Diplomamunka

Nyilas Attila, Karakas Péter

2010
Debrecen

Debreceni Egyetem
Informatikai Kar

Csoportmunka távolról

Témavezető:
Dr Rutkovszky Edéné

Készítette:
Nyilas Attila, Karakas Péter

Tartalom

Köszönetnyilvánítás

(Karakas Péter, Nyilas Attila)

Köszönöm Dr. Rutkovszky Edénének a témavezetői feladatainak az ellátásáért. Továbbá köszönöm Egri Zsoltnak, hogy engedélyezte a Webiris Kft-ben használt technológiák és módszerek ismertetését, továbbá szakmai tanácsokkal látott el és mint barátom támogatott. Köszönöm Veréb Viktóriának, hogy előzetes kutatásait a távmunkával kapcsolatban megosztotta velem. Köszönöm még Buzdor Attilának, hogy tanácsaival javított a dolgozat színvonalán.

Bevezetés

(Karakas Péter, Nyilas Attila)

Korunk információs társadalma egyre nagyobb térhódítási lehetőséget biztosít a távmunka számára.

Dolgozatunkban a távmunka információs technológiai cégeknél való alkalmazását vizsgáljuk meg. Mint munkavállaló és mint munkaadó szemszögéből. Megvizsgáljuk az előnyeit és hátrányait illetve konkrét gyakorlati megoldásokat javasolunk a problémák leküzdésére.

Abban a szerencsés helyzetben vagyunk, hogy mind a ketten több mint 2 éve dolgozunk távmunkában. Igaz más beosztásban, de ez segített abban, hogy még szélesebb képet kapjunk távmunka alkalmazásáról. Karakas Péter mint fejlesztési részlegvetető és Nyilas Attila mint fejlesztő. Ebből következően a dolgozat két eltérő egységet bontható. Első felében megvizsgáljuk a szervezet szempontjából lényeges elemeket. Milyen

projektirányítási módszereket és eszközöket érdemes használni. A dolgozat második része a fejlesztő szemszögéből mutatja be egy alkalmazás elkészítését.

A dolgozat és az elkészített program is egy éles próbája az itt leírtaknak. Az itt bemutatott technológiák és eszközök segítenek abban, hogy közösen a távolságot leküzdve értékeket állíthassunk elő. Hiszünk abban, hogy a távmunka a technológia fejlődésével egyre hétköznapibbá válik. Egyenlőre ez a terület még most bontogatja szárnyait és rengeteg még fel nem térképezett terület van és sok a meg nem válaszolt kérdés. A dolgozat megírása közben is számos ötlet és újítás látott napvilágot, amik segítettek tovább növelni a hatásfokot és az ügyfél elégedettséget.

Webiris Kft bemutatása

Hogy jobban megértsük az alkalmazott módszereket fontosnak tartjuk a Webirsi Kft rövid bemutatását.

A Webiris Kft 2007-ben alakult. Fő profilja a Webiris nevű program fejlesztése, mely látszerészek számára készült ügyviteli rendszer. Cégünk kiemelt figyelmet fordít a távmunkára. Már a kezdetektől fogva ezt a módszert alkalmazza. Az alkalmazottak leginkább a Debreceni Egyetem tanulói.

Ahogy később látni fogjuk a távmunka egyik hátránya a kisebb motiváció. Gyakorlati tapasztalatunk, hogy diákok esetében ez fokozottan érvényes, hiszen legtöbbünk számára a tanulás az elsődleges. Különösen vizsgaidőszak alatt tapasztalható a munkaórák csökkenése. Ezért különösen hatékony motivációs rendszert kellett kidolgozni, de ugyanakkor számolni kell a hullámzó teljesítménnyel.

További nehezítő körülmény a diákok tapasztalatlansága. Egy első, másodéves hallgató ritkán rendelkezik munkatapasztalattal és gyakran csak az egyetem évei alatt programozott. Továbbá magas a fluktuáció. Ezért különösen fontos a gyors és hatékony oktatás.

A Webiris Kft büszke arra, hogy ilyen körülmények mellet is képes a piacon életképes és folyamatosan fejlődő software-t létrehozni. Ezentúl fejlődési lehetőséget és munkalehetőséget kínál a diákok számára.

Ezt úgy tudtuk elérni, hogy folyamatosan keressük a nekünk legmegfelelőbb megoldásokat. Egy olyan hibrid projekt management

módszert sikerült kialakítani ami a SCRUM, agilis és extrame programming módszereken alapszik. Éppen ezért a dolgozat egy pillanatkép a jelenleg használt technológiákról, amely jelenlegi tudásunk szerint a legjobban működik.

Mit is nevezhetünk távmunkának?

"A távmunkának tekintjük a munkavégzésnek azt a formáját, amikor a munkavállaló nem a hagyományos munkahelyen, hanem attól távol végzi el rendszeresen napi munkáját, melynek eredményét a kommunikációs és információs technológiák alkalmazásával juttatja el munkaadójához" (http://www.tavmunkainfo.hu/miatavmunka.htm)

Ebből következik, hogy távmunkára olyan feladatok elvégzésére alkalmas, ahol az előállított eredmény, információ vagy legalább is nem materiális. Azonban nem tartjuk elképzelhetetlennek nem információs értékek előállítását sem. Egyszerűbb feladatok (pl, boritékolás) elvégzése műhely vagy speciális eszköz használata nélkül is lehetséges. De nyugodtan kijelenthetjük: alapfeltétel, hogy a munkavégzés helytől független legyen. Ne igényeljen speciális nehezen hozzáférhető eszközöket.

Távmunka fejlődése

A távmunka elsőnek az 1990-es évek végén jelent meg először. (http://www.tavmunkainfo.hu/EU99.htm). ezekben az időkben a kommunikáció leginkább telefonon és hagyományos levélen keresztül történt. Ezek drága és lassú adatcserét tettek lehetővé ezért az elvégezhető feladatok be voltak korlátozva. Az információ továbbítása körülményes volt, de már így is megjelentek a kezdeti lépések. Ezekben az időkben leginkább személyes értékesítők élhettek ezzel a lehetőséggel, hiszen ők egyébként is irodától távol dolgoztak. Az előállított értékük pedig egy aláírt szerződés, amit könnyen el lehet juttatni postán a központi irodába.

Az technika rohamos fejlődésével ma már alapvető közműnek számít az internet. Ez lehetőséget ad az információ gyors és olcsó továbbítására. Ezzel új lehetőségek nyíltak meg a távmunka számára. Az IT különösen szerencsés helyzetben van ebből a szempontból, mert legtöbb esetben az elállított érték adat, információ. Ezért nem véletlen, hogy főleg ezen a

területen terjedt el a távmunka használata. Már 2001-ben megjelentek a software fejlesztésre szakosodott távmunkát kínáló cégek. Ilyen volt az akkor még rent-a-coder.com, mára már vworker.com. Ez és sok más példa is bizonyítja, hogy a távmunkára van igény, és ez az igény folyamatosan növekszik.

A következő nagy ugrás a távmunka elterjedésében, az állami támogatások megjelenése után következett. Ma már Magyarországon is számos pályázat közül válogathat a munkaadó. Azonban ma még mindig ritkaságnak számít ha valaki távmunkában végzi feladatát.

Előnyök

A távmunka mindkét fél számára előnyös lehet.
Munkaadó előnyei:

- Olcsóbb üzemeltetés. Hiszen a munkaadónak nem kell irodaköltséget fizetnie. Ezen felül nem kell irodaeszközökre költeni. Hiszen a munkavállaló használhatja a saját eszközeit.
- Hatékonyabb a munkavégzés. Az otthoni munka akár 10-15%-al is eredményesebb lehet mint az irodai. (http://www.tavmunkainfo.hu/elonyok.htm#6)
- Eltűnnek a határok. Távmunka esetén a munkaerő nem feltétlenül a környező városból származi. Az egész ország vagy akár az egész világra kiterjedhet. Nem ritka az sem, hogy az időeltolódást kihasználva 24 órás ügyfélszolgálatot látnak el
- Nő a szervezet rugalmassága: könnyebb több és magasan képzett munkaerőt toborozni, hiszen a munkavállalók köre nem szűkül le az adott térségre.

Munkavállaló előnyei:

- Kényelmesebb munkavégzés
- Csökkenő munkahelyi stressz

Hátrányok

(http://www.tavmunkainfo.hu/hatranyok.htm)

A nyilvánvaló előnyök mellett azonban vannak hátrányok is. Ahogy az lenti listából kiderül a legtöbb veszély a munkavállalót fenyegeti.

Munkavállaló hátrányai:

- Izoláció: a munkavállaló legtöbbször magányosan otthon végzi munkáját és a szociális kapcsolatok leépüléséhez vezethet.
- Munka-alkoholizmus: távmunka esetén megnő azoknak a száma akik megszállottai lesznek a munkájuknak. Hiszen ők nem csak hazaviszik a munkájukat hanem az otthonuk a munkahelyük
- Motiváció hiánya: sok ember számára nehezen fenntartható a motiváció, ha csak magára számíthat.
- Kevesebb jövedelem: a távmunkában dolgozók általában kevesebb juttatást és kevesebb fizetést kapnak. Igaz, hogy sok költséget meg is tudnak spórolni.
- Több a becsapós álláshirdetés
- Rejtett költségek: legtöbb esetben a munkavállalónak kell beszereznie a munkavégzéshez szükséges eszközöket (számítógép, internet)

Munkaadó hátrányai:

- Nehezen mérhető egyéni teljesítmény
- A hagyományos szervezeti struktúra és módszerek nem működnek

Projektvezetés a Webiris Kft-nél

Már meglévő projektirányítási módszertanok

(Nyilas Attila)

A következő részben bemutatásra kerül, hogy melyek azok a már meglévő projektirányítási módszerek, amelyeket alapul vettünk. Ezek természetesen nem ültethetőek át egy-az egyben, hiszen módosítások szükségesek a távmunka alkalmazása miatt.

Az agilis módszertan

- Egyének és interakciók, szemben az eljárásokkal és eszközökkel.
- Működő szoftver, szemben a teljeskörű dokumentációval.
- Együttműködés az ügyféllel, szemben a szerződésről való alkudozással.
- Változásokra való reagálás, szemben a terv követésével.

Az agilis módszertant a legkönnyebb bemutatni ha összehasonlítjuk a klasszikus vízesés modellel.

A vízesés modellben azzal kezdünk, hogy az elején megnézzük, melyek a követelmények, az ügyfélnek összefoglaljuk, hogy így értette-e, majd ezek alapján megtervezzük a szoftvertermékét és lekódoljuk. Ha kell összeintegráljuk, teszteljük majd installáljuk és átadjuk.

Ezzel szemben az agilis módszertanok arra fókuszálnak, hogy a gyakran változó követelményekhez tudjanak alkalmazkodni. Iteratív módon történik a fejlesztés rövid időközönként. Ez általában 2-4 hét. Erre az időszakra hajszál pontosan megtervezik mit fognak elkészíteni, megvalósítják és át is adják. Így már elején is az ügyfél rendelkezésére áll a szoftver bár kevés funkcióval. De minden egyes iteráció során új teljesen tesztelt, működő funkciókkal bővül az alkalmazás.

Az egyik fontos különbség a hagyományos és az agilis módszertan között, hogy a hagyományos módszernél az ügyféllel mindössze két alkalommal találkozunk igazán: a projekt elején és a végén. Egy hosszabb terjedelmű, féléves, másfél éves projekt közben nem találkozunk az ügyféllel, hiszen ő sem igényli, meg mi sem. Jóllehet, mindkét oldalnak szüksége lenne rá, ennek nincsenek tudatában. A végén, amikor átadjuk a terméket, az ügyfél azt mondja, hogy ő nem ezt akarta. Akkor jön a vita, hogy ki hibázott. A követelményspecifikációnak hiába felel meg az alkalmazás ha azt ő máshogy értette, mint mi. Az ügyfél nem lesz elégedett.

A másik dolog, hogy az átadáskor nem ér véget rögtön a fejlesztés, nem tudjuk valójában átadni, hisz tovább kell fejleszteni, hogy olyanná alakítsuk, amilyenné az ügyfél *valójában* szerette volna. Tehát a plusz költségek mellett határidőcsúszással és bevétel kieséssel is számolhatunk.

A harmadik, mivel a vízesés modell fázisokból áll (követelményelemzés, tervezés, kódolás, integráció, tesztelés stb), ha bármelyik egy picit is megcsúszik, az összes utána következőt tolja maga előtt. Ismét határidőcsúszás az eredmény.

Nézzük a lehetséges problémákat: Ha követelmény változik, akkor mindig vissza kell térni az elejére, újra bele kell nyúlni a tervekbe. Ha tehát az ügyfél a projekt elindulása után rájön, hogy változtatnia kell a követelményeken, és ha ez a tervezés ideje alatt történik, akkor szól nekünk, és mi beletesszük a módosításokat. Ha viszont kódolási időszak

alatt történik mindez, akkor csak úgy tudjuk beépíteni ezt a változtatást, ha visszanyúlunk a tervezéshez.

