, kódot, amely az aktuális projektben felhasználható és ezt alakítják a szükséges formára. Az újrafelhasználás tehát egy eszköze lehet a fejlesztés gyorsításának. Az elmúlt években jelent meg egy olyan szoftverfejlesztési megközelítés (komponens alapú szoftverfejlesztés), amely az újrafelhasználáson alapul. Az újrafelhasználás alapú fejlesztés az elérhető komponensekre és azok egy szerkezeti egységbe történő integrációjára alapul. A fejlesztés általános folyamatmodellje a 4. ábrán látható.



4. ábra: Az újrafelhasználás - orientált fejlesztés folyamata

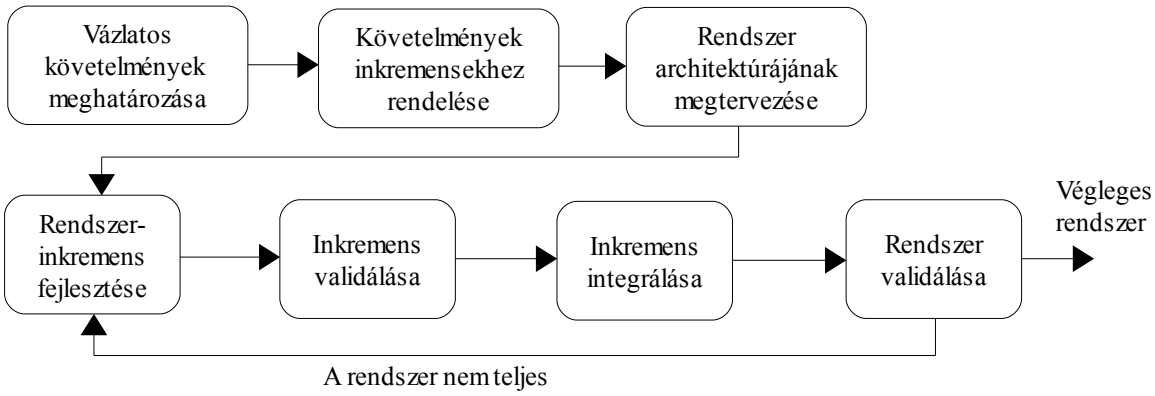
A követelményspecifikációs és validációs szakasz összehasonlítható más folyamtokkal, a köztük lévő szakaszok azonban különböznek az újrafelhasználás-orientált fejlesztésben.
A szakaszok:

1. *Komponenselemzés.* Adott a követelményspecifikáció, és keresni kell olyan komponenseket, amelyek implementálták azokat. Legtöbb esetben nincs egzakt illeszkedés, a komponens csak a funkciók egy részét biztosítja.
2. *Követelménymódosítás.* A megtalált komponensek információit felhasználva elemezni kell a követelményeket, és a komponenseknek megfelelően módosítani azokat. Ahol nem lehetséges a módosítás, ott a komponenselemzéshez kell visszafordulni.
3. *Rendszertervezés újrafelhasználással.* A rendszer szerkezetének kialakítása történik ebben a szakaszban. Figyelembe kell venni egy esetlegesen meglévő szoftvervázat, illetve azokat a komponenseket, amiket újra fel akarunk használni, illetve be akarunk építeni a rendszerünkbe. Ha nincsenek elérhető újrafelhasználható komponensek, akkor új szoftver kifejlesztése is előtérbe kerülhet.
4. *Fejlesztés és integráció.* A nem megvásárolható komponenseket ki kell fejleszteni, és egy COTS (dobozos) rendszerrel integrálni kell. Az integráció ebben a fázisban inkább a fejlesztési folyamat részét képezi.

A módszer előnye, hogy az újrafelhasználással időt - és adott esetben pénzt - spórolhatunk, de mindenképpen figyelemebe kell vennünk, hogy az újrafelhasználás hozadékaként kompromisszumokat kell kötnünk, amely nagyban befolyásolhatja az elkészült rendszerünk minőségét és a felhasználói követelmények teljesülését. Az evolúció során bekövetkező változások pedig azt okozhatják, hogy elvesztjük a rendszerünk feletti kontrollt.

2.5 Inkrementális fejlesztés

A vízesés modell néhány alapvető jelentőségű kérdésben megköti mind a megrendelő, mind a fejlesztő kezét! A megrendelőtől elvárja, hogy a fejlesztés és tervezés megkezdése előtt véglegesítse a specifikációt, hiszen a modell rugalmatlansága miatt a későbbi változtatások nagyon megnövelik a projekt időtartamát és jelentős költségvonzattal járnak. A fejlesztőtől pedig az implementáció megkezdése előtt követeli meg a tervezési stratégiák kiválasztását. Így egy robusztus rendszert kapunk eredményül, ami alkalmatlan a változások követésére.
Az evolúciós megközelítési mód ezzel szemben megengedi, hogy bizonyos kérdésekben elodázzuk a döntést, de ezzel felvállaljuk a nehezen karbantartható rendszer és a gyengén strukturáltság veszélyét.
Az inkrementális megközelítésben az említett két modell előnyeit alkalmazzuk, amit az 5. ábra mutat be.



5. ábra: Az inkrementális fejlesztés folyamata

Az inkrementális folyamatban a megrendelő rendelkezik azzal a szabadsággal, hogy a specifikáció mély részleteit csak később, a tervezés megkezdése után adja meg. Célszerű a magas prioritású funkciókat előbb implementálni! Az első inkremensek tehát a fő funkciókat implementálják, majd a továbbiakban a kisebb prioritású funkciókkal, illetve apróbb funkcionalitásokkal bővítve a rendszert jutunk el a végleges verzióig.
Az inkrementális fejlesztés előnyei:

1. A megrendelőnek nem kell végigvárni az egész folyamatot, hiszen már az első

inkremens nyújtja a fontosabb funkciókat és menet közben is használható.

2. Az első inkremens-használat során tapasztalatot szerez a felhasználó, így további információt adhat a fejlesztőnek a későbbi inkremensek fejlesztéséhez.
3. A teljes projekt kudarcba fulladásának kicsi az esélye, hiszen a felmerülő hibák a későbbi inkremensekben javításra kerülnek.
4. Ha a legfontosabb szolgáltatások implementálása készül el leghamarabb (már az első inkremensben), akkor a fejlesztés során ezen funkciók tesztelése minden egyes inkremens tesztelésekor megtörténik, így biztosítva a magas prioritású funkciók hibamentességét.

Az inkrementális fejlesztésnek is megvannak a hibái! Az inkremenseknek relative kis méretűnek kell lenniük, és minden inkremensnek szolgáltatnia kell valamilyen rendszerfunkciót. Ez azt vetíti előre, hogy nehézkes lehet a megrendelő követelményeinek megfelelő inkremensekre bontása. Az inkrementális fejlesztésből alakult ki az extrém programozás, amely a dolgozatom fő témáját jelentő agilis fejlesztés alapja.

2.6 Spirális fejlesztés

Ez a fajta modell széles körben ismert és alkalmazott. A szoftverfolyamatot nem a tevékenységek és a közöttük található esetleges visszalépések sorozataként tekinti, hanem egy spirálként. Minden egyes körben a spirál a folyamat egy fázisát reprezentálja. A legbelső kör a megvalósíthatósággal, a következő a rendszer követelményeinek meghatározásával, a következő a rendszer tervezésével … foglalkozik.

A spirál minden egyes ciklusát négy szektorra oszthatjuk fel:

1. *Célok kijelölése.* Az adott fázis által kitűzött célok azonosítása, a megszorítások definiálása, menedzselési terv felvázolása, kockázatok megismerése, alternatívák tervezése.
2. *Kockázat becslése és csökkentése.* A megismert kockázatok és azok hatásainak felmérése, és felkészülés a hatások csökkentésére. A nem megfelelően kialakított követelményrendszer esetében például egy prototípusrendszer fejlesztése válhat szükségessé.

3. *Fejlesztés és validálás*. A kockázat kiértékelése után választani kell egy fejlesztési modellt. Az eddig ismertetett modellek alapján például a biztonságkritikus rendszerek esetében a formális modell lehet a legalkalmasabb. A vízesés modellt az alrendszerek illesztésénél felmerülő kockázatok esetében érdemes alkalmazni.
4. *Tervezés*. A projekt áttekintésével döntést kell hoznunk a folytatásról. Léphetünk-e a spirál következő ciklusára, vagy sem? Ha a folytatás mellett döntünk, fel kell vázolni a projekt következő fázisát.

Az eddigi módszerektől eltérően a spirális modell nagy hangsúlyt fektet a kockázat kezelésére, és annak hatásait igyekszik minimalizálni. A kockázat csökkentésével a fejlesztés időtartamát és a rendszer megbízhatóságát tudjuk befolyásolni.

2.7 Agilis fejlesztés; Kialakulása; Összehasonlítás

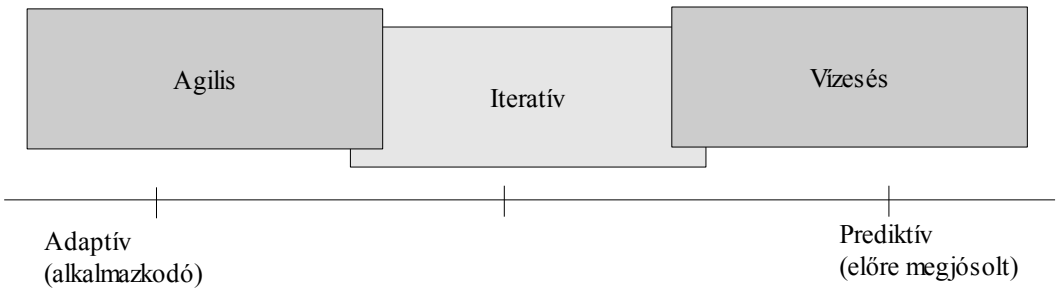
Az eddig ismertetett módszerek múlt századiak! Van, amire megoldást adnak; van, ami felett elsiklanak, más dolgokat csak részben oldanak meg! A fejlesztésre szánt idő, a végleges verzió kiadásának siettetése a 2000-es évek hozadéka! Nincs idő kivárni az inkrementálást, a vízesés modell statikussága és elnyúlása nem engedhető meg! Az erőforrás-idő-költség háromszögben eltolódtak a prioritások! Igény lett az „olcsót, jót, gyorsan” fejlesztésre. Ezen problémakör hatására született meg az agilis megközelítés.
Az agilitás nem csupán egy fejlesztési életciklust felölelő módszer! Magában foglalja a projektvezetés, a rendszertervezés és a szoftverfejlesztés eszközeit is, ami által egy igen hatékony eszköze lett a fentebb vázolt problémák megoldásának.

