Szakdolgozat

Horváth Dávid

Debrecen

2010

Debreceni Egyetem

Debreceni Egyetem
Informatikai Kar

Webes felülettel ellátott kérdőív-kezelő rendszer készítése nyílt forráskódú szoftverek felhasználásával

Témavezető
Pánovics János
egyetemi tanársegéd
Információ Technológia Tanszék

Készítette:
Horváth Dávid
programtervező informatikus

Tartalomjegyzék

1. Bevezetés....4
2. Miért nyílt forráskódú szoftver?....6
2.1. Multikulturális szociális hálózat 8
2.2. Újrahasználhatóság 10
3. Az alkalmazás célja....11
3.1. Felhasználói felület felépítése 12
3.2. Kezdetleges látványterv....12
4. Fejlesztés előkészítése....14
4.1. Univerzalitás 14
4.2. Szerep és jogkörök 15
4.3. Fejlesztés menete 16
5. Fejlesztés....17
5.1. Fejlesztési környezet kialakítása 18
5.2. Adatbázis felépítése 20
5.2.1. Modellezés 21
5.2.2. Normalizálás 23
5.2.3. Implementálás 27
6. Kódolás....27
6.1. Kódolás elkezdése....28
6.2. Beléptetés 29
6.3. Adatbázis kapcsolat 34
6.4. Táblák megjelenítése HTML-ben....35
6.5. Tesztelés és hibák keresése 36
6.6. Visszaállás egy korábbi verzióra 38
6.7. Refactoring 39
6.8. Dokumentáció kódolás közben 40
6.9. Version release – verzió kiadás 41
6.9.1. Mérföldkő alapú....42
6.9.2. Dátum alapú....42
7. Védelmi hibák és javításuk....43
7.1. SQL Injection 44
7.2. Directory Traversal 45
7.3. Autentikációs hibák 46
7.4. Remote Scripts (XSS) 46
8. Összefoglalás....47
9. Irodalomjegyzék....48
10. Függelék....48
10.1. Felhasznált alkalmazások....48
11. Köszönetnyilvánítás....49

1. Bevezetés

A számítástechnika és az ember világában egyre nagyobb szerepet kapnak a webes és szerveren futó, de az eredményt már kliens gépeken megjelenítő alkalmazások, technológiák. Azokat az időket, amikor a rendszer cseppet sem volt tekintettel redundanciára, vagy erőforrás elosztásra, lassan teljesen magunk mögött hagyjuk és egyre elterjedtebbek a sokoldalú, felhasználóbarát webes alkalmazások. Segítségükkel közelebb került a hétköznapi emberekhez a multimédián keresztüli élmény, mely jelentkezik mind szociális, szórakoztató és üzleti szférában. Manapság nem nagy teljesítményű számítógép, sokkal inkább egy gyors kapcsolat szükségeltetik a szerver irányába ahhoz, hogy használatba vehessük a szolgáltatást. Manapság sokkal könnyebb környezettudatos gépparkot, cégen belüli informatikai rendszert kiépíteni, mely a modern szervezetek információs gerinchálózatát alkotják.

A nyílt forráskód – mint eszme – pedig hozzájárul ezen szolgáltatások átlátható, biztonságos és helyes irányba fejlődéséhez. Ezen szempontok mellett persze nem elhanyagolható tény, hogy az eszközök tudása megmarad, és tanulhatnak belőle a következő generáció fiataljai, esetleg a nagyobb érdeklődésű veteránok. Az embernek mindig fontos volt a szabadság és a döntés szabad meghozatalának érzése, nemhiába alakult ki óriási interkulturális közösség, akik az informatika világában alkotnak nagyokat, projekteket és kollégákat saját maguk által megválogatva. Mivel az ilyen projektek élesen tagoltak, a szükség modularitás felé vitte a közösséget, amelynek az eredménye egymástól függetlenül fejlődő, de egymásra szervesen építő és erősen kooperáló projektek halmaza lett. A webes alkalmazások háttere is hasonló. Rengeteg szolgáltatás és funkció lett összeházasítva az évek során, felhasználva először az abszurdnak tűnő ötleteket és az aktuális trendeknek ellentmondó fejlesztéseket. Ez a párhuzam; ami a webes élmény fejlődését segítette elő és aminek az eredményeként ma már – igaz körülhatárolhatatlanul és homályosan is, de – a web 2-ről beszélhetünk. Ez főleg a magánszféra fogalmának megváltozásához vezetett és az üzleti – és egyéb szférában – komoly előrelépést jelent, átalakítva az addig használt informatikai eszközök használati struktúráját. Minőségi nyílt fejlesztésre és flexibilitásra való igény miatt egyre több eszköz íródik át valamilyen böngészőben futtatható alternatívává, biztonsági érdekeket szem előtt tartva. A rohamos fejlődésnek egyik nem kívánt mellékhatása a

biztonsági problémák elszaporodása. Lokálisan nem tárolt érzékeny információk megléte mellett biztonságérzetbe ringathatnánk magunkat, gondolván a vírusok és kártevők zárt forráskódú platformokon terjedése mellett. A valóság azt bizonyítja, hogy amit fel lehet törni, azt idővel fel is törik. Hasonló pontossággal és kitartással kell a biztonságra törekedni a fejlesztés során, mint amivel a feltörésre szakosodott profik rendelkeznek. Személy szerint azt sajnálom, hogy amíg a nyílt forráskódú közösségben ez az erő építésre – esetleg fork-olásra – használódik fel, addig a zárt forráskódú világban kódvisszafejtésre (reverse-engineering) és bináris fájlok manipulálására fordítódik.

Nagy szervezetek működéséhez rengeteg információ áramlására van szükség. Sokszor szimultán eseményekből kell információt kinyerni, minimalizálva az egyedek egymásra, vagy erőforrásra való várakozásának idejét. Egy olyan korban, amikor a személyi informatikai eszközök elérhető árúak és teljesítményűek, a szerverek pedig elképesztő mennyiségű információáradatot és sávszélességet képesek elnyelni, nem is csoda, hogy a vállalatok túlnyomó – ha nem éppen egész – része információtechnológiai gerincre épül.

Egyik nagyon fontos információáramlási irány az emberek véleményének megkérdezése és értékelése. Ez a feladat nagyobb lélekszámnál nehezen skálázható, ha egy többfunkciós rendszerről beszélünk ez hatványozottabban igaz. Webes alkalmazásként viszont ez könnyedén megvalósítható, megfelelő hozzáértéssel és tapasztalattal hozzávetőlegesen kevés idő és implementációs szakasz után kivitelezhető. Mivel ámulattal figyelem a nyílt forráskódú világ minden egyes új projektjét és érdekel a vállalati kommunikáció, úgy gondoltam ennek a szakdolgozatnak ezt választom témájául. Célom: betekintést nyújtani a nyílt forráskódú fejlesztés webes ágába, egy általános rálátást nyújtani az olvasónak úgy, hogy ezen szakdolgozat és egy-két telepítési útmutató segítségével gyorsan el tudjon kezdeni fejleszteni stabil alapokra építkezve. Valószínű, hogy az itt leírtak más vonatkozásban is használhatóak azoknak, akik még nem rendelkeznek többéves tapasztalattal fejlesztés terén.

2. Miért nyílt forráskódú szoftver?

Véleményem szerint a szoftver világ egyre inkább kétfelé szakad. Egyik oldal a zárt kódú (proprietary), zárt körülmények között fejlesztett, többnyire úgymond megvehető termékeket képviseli, a másik pedig a nyílt forráskóddal rendelkező (FLOSS - Free/Libre/Open Source Software), nyílt modellt követő, osztott fejlesztésű, általában magát a szoftvert közvetlenül nem áruló, szolgáltató oldal. Utóbbi általában adományokkal támogatható (nem nemhatározott összegű megvétel), de üzleti modellként kiegészíthető szolgáltatásokkal és szolgáltatási szint megállapodásokkal (SLA – Service Level Agreement) teszik vonzó ajánlattá a szóban forgó szoftver-megoldást.

Mint az informatikai világ történéseiből megtudhatjuk, mindkét modell életképes. A két modellt ötvözni is lehet, valójában egy skála két végpontját képviselik. A Jogi és engedély problémákat úgy tűnik új licencek létrehozásával sikerült részben megoldani. A probléma abból ered, hogy a zárt-kódú fejlesztést elősegítő licenc jogilag teljesen más megközelítést ajánl, mint például a sokak által használt GPL, mely a szabadságra helyezi a hangsúlyt. Egy köztes megoldás egy olyan duál licenc, ami megengedi a nyílt kódhoz zárt (előre lefordított) kódrészletek beépítését – bár ezt sokan elhibázott lépésnek tekintik. Tipikusan ilyen licencet használ az Android platform. A kernel és felhasználói programok nagy része nyitott, bárki számára lehívható, módosítható, de tartalmaz olyan meghajtóprogramokat (driver) vagy a szolgáltató által fejlesztett és karbantartott alkalmazásokat, melyek kódja zárt és csak dinamikusan vagy statikusan linkelhető formában érhető el. Ez lassítja a fejlest, mivel minden új kernel verzióhoz újra kell fordítani a meghajtóprogramokat, amihez csak a gyártó fér hozzá, egyfajta kontrollt biztosítva. Elképzelhető hogy ez leszabályozza egy termék élettartamát, mivel egy új kernel, vagy szoftver bevezetéséhez szükség lenne újra lefordított meghajtókhoz, amit megtagadhat. Ugyanez vonatkozik az hasonló jellegű alkalmazásokra.

Véleményem szerint a kitermelt tudás és technológia – emberi szabadság vonatkoztatásában – mindenkié. Mindenkinek szíve joga, hogy utánaolvas, tanulmányozza-e vagy esetleg tesz hozzá. A lényeg, hogy megteheti. Az egyetemi oktatás szerves részét képezi – magától értetődő okokból – az új ismeretek elsajátítása, melynek teljes egészében

elérhetőnek kell lennie, hogy azt bárki tanulmányozhassa, egészen addig a határig, amíg azzal vissza nem tud élni. Nagy hátránynak tartok olyan tudásanyagot oktatni, mely nem érhető el szabadon, mivel az már nem oktatás, hanem specializáció, ami elsősorban a terjesztő cégeknek a térnyerését segíti elő, nem a tudás szaporodását. Ha egy könyvet megveszünk, nem a tudást vesszük meg, hanem kifizetjük azt a költséget, amivel a könyv írása, szerkesztése, tördelése, minőségbiztosítása, nyomtatása és kiszállítása jár, ami persze magában foglalja azt az időt, energiát melyet a szerző ráfordított.

Ennek az írásnak a témája egy kérdőívkezelő rendszer, amelyet hogyha üzleti célra szeretnék felhasználni, akkor a telepítést, testreszabást és karbantartást végezném el pénzért, de nem a fejlesztést. Mivel jómagam vagyok a fejlesztő, én ismerem a legjobban a fejlesztett szoftvert, ezért a szoftver hosztolása is egy esetleges pénzforrás.

Ideális esetben a fejlesztéshez szükségem van a következő szoftverekre:

- Operációs rendszer
- Fejlesztői környezet (IDE)
- Teszt - produktív környezet (szerver)
    - Szerver operációs rendszer
    - Szerver szolgáltatásokat biztosító alrendszerek (SQL, HTML, PHP stb.)
- Dokumentáció kezelő rendszer
- Verziókövető rendszer

Ha a zárt világ megoldását választanám, odamennék egy a polcon lévő szoftvercsomaghoz és kifizetném a pénztárnál. A pénz eljutna a fejlesztőkhöz, akik kinyerve ebből az összegből a profitot felhasználnák az összeget a további fejlesztésekre, hibajavításokra és egyéb beruházásokra. Ebben az esetben a tudás a fejlesztőcégnél marad, ha csődbe megy a vállalkozás, a forráskód és vele együtt a tudás is elvész.

Nyílt modellt követve, letölteném ezeket a szoftvereket csomagként, felhasználnám az ezek által nyújtott lehetőségeket a fejlesztéshez és esetleg már hasonló projektek által kreált kódrészleteket (code-portions) is felhasználnék, felgyorsítva ezzel a fejlesztést. Mivel rengeteg dolgot kaptam a nyílt forráskódú közösségtől (operációs rendszer, IDE, alkalmazások, segítségnyújtás, tudásbázis, best-practices) ingyen – pedig a résztvevőknek is erőforrásaikba, idejükbe és tudásukba került amit aztán szabadon elérhetővé tettek –, ezért etikusan én is megosztom amit én alkottam, lehetővé téve másoknak az én munkámra való építkezést. Az is könnyedén elképzelhető, hogy a felhasznált rendszerek hibásak, és ahhoz hogy fel tudjam használni a munkám során, hibajavítást kell eszközölnöm; ezeket visszajuttatva a közösséghez, én magam is hozzájárulok a szoftver minőségének javulásához.