Egy tanulmány szerint, ha a projekt elején egy követelmény 1 forintba kerül, akkor annak beépítése a projekt 75%-ánál, már 100 forintos költséget jelent. Tehát minden ilyen változtatás plusz költséget generál, de nagyságrendileg is nagyobb kiadás, mintha az elején derült volna ki.

Ezzel szemben az agilis módszertanok a gyakori kommunikáció miatt mindig az igényeknek megfelelő kis lépéseket tesznek előre hétről hétre. Minden release-kor egy teljesen integrált és tesztelt rendszert adnak át az ügyfélnek.

Következőkben megnézünk 2 gyakori agilis módszertant: a scrum-ot és az extrém programozást.

Scrum

(http://webisztan.blog.hu/2010/05/17/megertik_es_atelik_a_munkajukat_a_scrum_modszerrol_1_resz)

Egy projekt általában egy ötlettel kezdődik. Először elkezdik címszerűen összeírni, hogy mit is kell tartalmaznia a terméknek ahhoz, hogy az ötlet megvalósuljon. Ezt a terméket leíró listát nevezzük *product backlog* listának. Majd ha az ötletek összegyűltek, elkezdik kidolgozni a mögöttes tartalmat is - kicsit hasonlóan a követelmény-specifikációkhoz. Az előnye az, hogy ahhoz, hogy elindítsuk a projektet, a teljes kép ugyan meg kell, hogy legyen, de a teljes lista (minden részletében kidolgozva) nem kell, hogy rendelkezésre álljon. Ennek az az előnye, hogy egyrészt hamarabb elkezdhetünk fejleszteni, másrészt úgyis meg fognak változni a projekt során a követelmények, és akkor legalább nem kell majd átírogatni a teljes dokumentumot. Ami nagyon fontos, hogy ez a lista egy jól meghatározott sorrendbe rendezett lista kell, hogy legyen. A sorrendet pedig a *Business Value-nak* nevezett paraméter határozza meg, ami azt jelenti, hogy azok a magasabb prioritású elemek, amik a legtöbbet jelentik a termék számára, és ezekkel kezdünk el foglalkozni először. Létrehozzuk a termék vázát és utána feltöltjük a legértékesebb feature-ökkel, minek eredményeként lehet, hogy már el is tudjuk kezdeni értékesíteni a terméket, miközben folyhat tovább a fejlesztési munka.

Aki képviseli a vevőt és tulajdonolja ezt a listát, őt hívjuk *Product Ownernek*. Ő mondja el a teljes képet a fejlesztőcsapatnak egy kick-off meeting keretében, hogy mindenkinek a fejében meglegyen, hogy miről szól ez a termék. Ez nagyon fontos. Ezt követően elkezdődnek az iterációk, amiket a Scrumban sprinteknek nevezünk. A product owner és a fejlesztőcsapat összeül, és ennek a listának egy, a tetején lévő részhalmazát, amiről a product owner úgy gondolja, hogy belefér egy sprintbe, odaadja a teamnek hogy ezt valósítsák meg az első iterációban. Nagyon részletesen elmagyarázza nekik ezeket az elemeket, a fejlesztők pedig addig kérdeznek, amíg világos nem lesz számukra minden részletében a feladat. Majd a fejlesztői csapat valamilyen módszer szerint (idő, komplexitás) megbecsli a feladatokat. Ezután a product ownerrel közlik, hogy most ennyi fér bele a "kívánságlistából". Megegyeznek, hogy minek kell elkészülnie és innentől nevezik ezt a listát *sprint backlog listának*. Ebben a pillanatban a sprint backlog lista befagy. Nem lehet sem hozzáadni, sem elvenni belőle úgynevezett *storykat*, vagyis listaelemeket. Csak és kizárólag közös megegyezéssel (PO és team), de az már kivételkezelést igényel.
A lista elemeit *taskokra* bontják, amelyek már a teljesség igényével készülnek. Egy story akkor lesz kész, amikor a hozzá tartozó összes task készen van. Az ideális sprint 2 vagy 4 hét. A sprint végén a fejlesztők leszállítják a feature-öket. Nagyon fontos, hogy a sprint backlog listában nem rétegeket felépítő dobozok vannak, hanem függőleges szoftver funkciók. Azaz ezekben benne van a UI, a backend munka, az adatbázis munka, és minden, ami csak szükséges a feature működéséhez. Így a sprint végére a feature-ökhöz többé már nem kell hozzányúlni, és így adjuk hozzá a termékhez minden sprintnél az újabb feature-öket.

Ez a módszer nem eredményez egy szép szoftverarchitektúrát, de megvan az ellenszer: a *refaktorálás*. Míg hagyományos módszereknél nem is biztos, hogy ismerik vagy alkalmazzák, a Scrum esetében szükséges dolog a refactoring. Azzal azonban nem ér véget a dolog, hogy a team elkészítette a bevállalt feature-öket. Nekik is kell eladniuk azokat a product ownernek. Egyenként végigmennek rajtuk és a product owner értékel. Majd a végén elfogadja vagy nem fogadja el a leszállított fejlesztéseket.

Szintén fontos elem, hogy minden nap van egy úgynevezett daily scrum, vagy *daily standup* nevű szeánsz, amikor a team maga (pont az önszervezésből adódóan) megbeszéli, hogy mit csináltak az előző standup

óta, mit tervezek csinálni a következő standupig, és van-e valami akadályozó tényező, amire azonnal ugrik is a *Scrum Master (SM)*, akinek feladata, hogy elhárítsa ezeket az akadályokat.

Az eXtrém Programozás

(http://www.valodi.hu/agile)
Az eXtrém Programozás az alábbiképpen foglalható össze:

- Az emberségesség és a hatékonyság összeegyeztetésére tett kísérlet
- Társadalmi jellegű változásokra irányuló mechanizmus
- A fejlődés egyik útja
- Fejlesztési stílus
- Szoftverfejlesztési diszciplína

Az XP fő célja, hogy csökkentse a változások költségvonzatát. A hagyományos rendszerfejlesztési módszertanokban, a rendszerrel szemben támasztott követelmények adottak a projekt elején, és gyakran nem is változnak meg. Ez azzal jár, hogy minél később kell változtatni a követelményeken, ami pedig szoftverfejlesztési projektekben a legvégén sem szokatlan, annál magasabbak lesznek a költségek.
Az XP arra törekszik, hogy ezeket a költségeket csökkentse azáltal, hogy más alapvető értékeket, elveket, és gyakorlatot vezet be. Egy XP-t használó rendszerfejlesztési projekt sokkal rugalmasabb lesz a röptében bekövetkező változásokkal szemben.

Az XP értékei

- Kommunikáció
- Egyszerűség
- Visszajelzés
- Bátorság
- Tisztelet

Kommunikáció

Az egyik alapvető fontosságú feladat a szoftverrendszer-készítés során, hogy valaki megmondja a fejlesztőknek, hogy mi a feladat. A korábbi

módszertanok során ez főként dokumentumok gyártásával történt meg.
Az XP technikákat tekinthetjük gyors információrendszerező és -terjesztő technikáknak is, amelyeknek célja, hogy a fejlesztőcsapat minél gyorsabban szerezze meg a szükséges tudást. A cél az, hogy minden fejlesztő ugyanúgy lássa a rendszert, ahogy a majdani felhasználók is látni fogják. Ezért az XP szereti az egyszerű terveket, metaforákat, a majdani felhasználók és a mostani fejlesztők együttműködését, a gyakori szóbeli kommunikációt, és a visszajelzéseket.

Egyszerűség

Az XP azt a megközelítést támogatja, hogy kezdjük el a lehető legegyszerűbben, és folyamatosan dolgozzuk át a programot, hogy egyre jobb és jobb legyen. A különbség eközött a megközelítés között, és a hagyományos megközelítések között, hogy a mai igényeknek megfelelő programot tervezünk és írunk, nem pedig a holnapi, a jövő heti, vagy a jövő hónapi igényeknek megfelelőt. Az XP ellenzői ezt hátrányként fogják fel, mondván, hogy így megeshet, hogy a következő hónapban több munkába fog kerülni átdolgozni a rendszert, ha nem készültünk fel előre új követelményekre, és azt állítják, hogy ez az időveszteség kitesz annyit, mint a megvalósított, de később szükségtelennek bizonyuló feature-ökre fordított idő. Az egyszerűség elve viszont pontosan azt mondja, hogy a program túlbonyolódik, ha tele lesz mindenféle feature-rel, amiről már rég mindenki elfelejtette, hogy mire való, és a bonyolultság okozta többletmunka viszont sokszorosan meghaladja a folyamatos átdolgozás által generált munkát.
Az előző értékhez, a kommunikációhoz kapcsolódóan, az egyszerűség megkönnyíti és elősegíti a kommunikációt, mert egy egyszerű tervet és a hozzátartozó egyszerű kódot a csapat minden programozója könnyen megért.

Visszajelzés

Az XP-ben a visszajelzés a fejlesztés több területére is vonatkozik:

- Visszajelzés a rendszertől: a részegység-tesztek készítésével a programozók közvetlen visszajelzést kapnak arról, hogy milyen állapotban van a rendszer egy-egy módosítás után.

- Visszajelzés az ügyféltől: a funkcionális teszteket a programozók együtt készítik az ügyféllel, így mindketten konkrét visszajelzést kapnak arról, hogy milyen állapotban van a rendszer funkcionalitása. Ilyenfajta közös tesztelést 2-3 hetente célszerű végezni, így az ügyfélnek megvan a lehetősége a fejlesztés irányítására.
- Visszajelzés a csapattól: amikor az ügyfél új igényekkel áll elő, a csapat rögtön tud rá reagálni, és visszajelzést tud adni, hogy mennyi ideig fog tartani a dolog, és mennyibe fog kerülni.

A visszajelzés szorosan összefügg az egyszerűséggel és a kommunikációval. A rendszerhibákat könnyű jelenteni, hiszen egy részegység-teszt megírása megmutatja, hogy mi nem jó a rendszerben, így a rendszer maga mutatja meg a javításra szoruló részeket. Az ügyfél is rendszeresen tudja tesztelni a rendszert, a saját igényeinek megfelelően, amiket az XP-ben 'felhasználói történet'-nek hívnak. Hogy Kent Becket idézzük, 'Az optimizmus szakmai ártalom a programozóknál, és a visszajelzés rá a gyógyír.'

Bátorság

Az XP bátorságra buzdító doktrínáját a legjobban gyakorlati példákkal lehet megvilágítani. Az egyik, hogy mindig a mai igények kielégítésére kell a programot tervezni és megírni. Ez az erőfeszítés azért szükséges, hogy ne gabalyodjunk bele a tervezésbe, és nehezítsük meg saját magunknak a hosszútávú munkát. (Értelemszerűen a 'mai igények' az összes olyan igényt jelentik, amiket ma ismerünk.) Bátorság kell ahhoz is, hogy ne ijedjünk meg attól, hogy a kódot folyton át kell dolgozni. Az átdolgozás azt jelenti, hogy átnézzük a kódot, és olyan változtatásokat eszközölünk rajta, amik egyszerűbbé és átláthatóbbá teszik. Bátorság kell ahhoz is, hogy eldobjunk már megírt kódot.

Tisztelet

Az XP-ben a tiszteletnek is többféle aspektusa van. Tiszteljük a többi csapattagot, mert dacára a folyamatos változásoknak és integrációnak, sosem adunk be olyan kódot, ami nem fordul le, vagy hibás abban az értelemben, hogy a részegység-teszt hibát eredményez rajta. Úgy általában semmit nem teszünk, ami a többiek munkáját hátráltatja. Magunkat is

tiszteljük annyira, hogy csak jó minőségű munkát adunk ki a kezünk közül, és mindig a legjobb tervet igyekszünk kitalálni, és az átdolgozásokat is lehető legjobban megtervezni.

Elvek

Az elvek, amelyek az XP alapját képezik, következnek a fent leírt értékekből, és arra valók, hogy segítsenek döntéseket meghozni. Az elvek konkrétabbak, mint az értékek, és jobban használhatók iránytűként konkrét helyzetekben.

Gyors visszajelzés

A visszajelzés akkor a leghasznosabb, ha hamar megérkezik. A konkrét esemény és a róla érkező visszajelzés között eltelt idő kritikus fontosságú a tanulság levonásának és az esetleges változtatások tekintetében. Az XP-ben, a hagyományos módszertanokkal ellentétben, az ügyféllel való kapcsolattartás sok pici esetben fordul elő, hogy az ügyfélnek tiszta képe legyen arról, hogy mi történik a fejlesztésben. Így a visszajelzései alapján lehet a projektet irányítgatni.

A részegység-tesztek is fontosak a gyors visszajelzés érdekében. Amikor az ember a kódot írja, a részegység-teszt a leggyorsabb közvetlen visszajelzés arra, hogy milyen hatása lett az új kódnak. Továbbá, ha a változás olyan funkcionalitást érint, ami nincs a programozó látóterében, és álmában sem gondolja, hogy arra hatása lehet az ő kódjának, a részegység-tesztek ezeket is észre fogja venni, és a bug nem akkor derül ki, amikor a rendszer már élesben üzemel.