Fel kellett továbbá ismerni, hogy a szoftverfejlesztés nem egy klasszikus gyártási folyamat! Mindig egy új szoftvert fejlesztünk, ami más és más felhasználói igényeket elégít ki, ezáltal egy gyakori kommunikációt kell hogy megvalósítson! Más megközelítést és más prioritásokat használ! A „Kiáltvány az agilis szoftverfejlesztésért” például így fogalmaz: „Mi a jobb szoftverfejlesztés módjait fedezzük fel azáltal, hogy fejlesztünk és segítünk másokat fejleszteni”.

Az agilis módszertan tehát:

- Az egyént a személyes kommunikáció, a módszertanok és eszközök elé helyezi.
- Elsődleges célként a működő szoftvert, nem az átfogó dokumentáció elkészítését célozza.
- A szerződésben foglaltak merev betartása háttérbe szorul a felhasználóval való együttműködéssel szemben.
- A „tervek mindenek felett” elutasítása, és a változtatásra való gyors reagálás szem előtt tartása.

Természetesen az előbb felsoroltak nem jelentik azt, hogy nem fontos a terv, a szerződés stb. Összehasonlításul a statikus vagy „múlt századi” modellekkel az alábbiakat mondhatjuk. (Lásd 6. ábra)



6. ábra: Az agilis módszer pozícionálása

Az ábráról az alábbi tulajdonságok olvashatók le:

- Átfedés tapasztalható a módszerek között
- Iteratív használata mindenhol felfedezhető
- Az agilis és a vízesés modell szembenállása

Továbbá az *adaptív* tulajdonságai:

- Nincs előre jóslás, hosszú távra tervezés
- Csak a közvetlen problémákra koncentrálnak
- De arra pontosan!
- Csak azt tudják „mit fognak a héten csinálni”

Ezzel szemben a *prediktív*:

- Előre megtervezett lépések
- Minden lépés az egészre (mint célra) optimalizálva; nehézkes változás- követés
- A változások kezelése már-már bizottságot igényel

Az *iteratív*, mint közbenső út:

- Nincs éles elhatárolódás
- Iteratív megjelenések oka: a vízesés modell rugalmatlanságának javítása
- Kulcs: a szoftvert rövidebb időközönként kiadni

Fontos tulajdonsága az agilis fejlesztésnek az idő, hiszen ez volt az egyik oka annak, hogy létrejött! Az eddigi modellekben az idő mértékegysége a hónap (régebben, nagyobb projekt esetében az év) volt! Ez mára hetekké alakult! A határidők elvesztették flexibilitásukat! A „majd a következő hónapban" kikopott! Sarkallatos pont lett a határidők pontos betartása! A határidők kőbe vannak vésve, alappillérek; a határidőkből nem lehet kicsúszni! Ha ez mégis megtörténik, akkor:

- A feladat „megoldhatatlan"
- A projektet „vissza kell mondani"
- Részfeladatokat kell definiálni.

Az eddigiekből is látszik, hogy itt alapjaiban másról van szó, mint az eddig megszokott esetekben! Az agilis fejlesztést egy 12 pontból álló kiáltványban definiálták, ami a következőket mondja (kiemelve a nézőpont kialakítása szempontjából fontosabbakat):

- Legnagyobb prioritása a megrendelő igényeinek megfelelő, értékes szoftver korai és folyamatos kiadása.
- A követelmények változása elfogadott, még a fejlesztés késői szakaszában is.
- A rövidebb periódusokat kell előnyben részesíteni.
- A megrendelőknek, üzleti szakembereknek és a szoftverfejlesztőknek naponta (!) együtt kell dolgozniuk a teljes projekt során.
- A projektet motivált emberekre kell építeni, akiknek megfelelő munkakörnyezetet kell kialakítani.

- Az információ átadásának csapaton belül a leghatékonyabb módja a beszélgetés.
- A legjobb architektúrák, követelmények és rendszertervek az önszerveződő csapatmunkából alakulnak ki.
- A fejlesztői csapat rendszeres időközönként megfontolja, hogy hogyan válhatnak hatékonyabbá és ennek megfelelően finomítják a viselkedésüket.

Természetesen az agilitás sem nyújt mindig, minden esetben jó megoldást! Ez sem egy varázspálca, vannak hibái:

- 2001-ben „született”, ami miatt még kiforratlan!
- Különösen érzékeny arra, hogy csak tapasztalt fejlesztők alkotta csapatokkal működik.
- Nincs eléggé megtervezve.
- A fejlesztési kultúra nagymértékű változtatását igényli a jó működés érdekében.

A fent leírtak általánosságban jellemzik az agilis módszertanokat, melyek közül példaként említek néhányat:

- eXterme Programming (XP)
- Test Driven Development (TDD)
- Feature Driven Development (FDD)
- Scrum (Leírása a 6. fejezetben.)

2.8 Agilitás és kódolás

Mint korábban említettem, az agilitás a projektvezetés, a rendszertervezés és a szoftverfejlesztés (ezáltal a kódolás) hármasát jelenti!
Minden jó szoftver egyik legfontosabb alapja a jól strukturált kód! Az agilis kódolást a [2] XXVII. alapján így definiálhatjuk:

- *Szervezés. (Seiri)* Létfontosságú, hogy tudjuk, mi hol található! A változóinknak megfelelő nevet kell választanunk!
- *Rendszerezés. (Seiton)* „Legyen helye mindennek, és minden legyen a helyén!”

Az egyes kódrészleteknek ott kell lenniük, ahol számítunk rájuk! Ha nem ott vannak, akkor újratervezéssel a megfelelő helyre kell azokat tennünk!

- *Takarítás. (Seiso)* Meg kell tisztítani a forrást a megjegyzésbe tett kódrészletektől! Kerülni kell a múltra, vagy a jövőre vonatkozó utalásokat!
- *Szabványosítás. (Seiketsu)* A fejlesztői csapatban mindenkinek egységes stílust kell követnie!
- *(Ön)fegyelem. (Shutsuke)* A fenti szabályokat fegyelmezetten tartsuk be, a munkánkat gyakran vizsgáljuk felül, és legyünk nyitottak a változásokra!

A kódolási szabályok részletesen megtalálhatók a [2] irodalomban. Itt - terjedelmi okok miatt- csak irányelveket fogalmaztam meg.

3. Agilis adatmodellezés

Ez a fejezet az adatmodellezéssel foglalkozik, különös tekintettel annak agilis megközelítésére. A fejezetben az alábbiakról olvashatunk:

- Az agilis DBA szerepe az adatmodellezésben
- Mi az adatmodellezés?
- Hogyan modellezünk?
- Hogyan modellezhetünk jobban?

3.1 Az agilis DBA szerepe az adatmodellezésben

Az adatmodellezés az egyik legnagyobb kihívást jelentő tevékenység egy agilis DBA számára. További tulajdonsága az adatmodellezésnek, hogy a legnagyobb vita alapja az agilis és a tradicionális szoftverfejlesztők között. Az agilis fejlesztők egy evolúcionális utat követnek, ahol az adatmodellezés csak egy a tevékenységek közül. A tradicionális fejlesztők ezzel szemben a „big design up front” (BDUF) megközelítést alkalmazzák, ahol a modellezés nem egy tevékenység, hanem „A tevékenység”!
Az agilis modellezés sajátossága továbbá:

- Mentor alkalmazásfejlesztők részvétele az adatmodellezésben;
- Tapasztalt fejlesztők és tervezők bevonása a modellezésbe;
- A fejlesztői csapat a garancia arra, hogy követik a modellezési szabványokat és konvenciókat;
- Kifejlesztenek és kiadnak egy adatmodellt, amely egy evolúció (iteratív és inkrementális) eredménye;
- Az adatbázis sémát szinkronban tartják a fizikai adatmodellel.

3.2 Mi az adatmodellezés?

Az adatmodellezés egy tevékenység, mellyel felfedezzük az adat-orientált struktúrákat.

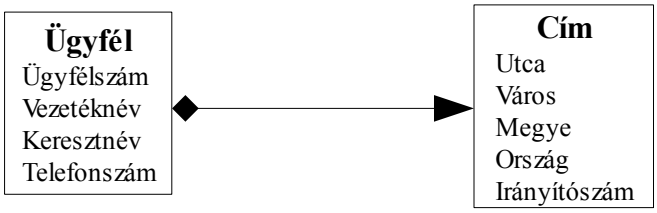
A modellezésben egy magas szintű koncepcionális modellt alakítunk fizikai adatmodellé. Egy objektum-orientált fejlesztő számára ez az osztálymodell kialakítást jelenthetné. Az adatmodellezéssel azonosítjuk az entitásokat, mint OO környezetben az osztályokat. Az adat attribútumokat hozzárendeljük az entitásokhoz, mint az attribútumokat és a funkciókat az osztályokhoz. Vannak asszociációk az entitások között, hasonlóan az osztályokhoz. Az adatmodellezésben is alkalmazhatjuk a kapcsolatokat, az öröklést, a kompozíciót, az aggregációt.

3.2.1 Gyakorlati alkalmazása

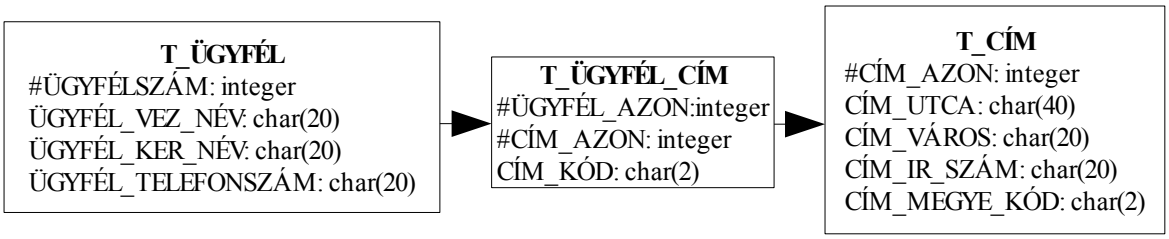
Az adatmodellezésben három alapstílust követhetünk:

1. *Koncepcionális adatmodell.* Ezeket a modelleket szokták domain modellnek is nevezni. Tipikusan ezt alkalmazzák a logikai adatmodell (LDM) előtt, vagy mint az LDM alternatíváját.
2. *Logikai adatmodell (LDM).* Az alapkoncepciók és azok kapcsolatainak feltárásán alapul. Az LDM leír egy logikai adat-entitást. Az adat attribútumok leírják az entitásokat és megadják az entitások közötti kapcsolatokat.
3. *Fizikai adatmodell (PDM).* A PDM-et használjuk az adatbázis belső sémájának leírására. Az adattáblák, azok oszlopai és a táblák közötti kapcsolatok is ezzel ábrázolhatóak.

Az LDM és PDM közötti különbség illusztrálására a 7. ábrán egy LDM, a 8. ábrán egy PDM modellt mutatok be.