A végeredmény mindkét esetben borítékolható, kérdés, hogy milyen hatása van ennek az informatika világára. Minőségbeli különbségeket sem tudtak kimutatni, bár ez egy véget nem érő vitatér. Mindkét oldalnak megvannak a pártfogói.

2.1. Multikulturális szociális hálózat

Minden kultúrának megvannak a sajátosságai, jellegzetes, felismerhető mintái. Mivel több ember dolgozik együtt nagyobb projekteken – és itt nem csak a programozásról beszélek – a kulturális differenciát mindenkinek kezelnie kell, különben széthúzás alakulhat ki. Szerencsére egy óriási előnnyel jár egy ilyen szociális hálózat: teljesen más szemszögből közelítődik meg egy-egy probléma, feladat. Csapatmunkában dolgozó egyének a cél kitűzése és definiálása után megtárgyalják a hogyanokat, megegyeznek minden – az implementálás elkezdéséhez szükséges – kérdésben és folyamatos kommunikációt folytatva esetleges kérdésekre adnak választ egymás között. Több nyelven is folyhat a kommunikáció, de a történelem úgy hozta, hogy alapértelmezett az Angol. A gyakorlat azt mutatja, hogy a dokumentáció, változtatás-napló (changelog), tennivaló-lista (todo-list) és levelezőlista angolul érhető el, gyakran mindenki számára. Gyakran ezek már a kezdetektől nyitottak a külső érdeklődők felé, hátha valaki kedvet kap a csatlakozáshoz, vagy szakmai tanácsokkal tud szolgálni. Megtörténhet, hogy ellentét alakul ki két csapattag között. Ez ha elfajul és a fejlesztőgárda nem tud közös nevezőre jutni – mint ahogy saját magam is tapasztaltam –, kettészakad és a fejlesztés két ágon folytatódik tovább, a különböző vélemények mentén.

Másik gyakran használt hozzáállás, hogy egy csapat összekerül, elkezdődik a fejlesztés zárt kereteken belül, és csak akkor helyezik licenc alá és teszik ki az ablakba, ha már egy bizonyos szintet elértek a projektben. Sokszor ez azt jelenti, hogy addig amíg egy-két esszenciális funkciót nem sikerül működésre bírni, addig szorosan együtt dolgoznak a résztvevők, megállapodva olyan kérdésekben melyek esetleg vitát robbanthatnak ki, majd a magra való építkezés gyorsításához, elősegítéséhez nyílt projekté mozdítják elő.

Kezdetekben a levelezőlistákat használtak hibákkal kapcsolatos beszélgetésekre, új funkciók tárgyalására és mérföldkövek bejelentésére, ami néhány projekt esetében mai napig működik; azonban a kevéssé megfogható Web 2 hatása itt is érezhető. Sok fejlesztő teljes értékű blog mellett micro-blogot is vezet, de fórumokon és levelezésen keresztül is kommunikál, nem szólva az azonnali üzenetküldők és egyéb technológiák használatáról. Sok érv szól a felsoroltak ellen, de a gyakorlat azt mutatja, hogy az emberek idomultak ezekhez az információ közlő csatornákhoz és fontosabb az árnyalatok és apróságok kifejezése, mint a több rendszer egyidejű használata, magánszféra csökkenése, vagy a személyes információk védelme. Másik érdekes téma ami gyakran ellentétekhez vezet, az az információ megmásíthatóságának ténye és a naplózás hiánya. Az interneten való kommunikálásnak a legnagyobb negatívuma a testközeli kommunikáció elmaradása. Személytelenné válnak beszélgetések, a valóságérzékelés csorbát szenved. Ilyen esetekben elhangozhatnak olyan mondatok, amelyeket nem gondol a személy komolyan, és később ezt be is látja, viszont a tekintélysérülés nagyobb fájdalommal jár, minthogy beismerje azt. Ilyenkor történnek meg olyan beszélgetések, melyeket naplókkal támasztanának alá, de azok vagy meg lettek változtatva, vagy a beszélgetések nincsenek lementve. Ezek a feszültségkeltő szituációk a való életben is megtörténnek, de teljesen más eszköztár érhető el és van használatban a két teljesen más törvényeken alapuló világban.

2.2. Újrahasználhatóság

Nagyobb programok írásánál elkerülhetetlen a modulokban való programozás, az egyes elemek paraméterezéssel ellátott eljárásokba való egyesítése és ezek gyűjtése könyvtárakba. Könnyen belátható, hogy ha van két csapat, mely egy ponton ugyan azt a funkciót szeretné bevezetni a kódba, vagy ugyanazon kódbázishoz kell hozzáférniük, vagy újra meg kell azt írniuk. Gyakran nem elérhető a forráskódja egy már kidolgozott megoldásnak, mert különböző okokból zárt berkekben zajlik a kódolás. Ez magas költségekkel jár és sok erőforrást felemészt, mivel muszáj karbantartani a már megírt kódot, előkészítve az újbóli felhasználást, megkönnyítve a keresést, működésének megértését és felhasználásának módját. Gyakorlatilag ez a folyamat megfigyelhető mindenhol ahol folyamatos fejlesztésről van szó, többnyire nem egy szoftverről beszélve. Egy zárt modellben csak azok férnek hozzá akik az entitás – többnyire cég – munkatársai, aktív fejlesztők, dokumentációért felelős esetleg jogász alkalmazottak. A kódok erősen védettek, közvetlenül szélesebb közönség számára még véletlenül se elérhetőek. Többnyire átdolgozott, optimalizált binárisok érhetőek el, a modellből kifolyólag többnyire pénzért megvehető formában, nem ritkán másolás és visszafejtés védelemmel ellátva. Az a személy, akit érdekel a hogyan, és szeretné szakmai szemmel átvizsgálni a kódot hogy tanuljon belőle, egyszerűen nem teheti meg. Amit tehet, hogy illegálisan visszafejti a kódot, binárisból alacsony szintű, de már olvashatóbb kódot készít, majd ezt tovább finomítva funkció felismerő algoritmusokat futtatva magas szintű kódot generál. Ez a kód nehezen érthető, magyarázatok nélküli és rosszul strukturált. Mivel szofisztikáltabb algoritmusokat bonyolultabb felismerni és megeshet, hogy a szemantikai elemzés téved, hibás kódot eredményezhet. Lefordítva nem ugyanazt a binárist kapjuk. Másik út ebben az esetben nincs, a kód zárt, ha valaki tanulni szeretne a kódból, egyszerűen nem teheti meg. Mivel az emberi kíváncsiság határtalan, gyakran megesik az utóbbi eset, illegális útra terelve ezzel a tehetséges programozót. Ha feltesszük, hogy szándékai nem rosszak, hibát észrevéve esetleg kijavítja azt, mégsem lesz felhasználva az eredeti projektben. S ha tudomásukra jut a fejlesztőknek, még perelhetnek is, annak ellenére hogy a szóban forgó személy nem akart rosszat.

Másik út, ami teljesen az újrafelhasználhatóság jegyében született és fejlődik, a nyílt forráskódú fejlesztési modell. Alapötlete, hogy egy már megírt kód legyen mindenki számára elérhető és szabadon hozzáférhető, minimális megkötésekkel, jogilag kifogásolhatatlan, mégis helytelen felhasználás ellen védő háttérrel. A megfelelő könyvtárak és funkciók megtalálása a webes keresők használatával percekre csökkent és optimalizálódott, megkönnyítve az információ áramlását és megosztását. Mivel nagyon sok ember dolgozhat és dolgozik is ilyen könyvtárakon a világ minden tájáról, különböző időzónákban, eltérő ütemben és stílusban, a strukturálás sarokkővé vált. A kódok letöltése, módosítása és újbóli megosztása embert próbáló feladat, a verziókövetés és külön ágakon való fejlesztésről nem is beszélve. Szerencsére ma már kifinomult eszközök érhetőek el, mint amilyen a verziókövető rendszerek bővülő palettája (teljesség igénye nélkül: SVN, CVS, GIT), melyek rendelkeznek több interfésszel, differenciakövető és migráló funkcionalitással is, vagy a fejlesztői környezetekbe integrált tároló (repository) kezelő, amellyel már meglévő könyvtárakat lehet letölteni és a kódoláshoz elérhetővé tenni, pár kattintással.

3. Az alkalmazás célja

Ezen alkalmazás elsődleges célja, hogy biztosítson egy egyszerűen kezelhető, hatékony kérdőívkezelő felületet minden felhasználónak. Legyen lehetőség készíteni, kitölteni, lekérdezni, statisztikát generálni, felhasználónevet felvállalva, illetve csoportokra bontva anonim módon kitölteni a kérdéssort. Jogkör szerint legyen adminisztrátor teljes hozzáféréssel, felhasználó és véleményező. Felhasználó kreálhat új kérdőíveket, rendelkezhet a saját ívei felől, de nincs joga más kérdéssorok meghívás nélküli lekérésére. A véleményező csak azokhoz fér hozzá, melyekre meghívást kapott, azokra is csak egyszeri kitöltési lehetőséggel. A kész programban intuitív módon lehessen szöveges, több választ is megadható illetve egyetlen választ is elérhetővé tevő kérdőíveket létrehozni. Legyen sokoldalú, átjárható és logikusan felépített.

Vannak másodlagos célok is, mivel csak bővítünk, használhatóságot javítunk. Ilyen többek között a többnyelvűsítés (localization), a grafikonok készítése, az exportálás és a kérdőívvel kapcsolatos értesítő levél küldése.

3.1. Felhasználói felület felépítése

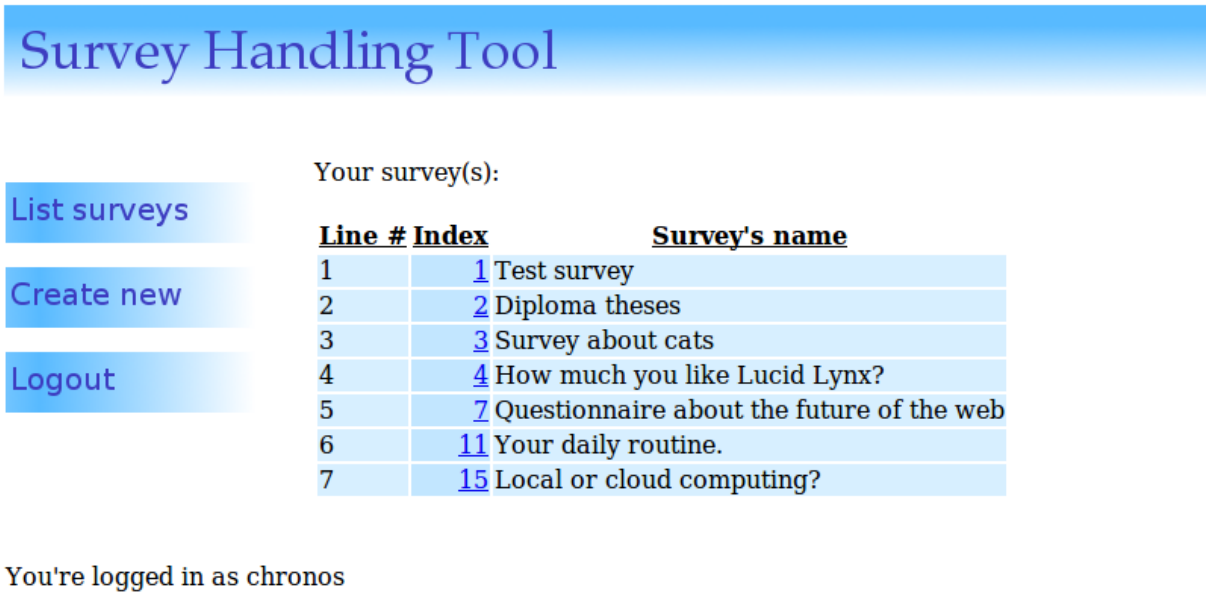
A felhasználói felületnek egyszerűnek, lényegre törőnek kell lennie, funkcionalitást előtérbe helyezve, jól strukturált menüszerkezettel, minden oldalon jelezve a felhasználónak, hogy épp hol jár. A kérdőívet lehessen félbehagyni, illetve onnan folytatni ahol elmentette a felhasználó. Legyen informatív, és közlékeny a felület, de a túl sok, vagy felesleges információ zavaró lehet. Személy szerint, jobban szeretem a minimalistább felületeket, ahol épp annyi elem van elhelyezve amennyi abszolút szükséges, de annyira jól szervezett és átgondolt, hogy a használhatósága nem hogy csökken, de még – letisztultsága révén – kényelmesebb is. Manapság rengeteg ilyen weboldalt, alkalmazást találhatunk az interneten, melyek egyre nagyobb népszerűségnek örvendenek. Az olyan oldalak ahol túl sok rendszerezetlen információ található és nem egyértelmű az éppen használt folyamat következő lépése, a felhasználók egyre kevésbé szeretnek. Gyakran egy eszköz funkcióbővítések sorozatán esik át és egy idő után nem megfelelő a felszín tervezettsége. Ilyenkor célszerű újragondolni a felhasználói felületet, melyet jobb hogyha a verziószám váltás is tükröz.