Inkrementális változtatások

Az XP fáklyavivői azt mondják, hogy Rómát sem egy nap alatt építették. Nem lehet egyszerre nagy változtatásokat csinálni egy rendszeren. Az XP fokozatos változtatásokat javasol. Megeshet, hogy ettől egy rendszernek három hetente lesz új kiadása, mindig csak apró változásokkal. A sok kis lépéssel az ügyfél is jobban látja, hogy merre halad a projekt.

Örömmel fogadott változások

Ez az elv azt mondja ki, hogy nem elég, hogy ne tegyünk semmit a változások ellen, pont ellenkezőleg, örüljünk nekik. Ha az egyik szokásos napi találkozó során kiderül, hogy az ügyfél igényei drámaian megváltoztak, akkor a programozók örüljenek neki, és kezdjék el tervezgetni az új iterációhoz szükséges terveket.

Tevékenységek

Az XP-ben alapvetően négyféle tevékenység van.

Kódolás

Az XP már emlegetett fáklyavivői szerint a rendszerfejlesztés egyetlen igazán hasznos végterméke a kód, bár a 'kód' kifejezést szélesebb értelemben használják, mint a hagyományos szemlélet hívei. Kódolás nélkül nincs semmi.

A kódolás jelentheti diagramok megrajzolását is, amikből aztán program lesz, vagy scriptek írását a webalapú rendszerhez, vagy egy C#-ban készülő objektum megírását, amit majd aztán le kell fordítani.
A kódolás néha ahhoz is kell, hogy a legjobb megoldást megtaláljuk. Az XP szerint előfordulhat az, hogy ha egy problémának több, látszólag ugyanolyan jó megoldása van, akkor mindet meg kell írni, és automatizált tesztekkel kell eldönteni, melyik a legjobb.

A kódolás az egyik eszköz a gondolatok kifejezésére, konkrétan a programozási problémákról keletkező gondolatok kifejezésére. Egy programozó, aki egy bonyolult programozási problémával küszködik, lehet, hogy nem tudja elmagyarázni rendesen a megoldás lényegét a kollégáinak, de meg tudja írni, és meg tudja nekik mutatni a kész kódot. (Adott esetben ez lehet akár pszeudokód is.) A kód, mondják eme álláspont hívei, mindig tiszta és egyértelmű, és nem lehet többféleképpen értelmezni, csak úgy, ahogy a számítógép. A többi programozó pedig úgy fejezheti ki a véleményét a témával kapcsolatban, hogy beleírnak a kódba, vagy hozzátesznek.

Tesztelés

Semmiben sem lehetünk biztosak, amíg nem próbáltuk ki. A tesztelés általában nem az ügyfél kérése, sőt nem is jól felfogott érdeke. Rengeteg szoftvert adnak ki rendes tesztelés nélkül, ami működik, többé-kevésbé. Az XP azt mondja, hogy nem lehetünk biztosak a kód működőképességében, amíg alaposan ki nem próbáltuk. Ez felveti azt a kérdést, hogy pontosan mi is az, amiben nem lehetünk biztosak.

- Nem biztos, hogy azt kódoltuk le, amire gondoltunk. Ennek a bizonytalanságnak az eloszlatására vannak a részegység-tesztek. Ezek automatizált tesztek, amik a kódot tesztelik. A programozónak annyi tesztet kell írnia, amennyit csak tud, hogy lehetőleg minden ágát letesztelje a kódnak. Ha minden teszt sikeresen lefut, akkor a kódolás készen van.
- Nem biztos, hogy amire gondoltunk, az tényleg az, amit az ügyfél akar. Ezért kellenek az elfogadási tesztek, amiket az ügyféllel kell elvégezni, a release-tervezés felfedező fázisában.

Meghallgatás

A programozók nem feltétlenül tudnak az üzleti oldaláról annak a projektnek, amin dolgoznak, noha a rendszerrel támasztott követelményeknek az üzleti oldalról kell érkezniük. Annak érdekében, hogy a programozók megértsék, hogy mire akarják a programjukat használni, meg kell hallgatniuk az üzleti oldal mondanivalóját is. Azt kell belőle meghallaniuk, hogy mire van szüksége az ügyfélnek. Ezen felül jó, ha azt is megértik, hogy miért van rá szüksége az ügyfélnek, hogy visszajelzést tudjanak adni az ügyfélnek a saját problémájáról, és ezáltal ő maga is jobban értse, hogy mi kell neki.

Tervezés

Csupán az egyszerűséget véve alapul, mondhatnánk, hogy a rendszert nem is kell tervezni, elég, ha kódolunk, tesztelünk, és odafigyelünk az ügyfélre. Ha ezeket jól csináljuk, a végeredmény egy működő rendszer lesz.

A gyakorlatban ez nagyon nincs így. Messzire el lehet jutni tervezés nélkül, de a végén biztos, hogy zsákutcába kerülünk. A rendszer túl bonyolulttá válik, és a belső függőségek átláthatatlanná válnak.

Ezt úgy lehet elkerülni, hogy logikus részekre kell a rendszert bontani. Ha logikus, és jól elkülönülő részekre sikerül a rendszert szétszedni, akkor a függőségek nem fognak gondot okozni, vagyis a rendszer egy részének megváltoztatása nem teszi tönkre az egész rendszert, és az egyes részegységek bonyolultsága sosem fogja túllépni a kritikus küszöböt.

Az XP ellentmondásos részei

- A legellentmondásosabb, legproblémásabb része a dolognak a változás-menedzsment. Mivel az ügyfél közvetlenül kommunikál a programozókkal, leginkább szóban, így két rossz dolog történhet. Az egyik, hogy az összevissza csapongó változásai költséges átdolgozásokhoz vezetnek, a másik az, hogy az ügyfél szép lassan egyre többet és többet követel.
- Nincsenek követelmény-listák, és specifikációk, tehát nehéz számonkérni bármit is. Értelemszerűen ennek következményeként olyankor érdemes XP-t használni, amikor vagy eleve szóba sem kerül a számonkérés, mert pl. belső az ügyfél, vagy olyankor, amikor az elszámolásnak kizárólag az eltöltött idő az alapja.
- Az ügyfél képviselője szerves része a projektnek. Ezt sokaknak stresszt okoz, hiszen minden hiba és tévedés azonnal nyilvánvaló az ügyfél számára is. Ugyanakkor ha az ügyfél képviselője rosszul végzi a munkáját, akkor azon az egész projekt megbukhat.

Csapat felépítése és feladatuk

(Karakas Péter)

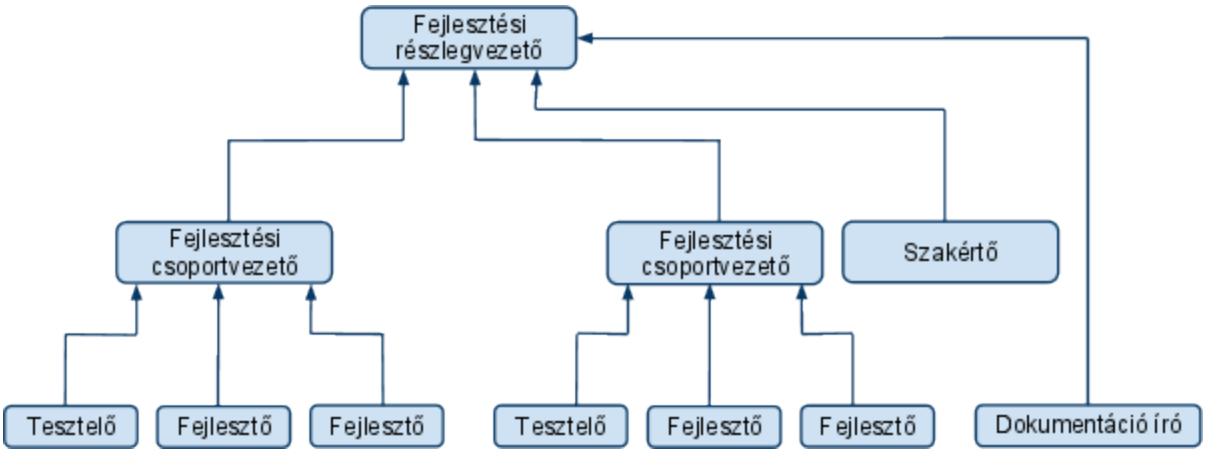
Akárcsak a hagyományos software fejlesztő cégeknél távmunka esetén is fontos egy jól felépített szervezeti struktúra. Sőt talán még fontosabb mint egy hagyományos cég esetén, hiszen a dolgozók nem tudnak egymással személyesen kommunikálni. Így kiemelkedően fontos, hogy mindenki tisztában legyen a feladatával és a hatáskörével. Ezért minden szereplő

pontos munkakörrel rendelkezik. Minden feladathoz tartozik egy ellenőrző lista. Ez az ellenőrző lista tartalmazza azokat a lépéseket, amiket a feladat végrehajtása során el kell végezni. Az elvégzett lépéseket fel is kell jegyezni, hogy ki mikor végezte el. Ez biztosítja, hogy egyetlen lényeges lépés se maradjon ki és visszakereshetővé teszi, hogy ki, mikor, mit csinált.

A távmunka ellenére nincs jelentős eltérés a tipikus projekt szereplők körében. Külön kitérünk azokra a plusz feladatokra, melyek a távmunka sajátosságai.

Nézzünk meg a gyakorlatban is működő példát, amelyet a Webiris Kft-nél alkalmazunk.

A szervezeti diagram mutatja meg, hogy milyen projektszereplők vannak és ki kinek tartozik elszámolással.



A fenti ábra fejlesztési részleg vázát mutatja. Ezek a legfontosabb szerepkörök. Egy ember akár több szerepkört is betölthet, hiszen az is előfordulhat, hogy valaki tesztelő és fejlesztő is egyben.
Nézzük meg az egyes szerepkörök feladatát, alapvető elvárásokat velük szemben.

Fejlesztő

Feladata egyértelműen a programozás, fejlesztés. A számára kiírt feladatok minél pontosabb végrehajtása. A fejlesztőnek nem feladata az ügyféllel való kapcsolattartás.

A fejlesztő a feladatait a Gforge ticket-kezelő rendszerben (a Gforge programról és annak használatáról későbbi fejezetben lesz szó) találhatja meg. Alapvető követelmény, hogy a feladatok leírásához interneten keresztül bármikor hozzá tudjon férni, ezzel nem gátolva a térben és időben kötetlen munkavégzés.

A fejlesztő a fejlesztési csoportvezetőnek tartozik elszámolással.

Tesztelő

A tesztelők gyakran alábecsült szereplői egy-egy projektnek. Egy tesztelőnél előny, ha rendelkezik programozási alapismeretekkel vagy esetleg maga is fejlesztő. Ebben az esetben fontos, hogy mindig más által írt kódot teszteljen. A tesztelőnek nem feladata a Unit test-ek megírása. Ez minden esetben a fejlesztő feladata.

Fejlesztési csoportvezető

Fejlesztési csoportvezető feladata, hogy a csoportjába tartozó programozók és tesztelők határidőre és megfelelő minőségben végezzék el a feladatukat.

Egy csoportba, tapasztalataink szerint, maximum 2-5 ember tartozhat. Ez az a mennyiség, amit egy csoportvezető kezelni tud. A kontrollt a csoportja felett gforge, hudson programok segítik.

Hetente minimum kétszer tart megbeszélést a fejlesztőkkel. A megbeszéléseknek három csoportját különböztetjük meg.

- Napi: alapvető feladatok átbeszélése. Közösen átnézik, hogy a határidők tarthatóak-e és a mérőszámok alakulását. Ekkor napi célokat tűznek ki. Ez szintén motivációs tényező, hiszen a cél elérése dicsérettel jár.
- Heti: a hét első megbeszélése. Ekkor történik meg az előző hét összegzése és a következő hétre kitűzendő feladatok meghatározása.
- Havi:Hudson pontverseny győztese kerül kihirdetésre. (A pontversenyről a Projektvezetés eszközei fejezetben lesz szó)

Természetesen az is előfordulhat, hogy egy megbeszélés egyszerre több kategóriába is beletartozik. Hiszen egy heti megbeszélés egyben napi is.

A csoportvezető feladata a ticket-ek lezárása. Ekkor személyesen is meg kell győződnie, hogy a feladat az elvárásoknak megfelelően lett-e elvégezve. Ellenőrzi, hogy elkészültek-e a szükséges unit test-ek, dokumentációk helyesek-e, lehetséges-e a kódon egyszerűsíteni, újrahasznosítani.

Fejlesztési részleg vezető

Feladata, hogy a csoportok határidőre elvégezzék a feladatokat. A csoport vezetőkkel hetente legalább egyszer megbeszélést tartanak. Továbbá az ő feladata a következő release-hez a lehetséges user story-k összeállítása. Ezt azonban nem önkényesen dönti el, hanem többi vezető és a felhasználók igényeihez igazodva. A felhasználói igények felmérése előzetesen minden második héten (sprint indulás előtt egy héttel) kérdőívek segítségével történik. Ő hozza létre a release indítási dokumentumot.

A csoportvezetők segítségével a lehetséges user story-kból kiválogatják az elvégezendő feladatokat. Az elvégezendő komplexitást az előző release-ek során teljesített komplexitások alapján és a következő két hét eseményeitől függ. Gyakorlati tapasztalatunk, hogy vizsgaidőszak alatt csökken a teljesíthető komplexitás. Ez alapján létrejön a release indítási dokumentum, ami tartalmazza az elvégezendő user story-kat és csoportot, amelyik elvégzi azt illetve a határidőket.