7. ábra: Egy LDM modell



8. ábra: Egy PDM modell

3.3 Hogyan modellezünk?

Az adatmodellezés során az alábbi fázisokat alkalmazzuk iteratív módon:

- Entitások azonosítása
- Attribútumok azonosítása
- Névkonvenciók alkalmazása
- Kapcsolatok azonosítása
- Adatmodellezési minták alkalmazása
- Kulcsok hozzárendelése

3.3.1 Entitások azonosítása

Az entitás OO megfelelője az osztály. Az entitás- típus egymáshoz hasonló entitások halmazát reprezentálja. Egy entitás- típus lehet például: emberek csoportja, helyek, dolgok, események vagy koncepciók. Ha gyakorlatiasabb példával élünk, akkor egy *nyilvántartó rendszer*ben az *Ügyfél*, a *Cím*, a *Rendelés*, a Tétel és az *Adó* is entitás típus.
Ezek alapján felfedezhetjük a különbséget az osztály és az entitás- típus között: az osztály adatot és viselkedést is tárol, míg az entitás- típus csak adatot.

3.3.2 Attribútumok azonosítása

Minden entitás rendelkezik egy, vagy több adat attribútummal. Például a 7. ábrán az *Ügyfél*

entitás tartalmazta a *Vezetéknév* és a *Keresztnév* attribútumokat. A 8. ábrán a T_ÜGYFÉL tábla tartalmazta az ÜGYFÉL_VEZ_NÉV oszlopot. (A relációs adatbázisokban az attribútum oszlopként van implementálva.)
Az attribútum tehát a táblában tárolt entitások valamilyen tulajdonságát írja le. Az attribútumok azonosítása igen nehéz lehet, és nagyban befolyásolhatja az adatbázis esetleges refaktorálását, illetve karbantarthatóságát.
Példaként vehetjük a 8. ábra CÍM_IR_SZÁM nevű attribútumát. Ennek formátumát sokféleképpen megválaszthatjuk annak megfelelően, hogy csak numerikus karaktereket tartalmazhat, vagy esetlegesen néhány betűvel vagy speciális karakterrel egyedivé tesszük.

3.3.3 Névkonvenciók alkalmazása

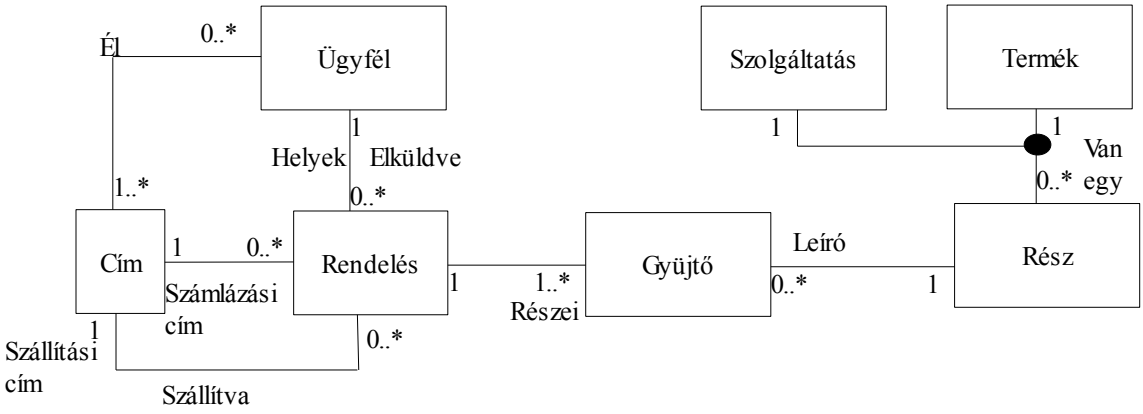
Minden szervezeten belül megtalálhatók azok a névadási „szabványok", melyek megkönnyítik és használhatóbbá teszik az alkalmazott adatbázisokat, illetve szoftvereket.
A névkonvenciók két csoportját különböztetjük meg:

- *Logikai névkonvenció*. Ez a humán felhasználó számára teszi olvashatóbbá a tábla attribútumait.
- *Fizikai névkonvenció.* Ez a technikai oldalról közelíti meg a névhasználatot.

A különbségek jól követhetőek és leolvashatóak a 7. és 8. ábra összehasonlításával.
Az agilis modellezésben ennek a témakörnek igen nagy jelentősége van! Ha visszalapozunk az „Agilitás és kódolás" fejezethez, akkor ott is feltűnik ennek az elvnek a jelentősége. (Kódolási konvenciók.)

3.3.4 Kapcsolatok azonosítása

A való életben nagyon fontos szerepe van a kapcsolatoknak! Minden entitás kapcsolódik egy másik entitáshoz. Például egy nyilvántartó rendszerben az Ügyfél *él* egy lakcímen.
A kapcsolatok tanulmányozására egy online rendelő-rendszer logikai-adat modelljét nézhetjük meg a 9. ábrán.



9. ábra: Online rendelő-rendszer logikai-adatmodellje

Az ábráról az alábbiakat olvashatjuk le:

Van egy kapcsolat az *Ügyfél* és a *Rendelés* között. Van két neve: *helyek* és *rendelő.* Az *Ügyfél* és a *Cím* közötti kapcsolat csak egy névvel rendelkezik. Ezek a nevek az olvasás irányának megfelelően értendők.

A nevek mellett a kapcsolatban résztvevők számosságáról is találunk információkat. Példaként nézhetjük a *Cím* és a *Rendelés* relációt. Minden rendeléshez csak egy cím tartozhat, de egy címhez nulla és „bármennyi” közötti darabszámú rendelést kapcsolhatunk.

3.3.5 Adatmodell minták alkalmazása

Az OO terminológiához hasonlóan az adatmodellezésben is alkalmazhatunk tervezési mintákat. Ezen minták alkalmazásával az adatbázisunkban olyan recepteket alkalmazhatunk, amelyek egy a korábbiakban már analizált probléma, szituáció megoldására nyújtanak kész és helyes megoldást.

3.3.6 Kulcsok

A *kulcs* egy, vagy több attribútum, amely egyértelműen azonosítja az entitást. Ha a kulcsot kettő, vagy több attribútum alkotja, akkor *összetett kulcs*ról beszélünk. Kulcsokat a mindennapokban is használunk. Ilyen például a személyi szám vagy a társadalombiztosítási

azonosító is.

A kulcs egzakt definíciója:

Azt mondjuk, hogy az egy vagy több attribútumból álló $\{A_1,A_2,....A_n\}$ halmaz az R reláció kulcsa, ha:

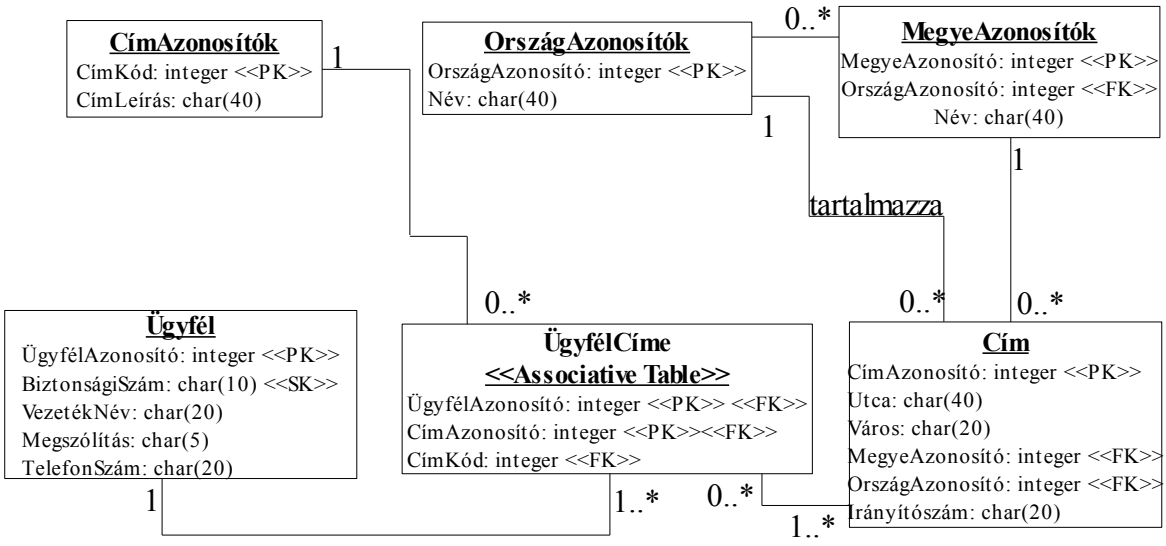
1. Ezek az attribútumok funkcionálisan meghatározzák a reláció minden más attribútumát, azaz nem lehet R-ben két olyan különböző sor, amely mindegyik $A_1,A_2,....A_n$-ben megegyezne.
2. Nincs olyan valódi részhalmaza $\{A_1,A_2,....A_n\}$-nek, amely funkcionálisan meghatározná R összes többi attribútumát, azaz a kulcsnak minimálisnak kell lennie.

A 10. ábrán egy olyan fizikai adatmodellt láthatunk, amelyen már a kulcsok is jelölésre kerültek.

<<PK>> : Primary (elsődleges) kulcs.

<<SK>> : Secondary (másodlagos) kulcs.

<<FK>> : Foreign (külső) kulcs.



10. ábra: Fizikai adatmodell a kulcsok jelölésével

A kulcsok hozzárendelésének alapvetően két stratégiája van:

- *Természetes kulcs alkalmazása.* Ekkor egy, vagy több létező attribútumból kialakítunk

egy kulcsot. A 10. ábrán ez az *Ügyfél* táblában figyelhető meg. Van két kulcsesélyes: az *ÜgyfélAzonosító* és a *BiztonságiSzám*. Ezekből az előbbit választottuk, míg az utóbbi <<SK>> bélyeget kapott.

- *Új attribútum felvétele.* Ebben az esetben felveszünk a táblához egy új oszlopot, és ezt használjuk kulcsként. A kulcsnak így nincs „jelentése", csak azonosításra szolgál. Erre példa a *Cím* tábla *CímAzonosító* mezője, ami egy mesterségesen létrehozott szám, egy címet azonosít.

A kulcsok hozzárendelése nagy tapasztalatot és körültekintést igénylő feladat.
A szakirodalmakból azonban értékes ötleteket kaphatunk ehhez a feladathoz:

- *„Egyszerű" kulcsok elkerülése.* Első ránézésre valamely attribútumból származtathatnánk kulcsot egy vágás vagy transzformáció alkalmazásával. Például a budapesti irányítószámokból majdnem minden esetben következtethetünk a kerületre és a postahivatal számára. A kivételek miatt azonban ennek alkalmazása veszélyeket rejthet magában.
- *A természetes kulcsok alkalmazása egyszerű „lookup" (megfeleltető, hozzárendelő) táblákban.* Vannak esetek, amikor a táblázatunk egy hozzárendelést tartalmaz. Például színek és színkódok összerendelését. Ebben az esetben a színkód kulccsá minősítése kézenfekvő.