Fontos hogy a programozó ne csak informatikus fejjel gondolkozzon. Egy jó alkalmazás elkészítéséhez az szükséges, hogy felhasználói szemmel használható és felhasználóbiztos legyen, ezzel figyelembe véve a célközönséget.

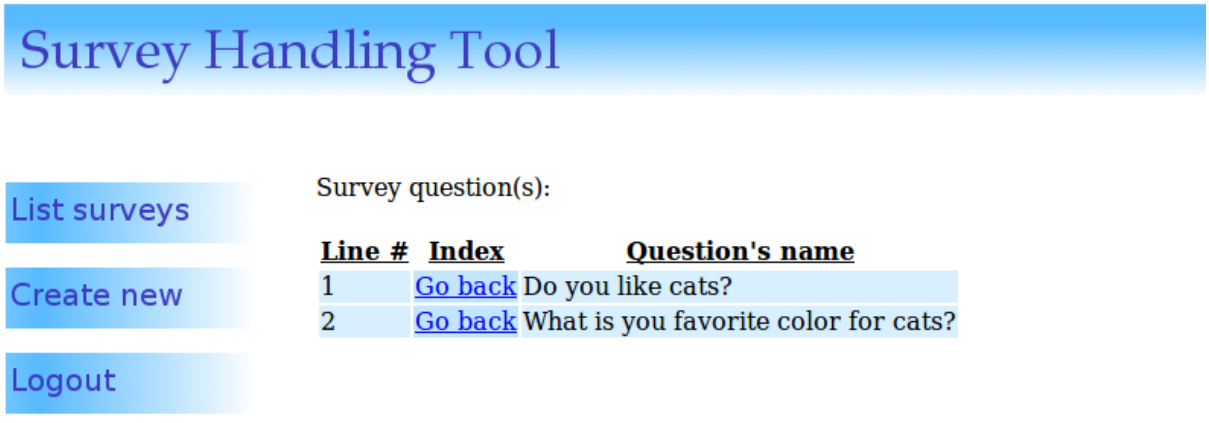
3.2. Kezdetleges látványterv

Az előbb felvázolt irányelveket szeretném levetíteni egy konkrét látványtervre, amivel az első verziót kívánom megcélozni. Ez egy prototípus amely még az első verzióig is változni fog, igényeimnek megfelelően.

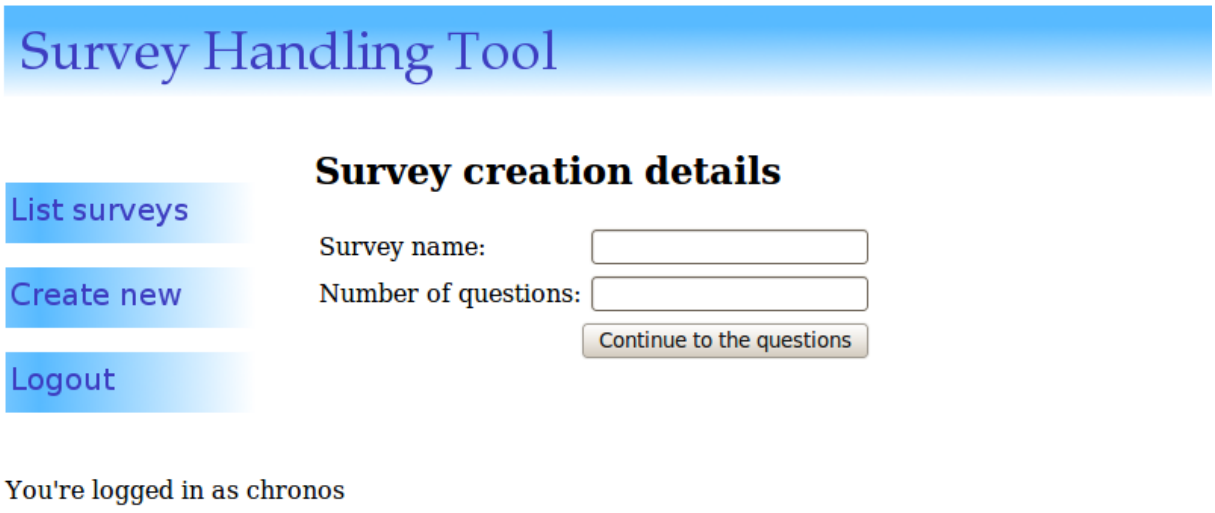
Belépéskor lehessen látni a saját kérdőíveket, az éppen elérhető funkciókat és azt, hogy épp melyik oldal az aktuális. Színnek világos és pár árnyalattal sötétebb kéket választottam, betűtípusnak pedig egy letisztult és könnyel olvashatót. A szövegek háttere kék-fehér átmenet lett, háttérszín pedig fehér. Így kellemesen átlátható az oldal felépítése. Kereteket szándékosan hagytam ki, nem akartam hogy túlságosan tagolt legyen a látvány.



A könnyű átjárhatóság kedvéért, minden bejegyzésben legyen egy hivatkozás a kérdésekre, ahol van lehetőség visszatérni a kérdőívek listájára, illetve a kérdésekre adható válaszokra. A kérdéseket mutató oldalt ilyennek képzeltem el:



Egyik fontos része ennek az alkalmazásnak az új kérdőívek létrehozása. Úgy gondolom hogy a felhasználó általi könnyebb átláthatóság és követhetőség fontosabb mint a gépi erőforrások, így maradtam a minimalista tervnél, nem szeretnék minden információt egy oldalra zsúfolni:



4. Fejlesztés előkészítése

4.1. Univerzalitás

Univerzalitás alatt egy olyan absztrakt entitást értek, amely minden esetben megállja a helyét: nincs szükség helyzetekhez, más entitásokhoz való igazításra. Viselkedése kiszámítható, működése előre megjósolható, még akkor is, ha eddig nem lett minden lehetséges házasítás gyakorlatban kipróbálva.

Az utóbbi fogalmat nehéz teljesíteni, nehéz egy olyan egységet létrehozni, ami mindig minden helyzetben úgy működik, ahogy mi szeretnénk. Nehéz, de nem lehetetlen!

Fejlesztés tekintetében az univerzalitás azért fontos, mert írhatunk olyan kódot – ami egy dolgot képes elvégezni, de azt nagyon jól – amelyet másoknak nem kell reprodukálni. Sőt mit több! Egy ilyen magasröptű elemnél óriási előny, ha különböző kultúrákból különböző

gondolkodású emberek veszik szemügyre a kódot és folyamatosan javítanak annak működésén. Apró, egyszerű alkalmazások használatával pedig a fejlődés módja lazul fel, kialakítva egy folytonosabban fejlődő alkalmazásarzenált. A későbbiek során lesz szó ilyen szoftverekről.

Elképzelhető, hogy mi már nem vesszük hasznát az általunk fejlesztett, esetleg elkészült alkalmazásnak – mert a cég profilt váltott, esetleg állást változtattunk –, vagy megváltozott egy processzus. Ilyenkor nem tűnik el kód – a bürokrácia rengeteg információt és dokumentációt felhalmozó rengetegében –, hanem mindenki számára elérhető formában, jól dokumentálva könnyen hozzáférhető. A másik indok, amiért érdemes ezzel foglalkozni, az egy szinttel feljebbi munkamegosztás. Képzeljünk el egy olyan projektet, ami kinövi a berkünket és más profillal rendelkező cég is ki szeretné fejleszteni azt a szoftvert, amit mi már elkezdtünk. Ilyenkor nem egy hagyományos értelemben vett elosztott fejlesztésről van szó – mely egyéneket köt össze –, hanem ember társulatokat, akik csoportonként együtt dolgoznak a szoftver adott részén, lazábban pedig a többi fejlesztővel is. Ha jól végezzük dolgunkat és sikerül univerzális kódot alkotnunk, hamar rádöbbenhetünk, hogy a mi kis közösségünk által fejlesztett megoldás, egy nagyobb absztrakciós szintű entitás része lett. Ez esetleg rekurzívan is igaz lehet bizonyos szintig, csoportonként. Sok ilyen példát láthatunk, egészen a GNU szoftver arzenálban gyakran használt grep parancsától a komplett operációs rendszerré bonyolódott disztribúciókig.

4.2. Szerep és jogkörök

Mint minden csoportmunkánál fontos, hogy mindenki tudja miért felelős és hol húzódnak a határok, kihez lehet vagy kell fordulni, akár abban az esetben is ha valami nem működik jól vagy nincs tisztázva. Ebben a modellben dinamikusabb szerepkörök vannak, de ez nem jelenti, hogy nem kell azokat kiosztani. Összehasonlítva más modellekkel, sokkal fontosabb a szerepkörök fix részének kiosztása, mint másoknál. Kettéválik ki mit tud elvállalni biztosan; és mi az, amit csak megpróbál, ahol a dinamikus rész nem lesz számon kérve. Az elvállalt rész nemteljesítése itt is következményekkel jár.

A gyakorlat azt mutatja, hogy a kezdeti szereposztás idővel változik, jobban széttagolódva eloszlanak a teendők a résztvevők között, lefedve ezzel teljes egészében a szükséges

feladatokat. Épp ezért nem érdemes az elején részletekbe menően definiálni a feladatköröket, idővel úgyis finomodnak és delegálódnak más személyeknek. Kiemelném a bizalom fontosságát, mivel a résztvevők többnyire nincsenek napi kontaktusban, így kommunikációjuk módosul a munkatársi viszonyhoz képest, Ezért teljesen más a nyomon követés és értékelés személyre gyakorolt hatása. Nagy szerepet kap az elhivatottság és az önkritika ebben a világban.

4.3. Fejlesztés menete

A konkrét implementálást munkálatok előzik meg, mint amilyen a teszt rendszer felépítése, fejlesztői környezet felállítása, adatbázis szerver konfigurálása, példa felhasználói fiók létrehozása és a teljes – egybefüggő – rendszer finomhangolása. Verziókövető rendszer használatakor fontos a jó beállítás, nehogy valaki rossz jogosultságokkal férjen hozzá a tárolókhoz, vagy hogy helyesen működjön az időnkénti mentés készítése esetleges katasztrófa bekövetkezése esetén.

Az Inkrementális fejlesztési modell használatából eredően egy olyan rendszer készítése az első lépcső, mely képes egy alapvető funkció ellátására. Ebben az esetben bejelentkezés és kijelentkezés képességét választottam, mivel így hozzászokok a fejlesztői környezethez, kapok egy működő weboldalt és az adatbázishoz való hozzáférés is letisztul. Több fájlt létrehozva differenciálom a szerepeket, mint amilyen a főoldal, a sikeres bejelentkezés illetve a kilépés. Ekkor még nem foglalkozok olyan témakörökkel mint dizájn, biztonság, átláthatóság vagy felhasználóbarátság. Több hangsúlyt fektetek az információgyűjtésre, informálódásra, igyekezve előrelátóan megtervezni az alapvető funkciókat, alapköveket. Fontos szerepet kap a dokumentáció és a szintaktikai strukturálási séma, mivel ezt célszerű betartani az egész projekten keresztül.