Fejlesztési részleg vezetője felelős a sprint lezárásáért. Ekkor személyesen meg kell győződnie, hogy az adott verzió kiadásra kész. A kiadás a csoportvezetők társaságában történik.

További feladatai közé tartozik a programozó felvétele, programozó elbocsájtása és ezekkel kapcsolatos adminisztratív teendők, továbbá kapcsolattartás az ügyfelekkel és szakértőkkel.

Szakértő

A szakértők olyan külső tanácsadók, akikhez az üzleti logikával kapcsolatos kérdésekkel fordulhat a fejlesztési részlegvezető. Tipikusan, jogászok, könyvelők vagy a Webiris esetében optikusok.

Karakas Péter

Dokumentáció író

Feladatuk a felhasználói kézikönyv karbantartása. Szinte minden release után frissíteni kell a felhasználó kézikönyvet, hiszen új funkció kerülhet bele. A dokumentálás a teszteléssel egy időben elkezdődhet, hiszen ekkor már rendelkezésre állnak a felhasználói felületek, így képernyőképek és részletes leírások is készülhetnek róluk. A dokumentáció író a legritkább esetben fejlesztő.

Új verzió kiadása a Webiris-nél

(Karakas Péter)

Nézzük meg, hogy a fentebb ismertetett módszereket, hogyan lehet alkalmazni a gyakorlatban és milyen problémákat vet fel a távmunka esetén illetve ezeket, hogyan lehet megoldani.

Akárcsak az agilis fejlesztésnél a Webiris-nél is úgy gondoljuk, hogy a változásokra a lehető leghamarabb kell reagálni. Ezért a Webiris Kft-nél a kéthetes kiadási stratégiát választottuk. Ez azt jelenti, hogy minden második héten újabb verziója jelenik meg a programnak. Ezek hibajavításokat és újabb funkciókat tartalmaznak.

Mielőtt részleteznénk egy új verzió kiadásának a folyamatát(ezt hívja a scrum sprint-nek) nézzük meg a Webiris technológiai hátterét. Azt már korábban említettük, hogy a Webiris látszerészek számára kifejlesztett ügyviteli rendszer. A program fejlesztés során is fontosnak tartottuk a távoli hozzáférés lehetőségét. Azaz az adatokhoz bármikor bárhonnan hozzá lehessen férni. Ezért már a kezdetekben web-es alkalmazásban gondolkodtunk. Így az ügyfelek egy böngészőn keresztül bármikor használhatják a programot.

Ez azzal az előnnyel jár, hogy minden felhasználó ugyanazt a verziót használja. Így nem kell párhuzamosan régebbi és újabb verziók frissítését megoldani. Ez jelentős könnyebbséget jelent.

Egy sprint atominak tekinthető abból a szempontból, hogy vagy sikeres vagy sikertelen. Nincs félig elkészült, vagy majdnem kész story. Ha nem sikerül határidőre elkészülni, akkor új sprint kezdődik. Amibe a nem teljesített story-k vagy átcsúsznak vagy törlődnek. Ezért különösen fontos a határidők tisztelete.

Egyik fontos eleme az agilis módszereknek a felhasználóval való kapcsolattartás. Ez a Webiris esetében nehezen megoldható, hiszen több ügyfél van, így lehetetlen minden egyes ügyfelet kikérdezni az elvárásairól, ötleteiről. Ezenfelül az ügyfelek kérései egymással ellentmondásosak is lehetnek. További probléma, hogy a kérések más és más prioritással rendelkezik ügyfelenként. Erre a problémára a lent olvasható megoldást dolgoztuk ki.

Sprint előkészítése, indítása

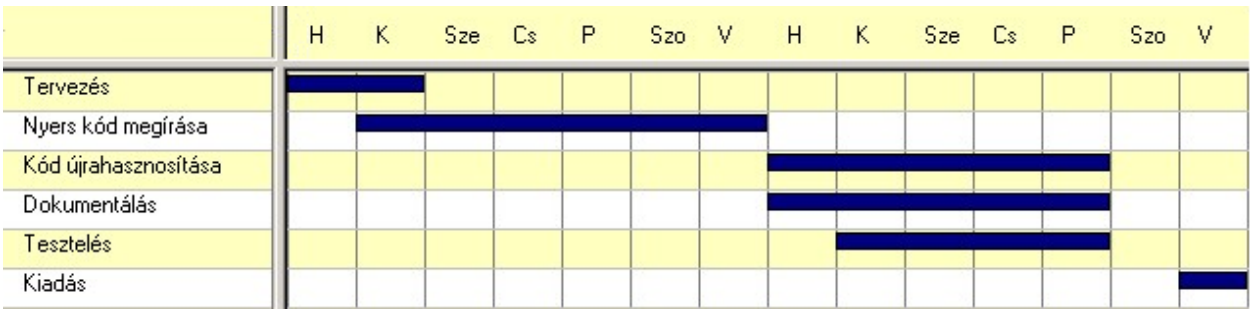
Egy sprint előkészülete már a kezdetét megelőző héten elkezdődik. Ekkor felmérjük a felhasználók igényét, kérdőívek segítségével, hogy milyen újítások szeretnének látni az új verzióban. A kérdőív egy demokratikus megoldás arra, hogy a többség által fontosnak tartott újítások kerüljenek bele a következő verzióba. A szavazásra kerülő újítások listája saját ötletek valamint az egyes felhasználóktól telefonon vagy email-en érkező kérések.

Ezután következik a legnépszerűbb felhasználó kérések komplexitásának a meghatározása. Minden csoportvezető meghatározza, hogy az adott story-ra hány koplexitást tud vállalni. A ügyfelekkel ezután közölni kell, hogy mire számíthatnak a következő verzióban. Ez tovább erősíti a bizalmat az ügyfelek és a Webiris között és motiválja a csapatokat, hogy ne okozzanak csalódást.

Ezt követi a release indtási dokumentum létrehozása, ami részletesen leírja, hogy melyik csoport milyen story-t fejleszt ki és milyen határidőket vállal.

Fejlesztés

A fejlesztés négy fázisra osztható: tervezés, implementálás (nyers kód írása, kód újrahasznosítása), tesztelés és dokumentálás. Ezek egymáshoz való lehetséges viszonya a lenti Gantt diagramon látható. Feladatoktól függően egy két nap eltérés lehet a folyamatok eleje és vége között.



A tervezés első része, hogy tisztázni kell a kérdéseket. Fontos, hogy minden fejlesztő részletesen átlássa a komplett feladatot, hiszen fejelesztés során számos apró döntést kell meghozniuk, de távolság miatt nem minden esetben van elérhető segítség. A tervezés mindig csapatmunka. A user story-k alapján a következő dokumentumok jönnek létre:

- CRC kártyák
- Domain model
- UI prototípusok
- Osztálydiagramok

Domain model: tartalmazza, hogy milyen adatelemek szerepelhetnek és azok, hogyan kapcsolódnak egymáshoz. Segítségével könnyedén létrehozható az adatbázisterv.

UI prototipusok: a felhasználói felületek drótváza. Ez egyfontos lépés hiszen gyakran ilyenkor dől el, hogy milyen új adatelemek jelennek meg.

CRC kártyák: segít feltérképezni, hogy az egyes osztályoknak mi lesz a feladatuk és milyen más osztályokkal lépnek kapcsolatba.

Osztály diagramok: Részletes leírás az osztályok adattagjairól és metódusairól.

Az agilis tervezésre jellemző, hogy az érthetőségre fektetni a hangsúlyt. Az UML egyik hátránya, hogy megértésük nem feltétlenül triviális, ellentétben a fenti megoldásokkal. Ezért ritkán használjuk. Így az oktatásból is kimaradhat az UML. Ez gyorsítja és csökkenti a költségét az új programozó betanításának.

Amint elkészülnek az első tervek megkezdődhet az implementálás. Az implementálás további két részre osztható: nyers kód megírása, kódok újrahasznosítása. Nyers kód megírása során az elsődleges cél a működő kód megírása, amelyik kielégíti az összes unit test-et. Azonban ez a kód legtöbbször nem optimális. Ezt követi a kód újrahasznosítása (refactoring). A

feladatot újragondolva gyakran egyszerűbb, gyorsabb megoldást lehet találni. Az előálló kód helyességéről a korábban megírt unit test-ek gondoskodnak. A kódok újrahasznosítása magában foglalja az esetleges feltárt hibák javítását is.

A tesztelés során meggyőződünk arról, hogy az újonnan kifejlesztett funkciók az elvártaknak megfelelően működik. Illetve, hogy a korábban már működő funkciók helyesen működnek-e. Ez a kézi tesztelés fázisa. Ha hibát tapasztalunk, azt jelezni kell a csoportvezetőnek.

A teszteléssel párhuzamosan folyhat a dokumentálás. Ekkor ugyanis már lehetnek elkészült programelemek. Ilyenkor történik meg a felhasználói kézikönyv frissítése. Fontos, hogy részletes leírás legyen arról, hogy mi hogyan használható. Az esetleges változásokról minden felhasználót tájékoztatni kell. Erre az email értesítést használjuk.

Sprint zárása

A lezárásáról a fejlesztési részlegvezető dönt. Amennyiben jónak látja kiadható az új verzió. Ez azt jelenti, hogy az ő felelősége, hogy ne kerüljön ki rosszul működő vagy hibás programkód. Szerencsére ebben sincs magára hagyva, hiszen Hudson és automatikus tesztek segítségével megbizonyosodhat arról, hogy a program helyesen működik.

Kiadás után következik a story-k elfogadtatása az ügyfelekkel. Ez a kiadás után egy héttel kérdőívben történik. Ahol osztályozhatják a végeredményt. Ez a megoldás azért szükséges, mert a felhasználók nagy száma miatt nem lehetséges a kiadás előtti egyenkénti elfogadtatás.

Projektvezetés eszközei

(Karakas Péter)

Távmunka esetén a hagyományos projekt vezetési eszközöket ki kell egészíteni, sőt gyakran le is kell őket cserélni. Például a hagyományos irodában megszokott cetlik vagy rajztáblák nem használhatóak.

Stand up

Tapasztalataink szerint jól működő módszer a SCRUM-ból át vett stand up távmunkás megfelelője. Hagyományos stand up esetén a résztvevők a munka megkezdése előtt állva tartanak egy rövid értekezést. Ez nyilvánvalóan nem használható távmunka esetén. Ezt kiváltva minden munkanapon 10:00 előtt el kell küldeni egy stand up e-mail-t, amely tartalmazza az előző munkanapon elért eredményeket, az aktuális napra tervezett feladatokat és azt hátráltató tényezőket.

Ez nem csak információ közlésre szolgál, hanem jelentős motiváló eszköz is. Ez helyettesíti a munkába járást, hiszen a dolgozó napi rutinjai közé beépül a reggeli stand up írás. Továbbá az üres stand up-októl való félelem munkára ösztönzi a dolgozót.

Skype, Team viewer: Online konferenciák, közös fejlesztés

Elő beszéd továbbítása nélkül szinte elképzelhetetlen a gyors kommunikáció. Erre tapasztalataink szerint a legmegfelelőbb eszköz a Skype. Alternatívája lehetne a Microsoft Messanger. Skype segítségével történnek az online megbeszélések lebonyolítása.

További hasznos eszközök a távolság leküzdésére a távoli asztalkapcsolatot megvalósító programok. Ennek segítségéven nem csak láthatjuk, de irányíthatjuk is a távoli kliens számítógépét. Ezáltal a monitora előtt érezhetjük magunkat. Egyszerűsége és hordozhatósága miatt a TeamViewer programot használjuk, de alternatívája lehet a LogMeIn is.

Ezen eszközök segítségével lehetőség nyílik a közös fejlesztés megteremtésére. Közös fejlesztésnek nevezzük azt az eseményt, amikor a csoport tagjai egyszerre online vannak és közösen fejlesztenek. Ez a módszer azon túl, hogy segíti az információ áramlást, hiszen mindenki egyszerre elérhető, növeli a teljesítményt is. Ekkor ugyanis mindenki számára kötelező a munka.

Dokumentációs rendszer: Miért fontos a dokumentáció?

Egy iroda esetén természetesnek számítanak a mappákba, iratkötőkbe gyűjtött dokumentumok, melyekhez az illetékesek szükség esetén

hozzáférhetnek. Azonban távmunka esetén nincs ilyen közös iroda. Ezért szükség van olyan eszközre, mely biztosítja a dokumentumok online hozzáférhetőségét. Szerencsére számos szoftver közül válogathatunk, melyek olyan lehetőségekkel szolgálnak mint a dokumentumok verziókövetése, hozzáférési napló, jogosultságok kezelése, egyszerű kezelhetőség és a biztonsági másolatok kezelése. Ezek az extra lehetőségek túlszárnyalják offline megoldások nyújtotta lehetőségeket.

A Webiris Kft-nél a Google Docs szoftvercsomag használata mellett döntöttünk, mivel telepíteni nem szükséges, ingyenes és könnyen használható.