3.4 Normalizálás

Az adatbázis tervezése során olyan adatstruktúrákat kell kialakítanunk, amelyek segítik a hatékony adatkezelést. Fontos, hogy egy-egy táblába csak a valóban logikailag összetartozó adatok kerüljenek, és hogy minél kevesebb ismétlődés legyen az adatok között.
A relációs modellben külön eljárás foglalkozik ezen kérdésekkel. Ez a módszer a *normalizálás*. Eredményeképpen logikailag konzisztens adattáblákat kapunk, amelyek áttekinthetőek és karbantarthatóak.
Az adatbázisok irodalmában különböző szintű normálformákat definiáltak. A legegyszerűbbek (első-, második-, harmadik normálforma) a gyakorlatban is sokszor előforduló tipikus esetek kezelésére szolgálnak, míg az összetettebbek (Boyce-Codd-, negyedik-, ötödik normálforma)

csak elméleti jelentőségűek. Ezek közül a legnagyobb jelentőséggel a harmadik normálforma bír, mivel a gyakorlati problémáknál is minden esetben használatos.

3.4.1 A normalizálás szabályai

A gyakorlatban az első három normálforma alkalmazása mindennapi. (Léteznek magasabb szintű normálformák is, ezek szakirodalma széles körű, dolgozatomban nem térek ki ezekre.) A normálformák kialakításához szükséges eljárásokat az 1. táblázat tartalmazza.

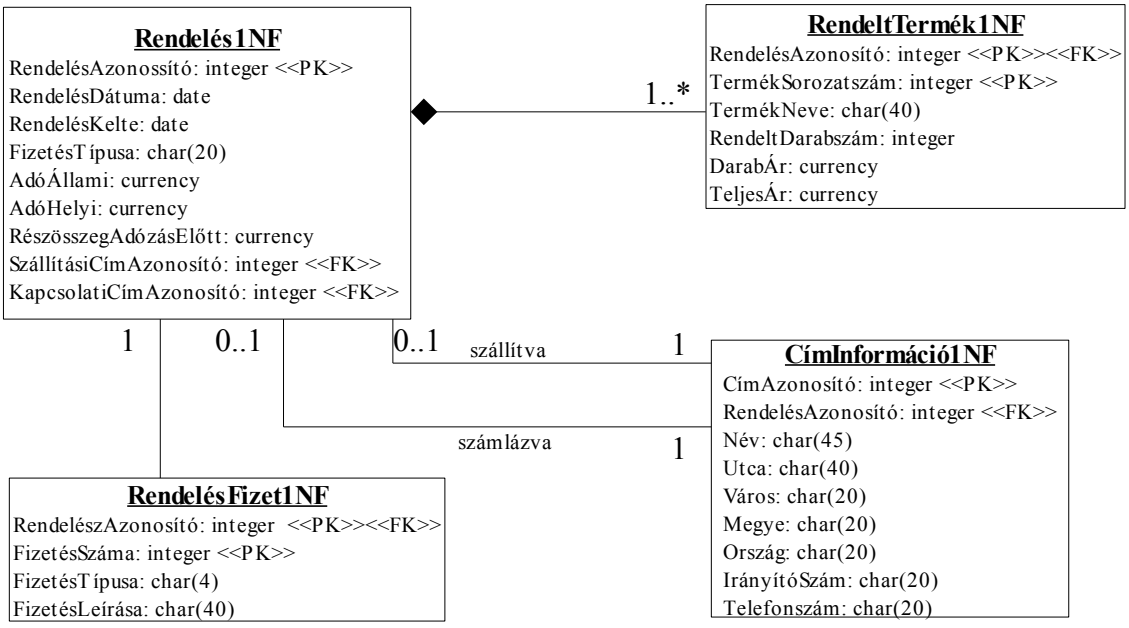
Szint	**Eljárás**
Első normálforma (1NF)	Egy entitás első normálformában van, ha minden attribútum függ az elsődleges kulcstól.
Második normálforma (2NF)	Egy entitás második normálformában van, ha első normálformában van, és nincs benne részleges függés.
Harmadik normálforma (3NF)	Egy entitás harmadik normálformában van, ha második normálformában van és nincs benne tranzitív függés.

1. táblázat: A gyakorlatban alkalmazott normálformák áttekintése

Az adatbázis tervezésekor felsoroljuk azokat az azonosítókat (attribútumokat), amik leírják az általunk modellezni kívánt világot (Nulladik normálforma 0NF). Ezzel a felsorolással nem definiáltunk semmilyen kapcsolatot, összefüggést.

3.4.2 Az első normálforma (1NF)

Egy entitás-típus első normálformában van, ha a relációban minden érték elemi, vagyis a reláció nem tartalmaz adatcsoportokat. A 0NF tartalmaz ismétlődő csoportokat, így az első NF-re hozáshoz ezeket kell megszüntetnünk. Megtehetjük ezt úgy, hogy az azonos logikai egységeket egy külön táblába rendezzük, és beazonosítjuk az így létrejött táblák között fennálló kapcsolatokat. Ezen fázis eredményét a 11. ábra mutatja.



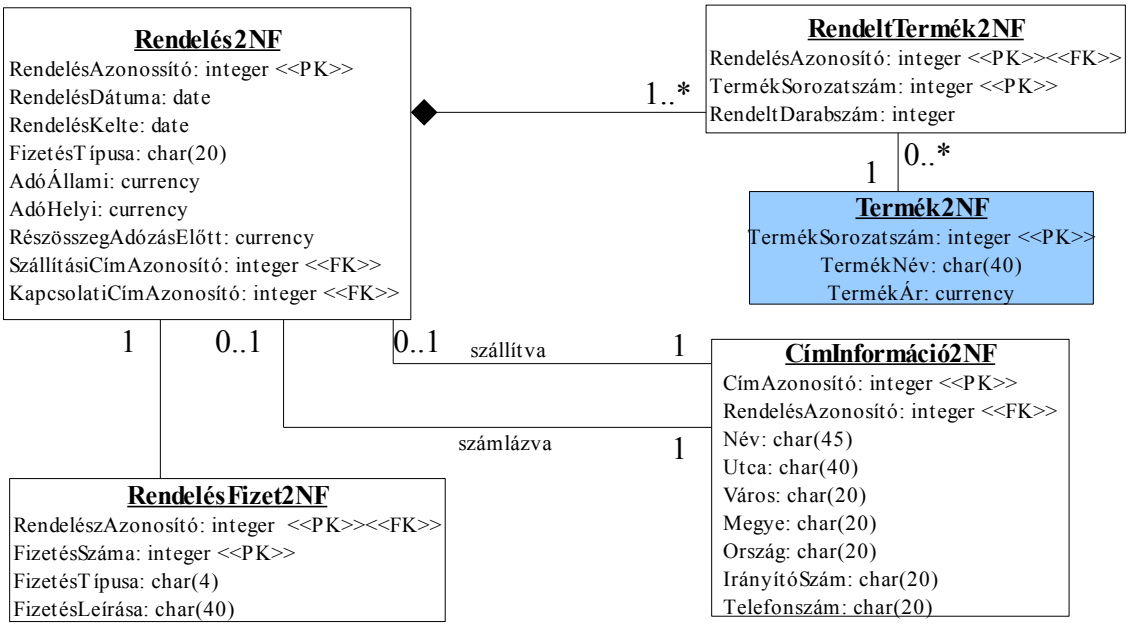
11. ábra: Első normálforma

Az ábrából látszik az is, hogy ebben a lépésben be kellett azonosítani a kulcsokat! Ez a fázis különösen fontos a végeredmény szempontjából, hiszen itt történik az első szétválogatása az attribútumoknak, és ez szolgál alapul a további átalakításokhoz.
Az első normálformát röviden így fogalmazhatjuk meg: minden sor, minden komponense atomi értékű legyen. (Nem megengedettek az összetett típusok.)

3.4.3 Második normálforma (2NF)

Egy entitás típus második normálformában van, ha első normálformában van és egyetlen másodlagos attribútuma sem függ egyetlen kulcsának valódi részhalmazától (nincs benne részleges függés). Ez azt jelenti, hogy a relációban nincsenek a kulcs részeitől való nem triviális függések. A definíciót összevetve a 11. ábrával láthatjuk, hogy az nem teljesül az *RendeltTermék1NF* táblára. A probléma a táblával az Termék információban van! Például, ha valaki többféle terméket rendel, akkor az ár rendben lesz, de a rendelés nevével lesznek problémák.
Ennek feloldására a sémát a 12. ábrának megfelelően alakíthatjuk át.

A második normálformának nincs nagy gyakorlati jelentősége. Tulajdonképpen a folytonosságot biztosítja, nem szigorú feltétel.

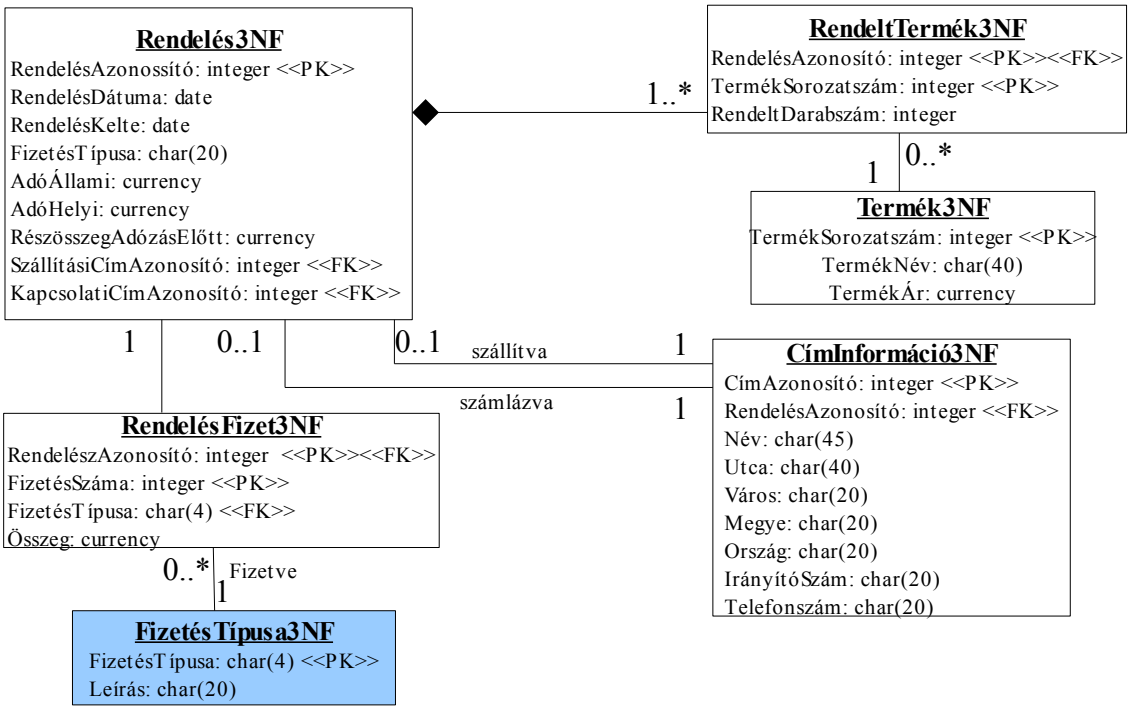


12. ábra: Második normálforma

3.4.4 Harmadik normálforma (3NF)

Egy entitás-típus harmadik normálformában van, ha második normálformában van, és egyetlen másodlagos attribútuma sem függ tranzitíven kulcstól. (Egy A→C funkcionális függést tranzitív függésnek nevezünk, ha létezik olyan B attribútumhalmaz, amelyre A→B és B→C.) Másként megfogalmazva: ha egy leíró attribútum nem csak egy kulcstól (vagy annak egy részétől), hanem egy másik leíróattribútumtól is függ. (Tranzitív függés)

Ha megvizsgáljuk a 12. ábrát, akkor azt találjuk, hogy az *RendelésFizet2NF* tábla fizetési típust leíró attribútuma (ami lehet például „*Bankkártya*”, „Készpénz”) csak a fizetés típusától függ, nincs kapcsolatban a megrendelés azonosítója (*RendelésAzonosító*) attribútummal. Ezen hiba feloldására egy új táblát hozunk létre *FizetésTípusa3NF* névvel. Az így kapott sémát a 13. ábrán láthatjuk.