5. Fejlesztés

A tervezési fázis nagyon nagy jelentőséggel bír. Eddigi projekt tapasztalataim azt sugallják, hogy sose lehet túl nagy jelentőséget tulajdonítani az előkészületeknek. Egy projekt levezetése nagyban hasonlít a nagyvállalatoknál alkalmazott feladat delegálás metodikájára:

- Követelmények feltárása
- Delegálás
- Nyomon követés
- Értékelés

A konkrét fejlesztés a követelmények lefektetésével kezdődik. Meghatározzuk mi az amit elvárunk a kész – vagy meghatározott verziójú – alkalmazástól. Ezt eredetileg a megrendelő és a programozó (projekt vezető) egyezteti, konkretizálva az elvárt szinteket, számításba véve a szoftveres és hardveres lehetőségeket, fejlesztői humán-erőforrásokat és minden egyéb a szoftverrel és fejlesztésével kapcsolatos faktort. Fontos hogy ezt komolyan vegyük, különben idővel de-motiváltság, érdektelenség alakulhat ki a fejlesztőgárdában. Reális követelmények és elvárások meghatározása után a nyomon követés és folyamatos figyelemmel tartás lényeges eleme a minőségbiztosításnak és időkeretek betartásának. Hiánya komoly gondokat okozhat mind a delegálás, mind az értékelés fázisában. Ha nem tudja valaki pontosan mi a dolga, nem csak hogy nem fog haladni a projekt, de még értéktelennek és haszontalannak is fogja magát érezni. Ha valaki nem tudja mi a feladata, de egyébként egyenlőre motivált, joggal a vezetőt fogja megkérdezni a részletek pontosításáért. Minél több részlet szorul pontosításra az első fázist elhagyva, annál több energia és erőfeszítés szükséges a korrekcióhoz. A folyamatos és önálló munkavégzés az egyik fontos motiváló erő egy résztvevő számára. Ha folyamatosan olyan akadályokba ütközik, melyeket önmaga nem képes terv illetve információ hiányában leküzdeni, sejthetjük, hogy a projekt végére akár teljesen elveszti az érdeklődését. Értékeléskor ez a jelenség komoly konfliktusokat szülhet. Gondoljunk csak bele: egy jó szakember minden tőle telhetőt megtesz, de az információéhség

miatt nem tud úgy haladni ahogy egyébként képességei lehetővé tennék, és így negatív kép tükröződik az eredményeken. Másik ilyen fontos faktor, a visszajelzés, angolul feedback. Visszacsatolás nélkül az egyén vesztheti el a munkája fontosságágának érzését.

Minden szervezet számára fontosnak kell lennie az előző bekezdésben taglalt négy fázis ismeretének, elfogadásának és betartásának. Már csak azért is, mert minden munkáért ellenszolgáltatást kap a dolgozó. Nyílt forráskódú világban az alappillér gyakran nem a pénz, hanem valami emberibb, más kifejezéssel: kevésbé materiális. Egy olyan embernek aki nem kap kézzel fogható ellenszolgáltatást, hanem mással éri be – nyilvánvalóan egyéni preferencia, hogy kinek mi értékes – ami többnyire elismerést, sikerélményt, vagy gyakorlati hasznot jelent, nagyobb de-motiváló hatása van a rajta kívül álló okokra visszavezethető rossz teljesítmény megállapításának, mint egyébként.

Ha megvannak a követelmények, akkor a fejlesztéshez szükséges környezet megteremtése, majd az első, működő verzió előállítása a cél. Ezt a programozó, esetleg a projekt vezető indítja útjára, a lehetőségekhez mérten választva ki az egyes modulokat. Nyílt forráskódú fejlesztés révén, némileg több hangsúlyt kell fektetni a fejlesztés univerzalitására, határok és megállapodások kötésére és elfogadására.

Ezek az elvek nagyobb fejlesztői csoportokra igazak, de ismeretük hozzásegített ahhoz, hogy minőségi munkát végezzek, hosszabb távon optimálisabb utat bejárva.

5.1. Fejlesztési környezet kialakítása

A fejlesztés elkezdése előtt érdemes utánaolvasni milyen eszközök és technológiák érhetők el! Ezekről rengeteg információ találhatunk az interneten és a szakirodalomban.

Elképzelhető, hogy már vannak olyan kidolgozott módszerek és eszközök, melyek meggyorsítják, elősegítik – esetleg valamilyen formában – leegyszerűsítik a folyamatokat. Ilyen lehet egy verziókezelő, hibakövető rendszer, vagy egy jobban átgondolt fejlesztői környezet. Mivel tiszta lappal kezdjük ezt a fejlesztést, lehetőségünk van jól informálódni, átgondolt döntéseket hozni és ezekre a stabil alapokra építkezni. Egy váltás, vagy döntés módosítása komoly erőfeszítésekbe kerülhet, ám ezeknek az alapköveknek a lecserélése

hosszas és idegtépő folyamat lehet, ami általában fő verziószám váltást is magában foglal.

Operációs rendszernek az Ubuntu 9.10-es asztali (desktop) verzióját választottam, mivel gazdag szoftverarzenállal és nagy létszámú aktív közösséggel rendelkezik. A gyakorlatban sok területen jól teljesítő Debian Linux disztribúció képezi az alapját, mely főleg stabilitásáról és jól átgondoltságáról híres. Fejlett csomagkezelő rendszerrel rendelkezik, amely az alkalmazások és komponensek kezelését – többnyire telepítését és eltávolítást, de nem legutolsó sorban frissítését is hivatott fájdalommentessé tenni. Mivel egy nagyon moduláris rendszerről van szó, a függőségek és verziók követése az egyik legnagyobb kihívás, de a gyakorlat azt mutatja, hogy az apróbb – könnyen lekezelhető – problémák ellenére, tökéletesen helytáll, magasan meghúzva a lécet a hasonló rendszerek számára. Kiemelnék itt egy érdekes funkciót, mely a Unix alapú rendszerek rendelkezésre állási idejét hivatott megnövelni. Nem egy olyan szerverről hallani mely akár 8-10 éven keresztül is, újraindítás nélkül üzemel. Az Ubuntu magját alkotó Linux kernelhez elérhető egy olyan csomag, a Ksplice (http://www.ksplice.com/), mely az éppen futó kernel foltozását (patching) teszi lehetővé valós időben, újraindítás nélkül. Magas szerver-rendelkezésre állási idő igénye mellett ez a funkció több mint kecsegtető.

Fejlesztői környezet terén nincs könnyű dolga az embernek, ha webes felületre szeretne fejleszteni nyílt forráskódú alapokon. Egyik oldalról ott vannak a már jól bevált fejlesztéshez is könnyen használható szövegszerkesztők – mint amilyen a Geany, a Notepad++, a Kate és társaik –, melyek bármely fázisban jól jöhetnek. Komolyabb eszközök után kutatva könnyen rátalálhatunk a méltán népszerű Eclipse és NetBeans párosra, melyek főleg funkcionalitásukkal hívják fel magukra a figyelmet. De vannak könnyedebb környezetek is, mint az Anjuta DevStudio, vagy a gPHPEdit. Én fő-környezetnek az Eclipse-et választottam, sokoldalúsága és funkcionalitásának bősége miatt.

Adatbázis rendszerek közül több is felvetődött, úgy mint az objektum-relációs PostgreSQL, a nemrég feltűnt klaszteres Cassandra és a jól bevált MySQL. Ha nagy leterheltségűnek terveznénk az oldalt, valószínű hogy a Cassandra lenne a legjobb választás, de én egyenlőre maradnék a MySQL-nél. Adatbázis motorok közötti migráció persze lehetséges, ha később igény merülne fel rá.

SQL munkaasztalok közül a MySQL Workbench és az SQL Workbench/J örvend nagy népszerűségnek, de én beérem egy egyszerűbb megoldással, a phpMyAdmin-al. Ez egy PHP alapú adminisztrációs felület, mely rendelkezik minden olyan funkcióval, melyre a fejlesztés során szükségem lehet.

Webszerverként a választás az Apache HTTP Server-re esett. Már több mint 15 éve van szolgálatban, fejlesztése mai napig aktívan folyik, 2009 szeptemberében a világ összes kiszolgálásának 54.48%-át teljesítette Apache szerver. Felépítése moduláris, beépülő modulokkal és új funkciókkal lehet bővíteni, mint amilyen az autentikációval vagy titkosítással kapcsolatosak. Meglepő módon a PHP, TCL, és a Python nyelvek is ide tartoznak.

A teljes környezet telepítése az operációs rendszerrel kezdődik, majd a frissítések letöltése után következhet a szerver alkalmazások működésre bírása. Utóbbit megtehetjük kézzel is, de én a már előre előkészített LAMP, avagy XAMPP csomaggyűjteményeket részesítem előnyben. Gyakorlatilag egy órán belül kapunk adatbázis és webszerverrel felszerelt operációs rendszert.

5.2. Adatbázis felépítése

Az adatok tárolásához egy jól átgondolt és konzisztens rendszert kell tervezni, ami jól beleillik az adatbázis-kezelő rendszerbe. A célom az volt, hogy egy olyan struktúrát hozzak létre, amely nem tárol fölösleges adatokat és nem kívánt duplikációkat – tehát nem redundáns –, és logikusan szervezett, az egyes lekérdezések szempontjából nem túlbonyolított.

Az adatbázis felépítése három fő-részből tevődik össze:

- Modellezés
- Normalizálás
- Implementálás

5.2.1. Modellezés

A modellezés során elemezzük a való világ modellezni kívánt részleteit, irreleváns részletek kizárásával emeljük ki a fontos – a modell által hordozni kívánt – információkat. Ez egy absztrakt feladat, többnyire ceruzát és papírt ragadok, esetleg egy tervezést, gondolkodást elősegítő alkalmazást használok (nagy favorit az XMind), mellyel több lépcsőben tudok finomítani a modellen. Több mind valószínű, változtatni fogok a modellen fejlesztés során, mivel elég valószínűtlen, hogy elsőre képes leszek mind a modellezés, mind a programozás követelményeinek eleget tevőt készíteni. Ebben az esetben két kategóriába tudom sorolni az információkat:

- Jogosultság és felhasználó kezeléssel kapcsolatos
- Kérdőívekkel és válaszokkal kapcsolatos

Az előbbinél szükség van az összes olyan információra, amellyel azonosítani lehet egy felhasználót: meghatározható a jogköre és szükség esetén értesíteni tudjuk a levelezési címén. Elnevezése ötletesen accounts, ami annyit tesz: felhasználói fiókok. Tehát a következőket fogja tartalmazni:

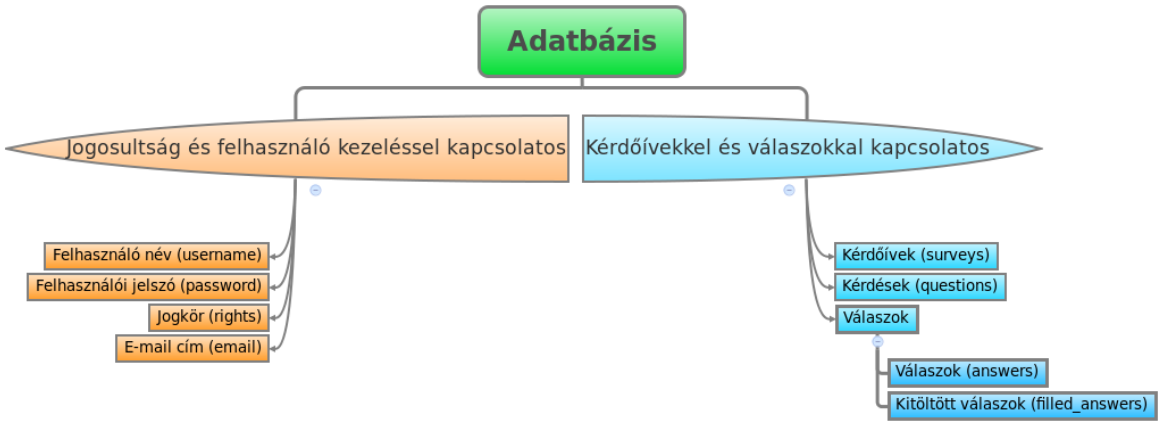
- Felhasználó név (username)
- Felhasználói jelszó (password)
- Jogkör (rights)
- E-mail cím (email)

Utóbbi jóval összetettebb, már átgondolásánál látszik, hogy nem lehet egy táblában összefoglalni az összes információt. Három nagyobb kategóriát tudok elhatárolni első ránézésre:

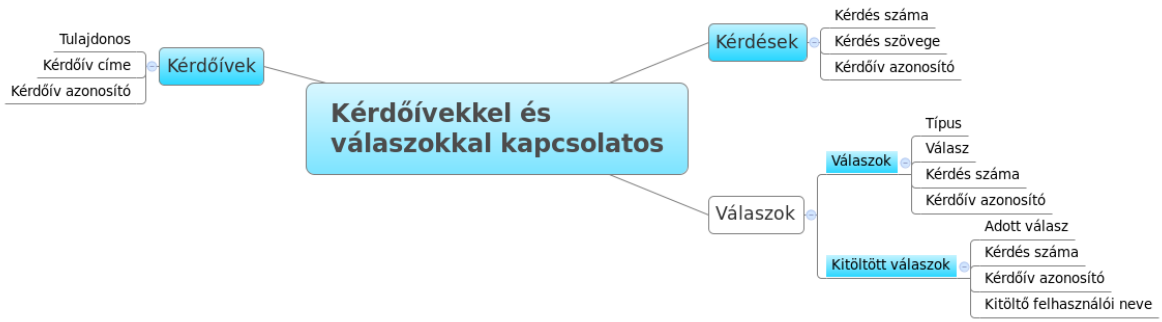
- Kérdőívek (surveys)
- Kérdések (questions)
- Válaszok (answers)

Újra átfutva rajta beláthatjuk, hogy a kérdőíveket felhasználókhoz tudjuk kötni, a kérdéseket kérdőívekhez, a válaszokat pedig a kérdésekhez. Kérdés, hogy a válaszokat miként kezeljük. Több válasz-típust kell definiálnunk, mint a választógomb, a jelölőnégyzet és a szöveges. Másik kérdés, hogy az adható válaszokat, hogy szeparáljuk el – ha egyáltalán szükséges – a felhasználó által megadott válaszoktól. Mivel a legnagyobb számú táblák valószínű a válaszokkal kapcsolatosak lesznek, ezért két táblára választom szét: az egyik az adható válaszokat tartalmazza majd, a másik pedig a már megadott válaszokat, melyeket köthetünk felhasználókhoz.