A dokumentumok kezelése és rendszerezése kiemelkedően fontos. A nehezebb kommunikáció miatt bárki bármikor egyszerűen hozzáférhessen a szükséges információkhoz. Különösen igaz ez a munkaköri leírásokra és a fejlesztéshez tartozó dokumentumokra. A munkaköri leírásoknak érthetőeknek és részletesnek kell lenniük. Szabatosan megfogalmazva, hogy mikor mi a teendő.

Svn

Ma már csoportos fejlesztés nem képzelhető el valamilyen verziókövetést támogató rendszer nélkül. Az SVN segít abban, hogy a kódban történt bármilyen változás visszakereshető, visszavonható legyen. Ezenkívül kezeli a konfliktusokat, amikor fét fejlesztő ugyanazt a file-t szerkeszti.

Fontos megemlíteni a branch-ek lehetőségét is. Egy branch akkor jön létre, amikor a fejlesztésnek egy új ága alakul ki. Ekkor két különböző verziója él a kódnak. Azonban a rövid release határidők miatt és a sprint atomossága miatt ez nem szükséges. Így megspórolható a merge. Ami egy rendkívül időigényes és nem automatizálható folyamat. Merge az amikor két branch egy új ágban találkozik. Azonban ekkor a konfliktusokat egyesével, kézzel fel kell oldani.

Hibajegyek kezelése: Gforge

A hibajegy kezelő rendszer talán a legfontosabb projektirányítási eszköz. Segítségével nyomon követhető, hogy kinek milyen feladatai vannak,

melyikkel mennyi időt foglakozott. Példák ilyen programokra: trac, red mine, gforge, jira.

A gforge alapértelmezetten minden olyan funkcióval rendelkezik, ami a Webiris Kft számára szükséges és számos kimutatást is készíthetünk benne.

Tracker Name	Description	Item Total	Open Count
To-Do	To-Do tasks for this project	5	4
Management	Nem programozással kapcsolatos munkák	21	10
Suggestion	Javaslatok, ötletek	15	15
Support	Ügyfelektől érkező kérések	187	90
Feature Request	Feature Requert System	569	116
Bugs	Bug Tracking System	491	80

Manage Trackers

A hibajegyeket csoportokra oszthatjuk fel. A hibajegy elnevezés megtévesztő lehet, mert nem csak hibákat tartalmaznak. Ez a magyar terminológiában terjed így el, de minden elvégezendő feladathoz egy ilyen hibajegy készül.

Submitted By:
Viktor Mészáros

Open Date
2009-08-02 16:35:14

Close Date
2009-08-12 18:21:36

Priority: Medium

Assignee: Tibor Kigyósi, Kiss Antal, Viktória Veréb, Attila Nyilas, Viktor Mészáros

Status:* Closed

Commit:* None

Found In Release: Devel::Devel1.7_Vonalkod_GEARS_alapok

Fixed In Release: Devel::Devel1.7_Vonalkod_GEARS_alapok

Duration (Days):

Estimated Effort (Hours):

Percent Complete (0-100):

Component: webiris-Szervíz

Severity: normal

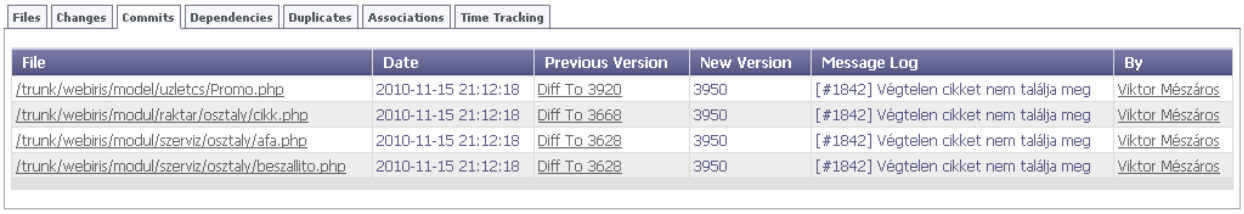
Resolution: None

Summary
Soap server vonalkód kép

Details (Edit)
Át kell irni hogy 90x15mm es képet csináljon.

Ez hibajegy számos információt tartalmaz. Ahogy a képen is látható.

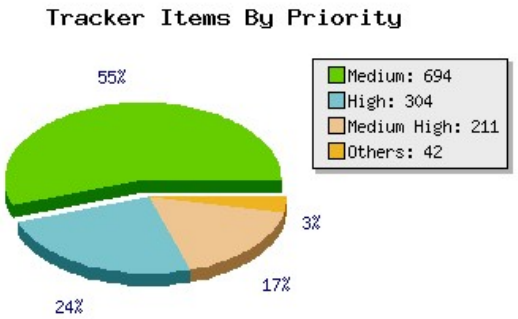
További fontos lehetőség, hogy a hibajegyhez tartozó svn commit-ok visszakereshetőek, ahogy a képen is látható:

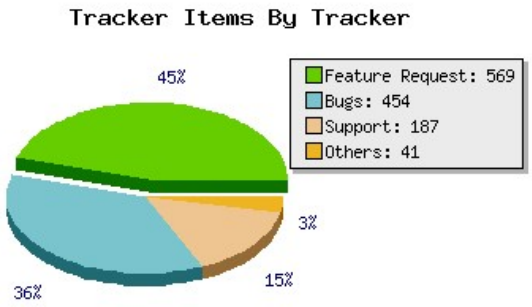


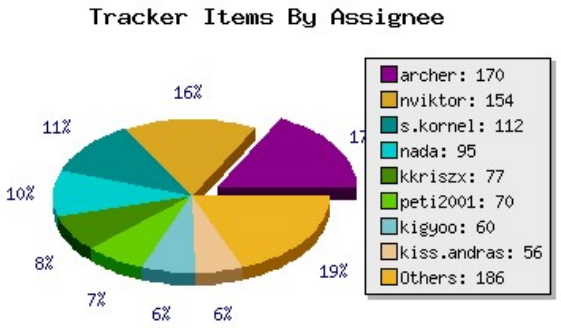
Files | Changes | Commits | Dependencies | Duplicates | Associations | Time Tracking

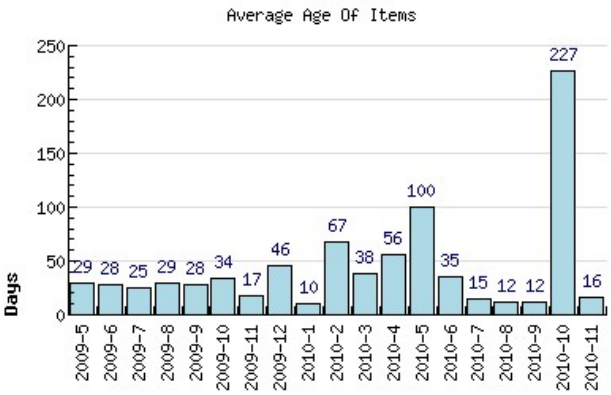
File	Date	Previous Version	New Version	Message Log	By
/trunk/webiris/model/uzletcs/Promo.php	2010-11-15 21:12:18	Diff To 3920	3950	[#1842] Végtelen cikket nem találja meg	Viktor Mészáros
/trunk/webiris/modul/raktar/osztaly/cikk.php	2010-11-15 21:12:18	Diff To 3668	3950	[#1842] Végtelen cikket nem találja meg	Viktor Mészáros
/trunk/webiris/modul/szerviz/osztaly/afa.php	2010-11-15 21:12:18	Diff To 3628	3950	[#1842] Végtelen cikket nem találja meg	Viktor Mészáros
/trunk/webiris/modul/szerviz/osztaly/beszallito.php	2010-11-15 21:12:18	Diff To 3628	3950	[#1842] Végtelen cikket nem találja meg	Viktor Mészáros

Továbbá a beépített kimutatások hasznos segítség jelenteknek a csoportvezetők és fejlesztési részlegvezető számára:









Összességében a gforge egy olyan eszközt nyújt a csoport számára, ami segít nyomon követni az aktuális sprint állapotát, a kiosztott feladatokat, további minden a kódban történt változás visszakereshető.

Hudson: kódminőség biztosítása

A csoportvezetők és a fejlesztési részlegvezető jobbkeze, amivel ellenőrizhetik a kód minőségét. Hudson segítségével a rendszeresen ellenőrizendő feladatokat lehet automatizálni. Mielőtt részletesen beszélnénk a Hudson-ról fontos megemlíteni a kód minőségét ellenőrző mérőszámokat.

Minden programozó saját "nyelvjárással" rendelkezik. Ez az a stílus, ahogyan a fejlesztő a programot írja. Ez fejlesztőnként változik. Ez azonban egy közös projekt esetén kavarodás eredményezhet. Nem formázhatja minden fejlesztő az általa megszokott formára a kódot. Ezért fontos, hogy fejlesztési konvenciókat határozzunk meg. Ilyen például, hogy a függvény deklaráció így nézzen ki:

```
public function függvény_neve()
{
    //függvény törzse
}
```

A **konvenciótól való eltérést** mérhetjük a CheckStyle nevű programmal.

Egy metódus bonyolultságát a **ciklomatikus komplexitással** mérhetjük. Adott szoftver forráskódjának alapján határozza meg annak komplexitását, egy konkrét számértékben kifejezve. A komplexitás számítása a gráfelméletre alapul. A forráskódban az elágazásokból felépülő gráf pontjai, és a köztük lévő élek alapján számítható.(http://hu.wikipedia.org/wiki/Ciklomatikus_komplexit%C3%A1s). Ha egy kódrész komplexitása meghalad egy megengedett értéket, akkor figyelmeztet.

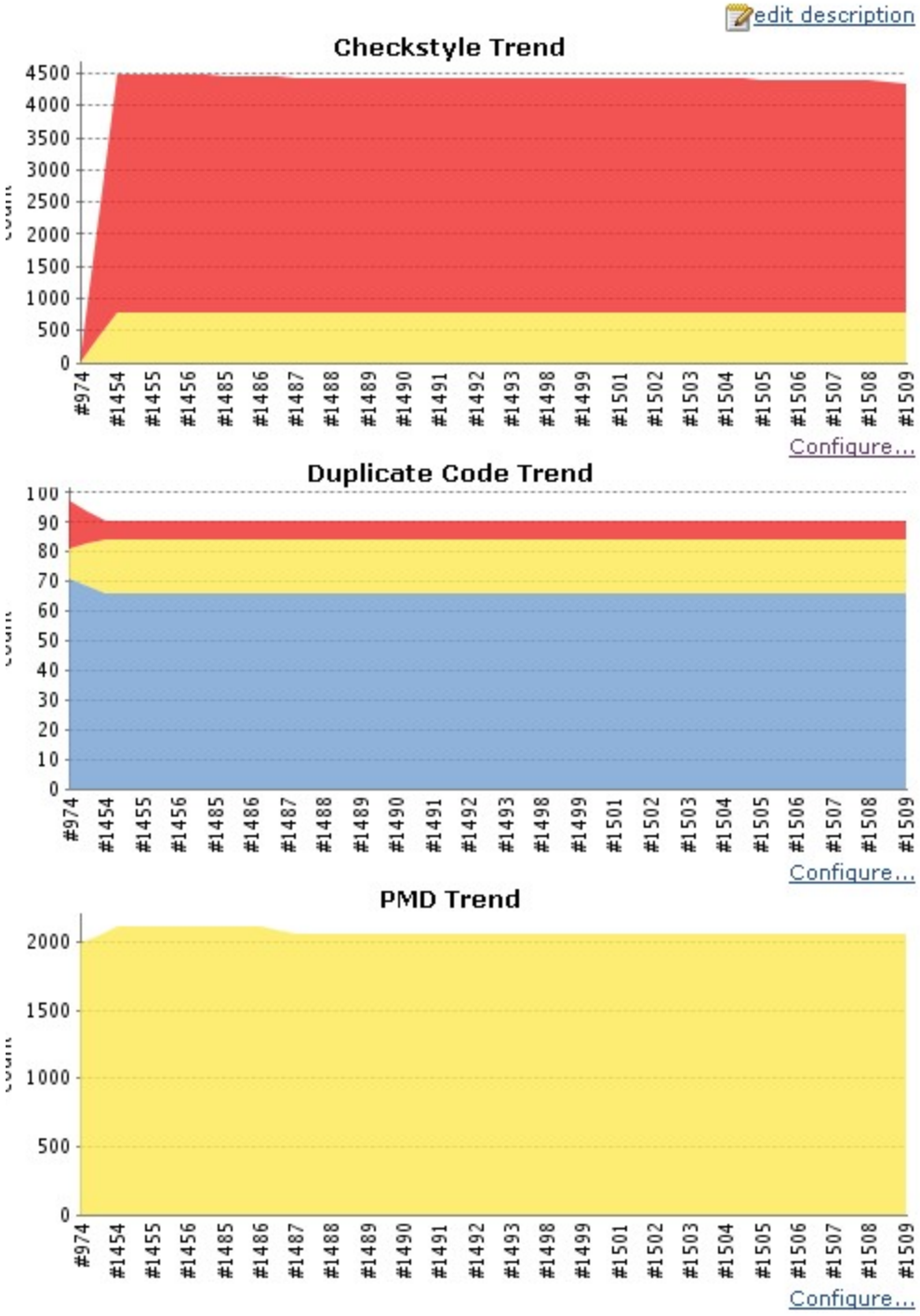
Ma már a legtöbb nyelv esetén lehetőség van a unit test-ekkel **lefedett kódok mérésére** is. Röviden csak kódlefedettség (code coverage). Itt törekedni kell 90% körüli értékre. Ez vizuálisan is ábrázolható.

```
  :     /**
  :      * Sets the bank account's balance.
  :      *
  :      * @param  float $balance
  :      * @throws InvalidArgumentException
  :      * @access public
  :      */
  :     public function setBalance($balance)
  :     {
1 :         if ($balance >= 0) {
0 :             $this->balance = $balance;
0 :         } else {
1 :             throw new InvalidArgumentException;
  :         }
0 :     }
```

Piros sorok jelölik a még tesztelésre váró sorokat.