13. ábra: Harmadik normálforma

3.4.5 A harmadik normálforma után

A 14. ábrán lévő séma már nem tartalmaz redundáns attribútumokat, a származtatható vagy kiszámítható attribútumokat eltávolítottuk. További egyszerűsítésekre azonban még van lehetőség! Például a *RészösszegAdózásElőtt* mező az *Rendelés3NF* táblából eltávolítható. Az *RendeltTermék*3NF tábla *TeljesÁr* oszlopára sincs szükség, így azok törölhetőek.
Ezen néhány művelet alkalmazásával egy jól használható sémához jutottunk.

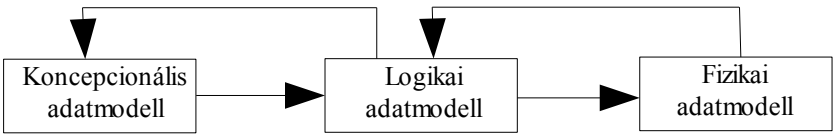
4. Adatbázis tervezés

4.1 Adatbázisok tervezésének áttekintése

Az adatbázisok tervezése során - mint minden szoftver-termék tervezése esetében - nem hajthatjuk végre minden esetben ugyanazt a tervezési menetet. Minden projekt valamiben más, mint az előző, ugyanakkor vannak hasonlóságok, amik kiemelése és egy projektvezetési tervvé – módszertanná - való fejlesztése nagy segítséget nyújhtat a későbbi munkákhoz.
Az adatbázistervezés során nem szabad figyelmen kívül hagyni, hogy ebben az esetben gyakorlatilag többszöri tervezéssel kell eljutnunk a végeredményhez. A projekt során több modellt alkotunk, amik kihatással vannak a termék végső verziójára. Ezek a modellek:

- Koncepcionális modell
- Logikai adatmodell
- Fizikai adatmodell

Ezen modellek tervezése egy-egy önálló folyamat is lehetne, de az egymásrahatás miatt ezeket egységes rendszerként kell kezelni.
A modellek egymásra gyakorolt hatását, sorrendiségét a 14. ábra mutatja.



14. ábra: Modellek az adatbázisban

A modellezés során tehát a fenti modellek megalkotásával érjük el a kialakítani kívánt rendszer modelljét.

4.1.1 A koncepcionális adatmodell

A modell *definíció*ja: Véges számú tulajdonságtípussal megadott véges számú egyedtípus, és a közöttük fennálló véges számú kapcsolattípus összessége.

Látható, hogy ez nem más, mint egy absztrakció. A modellezni kívánt világ objektumait és az azokat összekötő kapcsolatokat szűkítjük le olyan formára, hogy azt adatbázisban tárolhassuk. Természetesen - mivel ez egy koncepció, egy első lépés - nem arra koncentrálunk elsősorban, hogy milyen adatbázis táblákat hozunk majd létre, ott milyen típusokat használunk. Itt csak megadjuk azt a szűkebb világot, amit az adatbázisban tárolni akarunk.

4.1.2 A logikai adatmodell

Ezen fázis elsődleges feladata, hogy az adatokról és azok szerkezetéről részletes logikai leírást adjon, támaszkodva a koncepcionális modellre.
Látható, hogy itt már informatikai eszközök tervezése zajlik. Döntéseket kell hoznunk az adatbázis tábláinak szerkezetéről, a kapcsolatok realizálásáról. (Az adatmodellezés teljességét és történetét figyelembe véve meg kell említeni, hogy különböző adatmodellek léteznek. A hálós, hierarchikus modellek már nem bírnak jelentőséggel. A relációs modell hasnálata napjainkban a legelterjedtebb, míg az objektumorientált megközelítés teljes kidolgozása és alkalmazása még várat magára.)

4.1.3 A fizikai adatmodell

Ez a szint már csak és kizárólag a megvalósításra koncentrál. A főbb kérdések:

- Az adatok fizikai helyének és tárolásának megadása
- Fájlok struktúrája
- Az adatelérés módja
- A redundancia kérdése

Ezen lépésben már csak a megvalósítás részleteiben lehetnek változások. Azonban észre kell venni ezen modell fontosságát! Az üzemeltetés és esetleges továbbfejlesztés szempontjából fontos kérdések dőlnek itt el, tehát a nagy körültekintés elengedhetetlen.

4.2 Az AMDD (Agile Model-Driven Development)

A modellezés és a dokumentálás nagyon fontos része a szoftverfejlesztés folyamatának és a tervezők, illetve fejlesztők munkájának. A fejlesztők az előző fejezetekben leírt modellek megalkotásával hozzák létre a megvalósítandó rendszer tervét.
Az ebben a fejezetben ismertetésre kerülő Agilis Modellezés (AM) választ ad a tervezési és dokumentálási fázisok, munkák során jelentkező feladatok megoldására, az effektív munkavégzésre.

4.2.1 Mi az agilis adatmodell? (AM)

Az agilis modellezés egy gyakorlatorientált - gyakorlati eredményeket használó - szabályrendszer, amely egy szoftverfolyamat (és ezzel együtt egy adatbázistervezési folyamat) modellezését és dokumentálását foglalja össze. Tartalmazza a folyamatok leírását, a jó megoldásokat és a napi munka szervezésének leírását.
Azonban nem részletes leírásokat tartalmaz! Nem határozza mag a modellezés részleteit, csak ötleteket ad az effektív döntéshozatalhoz! Több helyen megtalálhatjuk azt a hívatkozást, hogy az AM nem egy tudomány, hanem egy művészet!
Az AM három célja:

1. Definiálja azt a módszert, amivel a felhalmozott gyakorlati tudást átültethetjük a modellezésbe.
2. Hogyan lehet alkalmazni a modellezési technikákat a szoftver projektekben?
3. Hogyan lehet a fejlődés érdekében felhasználni a modellezési technikákat?

4.2.2 Az AM előnyei

Az AM filozófiai alapokat biztosít az alábbi témakörökben, amik a modell előnyeit és értékeit biztosítják:

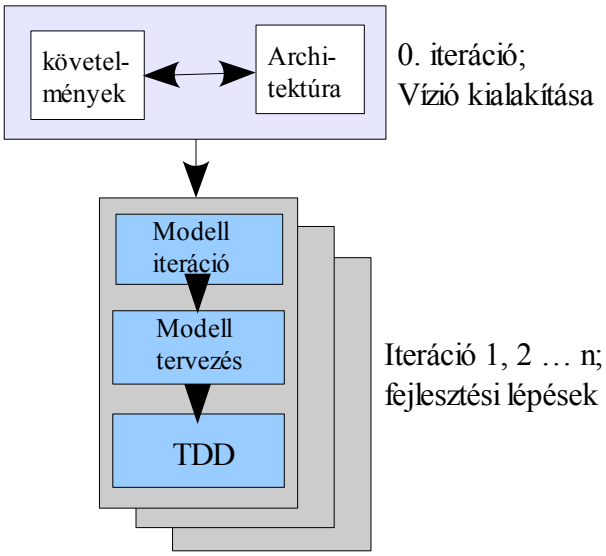
- *Kommunikáció*; Különös fontossággal bír a fejlesztők és a projekt egyéb résztvevőinek kommunikációja. Bővebben lásd a Scrum leírásánál.

- *Egyszerűség*; A legegyszerűbb megoldásokat alkalmazza annak érdekében, hogy a termék a lehető legjobban lefedje a követelményekben meghatározott rendszert.
- *Visszacsatolás*; A napi ellenőrzésekkel – visszacsatolásokkal - elejét venni a „vakvágányra” futásnak és ezzel a fölösleges munkának.
- *Bátorság*; A döntések támogatásával arra kell bíztatni a fejlesztőt, hogy legyen kreatív és effektív.
- *Alázatosság*; Ezzel fejezi ki, hogy a hozzáadott munka és tudás előrevetíti a projekt sikerességét, viszont megköveteli a szabályok betartását és betartatását!

4.2.3 Az AMDD

Az AMDD az Agile Model Driven Development kezdőbetűiből alkotott betűszú, amelynek jelentése: agilis közelítésű fejlesztés. Alapja az OMG által kiadott MDA (Model Driven Architectura) szabvány.

Az AMDD ciklust a *15. ábra* mutatja.



15. ábra: Az AMDD életciklusa

A folyamat leírása:

0. iterációs lépés: A fejlesztés elején a team kialakít egy kezdő koncepciót a megvalósítandó rendszerről, ami a többi fejlesztési fázis bemenetéül szolgál. Itt a követelmények alapos

vizsgálata, priorizálása mellett az architektúra alapjait is meghatározzák.
1, 2 … n iterációs lépés: Ezekben a lépésekben a fő feladat a részletek kidolgozása és az esetlegesen felmerülő új, vagy megváltozott igények kielégítése.