Az eszmefuttatásom végére a következő ábra tárult elém:



Sajnos ez az ábra nem fedi le az összes, az adatbázis létrehozásához szükséges részletet, a logikai viszonyok és egyes részletek még hiányoznak. Ezt láthatjuk a következő ábrán:



Ezen az ábrán már sokkal több információval rendelkezünk, egyértelműen látszanak duplikációk, amelyeket felhasználhatunk a táblák összekapcsolásához. Vannak olyan tulajdonságok, melyek maguktól értetődnek – mint például a felhasználói név egyedisége –, de vannak olyanok is, melyeket csak az implementáció közben veszünk észre. Ebben az esetben az automatikus számozás funkció könnyítette és egyszerűsítette a munkám. Lehetőség van automatikusan növekvő számozás használatára MySQL-ben, ahol az attribútum kitöltése nem kötelező, így a rendszer választja ki a következő sorszámot. Megjegyezném, hogy ez a számozás nem egyedi kulcs és van lehetőség manuálisan beszúrni sorokat, ugyanúgy, mint törölni. Változás esetén vigyázni kell, hogy kezeljük le az anomáliákat. Több variáció is lehetséges, úgy mint a tábla elkészítése újra, de a következő SQL parancs is hatásos:

```
alter table "table_name" auto_increment=1
```

Hasznos technikai segítség volt számomra, hogy van lehetőség lekérni a beszúrt sor kapott sorszámát egy SQL paranccsal:

```
last_insert_id();
```

Ezt az információ munkamenetenként egyedi, így más kapcsolatoktól függetlenül van lehetőségünk lekérdeni. Mindig a legutóbbi változtatáshoz fűződő számlálóhoz férünk hozzá, vigyázzunk, nehogy felülíródjon még mielőtt felhasználnánk.

5.2.2. Normalizálás

Normalizálás során arra törekszünk, hogy megszüntessük az egyes információk többszöri – és fölösleges – tárolását, és olyan struktúrát építsünk ki, melyben minden adat közvetlenül meghatározható, vagy származtatható. Az adatbázis normalizálásnál normálformákról beszélünk egytől ötig, melyből az első három elegendő ahhoz, hogy elkezdjük a munkát. Igyekeztem tudatosan felépíteni az eddigi struktúrát, de a biztonság kedvéért vegyük ezeket sorra az eddigi adathalmazainkkal.

1NF – Első normálforma:

Egy reláció első normálformában van, ha minden eleme egyszerű, tehát oszloponként egy értéket tárolunk. Ez teljesül mindegyik adatra, gondoljunk csak a felhasználói névre, vagy a kérdőív azonosítójára. Ha egy oszlopban több adat is megjelenne, akkor azt egyenként le kell választani, új oszlopokat hozva ezzel létre.

2NF – Második normálforma:

Egy reláció második normálformában van, ha az első normálformában van és minden másodlagos attribútum funkcionálisan függ a reláció valamely kulcsától.

Ezt úgy tudjuk a legegyszerűbben felismerni, ha megnézünk a tábla egy sorát és ellenőrizzük, hogy nincs-e több olyan oszlop, amelynek tartalma egyszerre jelenik meg, és van kapcsolat köztük. Feloldani úgy lehet, hogy külön táblába helyezzük át a kötődő oszlopokat.

3NF – Harmadik normálforma

Egy reláció harmadik normálformában van, ha második normálformában van és a reláció nem tartalmaz funkcionális függőséget a nem elsődleges attribútumok között.

A normalizálás folyamatát egy olyan táblával akartam kezdeni, ami hozzávetőlegesen egyszerű és nem áll sok elemből, majd bonyolultabbakkal folytatni. Az eddigi adattárolásról folytatott gondolataimból kiderül hogy a felhasználói információkat tartalmazó tábla lesz a legkevésbé változó, a funkciója minden esetben ugyanaz.

accounts		
username	Egyedi, nem lehet nulla	varchar(30)
password	Nem lehet nulla	varchar(30)
right	Nem lehet nulla, alapértelmezett: “u”	set('a','c','u')
email	Nem lehet nulla	text

Ennél a táblánál semmi dolgunk nincs, minden tulajdonság a felhasználói névhez kötődik, minden bejegyzés ebben a táblában egyedi lesz, mivel nem engedjük meg hogy egy username-ből legyen több is.

Következő tábla nagyobb lesz, mivel tartalmaznia kell a kérdőívekkel kapcsolatos információkat.

surveys		
PRIMARY	Egyedi, nem lehet nulla és auto increment	int(11)
survey_name		text
question_number	Nem lehet nulla	int(11)
question_text		text
answer_number	Nem lehet nulla	int(11)
answer_type	Nem lehet nulla	set('txt','cb','rb')

Ebben a táblában sok ismétlődő információ megjelenik ha elkezdjük feltölteni adatokkal. Ez nyilvánvalóan nem előnyös, minél hatékonyabban tároljuk az adatokat, annál gyorsabb és tisztább lesz az adatbázisunk. Ebből a táblából ki kell venni és újba kell helyezni azokat az elemeket, melyek függnek egy másik elemtől, de nem az elsődleges kulcstól. Ha beszúrnánk egy új választ az egyik kérdésre, a sor kapna egy egyedi azonosítót amivel csak bonyolítanánk a lekérdezéseket, ráadásul rengeteg fölösleges információt kellene tárolnunk.

Ezek alapján abszolút szükséges a surveys táblába az elsődleges kulcs és a közvetlenül kérdőívhez kapcsolódó információk, mint a neve és a készítő felhasználóneve. Íme:

surveys		
PRIMARY	Egyedi, nem lehet null és auto increment	int(11)
survey_name		text
owner	Accounts tábla username-el áll kapcsolatban	varchar(30)

Az előbbiből tanulva, más megközelítésben kezdtem neki a megmaradt adatok rendszerezésének. Első nekifutásra úgy tűnik szükség van egy kérdésekkel és egy hozzá kapcsolódó válaszok táblához. Az is felvetődött bennem hogy az adható válaszok és a felhasználók által adott válaszok is külön információk halmaza, nem lehet őket egyszerre kezelni.

questions		
survey_number	Surveys tábla elsődleges kulcsára mutat	int(11)
question_number	Nem lehet null, elsődleges kulcs	int(11)
question_numbering		int(11)
question_text		text

answers		
question_number	Questions tábla elsődleges kulcsára mutat	int(11)
answer_number	Elsődleges kulcs, auto increment, nem lehet null	int(11)
answer_numbering		int(11)
answer	Nem lehet null	text

user_answers		
answer_number	Nem lehet null,	int(11)
answer_counter	Auto increment, nem lehet null	int(11)
username	Opcionális	varchar(30)

Normalizálás során elbizonytalanodtam egy bonyolultsági fok elérésekor, de sokat segített, hogy leírtam mit szeretnék, majd rajzokkal rajzokkal bizonyosodtam meg róla hogy a megoldás helyes. Úgy gondolom hogy a létrehozott sémával már el lehet kezdeni dolgozni. Nyilvánvalóan szempontok alapján lehet optimalizálni, de érdemes ezt későbbre hagyni. Ilyen például a kapcsolótáblák használata, amivel megnöveljük a táblák számát, de átláthatóbb, kezelhetőbb rendszert hozunk létre. Vigyázzunk viszont, ha van rá mód, ne halasszuk éles felhasználás elkezdése utáni időpontra. Már feltöltött adatbázissal dolgozni nagy tapasztalatot és óvatosságot igényel.

5.2.3. Implementálás

A táblák feltöltésére több mód is kínálkozik. Ha szeretnénk gyakorlati megközelítést alkalmazni, PHP-ből meghívott SQL parancsokkal kreálhatjuk a táblákat és tulajdonságaikat. Ennek nagy hátránya, hogy ezt egy felhasználó se fogja használni, így nagyobb a valószínűsége, hogy biztonsági rést hagyunk magunk után. Kényelmesebb megoldást nyújt a phpMyAdmin, amivel szintén használhatunk SQL lekérdezéseket adminisztrátori jogkörrel, vagy van lehetőség kézzel megszerkeszteni a táblákat, kiválasztva a kívánt tulajdonságokat, megadva a részleteket. Mivel valószínűleg módosulni fog az adatbázis sémánk, ezért többnyire itt fogunk változtatásokat végezni.

6. Kódolás

A konkrét fejlesztést két oldalról kell megkezdenünk, Eclipse-ben és phpMyAdmin-ban. Eclipse-ben egy új projektet kell létrehoznunk, úgy hogy az összes fájl elérhető legyen a webszerver számára. Ezt a legegyszerűbb módon a /var/www könyvtár használatával tehetjük meg, beállítva jogosultságokat úgy, hogy legyen hozzá írási jogunk. Mivel elég gyakran kísérletezgetem, és nincs verziókövető rendszer feltelepítve, ezért több könyvtárral érem el azt, amit GIT vagy SVN segítségével nagyobb projektek. Van több teszt könyvtár és egy fő „branch”; ha valami a teszt rendszeren jól működik, be lehet olvasztani a fő ágba. Kíváncsi természetem és kísérletező kedvem miatt még nem alakítottam ki stabil szokásrendszert és nincsenek bevált módszereim, illetve így sokkal könnyebben tudom követni a változásokat. Hátránya a lassúság: néha órákat kutatok egy probléma optimálisnak vélt megoldása, esetleg egy már megírt implementáció után, többnyire sikerrel, illetve nem módosított fájlok többszörös tárolása.

A projektünkön belül hozzunk létre php kiterjesztésű fájlokat és teszteljük le hogy működnek-e. Apache szerver teszteléséhez egy bevált szokás, hogy akár egyszerű szövegszerkesztővel létrehozunk egy – például – phpinfo.php nevű fájlt, a következő tartalommal:

```
<?php phpinfo(); ?>
```

Ezt elmentve egy a /var/www alatti könyvtárba már hivatkozhatunk is rá. Amennyiben a gyökér könyvtárba helyeztük el, böngészőnkben nyissuk meg a **http://localhost/phpinfo.php** URI-t, ahol az Apache szerverrel kapcsolatos információkat fogjuk látni. Ha sikerült, azt jelenti hogy egy általunk létrehozott kód lefut, egyébként valami felett elsiklottunk és korrigálásra szorul. Fontos megjegyezni, hogy ha nincs szükségünk az itt látható paraméterekre, töröljük az említett fájlt, ugyanis tartalmazhat olyan információkat melyekkel biztonsági rések használhatóak ki.