Fejlesztés során a kód másolás egy veszélyes művelet. Ugyanis ekkor a változásokat több helyen is el kell végezni. Azonban nincs garancia, hogy az összes helyen elvégeztük a változtatásokat. Ezért a kód másolást a minimálisra kel szorítani. Ennek a mérőszáma a **kód duplikáció**.

Ezen mérőszámok segítik a csoportvezetőket és fejlesztési részlegvezetőt, hogy kódot olvasható és könnyen karbantartható minőségben tartsa. A Hudson automatizálja ezt az ellenőrzési folyamatot. Minden commit után kigenerálja ezeket a mérőszámokat és grafikonon ábrázolja őket.



Ha hibát észlel (például valamelyik unit test sikertelenül fut le), akkor az illetéseket tájékoztatja. Ezzel gyakorlatilag azonnal kiszűrhető, ha olyan változás került a kódba, ami hibás működést eredményez.

Ezen kívül a Hudson remek motivációs eszköz is lehet. Az olyan commit-ok után, amik javítottak a kód minőségén pontokat kap a fejlesztő. Ha pedig

romlott a kód minősége, akkor pontlevonást kap. Ezzel egy pontverseny alakul ki a fejlesztők között. A pontok minden hónapban nullázásra kerülnek.

Automatizált tesztek

Az automatizált tesztek, amelyeket a Hudon futtat, segítenek az esetleges hibák minél gyorsabb kiszűrését. Az automatizálás nagyon fontos, hiszen ezáltal megkerülhetetlenné válik és időt takarít meg a csoportvezetőknek. A teszteknek két típusuk van: unit testek, felület test-ek.

A Unit test-ek a program egy önálló egységét tesztelik csak. Egy metódust, a környezettől függetlenül. A függetlenség nagyon fontos, hiszen így pontosan kiszűrhetőek a hibás elemek és csak a hibás elemek.

Unit test-ekkel azonban nem oldható meg a felhasználó felületek tesztelése. Erre a Selenium kiegészítőt használhatjuk. Segítségével előre meghatározhatunk lépéseket, melyeket a Selenium automatikusan végig kattint és ellenőrzi a böngészőben megjelenő tartalmat.

Csoport összeállítása

(Karakas Péter)

A csoport egy csoportvezetőből és maximum 5 főből áll, akik lehetnek fejlesztők és/vagy tesztelők. Csoport toborzásánál érdemes online eszközöket igénybe venni. Ez logikusnak tűnik, hiszen aki internetek keres munkát nagyobb valószínűséggel alkalmas távmunkára is.

Felvételiztetés

Olyan személyeket kell keresni akik megfelelnek az alap elvárásoknak. Ezeket az alap elvárásokat nem kell a megtanítani, így csökkenthető az oktatási ideje. Érdemes azonban egy vizsgát készíteni, ami rákérdez az alapokra és csak azokat elindítani a tréningen akik sikeresen teljesítették a vizsgát. Az alsó határt kétféle képen is megadható. Vagy az X legjobbat, vagy egy bizonyos százalékot teljesítők kezdhetik meg a tanulást. Ez cégpolitikától függően változik. Az elsőt akkor érdemes, ha sok jelentkező van és a csoport oktatását egyben szeretnénk indítani. Azaz például félévente van munkaerő felvétel. A százalékos felsőhatár pedig abban az

esetben hasznos, ha egész évben van munkaerő felvétel és csak egy bizonyos szintet teljesítők kezdhetik meg a tanulást. Ebben az esetben nincsenek csoportok, hanem mindenkinek más és más időben kezdi az oktatást. Ezzel folyamatos munkaerő utánpótlást lehet elérni. Azonban mindig érdemes több emberrel indítani a tanfolyamokat, mint a betöltendő hely, mert a tapasztalatok szerint minden esetben vannak, akik nem csinálják végig a tréninget.

Oktatás

Minden cégnél fontos a jól képzett munkaerő. Ehhez biztosítani kell az új dolgozók betanítását és meglévők továbbképzését. A Webiris Kft-nél az új dolgozók gyakran tapasztalatlanok. Ez a diákmunkából és a távmunkából adódóan nagy a fluktuáció. Ezért a hatékony és gyors oktatás kiemelkedően fontos. Értelemszerűen itt is a távoktatás eszközeit érdemes használni.

Rengetek táv oktatást segítő szoftver létezik. Ezek közül a legismertebb, ingyenesen is használható eszköz a Moodle. Ezzel a programmal lehetőség van a tananyag összeállítására, ütemezésére és a vizsgáztatásra is.

Szintén kérdéses lehet, a oktatás időbeosztása. A tananyagot leckékre kell bontani. Minden lecke egy jól meghatározott témáról szól. Az, hogy milyen ütemben érkeznek a leckék kétféle lehet fix és kötetlen. Fix esetén az újabb lecke meghatározott időnként érkezik. Aki addig nem tudta teljesíteni a feladatát az kiesik a csapatból. Ezt a technikát csoportos oktatással érdemes együtt használni, amikor együtt kezdenek és együtt végeznek a betanulást végzők. Kötetlen időbeosztás esetén, ha valaki sikeresen teljesítette a feladatát azonnal léphet tovább a következő leckére. A tananyagot érdemes úgy összeállítani, hogy teljesíthető legyen 1 - 1,5 hónap alatt.
A fentiekből következik, hogy érdemes minden lecke után egy feladatot vagy/és vizsgát is betenni amivel felmérhető ki mennyire tanulta meg az adott leckét.

Menedzsment

(Karakas Péter)

Ledolgozott óra és fizetés

A távmunka egyik problémája az egyéni teljesítmény mérése. Hagyományos munkahelyeken a munkahelyen eltöltött idő után fizetnek. Ez kétségtelenül objektív mérőszám, de munkáltató szempontjából nem feltétlenül kedvező, mert nem az eredményes munka után fizet.

Ezzel szemben távmunka esetén a az elvégzett munka az, ami számít. Például egy értékesítőnél vagy ügyfélszolgálatos esetén könnyedén mérhető az aláírt megrendelések vagy a megválaszolt levek száma. Azonban software fejlesztésnél nem adhatunk meg ilyen objektív mérőszámot. A fizetés megállapítására az alábbi lehetőségek vannak:

- Fix bérezés
- Bevallott óraszám
- Ticket vagy projekt alapú

Nézzük meg ezeket részletesen

Fix bérezés

Fix bérezés esetén a fejlesztő napi meghatározott óraszámban dolgozik és ezért kapja meg a fizetését. Ez azt jelenti, hogy semmilyen kontroll és adminisztáció nem szükséges. Ez igen magas fokú szabadságot ad a fejlesztőnek. Ekkor a vezetőknek csupán szubjektív eszközeik vannak arra, hogy eldöntsék, hogy a fejlesztő az előre meghatározott óraszámot ledolgozta-e. Bérezés ezen módja a legkönnyebben megvalósítható hiszen nem igényel külön adminisztációt.

Bevallott óraszám

Ekkor az elvégzett munka mennyiségét óraszámban mérjük. Arról, hogy hány órát dolgozott a programozó saját maga nyilatkozik. Ekkor azonban felmerül a csalás veszélye. Hiszen nehezen ellenőrizhető, hogy ténylegesen hány ledolgozott órát jelent. Azonban a bizalom ilyen magas foga jótékony hatással is lehet a munkára (Viki 31.o). Saját tapasztalataim alapján előfordul az is, hogy a programozó szégyenli bevallani a valósan eltöltött munkát, ha az túlságosan elhúzódott. Ez is azt bizonyítja, hogy munkaerő válogatásánál különösen fontos tényező a lojalitás. Kevésbé lojális

munkatársak esetén megoldás lehet, hogyha a munkavégzés web-kamerás, vagy csoportos hanghívás közben történik. További lehetőség lehet a meghatározott időben történő fejlesztés. Ekkor azonban elveszíti a munkavégzés az időbeli rugalmasságát. Automatikus kontrollt lehet úgy megvalósítani ha a munkavégzés valamilyen speciális felületen történik amiben mérhető az aktívan eltöltött idő. A vezetőknek ezen kívül érdemes összehasonlítani a csoportdolgozóit. Illetve kiszűrni a kirívó eseteket. Például napi túl magas óraszám vagy nem megmagyarázható ledolgozott órák.

Összeségében elmondható, hogy becsület alapú elszámolás egy könnyen megvalósítható módja teljesítmény mérésének, mely kényelmes a munkavállaló és a munkáltató számára is és erős bizalmat épít ki közöttük.

Ticket vagy project alapú

Ebben az esetben minden elvégezendő feladat előtt megállapodnak, hogy a feladat teljesítése esetén mennyi fizetés jár. Ekkor nincs órabér hanem az elvégzett feladat után jár az előre meghatározott összeg. Tehát a ledolgozott órának nincs jelentősége. Ennek a módszernek a hátránya, hogy ha egy feladat megvalósítása elhúzódik, akkor igazságtalannak érezheti a fejlesztő a kapott összeget. Továbbá nehezen megbecsülhető, hogy mennyi a reális értéke egy-egy feladatnak. Ezen felül az a kérdés is felvetődik, hogy mikortól számít egy feladat elvégezettnek.

Az alkalmazott fizetési módszer alkalmazottanként változhat. Például egy ügyfélszolgálatos esetén, aki mondjuk naponta 4 órás ügyeletben van, érdemes a fixbérezést választani. Azonban például egy-egy design elem elkészítése esetén, ami viszonylag ritka, érdemes ticket alapú bérezést választani. Fejlesztők esetén a Webiris Kft-nél a bevallott óraszám alapján kalkuláljuk a fizetéseket.

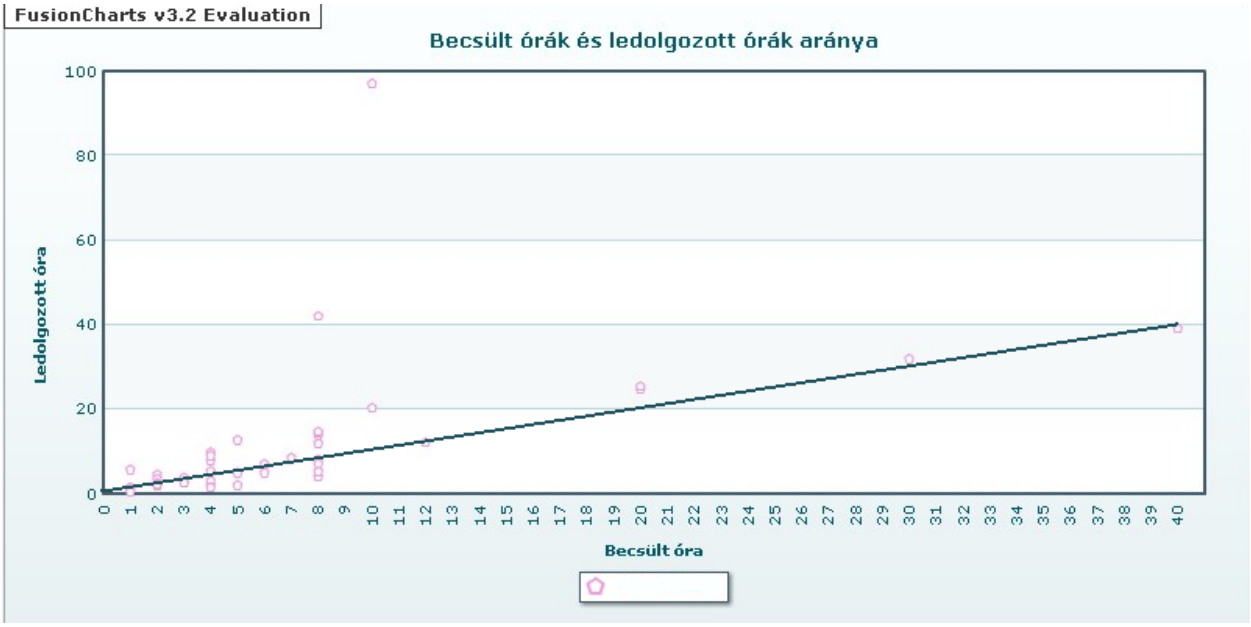
Velocity

Az angol terminológiában csak velocity néven emlegetett mérőszám lényege, hogy a programozó azon képességét határozza meg, hogy mennyire jól képes megbecsülni egy elvégezendő feladathoz szükséges időt(nehézségét). Kiszámításnak módja:

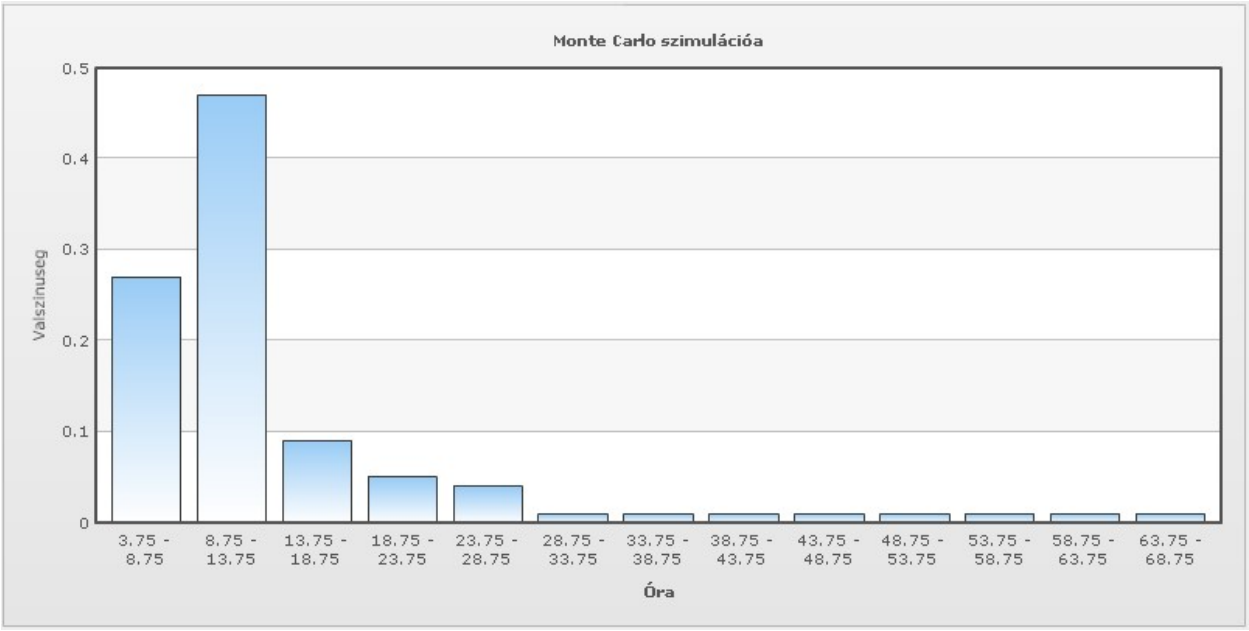
$v = h_{becsült}/h_{tényleges}$

Azaz a becsült órát elosztjuk a tényleges óraszámmal, azaz ami ahhoz kellett, hogy elkészüljön az adott feladat. Ennek előfeltétele, hogy minden programozó mielőtt nekilátna a feladatának becsülje meg, hány óra alatt tudja elkészíteni azt.

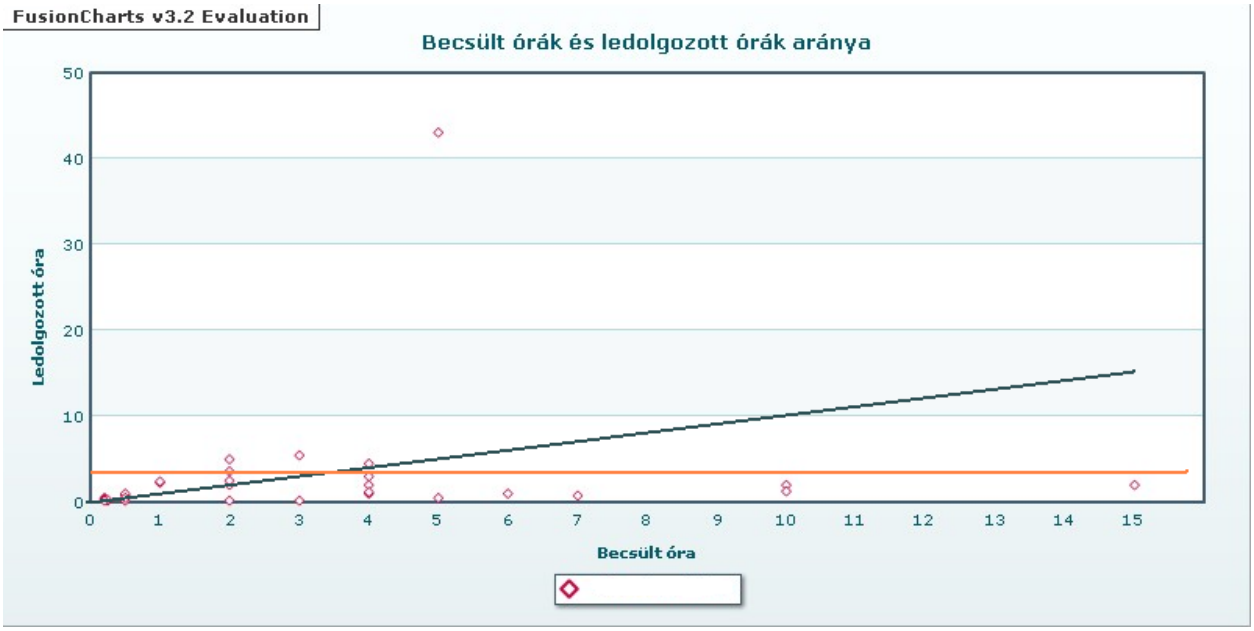
Ha $v < 1$, akkor alulbecsülte ha $v > 1$, akkor túlbecsülte a szükséges óraszámot. Jó becslőnek az számít akinek a v-k egy adott szám körül mozognak. Minél kisebb a szórás. Ez alapfeltétele annak, hogy projeckt-ek várható elkészülési idejét jól lehessen becsülni.
(http://www.joelonsoftware.com/items/2007/10/26.html)
Nézzünk egy életből vett példát:
Az ábrán egy programozóhoz tartozó grafikon látható.



A sötétkék egyenes jelöli az y = x egyenest. Ebből következik, hogy a programozó alá és felé is becsülte a feladatokat és két kivételtől eltekintve egyenletesen.
Ehhez tartozó Monte Carlo szimuláció (a Monte Carlo szimulációról a Program dokumentációban lehet olvasni) a következő paraméterekkel:
100 szimuláció 1 darab 10 órára becsült feladat.

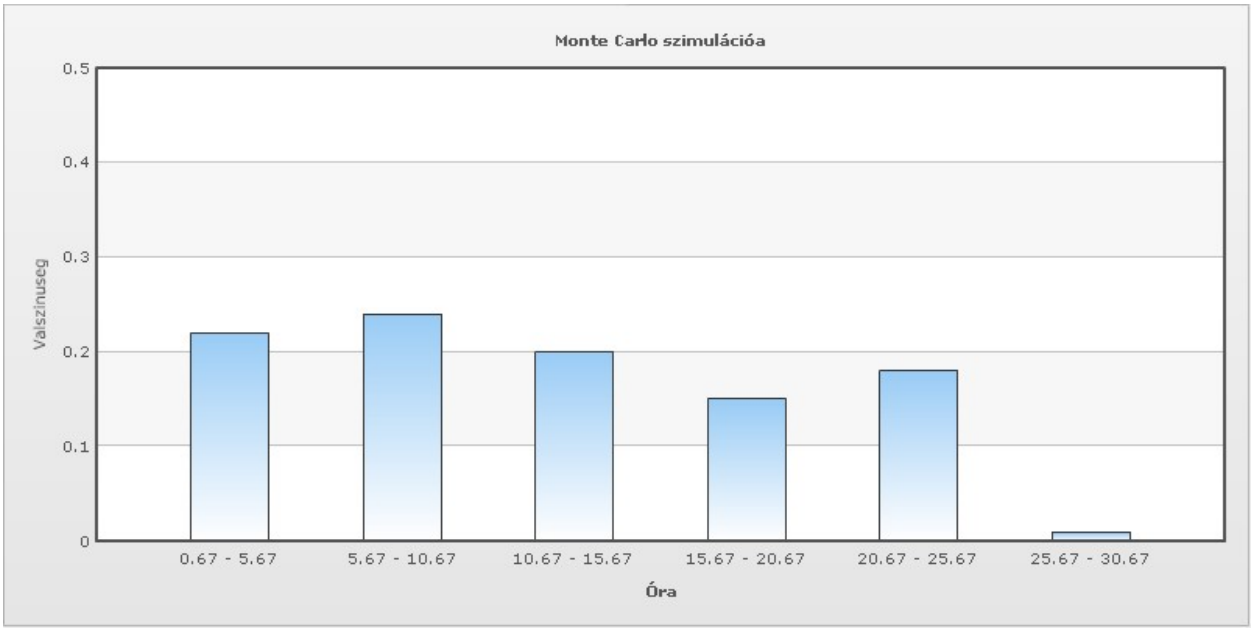


Az ábráról az olvasható le, hogy legnagyobb valószínűséggel 3.75 és 13.75 óra között készül el. Ez a velocity értékekből várható is volt. Azaz általánosan elmondható, hogy jól tervezhető ez az eset, hiszen egy kiugró értékünk van a többi lehetőség hozzá képest elhanyagolható. Az esetek 70%-ban 13 óra elég feladat elvégzéséhez.
Nézzünk meg egy másik esetet, amikor a programozó nem ilyen kedvezően becsüli meg a feladat nehézségét.



Az ábrát megfigyelve látható, hogy az ideális egyeneshez képest elszórva helyezkednek el a pontok. Továbbá az y = 3 egyeneshez képest

egyenletesen helyezkednek el. Ebből azt a következtetést vonhatjuk le, hogy a programozó az esetek többségében 3 óra alatt befejezte az adott feladatot, de sokszor alábecsülte a feladatot.



A grafikonról az olvasható le, hogy 0.67 és 25.67 óra között szinte ugyanolyan valószínűséggel készülhet el a feladat. Ez nehezen tervezhető.

Kommunikáció

(Karakas Péter)

Kapcsolattartás az ügyféllel

Különösnek tűnhet, de akár az ügyfeleknél is megoldható, hogy elkerüljük a személyes kapcsolattartást. Ez különösen akkor igaz, ha az ügyfelek megszerzése is online történt. Ekkor az ügyfelekben ki sem alakul a személyes kapcsolattartás igénye. Azonban ez csak bizonyos üzletágakban teljesülhet. Ekkor a kapcsolattartás eszköze a telefon illetve a email illetve akár a hagyományos levél is lehet.

Kapcsolattartás a dolgozókkal

A dolgozók egymás közötti kommunikációja során törekedni kell az online eszközök használatára. Míg az ügyfelek kevésbé rugalmasak-e téren, addig a dolgozóknál jelentős összegeket lehet megtakarítani, ha minimálisra vagy akár nullára redukáljuk a telefonos vagy postai kommunikációt.

A dolgozók nem csak a feletteseikkel, de egymással is kommunikálni szeretnének. Fontos, hogy mindenki tisztában legyen azzal, hogy hol keresse a kollégáját. Teremteni kell egy olyan egységes felületet, ahol elérhetik egymást a dolgozók. Érdemes egy szoftvert előírni és kötelező venni, (például Skype) így elkerülhető, hogy különböző programok használata miatt a dolgozók elszeparálódva külön csoportot alkossanak.

Az online kommunikáció másik fontos eszköze az email. Fontos, hogy minden dolgozó email címe elérhető legyen bárki számára. Továbbá fejlesztői levelező listák is fontos szerepet töltenek be a kommunikációban. Ezekre híreket és egyéb közérdekű közleményeket lehet kihirdetni.

Symfony PHP-s keretrendszer

(Nyilas Attila)

Egy keretrendszer előnyei

A PHP nyelvet igen könnyű megtanulni. Ha valakinek van C nyelven tapasztalata, akkor igen hamar készíthet egyszerűbb alkalmazásokat. A nyelv számos könnyítést is ad a C nyelvhez képest. Lazán típusos nyelv. Nem szükséges a változó deklaráció és futás közben válthatunk típust. Használhatunk osztályokat, de nem kötelező. Így procedurális és OO programozásra is van lehetőség.

Ezek első hangzásra hasznosnak tűnnek, mert pillanatok alatt lehet kisebb alkalmazásokat írni. De ezek a könnyítések adják a nyelv gyengeségét is, hiszen ilyen enyhe szabálykörnyezetben nagyon könnyen lehet rossz minőségű kódot írni.

A keretrendszer használata hosszútávon azért is fontos, mert olyan plusz szabályokat vezet be, ami megnehezíti a zavaros kódok írását. Előírja, hogy

minek hol a helye és a keretrendszer eszközeit használva átlátható kódot kapunk.

Sokan fognak saját keretrendszer írásába. Ezt semmi képen nem javasoljuk. Egy meglévő keretrendszer használata rengeteg plusz munkától kíméli meg a fejlesztőket. Hiszen ezzel megspórolható a keretrendszer kifejlesztésének a költsége. Sőt külön szervezet foglakozik a továbbfejlesztésével. Így tőlünk függetlenül is újabb és újabb újítások, hibajavítások kerülnek bele a keretrendszerbe. Oktatás szempontjából is előnyösebb egy meglévő keretrendszer választása. Hiszen egy piacon lévő keretrendszerhez már számos dokumentáció, tananyag, segítség létezik. Így megspórolható a saját tananyag írásának költsége is.