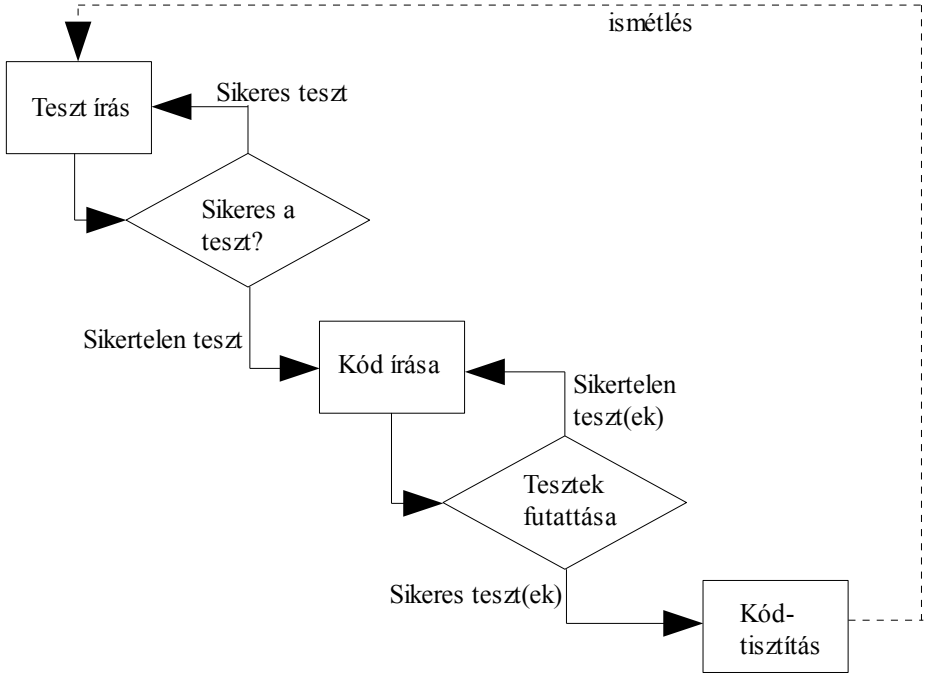
Az egyes lépések részletesebb ismertetése:
Vízió kialakítása: Jellemzően a projekt elején kell ezt a lépést megtenni. Hossza a projet hosszával arányos. Néhány hét alatt befejeződő fejlesztés esetén órákban mérhető, míg a több hónapos, vagy több mint egy év alatt lezáródó projekt esetében akár hetek is lehetnek. Célja, hogy a magas szintű követelmények figyelembevételével architektúrális megoldásokat adjon, és a projekt egészét meghatározó stratégiákat fogalmazzon meg.
Magasszintű követelmények modellezése: A feladat nagyon egyszerű, ugyanakkor nagyon bonyolult! Elő kell állítani egy olyan rendszert, ami a felhasználókban felkelti az együttműködési hajlandóságot. Ki kell alakítani a magas szintű követelmények egy (működő, kipróbálható) modelljét, implementálni kell a főbb üzleti folyamatokat (vagy azok egy modelljét), és meg kell tervezni a felhasználói felületet. Ezek a lépések nagyon fontosak, hiszen a megrendelővel itt kell kialakítani azt a bizalmas viszonyt, ami a további munka alapját képezi!
Architektúra modell: Ebben a lépésben meg kell határozni azt a vázat, amire a rendszert felépítjük. Látszik, hogy nem egyszerű kérdésekről kell döntést hozni, de ebben nagy segítség az agilis modell. Mivel nem egy soros elvű modellel dolgozunk, ezért a meghozott architektúrális döntések nem életre szólóak! A fejlesztés további szakaszában (az iterációkban) a hibás megoldásokat javíthatjuk. Ez a megközelítés adja az agilitás erősségét, és ez a legnehezebben átvehető azon fejlesztők számára, akik egy nem iteratív modellezést akarnak az agilitással kiváltani.
Iterációs modellezés: A fejlesztés ténylegesen ebben a fázisban (több fázis, mivel iteratív) zajlik. A fejlesztő csapat a követelményeket fontossági sorrendben implementálja, ezzel kezdetben nagy lépéseket tesz a végleges rendszer kialakítása felé. A további részletek kidolgozása más iterációs lépésekben történik. Fontos kérdés ebben a fázisban a munka szétosztása, illetve a költségek (idő) becslése. A nagyobb munkák megvalósítása egy bemeneti paraméter a későbbi funkciók megvalósulási idejének becslésekor! A megvalósításkor keletkezett tapasztalatokat és eredményeket a további iterációk tervezésekor figyelembe kell

venni.

Model storm; Modell tervetés: Az elnevezésből is következik (modell vihar), hogy ez egy nagyon gyors lépés. Néhány percben kell a csapatnak választ adni a felmerülő kérdésekre. A megvalósítás is igen ötletes. Egy tábla vagy papír előtt vázolja valaki a problémát, amit egy másik csapattag megválaszol és feladatul kapja a megvalósítást is. A feladat kiosztása után a csapattagok tovább folytatják az eddigi munkát.

4.3 A TDD (Test-Driven Development)

A TDD rövid fejlesztési ciklusokra támaszkodik. Első lépésben a fejlesztő egy tesztet ír, amely megvizsgálja, hogy a kód helyesen működik-e. Ha nem, akkor egy új kódot ír, ezen futtatja az előzőekben definiált tesztet, majd javítja a kódot. Ezt a folyamatot szemlélteti a *16. ábra*.



16. ábra: TDD folyamat

A tesztvezérelt fejlesztés első lépésében minden esetben egy teszt-esetet kell definiálni, amely a követelményekre és/vagy a megrendelővel folytatott megbeszélésre támaszkodik. Ennek egy lényeges következménye, hogy a teszt megírását csak akkor lehet elvégezni, ha ismerjük a követelményeket! A kód fejlesztése csak ez után következhet! Ez a TDD alapvető

koncepciója.

A kód elkészülte után az összes tesztet le kell futtani, amit a kezdetekben definiáltak. Ez egy hosszas iteratív folyamat kezdetét jelentheti, hiszen az ábrából következik, hogy a tesztelési fázis csak abban az esetben fejeződik be, ha minden teszt sikeresen lezárult. Ez egyfajta garanciát is jelenthet arra nézve, hogy a funkciók és ezzel együtt a teljes rendszer megfelel a követelményeknek.

Ezek után jogosan merülhetnek fel az alábbi kérdések:

- Alkalmazható a TDD az adatbázisok tervezésekor?
- Írhatunk teszteket az adatbázis séma megváltoztatásához?
- ...

A válaszok alapos meggondolást igényelnek és nem egyértelműek!

A szakirodalmat tanulmányozva találunk arra vonatkozó példát, hogy igen, alkalmazható! Ha olyan fejlesztőt keresünk, aki azt válaszolja a kérdésekre, hogy igen, én alkalmazom, akkor nehéz dolgunk van!

Az adatbázis tervezése - véleményem szerint - inkább az AMDD modellt támogatja, használja. A TDD megközelítés tesztorintáltsága inkább az OO szemlélet továbbgondolásának tekinthető, ezáltal jobban beleillik egy alkalmazásfejlesztési környezetbe.

4.3.1 A TDD és az AMDD

Az előző pont végén felvetett kérdések megválaszolása előrevetíti a lehetőségét a TDD és az AMDD egy –nem teljes- összehasonlításának, amit a 2. táblázat tartalmaz.

TDD	***AMDD***
Lerövidített programozási visszacsatolás.	Lerövidített modell visszacsatolás.
Részletes specifikációt kíván.	Tradicionális specifikáció.
Támogatja a fejlesztőket a jó minőségű kód előállításában.	Tökéletes kommunikáció szükséges a fejlesztők (csapat) és a megrendelő között.
Konkrét problémák megoldására van kitalálva.	Támogatja a csapat és a megrendelő további elképzeléseit. (Nem túl érzékeny a változásokra.)
A programozóknak szól.	Az adatbázis fejlesztőknek szól.

Azonnali visszacsatolást vár.	A szóbeli visszacsatolás elégséges. (Sok kommunikációra épül.)
A fejlesztőt segíti azáltal, hogy a konkrét megoldásokat, az üzemeltethetőséget és a tesztelhetőséget helyezi előtérbe.	Az architektúrális tervezés megelőzi a kódolást, így nagy projektek esetében előnyösebb.
Nem a vizualitást helyezi előtérbe.	Vizuális modellezéssel és a megrendelő bevonásával dolgozik.
Mindkét megoldás számos újdonságot tartogat a tradicionális fejlesztési módokkal szemben, valamint mindkettő támogatja a szoftverek evolúciós fejlesztését.	

2. táblázat: A TDD és az AMDD összehasonlítása

A választás tehát projek és személyfüggő. A vizuális típusok inkább az AMDD-t, a kevésbé vizuálisok a TDD-t használják. A projekt jellegét tekintve az AMDD adat-orientált, a TDD inkább kód-orientált.

Az extremitást és a hatékonyságot tovább növelhetjük, ha a két módszert vegyesen használjuk! (Akár projekten belül is!)

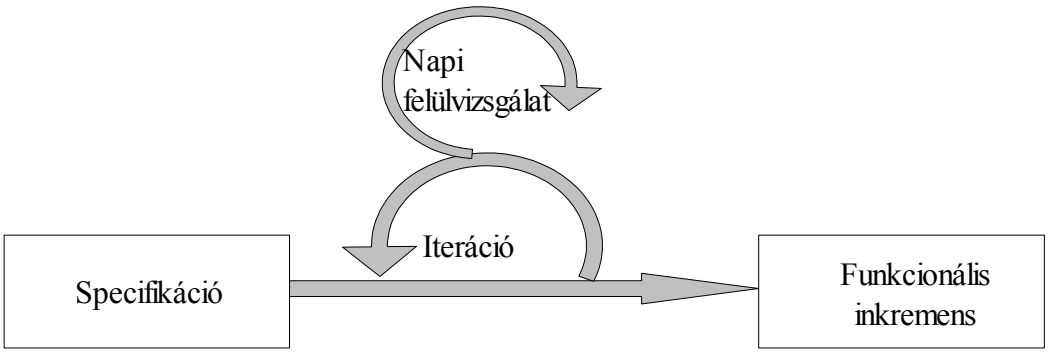
Például az AMDD segítségével egy modellt, architektúrális tervet hozhatunk létre (természetesen a megrendelő bevonásával), majd a TDD-t alkalmazhatjuk a kritikus részek fejlesztésekor. Ezzel a követelménykezelést az AMDD-re bízzuk, míg a TDD a tiszta és működő kódot biztosítja.

A végeredmény egy előreláthatóan validált és verifikált szoftver lesz.

5. Agilis projektmenedzsment – A Scrum

5.1 A Scrum háttere

A szoftverfejlesztés egy komplex feladat, amit természetesnek mondhatunk, hiszen a minket körülvevő világ is nagyon összetett. A fejlesztés során sok szempontot és követelményt kell szem előtt tartani, amik lehetnek például technikai feltételek és felhasználói szempontok is.
Az ebben a fejezetben ismertetésre kerülő Scrum egy olyan módszer, amely nagyban megnöveli a projekt sikeres befejezésének valószínűségét.
A Scrum alapvetően egy inkrementális, iteratív folyamat, melynek a váza a 17. ábrán látható.