6.1. Kódolás elkezdése

A programkód írását akkor ideális elkezdeni, ha minden rendszer működik, legalább annyira hogy hivatkozni tudjunk rá, így jobban oda tudunk figyelni a konkrét programozási problémák megoldására és a fejlesztés folyamatának mederben tartására. Ebben az esetben ideális ha a MySQL, Apache, Eclipse és phpMyAdmin stabilan működnek, ergo a számítógép újraindítása esetén sem kell a beállításokkal, vagy rendszerek elindításával időt pazarolnom, és az operációs rendszer képességeivel is jó hogyha tisztában vagyok. A következő lépés fájlok létrehozása a webszerver alapértelmezett könyvtárába, a /var/www-be pozicionált projektünkbe.

Kezdetben szükség van egy belépést és munkamenet kezdetet kezelő, egy regisztrációt szolgáló, egy kiléptető és egy a kérdőíveket kezelő lapra. Elképzelhető, hogy az utolsóként felsoroltat több részre kell majd bontani, mivel ebben lesz a lényegi kód, és úgy tűnik mennyiségileg ez fogja a legnagyobb részét alkotni. Kézenfekvő megoldás lenne minden egyes funkcióra külön PHP fájlt létrehozni és eszerint létrehozni a fájlok közötti linkeket, de az is megvalósítható, hogy egész végig egy fájllal dolgozunk és paraméterek változtatásával frissítjük a lapot. A második esetben teljesen más jellegű strukturálást kell végeznünk, külön fájlokba delegált eljárások meghívásával jobban követhető és fejleszthető állományok jönnek létre. A disztribúciós feladatot a fő fájl adja, ami kódolási szempontból nem ideális, mivel ha többen dolgoznak rajta, elképzelhető, hogy várnia kell valakinek, amíg az éppen állományon dolgozó kolléga befejezte a módosítást.

Mint már említettem, a kód szerkezetét folyamatosan alakítjuk, így ez nem két összeférhetetlen módszer, annál inkább két irányvonal, amivel jó ha tisztában vagyunk. Én a kettő között lavírozva szeretném kihasználni mindkettő előnyeit.

6.2. Beléptetés

Az oldal kezdőlapja legyen index.php, amin keresztül be fogunk tudni lépni az oldalra. Ez az alapértelmezett kezdőoldal a szerveren, ha csak az elérési útvonalat adjuk meg fájlnév nélkül, akkor is ez az oldal fog betöltődni. Szükségünk van egy HTML vázra, mely tartalmaz minden szükséges mezőt és szöveget, majd módosítjuk úgy hogy megfelelően reagáljon az adatok elküldése esetén.

```
<html>
<title>Login page – Survey Handling Tool</title>
<body>
        <h1>Login</h1>
        <form action="" method="post">
        <table align="left" border="0" cellspacing="0" cellpadding="3">
        <tr><td>Username:</td><td><input type="text" name="user"
                maxlength="30"></td></tr>
        <tr><td>Password:</td><td><input type="password" name="pass"
                maxlength="30"></td></tr>
        <tr><td colspan="2" align="right"><input type="submit"
                value="Login"></td></tr>
        </table>
        </form>
</body>
</html>
```

A fentebb látható kód ezt eredményezi:

HTML kód PHP-be való ágyazására két utat választhatunk. Az egyik hogy megszakítjuk a kódot, beszúrjuk a kívánt sorokat, majd jelezzük az értelmezőnek, hogy innentől ismét futtatható kód áll rendelkezésre. Fontos érzékelnünk, hogy amíg az előbbi a kliens gépén futó, szerver szempontjából egy „üres" szöveges rész, addig az utóbbi egy forráskód, amit értelmezni kell. A leggyakrabban a PHP HTML kódot generál, és a felhasználóhoz sose jut el PHP kód. Mivel a HTML önmagában kevés lenne a felhasználóval való interakcióra, ezért más technológiákkal kiegészítve használják, mint amilyen a JavaScript, AJAX vagy CSS. Lássunk egy példát a kódmegszakításra:

```
function survey_details(){ ?>
        <h2>Survey creation details</h2>
        <form action="creation.php?action=creation&amp;creation=1"
              method="post">
        <table align="left" border="0" cellspacing="0" cellpadding="3">
        <tr><td>Survey name:</td><td><input type="text" name="s_name"
              maxlength="30"></td></tr>
        <tr><td>Number of questions:</td><td><input type="text"
              name="s_qnumber" maxlength="30"></td></tr>
        <tr><td colspan="2" align="right"><input type="submit" name="creation"
              value="Continue to the questions"></td></tr>
        </table>
        </form>
<? }
```

Itt a két PHP token („<?” és „?>”) között egy statikus szövegrész található, melyet a felhasználó böngészőprogramja fog értelmezni. Ilyenkor nincs lehetőségünk módosítani a szövegrészt, lekéréskor egy az egyben el lesz küldve a kliensnek.

A másik módszerrel egy eljárást hívunk meg az információ közlésére, mely a kimeneti folyam jelenlegi végére fűzi a kívánt karaktereket. Itt lehetőség van dinamikus, bonyolultabb tartalom beszúrására is, melynek segítségével például adatbázis táblákat tudunk megjeleníteni, vagy változóktól teszünk függővé bizonyos elemek megjelenését. Lássunk tehát egy példát a HTML-be való beágyazásra:

```
<html>
 <title>Registration Page</title>
 <body>
                    <? displayStatus(); ?>
 </body>
</html>
```

Itt egy eljárást hívunk meg paraméter nélkül és íratjuk ki az éppen aktuális státuszt. Ez egy primitív példa, ahogy tapasztaltam, más programozástechnikai eszközökkel kombinálva könnyedén kreálhatunk magas bonyolultságú kódot.

A két megközelítés egymást kiegészítően kell használnunk. Az első módszer akkor előnyös ha egy nagyobb HTML kódrészletet szeretnénk beilleszteni. Ilyen tipikusan a fájl eleje és vége, a fej és lábrész vagy egy statikus rész. A második pedig minden egyéb esetben hasznos.

A következő lépés a HTML kód által elküldött információk feldolgozása, elkapása szerver oldalon. A $_GET és $_POST metódusok szolgálnak a böngészőből indított kommunikáció fogadására az Apache szerveren. A különbség az köztük, hogy amíg a $_GET eljárásnál a felhasználó által is látható böngészősávban adódik át az információ, addig az utóbbinál a háttérben történik mindez.

URI-ben átadódó információra egy példa:

```
http://localhost/test/creation.php?action=link&what=list.php
```

A PHP fájlneve után egy kérdőjel kezdi a változónevek felsorolását, egyenlőségjel után következik a változó neve, majd és jel előzi meg a következő változót. Ezen információk elkapása és felhasználása (jelen esetben az *action* és *what* változók) PHP kódban így néz ki:

```
if (isset($_GET['action'])) {
 switch (strtolower($_GET['action'])) {
                        case 'link' : do_link(@$_GET['what']); break;
                        case 'creation' : do_link("create.php"); break;
                        default : break;
                        }
 }
```

$_POST metódusnál hasonló a helyzet, annyi különbséggel hogy ott a különböző HTML elemek neveire kell hivatkozni, például egy editbox vagy checkbox nevére. A változó nevek itt a *user* és a *pass*.

```
Username:<input type="text" name="user" maxlength="30">
Password:<input type="password" name="pass" maxlength="30">
```

A PHP kód pedig a következő:

```
$_POST['user'] = trim($_POST['user']);
  if(strlen($_POST['user']) > 30){
    die("The username is longer than 30 characters, please use a shorter one..");
  }
```

Ezen változók tartalmától függővé tehetjük a megjelenő tartalmat. Egy lehetséges felhasználás a rejtett elemek használata, amik értékét a folyamattól függően változtatjuk.

```
<form action="test6.php" method="post">
<?php
if (isset($_POST['process'])){
 if ($_POST['process'] == 1){
        $number = 2;
        echo "Second page<br>
        <input type=\"hidden\" id=\"process\" name=\"process\" value=\"$number\">";
 }
 if ($_POST['process'] == 2){
        $number = 3;
        echo "Third page<br>
        <input type=\"hidden\" id=\"process\" name=\"process\" value=\"$number\">";
 }
 if ($_POST['process'] == 3){
        $number = 3;
        echo "Last page<br>
        <input type=\"hidden\" id=\"process\" name=\"process\" value=\"$number\">";
 }
}
else {
 $number = 1;
 echo "First page<br>
 <input type=\"hidden\" id=\"process\" name=\"process\" value=\"$number\">";
}
?>
<input type="submit" value="Next page">
</form>
```

Ez a kódrészletet arra használtam hogy kikísérletezzem hogy lehet egy PHP fájl több funkcióra is felhasználni. Ezen módszer segítségével könnyebben csoportosíthatjuk – többek között – a belépéssel kapcsolatos teendőket, logikát.

6.3. Adatbázis kapcsolat

Az adatbázis hozzáféréshez először kapcsolatot kell teremtenünk a szerverrel, majd ezt a kapcsolatot fogjuk használni az SQL utasítások lefuttatására. Érdemes egy database.php nevű fájlt létrehozni, melyet minden szükséges alkalommal meg tudunk hívni. A kód a következő lesz:

```
<?
/**
 * Connect to the MySQL database.
 */
$conn = mysql_connect("localhost", "password_ch", "Secret") or die(mysql_error());
mysql_select_db('database1', $conn) or die(mysql_error());
?>
```

Mint láthatjuk, ezzel kijelöltük a használni kívánt adatbázist a szerver címével (localhost fel lesz oldva a rendszer által 127.0.0.1 -as IP címre), a kívánt felhasználói névvel és a hozzá tartozó jelszóval. Utóbbi kettőt is statikusan adom meg, nem teszek különbséget felhasználók között, ezt külön általam írott kóddal fogom elvégezni. Külön hozzáférés akkor lehet hasznos, hogyha több kiszolgáló is kapcsolódik az adatbázis szerverünkhöz, kiszolgálónkként külön fiókokat hozva létre. Ha az egyik szervert fel is törik, az SQL szerverhez csak mértékkel tudnak majd hozzáférni.

A lekérdezéseket a *runSqlStatement();* eljárással tudjuk intézni, mely a következőképpen néz ki:

```
function runSqlStatement($statement){
 $return = mysql_query($statement) or die (mysql_error());
 return $return;
}
```

Ügyeljünk rá, hogy lekérdezések paramétereinél hívjuk meg a *mysql_real_escape_string();* metódust, védekezve az SQL Injection támadások ellen.

6.4. Táblák megjelenítése HTML-ben

A látványtervben már látott táblázatokat dinamikusan kell tudni kezelni, SQL-ből érkező adatokat olyan formába kell önteni, ami könnyen követhető, táblázatszerűen hat. Többszöri nekifutásra egy ciklusokkal szervezett metódust írtam, amely különböző színekkel jelzi az oszlopokat és linkeket is beszúr a szövegrészekhez.

```
if (isset($_GET['index'])){
 $index = $_GET['index'];
 $username = $_SESSION['username'];
 $q = "SELECT * FROM questions INNER JOIN surveys on
 questions.PRIMARY=surveys.PRIMARY WHERE questions.survey='$index'";
 $return = runSqlStatement($q);
 $numberOfLines = 0;
 ?>
 Survey question(s):<TABLE>
                        <TBODY>
                            <TR><br><br>
                                <TH><U>Line #</U></TH>
                                <TH><U>Index</U></TH>
                                <TH><U>Question's name</U></TH>
                            </TR>
 <?
 while($nt=mysql_fetch_array($return)){
                    if ($nt[owner] == $username){
                        $numberOfLines++;
                        echo " <tr ><td vAlign=\"top\"
 bgColor=\"#d7efff\">$numberOfLines</td>
                                <td vAlign=\"top\" align=\"right\" bgColor=\"#c2e6ff\">
 <a href=\"creation.php?action=link&amp;what=list.php\">Go back</a></td>";
                        echo "<td vAlign=\"top\"
 bgColor=\"#d7efff\">$nt[question]</td>";
```

```
                    echo "    </tr>";
                }
}
if ($numberOfLines == 0){
                    echo " <tr >
                    <td vAlign=\"top\" bgColor=\"#d7efff\">none</td>
                   <td vAlign=\"top\" align=\"right\"
 bgColor=\"#c2e6ff\">none</td>";
                    echo "<td vAlign=\"top\" bgColor=\"#d7efff\">none</td>";
                    echo "    </tr>";
                }
?></TBODY></TABLE><?
}
```

A kód elején megvizsgálom hogy az *index* változó értéke igaz-e, ha igen akkor lefuttatom a megfelelő SQL utasítást, majd mysql_fetch_array(); eljárással bejárom a kapott adathalmazt. Bejárás során a kimeneti állomány végére fűződik a HTML kódrészlet, így kontrollálva a sorok számát. Kivételt képez, amikor egy sor sincs kitöltve az adattáblában, ilyenkor is van szükség kimenetre. Ezt láthatjuk az utolsó blokkban.