17. ábra A Scrum folyamat

Az alsó nyíl egy itárációt szemléltet, aminek a végeredménye egy inkremens. A felső nyíl egy felülvizsgálatot szimbolizál, amelyet a projekt team tegjai végeznek. Ekkor ellenőrzik a követelmények teljesülését, javaslatot tesznek a módosításra.
A projekt a következő állomásokat „járja be”:

- A csapat tagjai áttekintik a teendőket, azaz, hogy mit is kell csinálni. „What to do..”; Meghatározzák, hogy az adott iterációs folyamatnak milyen funkcionalitású végeredményt kell szolgáltatnia.
- Az iteráció következő fokozatában az előző folyamat végeredményét használva kiindulásként, tovább iterálják az inkrementst, addig, amíg az teljes egészében nem felel meg a követelményeknek.

Az eddig leírtak nem tartalmaznak újdonságot. Szimpla iteratív folyamat leírása. A Scrum ezt azzal fejleszti tovább, hogy egy napi ellenőrzési rutin beiktatásával a hibák javítása azonnal metörténik, ezzel jelentős időmegtakarítást ér el.

5.1.1 Scrum szerepkörök

Egy projekt sikere nagyban függ attól, hogy a projektben résztvevők mennyire elkötelezettek a projekt iránt, és mennyire vállalják a módszertan szabályainak betartását. A Scrum arra épít, hogy a lefektetett szabályokat maximálisan megköveteli, ezzel igyekszik kiküszöbölni az emeri hibákból adódó késlekedést.

Alapvetően három szerepkörről beszélhetünk:

1. *A terméktulajdonos;* Ez a szerepkör egy megrendelő szerepkörével azonos. A projekt folyamatában az a feladata, hogy kitűzze a követelményeket, meghatározza azok prioritását, ellenőrizze az iterációk során, hogy az implementált rész megfelel-e az elvárásainak. A hagyományos fejlesztések esetében ez a szerepkör csak a fejlesztés indulásakor volt „hasznosítva". A későbbiekben, amikor már a fejlesztőkben kialakult a megvalósítandó szoftver képe, kevesebb figyelmet kapott.
2. *A Scrum master;* A projekt irányítója. Feladatköre nagyon hasonlít egy projektvezető feladataihoz, azonban itt néhány dolgot még szükséges felügyelnie. A módszer sajátosságait figyelembe véve (ld. bővebben a további fejezetekben) menedzselni szükséges a projektmegbeszéléseket és az iterációk különlegességéből adódó ütemezéseket.
3. *A team;* Gyakorlatilag ez a legkevébé megváltozott szerepkör. Klasszikus implementációs feledatok tartoznak ide. A másságot az jelenti, hogy az agilitásból adódó kötöttségeket el kell fogadni.

5.1.2 A Scrum folyamat

A folyamat első lépéseként az elkészítendő rendszer egy viziójának a kialakítása szükséges. Ekkor nagyon fontos feladat hárul a megrendelőre, hiszen ekkor kell meghatározni a funkcionális és nem funkcionális követelményeket. Nagy különbség a korábbi fejlesztési módszertanokkal szemben, hogy itt nagy hangsúlyt kell fektetni a követelmények priorizálására. Fontos megjegyezni és lefektetni ebben a szakaszban, hogy a követelmények megváltozása - a prioritások megváltozása - hatással van az üzleti követelményekre is, továbbá befolyással bír a projekt időtartamára.

Minden munkafolyamatnak el kell készülnie 30 munkanapon belül. Ezt az időintervallumot *Sprint*-nek nevezzük. Minden sprintet inicializálni szükséges, melyet a *sprint planning meeting*-en tesznek meg, amin a projekttulajdonos és a team vesz részt. Ezen a meetingen kell döntést hozni a legnagyobb prioritású funkciókról. A team nyilatkozik arról, hogy a következő sprint-ig milyen inkremest szállít le a megrendelőnek. A meeting jellemzője, hogy az ajánlás szerint nem lehet hosszabb 8 óránál! További jellemző a tagoltság; két részt különböztetünk meg, 4-4 órás időtartammal. Az első négy órában a megrendelő ismerteti a követelményeit, azok prioritását, megbeszéli a team tagjaival az inkremens előállításához szükséges feltételeket, a fejlesztők pedig ismertetik, hogy milyen úton képzelik el a fejlesztést, mi lenne a legmegfelelőbb irány. A második négy órában a sprint tervezése folyik.

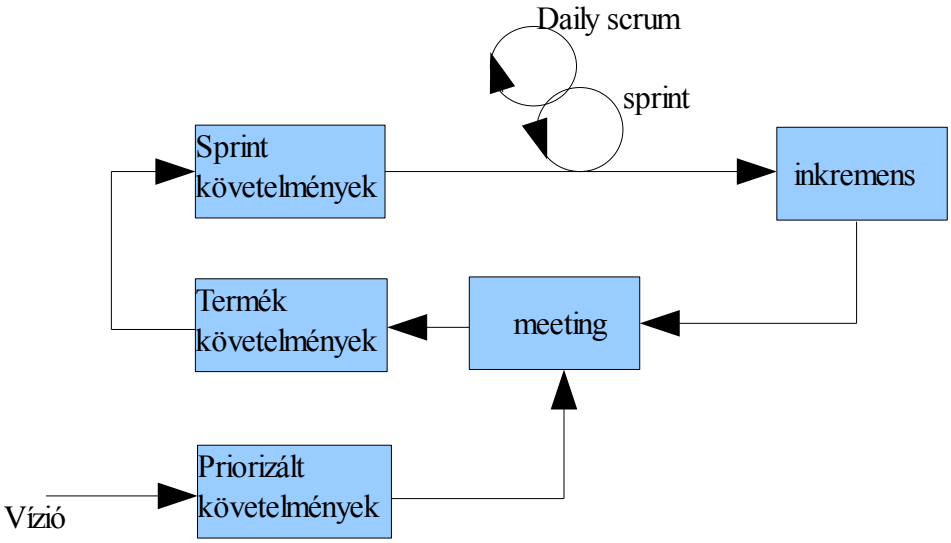
Az eddigi folyamatleírásból is sejthető, hogy a Scrum egy nagyon „merev” (az időfelhasználás tekintetében), időzített folyamat. Ennek egy következő bizonyítéka, hogy az ajánlás szerint a team minden nap tart egy 15 perces megbeszélést, amit *Daily Scrum*-nak neveznek. Itt minden csapattag megjelenik és az alábbi három kérdésre ad választ:

- Milyen munkát végzett a projekten az utolsó -előző napi- Daily Scrum óta?
- Milyen munkát tervez a következő Daily Scrum-ig? (azaz ma)
- Milyen akadályok látszanak, amik a sprint és a projekt célkitűzéseit hátráltatják?

A kérdéseken kívül meg kell beszélni az együttműködés feltételeit és egymás munkájának a segítését.

A sprint lezárását egy sprint lezáró megbeszélés adja, ahol a csapat bemutatja a

megrendelőnek az elkészült munkát. Ez egy informális megbeszélés, fél óra időtartamban.
A következő megbeszélésforma a „Srum retrospektiv”, amit a Scrum master tart a team tagjainak. A megbeszélés időtartama három óra, a fontosabb témakörök: processzek állapota, személyes inspirálás és a fejlesztés egészének állása.
A folyamat kibővített menete a 18. ábrán látható.



18. ábra: A kibővített Scrum

5.2 Scrum - A részletek

5.2.1 A Scrum master

Az 5.1.1 pontban már leírtam a Scrum master főbb szerepkörét, a projekt során megoldandó feladatait. A scrum szempontjából kucsfontosságú szerepkör bővebb ismertetése egy kérdéssel kezdődik! Miért nem volt jó a már említett projektmenedzser elnevezés? A válasz kézenfekvő! Mert ez egy kibővített, más tudást és tapasztalatot igénylő feladat! Más a filozófia, más embert kíván!

5.2.2 Projekttervezés

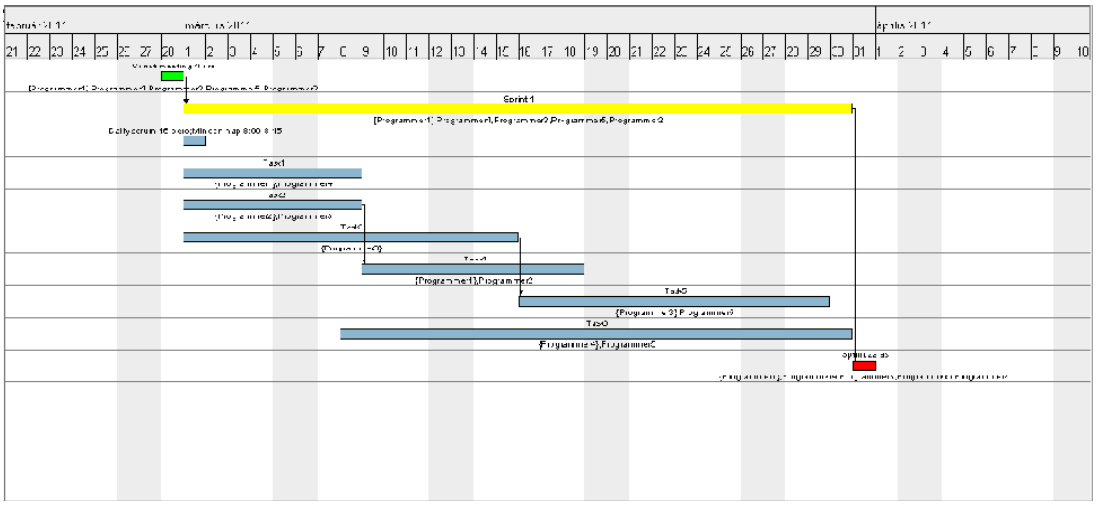
A projekttervezés egy olyan út megtervezését takarja, mely szinkronizálja a megrendelő igényeit a fejlesztő lehetőségeivel.
A projektterv számos időzítési és technikai kérdésben is segít dönteni! Elkészítése során ismereteket szerezhetünk a megrendelő elvárásairól, tanácsokkal és ötletekkel láthatjuk el, aminek következtében hatékonyabbá válhatunk. A terv másik fontos tulajdonsága a sprint tervezése során mutatkozik! A sprinttervet a projekttervvel összhangban kell elkészíteni, így az inkremensekre is hatással van.
A projekttervezés során megválaszolandó kérdések:

- A projekt finanszírozói (a megrendelők) számíthatnak-e arra, hogy a projekt lezárása után egyes követelmények megváltoznak?
- Milyen eredményeket fogunk elérni az egyes sprintek végére?
- Miért hoz a projekt hasznot a megrendelőnek? Milyen haszonnal jár a sikeres befejezés?

A leírtakból észrevehetünk egy nagyon hasznos scrum tulajdonságot! A projekt nagysága előrevetíti, hogy a kezdetekkor - a követelmények definiálásakor - nem tudjuk a megrendelő teljes igényrendszerét felmérni. Nem láthatunk minden részletet előre, a megrendelő sem tudhatja előre, hogy milyen apró funkciókat akar implementáltatni. Ezért a scrum az elején nem veszik el a részletekben, azok kidolgozását meghagyja a sprint-ekre. A sprint megbeszéléseken pedig hagy időt a megrendelőnek az igényei teljes feltárására! Ezen felül az igények felmerülésekor már az implementálást végzők - a team - jelenléte a garancia arra, hogy azonnal megértsék a kérést, technológiai oldalról közelített választ adjanak rá.

5.2.3 A projekt riportolása

Már közepes nagyságú projekteknél is nagyon nehézkes az idő-költség-erőforrás ripotolása, szemmel követése, és ami a legfontosabb: menedzselése. Talán ez az a terület, ahol a Scrum a legkevésbé tér el megszokott projektvezetési stratégiáktól. Itt is nagy hasznát vehetjük az ún. Gant-diagramnak. Erre mutat példát a 19. ábra.



19. ábra: Projektidőzítés Gant-diagramon

Az ábán megfigyelhetőek a megbeszélések, processzek és a fontosabb események. Ezen diagram segítségével egy helyen tárolhatjuk a projektünk részleteit, állapotokat és könnyen érthető, riportolható formátuma miatt a megbeszéléseken segítségül hívhatjuk.

5.2.4 A projektcsapat

A csapat felépítése és összetétele befolyásolja a projekt sikerességét. A mindenek felett ellenőrzést gyakorló Scrum master kiválasztása nem könnyű feladat, talán az első olyan döntés a projekt során, ami meghatározza annak végeredményét.
Ebben az alfejezetben néhány szempont és gondolat erejéig szeretném megvilágítani azt a rendszert, ami alapján a projektcsapat tagjait kiválaszthatjuk.

Egy nagyon meggondolandó kérdéskörrel kezdeném! Mi a fontos egy csapat hatékonysága kapcsán? Hogyan fokozhatjuk a hatékonyságot a végsőkig? Minek van nagyobb prioritása? Az egy nap alatt begépelt kódsoroknak vagy a funkció fejlesztésének? Ki dönti ezt el?
A Scrum egy sajátossága, hogy sok döntést a csapatra hagy! A csapat felelőssége, hogy a kihasználtságuk és a hatékonyságuk maximális legyen, ők tervezik és hajtják végre a munkát.

A Scrum master feladata, hogy tájékoztassa a csapatot, tanácsokat adhat, de a team magát irányítja. Ezekből következik, hogy a scrum - és ezzel az agilis módszertan - nem a kezdő fejlesztők világa! Nagy tudást, önállóságot igénylő munka.
A fejlesztés - mint már említettem - 30 napos sprintekben történik. A fejlesztők és a megrendelő döntése alapján elkészült funkciók implementálása zajlik, a sprint lezárása után a csapat beszámol a megrendelőnek. A sprint alatt mindent alárendelnek a feladat teljesítésének. Mindenki tudja a feladatát, önállóan dolgozik és megoldást keres a felmerülő problémákra.
Láttuk a team feladatát és azokat a tulajdonságokat, amikkel egy csapattagnak rendelkezni kell. Most nézzük meg a Srum master-től elvárt tulajdonságokat!
A Scrum master feladata akár kicsinek is tűnhet, hiszen egy tapasztalt, önálló munkavégzésre képes csapatot irányít! Felmerülhet a kérdés, hogy akkor miért is fontos a személye és a kvalitásai? A választ a daily scrum megbeszélés tartalmazza. Ezen a rövid egyeztetésen sok múlik! A Scrum master és a csapat áttekinti az elvégzett és a csapat előtt álló feladatot. Röviden értékelik azt, konzultálnak és próbálják optimalizálni. Ebben van a Scrum master felelőssége! Úgy kell tehát a csapat működését vezetni, irányítani, hogy a napi feladatmegoldást a csapatra bízza, mégis optimális szinten tartja a hatékonyságot!

5.2.5 A projektmegbeszélés és a sprint

Az eddigiekben számos szó esett a különböző megbeszélésekről. Most részletezem ezek tartalmát.

5.2.5.1 A napi scrum megbeszélés

- Időtartama 15 perc.
- Általában ugyazon helyen és időben tartott megbeszélés, az előző napi és az aktuális napi feladatok átbeszélésével.
- Minden csapattagnak kötelező a részvétel. Az esetleges hiányzásról mindenképpen tudnia kell a csapatnak, akár a telefonos bejelentkezés is támogatott!
- Gyors megbeszélésről lévén szó, a pontos megjelenés mindenképpen elvárt! A későket előre megbeszélt büntetéssel sújtják!

- A Scrum master előre rögzített sorrendben halad végig a csopot tagjain. (pl óramutató járásával megegyezően)
- A megbeszélés során - a már ismertetett - három kérdésre kell választ adni.
- Tilos a kérdésektől való eltérés! Rövid, lényegre törő válaszokat kell adni.
- Egyszerre csak egy ember beszélhet! Nincs idő arra, hogy megvitassanak egyéb kérdéseket!
- A megbeszélés során felmerülő ötleteket, kérdéseket egy azonnal szervezett megbeszélésen vitatják meg, semmiképpen nem a daily scrum-on!

5.2.5.2 A Sprint

- A Sprint időtartama 30 naptári nap.
- A sprint alatt kérhető külső segítség, támogatás.
- A munkaszervezés tekintetében a team egy önálló entitás! Senki nem vezetheti, önszerveződő.
- A sprint előtt megállapított bemeneti követelmények nem változhatnak meg a sprint alatt! A csapat nem enged semmilyen beleszólást.
- Ha egy sprintről kiderül, hogy járhatatlan útra tévedt, akkor a Scrum master kezdeményezi a megrendelőnél a követelmények újradefiniálását, ezzel a sprint lezárását és egy új beindítását.
- Ha a sprint közben kiderül, hogy a sprintet a sok feladat miatt nem képes a csapat befejezni, akkor konzultálni kell a megrendelővel. Egy megbeszélés tárgyát képezi, hogy melyek azok a funkciók, amelyek kikerülnek az adott sprintből. Ha nagyon sok tétel kerül ki a projektből, akkor a Scrum master döntése, hogy folytatódjon-e a sprint. (l. előző pont)
- Ha a team arra a következtetésre jut, hogy a sprint időtartama jóval több, mint az elvégzendő munka, akkor a megrendelővel egyeztetve újabb feladatokat emelhet be a sprintbe.
- A csapat tagjainak két adminisztratív feladata van:
    1. Részt venni a napi scrum megbeszélésen.

2. Az elkészült és a hátralévő feladatok naprakész adminisztrálása egy nyilvános szerveren vagy mappában, megadva a feladattal eltöltött munkaórák számát.

5.2.5.3 A sprint megbeszélés

- Időtartama 4 óra.
- A sprint áttekintésére maximum 1 óra áll rendelkezésre.
- A megbeszélés célja, hogy a már megvalósított funkciókat bemutassák a megrendelőnek, illetve a megvalósításra váró funkciókat átbeszéljék vele.
- A funkció nincs kész, amíg nincs bemutatva.
- Alapvetően csak a funkciókról kellene szólni ennek a megbeszélésnek. A nem funkcionális termékeket el kell fedni a megrendelő elől, hiszen nem képezi a megrendelés tárgyát.
- A funkció bemutatását egy csapattag végzi a munkaállomásán.
- A bemutatás során felmerülhetnek a funkcióra vonatkozó változtatási kérelmek, amik megvalósítása egy következő sprint feladata.
- A továbbiakban értékelnek a csapattagok, a Scrum master és a megrendelő.

5.2.5.4 Sprint záró megbeszélés

- Időtartama 3 óra.
- A résztvevők: a csapat, a Scrum master és a megrendelő (opcionálisan)
- A megválaszolandó kérdések:
  1. Mi ment jól a sprint alatt?
  2. Mit javíthatnának a következő sprintben? (Ami az előzőekben nem ment jól.)
- A csapat priorizálja a javításra váró dolgokat.
- A felmerülő kérdésekre a Scrum master nem ad azonnali válaszokat.

6. Összefoglalás

A szoftvertermékek fejlesztés-elmélete majdnem egyidős a szoftverfejlesztéssel. A kezdetben mutatkozó „elmélet-mentesség”-ről hamar kiderült, hogy nagyobb projektek esetében nem tartható. Erre válaszul születtek meg azok a modellek, elméletek, melyek mentén elindult a szoftverfejlesztési technikák evolúciója. Dolgozatom első részében röviden bemutattam ezt az utat, és próbáltam rávilágítani azokra az előnyökre és hátrányokra, melyek korlátját illetve előnyét jelentik az adott metódusnak.

A következő nagyobb fejezetet az adatbázisok elméletének szenteltem, és néhány helyen -hol bújtatva, hol külön kiemelve- említést tettem az újonnan kialakuló fejlesztési irányvonalakról és technikákról.

Rávílágítva a hátrányokra az is kiderült, hogy minden eddig ismertetett elv kritikus pontja az idő! Az idő, amely minden esetben kevésnek bizonyul, ezzel mutatva irányt a következő elméleteknek, fejlődési irányoknak.

Visszatérve az adatbázisokhoz, leírtam az adatbázis-fejlesztés koncepcióját, és ismertettem az agilis rendszerfejlesztés ide vonatkozó - nagyobb részben - elméleti, kisebb részben gyakorlati eredményeit.

Az 5. fejezet a gyakorlatról szól, hiszen az agilis projektmenedzsmenthez szorosan kapcsolódó scrum menedzsment leírását tartalmazza. Áttekintettem mindazon vonatkozásokat, melyek egy projekt - egy agilis projekt - elindításához alapvetően szükségesek. Szót ejtettem a szerepköröről, a megbeszélésekről és magáról a folyamatról.

A dolgozatomban összefoglalt elméleti eredmények azt mutatják, hogy a rendszerfejlesztés elméleti megfontolásait a gyakorlatba átültetve egy olyan módszerhez jutunk, amely választ ad a 21. század igényeire, ezzel mind a mérnökök, mind a felhasználók igényeit kielégíti.

7. Irodalomjegyzék

[1]	Scott W. Ambler	Agile Database Techniques
[2]	Robert C. Martin	Tiszta kód
[3]	Ian Sommerville	Szoftverrendszerek fejlesztése
[4]	Scott W. Ambler Pramod J. Sadalage	Refactorin – Adatbázisok újratervezése
[5]	Jeffrey D. Ullmann, Jennifer Widom	Adatbázisrendszerek – Alapvetés
[6]	Egyetemi jegyzet: Juhász István Szoftverrendszerek tervezése c. előadásai alapján.	

Internetes hivatkozások:

Agile Modeling (AM) Home Page	http://www.agilemodeling.com/
Manifesto for Agile Software Development	http://agilemanifesto.org/
Agile Alliance	http://www.agilealliance.org/

Köszönetnyilvánítás

Köszönetet mondok témavezetőmnek, Dr. Juhász István egyetemi adjunktusnak a diplomamunkám készítése során nyújtott szakmai segítségért, felmerült kérdéseim alapos és gyors megválaszolásáért, amelyekkel segítette a dolgozatom elkészítését.