6.5. Tesztelés és hibák keresése

Kódolás során még a leggyakorlottabb programozók is ejtenek szemantikai hibákat. Ha a megszokott bonyolultsági fokon túllépünk, megeshet, hogy nem tudjuk követni a kóddal az elgondolt vázat. Nem árt néha megtorpanni és megnézni: amit eddig építettünk vajon úgy működik-e, mint ahogyan azt elterveztük? Sokszor – főleg az objektum orientált világban igaz az – hogy nem tudjuk minden apró módosítás után letesztelni a programunkat, mert elég absztrakt a tervezet a fejünkben, hogy minden komponensnek működnie kell valamennyire ahhoz, hogy egyáltalán lefusson, eredményt produkáljon. Lehetőség van ilyenkor néhány egységet úgy megírni, hogy olyan kódot alkalmazunk, melynek funkciója nem egyezik a véglegesével, csak tesztelési célra szánjuk. Ez általában fölösleges energia – de főleg – időpazarlás. Sokkal célravezetőbb, ha jól megtervezett és megértett vázat dolgozunk ki, akár papírra vagy elektronikus formában lerögzítve és amikor magát a kódolást végezzük, nyomon

követjük, hogy épp hol járunk, mi szükséges ahhoz hogy lefusson a program. Később, több tapasztalattal rendelkezve már valószínű el fogjuk hagyni ezt a jó szokást és nem is állunk meg az első tesztelési lehetőségnél, inkább magabiztosan folytatjuk a kódolást egészen addig amíg úgy érezzük, tudjuk, hogy hol tartunk. Amikor kezdjük elveszteni a fonalat érdemes megállni, pihenni egyet és megnézni, hogy amit alkottunk azt csinálja-e amit elképzeltünk.

Az egyik általam használt módszer a változók értékeinek nyomon követése. Mivel itt a kód a szerveren fut le, és nem a fejlesztői környezetünk által kontrollált közegben, joggal kérdezhetnénk, hogy hogy helyezünk el itt töréspontokat (breakpoints)? PHP-ben erre is van lehetőség, de ne haladjunk ennyire előre. Ha úgy gondoljuk ez egy egyszerűbb hiba, esetleg már arról is van ötletünk hol keressük, és ezen variációk közül kell eldöntenünk merre induljunk, egy rövid gondolkodás után tegyünk be egy pár plusz érték kiíratást a kódba. Fontos hogy ezeket egyenként jelöljük külön-külön egyértelműen felismerhető módon, még hogyha egymás után is íródnak ki, tudjuk mi-mihez tartozik. Következzen egy példa erre. Felhasználói nevet, jelszót és jogkört követünk:

```
chronos123ASDa
```

Itt az elején van a felhasználói név, a jelszó következik, majd jogkör zárja. A kódban ezeket külön hívtuk meg, de mivel nem tettünk be semmi megkülönböztető jelzést, ezért nem tudjuk hol határolódik el az egyik a másiktól. A kiíratás látszólag teljesen rendben van. Példa az elhatárolásra:

```
#chronos#!123AS!=Da=
```

vagy

```
#chronos#
!123AS!
=Da=
```

A hibát meg is találtuk, olyan jogkör hogy „Da” nincs, az első karakter valahogy átcsúszott a következő változóba.

Ezt a módszert gyorsan be tudjuk vetni – bár képességei nagyon korlátoltak –, számomra hasznosnak bizonyult.

A másik módszer – amiről már esett pár szó – a töréspontokkal való követés. Ehhez szükségünk van egy beépülő modulhoz (plugin), nevezetesen a DBG-hez, ami a PHP Debugger és profiler modulja. Ha ez megvan, szúrjuk be a következő eljárást a kódunkba:

```
function break-point(){
   ob_flush();
   flush();
   sleep(.1);
   debugBreak();
}
```

Ebben az eljárásban az ob_flush(); parancs elküldi a felhasználó böngészőjére az összes egyéb pufferelt adatot (other buffered flush). A flush() hasonlóképpen tesz, csak a normál, egyébként is a böngészőben landoló adatokkal. A sleep(.1); időt biztosít az Apache-nak hogy minden információt el tudjon küldeni, mielőtt a következő parancs teljesen megállítaná a kód végrehajtást. Használatához a kívánt sorok közé kell szúrnunk a break-point(); parancsot, melynek hatására a töréspont – időrend szerint – utáni részletek nem hajtódnak végre. Eclipse-ben a folytatás (resume) parancs hatására persze a következő töréspontra „ugrik” az értelmező és ott fog megállni a legközelebbi alkalommal a végrehajtás.

6.6. Visszaállás egy korábbi verzióra

Az alkalmazás fejlődése során előfordulhat, hogy egy új funkció implementálásakor más funkciók megsérülnek, esetleg olyan magas számú változtatást végzünk egyszerre a kódbázison, hogy az összeomlik. A kijavítás hosszas és bosszantó, sokszor célravezetőbb, ha egyszerűen visszatérünk a kezdeti állapothoz. Ahhoz hogy ezt megtehessük szükség van egy mentéssel kapcsolatos szokásrendszerre, amely módot biztosít arra hogy valamilyen folyamat után visszakaphassunk egy korábbi állapotot. Ez lehet egy manuális verziókezelés vagy egy automatizált (revision control), mint amilyen a GIT vagy CVS.

6.7. Refactoring

Kódolás során megeshet, hogy nem a legoptimálisabb módon strukturálunk, vagy menet közben megváltozik a felépítés. Ilyenkor szükség van újrastrukturálásra, újratervezésre, amit legtöbbször a refactoring kifejezéssel fedhetünk le. Szerencsére a legtöbb modern IDE valamilyen formában támogatja, persze a lényegi munkát nem végzi el helyettünk. A folyamat során nem változik meg a program működése, kizárólag a program belső szerkezete. Ha jól csináltuk, a végén a felhasználó semmit se fog észrevenni a változtatásokból, mert az alkalmazás reakciói, kimenetei és viselkedése egyáltalán nem más mint előtte, csak számunkra vált átláthatóbbá, modulárisabbá és kezelhetőbbé.

A gyakorlatban két indokkal kezdhetünk neki az újraszervezésnek:

- Karbantartási okokból
- Bővíthetőségi okokból

Az első indoknál abból induljunk ki, hogy ha van egy jól, de nem a leghatékonyabban működő része a programunknak, annak cseréje akkor a leginkább fájdalommentes, ha azt modulárisan felépített kódon végezzük. Jobban járunk, ha különálló, önmagukban is működni tudó egységeket hozunk létre, minthogy egy ömlesztett kódbázison dolgozzunk. Mellékhatásként a kódrészlet újrafelhasználhatóvá is válik, így hamar megtérül a befektetett energia.

Bővíthetőségi okból való újrafaktorizálásnál leginkább az alkalmazás funkcióinak bővítése van szem előtt, mivel a jelenlegi kódbázissal ez nagyon bonyolult, hosszas folyamat lenne. Újraszervezés során elkülönülnek a funkciók, egyértelműen lehet látni, hogy mely funkciókat lehet összevonni, és melyek helyett lehetne univerzálisabb kódot alkalmazni. A folyamat lezáródása után sokkal tisztább kóddal van dolgunk és könnyebb átlátni, miként lehetséges implementálni az új funkciót.

Nagyon hasznos az egyes többször használt funkciókat, többsoros kódrészleteket eljárásokba szervezni és az áthelyezett részlet helyére eljáráshívást szúrni be, akár

paraméterekkel ellátva. A változók és eljárásnevek újraértelmezése, esetleg egységesítése – ami a modern fejlesztői környezetek segítségével pár kattintással megvalósítható – is nagyon népszerű. Sok ide tartozó technika van, ezeket nem tárgyalom, könnyen fellelhetőek a szakirodalomban.

6.8. Dokumentáció kódolás közben

A dokumentálás fontos részét képezi a fejlesztésnek, nemcsak azoknak segít megérteni a program működését, akik később csatlakoznak a folyamathoz, hanem önmagunknak is ad útmutatást, hogy eddig mit készítettünk el, és az komponensenként, hogy működik. Párhuzamosan – esetleg utólagosan – készíteni a dokumentációt sok plusz munkával jár, és ráadásul nem is biztos hogy a kívánt eredmény tárul majd elénk. Szerencsére a fejlesztés technológiája és kultúrája kidolgozott egy gyakorlatban könnyen és hatékonyan használható módszert, inkább szokásrendszert amire dokumentáció generátorként hivatkozunk.

Dokumentáció generátor egy programozási eszköz, mely a kód struktúráját elemezve, a forráskódban elhelyezett speciálisan jelölt megjegyzéseket használja fel a dokumentációs anyag létrehozására. Ennek értelmében annyi a dolgunk, hogy a programírás közben az egyes modulok – stílustól függően – elkezdése vagy befejezésekor, leírjuk, hogy az adott rész mire hivatott, hogy működik, milyen adatok tartoznak hozzá. Értelem szerűen nem csak új részek írásakor nyúlhatunk hozzá a megjegyzésekhez, hanem átstrukturáláskor, vagy viselkedés módosuláskor is újrafogalmazhatjuk a szöveget. Ha szeretnénk látni, hogy néz ki a kész dokumentáció, egy szkriptet kell lefuttatnunk, ami gyakran a fejlesztői környezetbe integrálva van, mint ahogyan az Eclipse-be is. Változtatás esetén csak újra le kell futtatni a generáló kódot, és kapunk egy általunk kiválasztott formátumú leírást a program működéséről. Ez akkor a leghasznosabb, ha egy központi szervert használunk a verziókövetésre és mindenki oda tölti fel a módosított vagy új fájlokat, majd a szerver automatikusan legenerálja a leírást, frissítve ezzel a honlapon található web alapú tudásbázist.

A legtöbb programozási nyelvhez létezik ilyen eszközkészlet: Java-hoz a JavaDoc, Python-hoz a pydoc, PHP-hez pedig a phpDoc, teljes nevén a phpDocumentor tartozik.

6.9. Version release – verzió kiadás

A fejlesztés során érdemes verziókat kiadni, így a felhasználók nyomon tudják követni, hogy hol jár a folyamat. Sok fajta verziószámozási séma létezik, a legjobb az, ha átgondoljuk számunkra melyik megfelelő és megállapodunk. Ennek a kérdőívkezelő rendszernek az esetében két megközelítés tűnik kézenfekvőnek. Egyik a mérföldkő alapú, másik pedig a dátumozott verziókiadás.

Mérföldkő alapúnál vannak bizonyos funkciók és állapotok, melyeket verziókhoz kötünk és ha sikerül működésre bírni az adott funkciót, vagy elérni az adott állapotot, akkor növelhetünk az adott részén a verziószámnak. Ilyen például, ha szeretnénk átstrukturálni, átdolgozni az alkalmazást, de a meglévő funkciókat nem akarjuk bővíteni. Felhasználói felület újragondolása, esetleg felhasználói folyamatok sorrendje lehet ilyen. Ebben az esetben lehet azt mondani, hogy emeljünk egyet a fő verziószámon, hogy tudja a felhasználó, hogy nagyobb változtatások lettek bevezetve és jobban tud időzíteni. Kellemetlen lenne ha használhatóságot befolyásoló változtatást jelölnénk al-verziószám növeléssel, nagy szervezeti terheltség mellett bevezetnék, felkészületlenül érve a felhasználókat.

Dátumozott verziószámozást használ többek között az Ubuntu, az úgynevezett nighly build-ek, illetve sok olyan szoftver, amelynél nincsenek mérföldkövek, az éppen aktuális verziót egy automatizált rendszer szolgáltatja. Utóbbinál általában a csomag fájlneve legalább tartalmaz a program neve után egy év-hónap-nap kombinációt, illetve egy számot, ami a kód beküldések számát jelzi. Utóbbit angolul commit-nek nevezzük és forráskód kezelő rendszerbe (pl. GIT, CVS) való kódbeküldés sorszámát értjük rajta.

Ennek az alkalmazásnak a fejlesztésénél függővé teszem a verziószámozási sémát a fejlesztés módjától és a felhasználás körülményeitől. Felvázolom, hogy ha mindkét típusra szükség van, hogy nézhetnek ki a sémák.

6.9.1. Mérföldkő alapú

A.B.C[.D]

Ahol:

A – fő verziószám. Nagy változtatásokat jelölünk, melyeket nem lehet vagy érdemes apró lépésekben megtenni.

B – mellék verziószám. Új funkciók és közepes mértékű változtatások.

C – al-verziószám. Hibajavítások és kódtisztítások (code cleaning) illetve elírások javítása.

D – opcionális. Itt az állapotokat adjuk meg, például hogy éles használatra kész, vagy tesztelői változat, esetleg már kiadásra jelölt. Teljesség igénye nélkül: Rc – Release candidate, RTM – Release to Manufacturing, Alpha – tesztelésre kész, Beta – használhatósági tesztek.

A kódolást 0.0.0-tól kezdjük és haladunk az 1.0.0 felé. Az első lépcső a 0.1.0 lesz, majd apró lépésekben 0.1.1, 0.1.2 és így tovább, majd 0.2.0. Az 1.0.0-t megelőzően kiadhatunk 1.0.0.Beta verziót is, jelölve ezzel hogy ez még nincs befejezve, illetve igénytől függően haladhatunk a teljes első verzió felé.

6.9.2. Dátum alapú

Az alapelgondolás itt tehát az, hogy a verziók automatikusan kerülnek csomagokba, tükrözve az éppen aktuális állapotot. Ezeket a csomagokat pillanatfelvételnek, angolul snapshot-nak nevezzük. A séma pedig legyen a következő:

SHT_YYYYMMDD_rC..C

Ahol:

SHT – Survey Handling Tool, az alkalmazás egyértelműen azonosítható neve.

YYYYMMDD – A Year, Month és a Day angol kifejezéseknek megfelelő év, hónap és nap.

rC..C – Ahol az „r” jelöli a release, kiadás szót, a C..C pedig a commit-ok számát.

Nézzünk erre egy példát. 2010.04.18-án készül egy csomag, amikor már a 126-ik beküldés megtörtént. A csomag neve a következőképpen alakul:

SHT_20100418_r126

7. Védelmi hibák és javításuk

Minden alkalmazásnak vannak gyenge pontjai, amiket jó, vagy rosszindulatú támadók ki tudnak használni saját céljaikra, nem rendeltetés szedrűen használva az adott programot. Az informatikai paletta megváltozott az elmúlt 10 évben és egyre nagyobb teret kaptak a nem csak kizárólag helyi gépen futó kódok. Legelterjedtebbek a vegyes alkalmazások, melyek több technológiát felhasználva részben szerveren (szervereken), részben a felhasználó gépén fut, így azok erőforrásait is igénybe veszi. A zárt forráskódú lokális gépen futó alkalmazásoknál a támadási felületet 90%-ban a kód visszafejtése adta, de manapság a szervereken futó SQL, vagy PHP kód még annak ellenére sem hozzáférhető, hogy nyílt forrású szerver alkalmazások futtatják. A kód csak közvetett kapcsolatokon keresztül fut és összetett rendszeren keresztül zajlanak a lekérdezések és adatközlések is. A paraméter-átadással kapcsolatos hibák és hiányosságok adják ebben az esetben a legkönnyebben kihasználható támadási felületeket. Ilyen többek között:

- SQL Injection
- Directory Traversal
- Autentikációs hibák
- Remote Scripts (XSS)

7.1. SQL Injection

Az alapelgondolása ennek a támadási mechanizmusnak az, hogy egy SQL lekérdezést is érintő esemény hatására olyan információkat adunk ki a támadónak, amelyekhez nem lenne normál esetben jogosultsága. Példaként vegyünk egy olyan esetet, amikor egy HTML kódban lévő változót veszünk bele a lekérdezésbe, melyet a PHP kód közvetít az SQL motornak. A felhasználó kitölti a HTML form-ot, aktiválja az adatcserét és várja az eredményt. Abban az esetben ha ez egy szöveg, a lekérdezés a következő formában nézhet, és többnyire így is néz ki:

```
SELECT username, rights FROM accounts WHERE user = '$username';
```

Itt a PHP kicseréli a $username változót az értékére, melyet GET vagy POST módon vettünk át a HTML kódból. Tételezzük fel, hogy a felhasználónév Ádám. A lekérdezés ebben az esetben így néz ki:

```
SELECT username, rights FROM accounts WHERE user = 'Ádám';
```

Ez a lekérdezés még ártalmatlan és helyes. Gondoljunk bele, hogy mi történik hogyha a nevünk helyére a következőt irjuk be:

```
'; SELECT * FROM * WHERE 'x' = 'x
```

Ebben az esetben az x = x feltétel minden esetben igaz, ezért minden sort ki fog adni.

A teljes lekérdezés pedig ez lesz:

```
SELECT username, rights FROM accounts WHERE user = ''; SELECT * FROM *
 WHERE 'x' = 'x';
```

Az első lekérdezés eredménye minket nem érdekel, de a második értékes információkkal szolgálhat, mivel közvetetten visszaadja gyakorlatilag az egész táblát.

Amilyen könnyen ki lehet ezt a sebezhetőséget játszani, olyan könnyű ellene védekezni. Egyik megoldás, hogy letiltjuk a veszélyes karaktereket, mint amilyen az idézőjel, aposztróf vagy a pontosvessző, mivel ezek alkotják a támadás pilléreit. Másik megoldás MySQL felhasználóknak a *mysql_real_escape_string();* meghívása, ami átalakítja a veszélyes karaktereket olyan karakterláncokká, melyeket az SQL motor nem értelmez. Egyszerűsége miatt én az utóbbit preferálom.

7.2. Directory Traversal

Ez a támadási mechanizmus „../” (dot dot slash) vagy könyvtár mászás (directory climbing) néven is ismert. Lényege, hogy elérési útvonalak betöltésekor (többnyire sütikből) olyan útvonalat adunk meg, ami nem a könyvtár-hierarchia megfelelő szintjére mutat – amihez egyébként jogosan férünk hozzá – hanem sokkal felette, rendszerkönyvtárakban lévő fájlokra. Annak ellenére hogy ez fájlrendszerre vonatkozik, minden operációs rendszeren kivitelezhető. Többnyire jelszavak és felhasználónevek megszerzésére használható, amennyiben nincs megfelelően védve az ilyen és hasonló támadások ellen. Például ebben a PHP-ben megírt kódnál ez a támadás kiaknázható:

```
<?php
$temp = 'red.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
  $temp = $_COOKIE['TEMPLATE'];
include ( "/var/www/" . $temp );
?>
```

HTTP protokollon a következő utasitást kiadva:

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../etc/passwd
```

Hozzáférhetünk a jelszavakat tároló fájlhoz.

Unix szerű rendszerekben a példában bemutatott fájl már rég nem használatos és jól kidolgozott védelmi szabályok vannak érvényben. Ettől függetlenül jó ha tudunk erről a támadási felületről. Megelőzésére egyszerűen meg kell tiltani a fájlok közvetlenül való betöltését olyan útvonalról, amit a felhasználó adhat meg.

7.3. Autentikációs hibák

Már egyszerű alkalmazásoknál is fontos szerepet tölt be az azonosítás és jogosultságok ellenőrzése, így nem csoda, hogy ezzel kapcsolatos problémák is felhasználhatók illetéktelen behatolásra. Gyakorlatilag egy alacsonyabb hozzáférési szinttel rendelkező felhasználó szert tesz egy olyan sütire ami nem hozzá tartozik, és ezt felhasználva emel a hozzáférésének szintjén. Megfelelő ellenőrzések és jól átgondolt azonosítási protokoll esetén ez a módszer nem aknázható ki.

7.4. Remote Scripts (XSS)

Remote Scripts technológiának a lényege, hogy egy lokális gépen futó szkript egy szerveren futó eljárást hív meg, majd az visszatérési értéket feldolgozva jut hozzá a folytatáshoz szükséges információhoz. Megfelelően használva nagyban elősegíti az oldalak interaktivitását, dinamikusabb oldal tartalmakat lehetővé téve. Egyes felmérések szerint a világhálón fellelhető oldalak csupán 1/3-a védett az ilyen támadások ellen, így beláthatjuk a támadók egyik nagy kedvencéről van szó.

Két kategóriába sorolhatjuk:

- Ideiglenes (non-persistent)
- Állandó (persistent)

Az elsőnél egy kliens kérés történik a szerver felé, mely generál egy újabb lapot a kérés függvényében, anélkül hogy ellenőrizné a hozzáférési jogosultságot, vagy magát a közlendő tartalmat. Egy gyakorlati példa, hogy a keresőoldalaknál a keresett szövegrész az eredmények oldalán is szerepel. Az SQL injection-höz hasonlóan be lehet szúrni olyan kódrészletet, mely a felhasználó gépén fut le és fér hozzá a böngészőben használt adatokhoz, sütikhez.

Alapesetben ez azt jelentené, hogy a felhasználó jut hozzá a saját fájljaihoz, de sajnos lehetőség van rávenni a böngészőt, hogy akár a háttérben elnavigáljon egy fertőzött oldalra és megtörténjen a támadás második lépcsője.

Az állandó XSS támadásnál komolyabb problémáról beszélhetünk. Manapság nagy népszerűségnek örvendenek a Web 2.0-s alkalmazások, közösségi oldalak, olyan eszközök, melyek a csoportos munkát, csoporton belüli kommunikációt segítik elő. Egy szövegrész tárolódik az oldal adatbázisában, majd egyes oldalakon betöltődik és automatikusan lefut a kliens böngészőjében. Ilyen például egy mikroblog oldal, ahol egy felhasználó tartalmat tesz közzé, majd az összes ismerősének az oldalán megjelenik, terítve ezzel információt. A kártékony kód ilyenkor futótűz szerűen terjed, férget kreálva ezzel. Terjedése sebessége miatt ez a támadási mód sokkalta veszélyesebb mint a többi. Jól pozicionálva órák alatt képes megfertőzni nagy számú felhasználót.

8. Összefoglalás

Az informatikai világ egyre bonyolultabb eszközöket hoz létre, ezek összekapcsolt használatához szerteágazó érdeklődés és tapasztalat szükséges. Ezen szakdolgozat keretein belül a célom, hogy a rendszerek viszonyait tisztázzam egy nyílt forráskódú webes alkalmazásra vetítve úgy gondolom sikerült. Sajnálatomra nem tárgyaltam olyan részletességgel mint amilyennel szerettem volna. Kész alkalmazást nem, de annál több tapasztalatot sikerült gyűjtenem a különböző fejlesztéssel kapcsolatos elemek próbálgatása közben. Remélem másoknak is hasznosak lesznek az itt elhangzott eszmefuttatások és gyakorlati megközelítésű példák.

Célom hogy folytassam a fejlesztést – esetleg más programnyelvek bevonásával is –, illetve szeretnék PHP és Python programozási nyelvekkel tüzetesebben foglalkozni, nyílt forráskódú keretek között.

9. Irodalomjegyzék

http://www.ibm.com/developerworks/opensource/library/os-debug/
http://en.wikipedia.org/wiki/Code_refactoring
http://en.wikipedia.org/wiki/Code_injection
http://en.wikipedia.org/wiki/Directory_traversal
http://www.acunetix.com/websitesecurity/php-security-1.htm
http://unixwiz.net/techtips/sql-injection.html
http://php.net/manual/en/function.mysql-real-escape-string.php
http://news.netcraft.com/archives/2010/02/22/february_2010_web_server_survey.html

10. Függelék

10.1. Felhasznált alkalmazások

Ubuntu 9.10	http://www.ubuntu.com/
Xmind 3.1.1	http://www.xmind.net/
OpenOffice.org 3.1.1	http://hu.openoffice.org/
Eclipse Ganymede	http://www.eclipse.org/
gedit 2.28.0	http://www.gedit.org
Apache 2.2.12	http://www.apache.org/
MySQL 5.1.37	http://www.mysql.com/

11. Köszönetnyilvánítás

A következő személyeknek szeretnék köszönetet mondani a szakdolgozatom elkészülésében nyújtott közvetlen és közvetett segítségükért: Pánovics Jánosnak a készséges segítségéért és támogatásáért, Michael Dazertnek és Zsenyuk Ritának a biztatásért és lelkesítésért, a közösségnek hogy ez a szakdolgozat létrejöhetett, Tóth Jánosnak a szakmai véleményéért és a családomnak a türelmükért és megértésükért.

Utoljára, de nem utolsó sorban köszönöm – és ajánlom ezt a szakdolgozatot – a páromnak, Csepregi Tímeának a biztatásért, türelméért és mindenért amit tőle kaptam az elmúlt időszakban.