Akárcsak Java-ra, PHP-ra is sok Open source keretrendszer érhető el. Manapság a Zend framework, Yii, PHPCoke és a Symfony a legelterjedtebbek. Azért választottuk a Symfony keretrendszert, mert véleményünk szerint jelenleg ez a keretrendszer képes a legjobban támogatni a fejlesztőt és megkíméli őt a felesleges kódolástól. Továbbá egyre növekedő közösség áll mögötte ami biztosítja, hogy a közeljövőben nem marad abba a fejlesztése.

A Symfony telepítése Linux-ra

Az Apache-MySql-PHP környezet telepítése *sudo apt-get install mysql-server phpmyadmin* paranccsal a legegyszerűbb.
Ezek után telepíthetjük a Symfony-t:

- apt-get install php-pear
- pear channel-discover pear.symfony-project.com
- pear install symfony/symfony-1.4.8
- /etc/init.d/apache restart

Ha nem sikerül újraindítani az apache servert a 80-as vagy a 443-as portok foglaltsága miatt, akkor az /etc/apache/ports.conf fájlban az esetleges bejegyzés duplikációt töröljük.

Ezzel telepítettük a keretrendszert.

Alkalmazás létrehozása:

- mkdir *projekt* (ezzel létrehozzuk a projekt könyvtárat)
- cd *projekt*
- symfony generate:project *projekt*
- symfony generate:app frontend (alkalmazás létrehozása)

Tervezési minták
Nézzük meg melyek azok a tervezési minták, amiket a fejlesztés során használunk és a keretrendszer támogatja.

MVC Design pattern

(*http://ikon.inf.elte.hu/wiki/index.php?title=MVC*)
(*http://hu.wikipedia.org/wiki/Modell-n%C3%A9zet-vez%C3%A9rl%C5%91*)

Ma már de facto szabálynak mondható, hogy ha web-es alkalmazás fejlesztésről van szó, akkor MVC tervezési mintát kell követni. Talán ez a legfontosabb alapelv amit követnünk kell, hogy hosszútávon átlátható és karbantartható kódot kapjunk. Az MVC a Model, View és a Controller szavakból képzett mozaikszó. Lényege, hogy erre a három egységre bontja fel a kódot. Így nem keveredik az adatok elérése, megjelenítés és a vezérlés.

A Modell

A modell az ahol az üzleti logikát tároljuk. Ez felelős az adatok kezeléséért. PHP-s, illetve általában a webes alkamazásokban a modell komponenst egyre inkább relációs adatbázisok, tárolt eljárások és függvények segítségével valósítják meg. Ez szolgáltatja az adatokat a Controller-nek. A modell részei lehetnek a munkamenet-változók és az adatfájlok is. A gyakorlatban általában elmosódik a határvonal a modell és controller között

A Controller

Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat.

1. A felhasználó valamilyen hatást gyakorol a felhasználói felületre (pl. megnyom egy gombot).
2. A vezérlő átveszi a bejövő eseményt a felhasználói felülettől, gyakran egy bejegyzett eseménykezelő vagy visszahívás útján.
3. A vezérlő kapcsolatot teremt a modellel, esetleg frissíti azt a felhasználó tevékenységének megfelelő módon (pl. a vezérlő frissíti a felhasználó kosarát). Az összetett vezérlőket gyakran alakítják ki az utasítás mintának megfelelően, a műveletek egységbezárásáért és a bővítés egyszerűsítéséért.
4. A nézet (közvetve) a modell alapján megfelelő felhasználói felületet hoz létre (pl. a nézet hozza létre a kosár tartalmát felsoroló képernyőt). A nézet a modellből nyeri az adatait. A modellnek nincs közvetlen tudomása a nézetről.
5. A felhasználói felület újabb eseményre vár, mely az elejéről kezdi a folyamatot.

A View

A megjelenítő rész feladata, hogy "megmutassa" a felhasználónak a modell-t. PHP-val (illetve a webes nyelvekben általában) sablonokat (template-eket) használunk erre a célra. A template lehet egy olyan bonyolult, összetett rendszer is, mint a Smarty, vagy lehet annyira egyszerű, hogy sima php fájlokat készítünk, amiben majd főleg csak html-t írunk, elvétve fordul elő egy-egy print, echo, for, while, if...

A legjobb azt a szabályt követni, hogy template-ben igyekszünk csak echo-t, vagy print-et használni, táblázatokhoz for-t, vagy while-t, és lehetőség szerint messzire elkerüljük az értékadást és a függvényhívást. Ezzel megakadályozzuk, hogy a modell-t vagy controller-t kezelő kód beleszövődjön a view-ba.

Helper vagy Partial: Olyan html és php kódok amelyek néhány paramétert leszámítva nem változnak és sok helyen meg akarjuk jeleníteni. Ezeket kiemeljük egy külön fájlba. Így nem kell megírni minden egyes view-ban ahol szükség lenne rá hanem elég csak hivatkozni rá.

Mit nevezünk Action-nek?

Az action-ök vagy magyar terminológiában ’műveletek’ jelentik egy alkalmazás szívét, mivel ezek tartalmazzák az alkalmazás logikáját. Ezek hívják meg a modellt és adják át az adatokat a view-oknak változók formájában. Minden webes kérésnél az URL egy action-t hivatkozik a benne lévő paraméterekkel.

Symfony keretrendszerben minden action-t az alábbi formában kell elnevezni: execute*Név*. Mivel az alkalmazásunkat modulokra bontjuk ezért ezek az action-ök a modul-nak megfelelő *modulnév*Actions osztályban lesznek. Ez az osztály az sfActions osztályból vagy annak egy leszármazottjából kell hogy származzon.

```
class AuthActions extends sfActions
{
    public function executeLogin(sfWebRequest $request)
    {
        $this->form = new LoginForm();
        $this->message = '';
        if ($request->hasParameter('name')) {
            $programmer = Doctrine::getTable('Programmer')
                ->findByNameAndPassword(
                $request->getParameter('name'),
                $request->getParameter('password')
            );
            if ($programmer) {
                $programmerBO = new ProgrammerBO();
                $programmerBO->setModel($programmer);
                $programmerBO->login($this->getUser());

                $this->forward('index', 'index');
            } else {
                $this->message = 'Hibás bejelentkezés';
            }
        }
    }
```

```
    public function executeLogout()
    {
        $this->getUser()->setAuthenticated(false);
    }
}
```

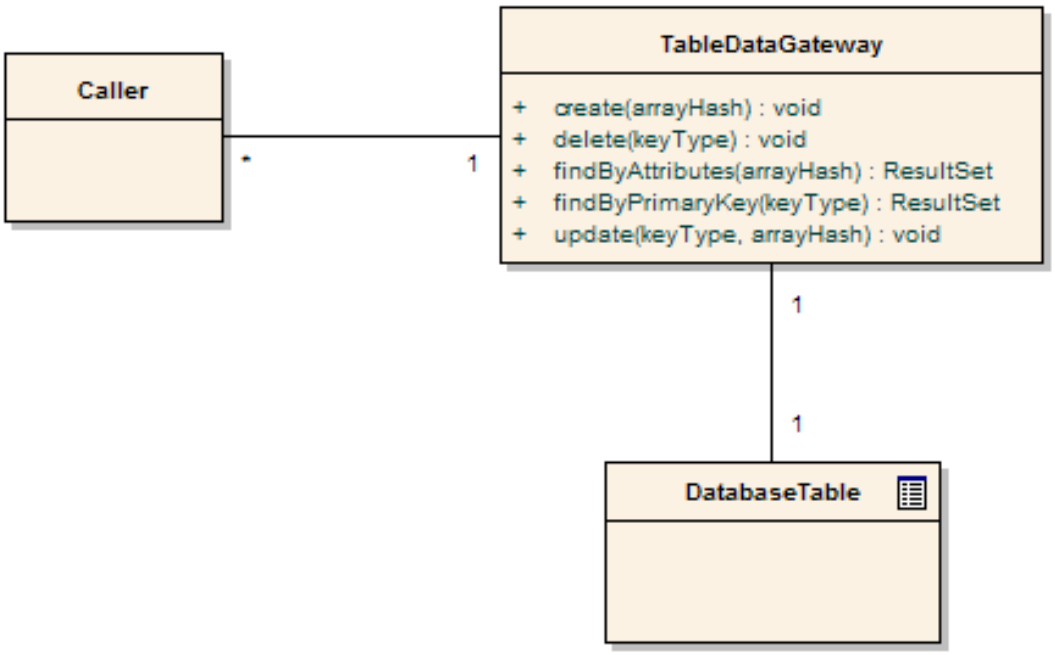
A fenti példakód a felhasználó bejelentkeztetését és kiléptetését oldja meg. Jól látható, hogy a controller csupán annyit tesz, hogy összekapcsolja a felhasználótól érkező kéréseket a hozzá tartozó üzleti logikával és view-val.

Front Controller design pattern

Minden web-es kérés először a Front Controller-hez kerül. Ez megkerülhetetlen. Web-es kérés például: `http://localhost/backend.php/programmer/new`. Ekkor a Front Controller a backend.php. Ide gyűjthetően azok a feladatok, amiket minden futás előtt el kell végezni. Ilyen például: futási környezet beállítása, jogosultság ellenőrzés, URL dekódolása, szűrők alkalmazása. Ezeket természetesen nem nekünk kell megírni hanem a Symfony eszközeit kell használni. Többféle Front Controller is megadható. Ezekkel különböző futási környezetek deklarálhatóak. Így változtatható a log-olás részletessége, hibaüzenetek megjelenítése, e-mail küldés. Például devel módban a hibakonzol számos fontos információt szolgáltat a fejlesztőnek, de éles környezetben teljesen felesleges, sőt veszélyes is lehet.

Table Gateway design pattern és a Doctrine

SQL utasítások használata egy webes alkalmazásban nem célszerű. A fejlesztő kényelmetlennek érezheti vagy ha nem akkor sem biztos, hogy jó SQL utasítást fog írni. Ezért az SQL utasításokra egy absztrakciós réteget építünk. Ezért olyan osztályokat alkotunk amelyek egy adatbázisbeli táblát, nézetet vagy rekordot reprezentál és képes szinkronizálni az adatbázissal. Így egy objektumon keresztül elérhetjük az egész táblát vagy rekordot és a beépített delete, insert, select, update függvényekkel könnyen végrehajthatjuk az utasításokat.



Mi az az ORM

Az Object Relational Mapping egy olyan programozási eljárás amely a programozási nyelv és a relációs adatbázis közti adattípus eltéréseket feloldja. Tipikus példa amikor például egy usernek (objektum) vannak telefonszámai egy listában vagy tömbben amik telefonszám objektumok. Ezt az adatbázisnak nem adhatjuk át mivel az csak egyszerű típusokkal tud dolgozni. Ekkor az ORM feladata az összetett objektumok egyszerű típusokká való átkonvertálása és az adatbázissal való szinkronizálása.

Előnyei:

- Elég php nyelven kódolni
- Nem kell ismernem az adatbázis működését
- Könnyebben alkalmazhatunk tervezési mintákat
- Megszokott OO eszközöket használhatunk

Hátrányai:

- Ez az extra réteg az adatbázis és a nyelv között sebességbeli csökkenéssel jár

- Bőbeszédű: Egy viszonlag rövid SQL utasítás így több soros kóddá alakul
- Az OO nem mindig a legjobb megoldás
- Keretrendszertől függhet

Sok ingyenes és kereskedelmi szoftver létezik, amelyek lehetővé teszik ezt, de néha saját ORM kifejlesztése is szükséges lehet.

Mi a symfonyban alapértelmezetten beépített Doctrine ORM-et használjuk.

Sor átjáró (Row data gateway)

A sor átjáró egy objektumot ad meg, ami pontosan olyan struktúrájú mint az általa reprezentált adatbázis rekord. Ezáltal a programozási nyelv összes eszköze használható vele és elfedi az adatbázis műveleteket a felhasználó elől.

Tábla átjáró (Table data gateway)

A tábla átjáró a sor átjáróhoz hasonlóan egy objektumot képez csak nem az adatbázis egy rekordjáról hanem egy egész tábláról. A visszakapott adat egy tömbben lesz. A tömb egyes elemei pedig sor átjátók. Kezeli a táblák közti kapcsolatot.
Doctrineban ez egy Doctrine_Collection-t eredményez.

Lehetőségünk van létrehozni saját Doctrine_Record objektumot vagy Doctrine_Collection-t és feltölteni adatokkal.
Doctrine_Recordnál egyszerű értékadást végezhetünk:
$record->vezeteknev = 'Nyilas';
A $record->save(); utasítás hatására az adatbázisba is bekerül az új rekord.
Doctrine_Collection létrehozásánál egyszerűen tudunk hozzáadni rekordokat:
$collection->add(*Doctrine_Record* $rekord);
Adatbázisba mentést szintén a save() metódus segítségével tehetjük meg:
$collection->save(*Connection*);

Doctrine Model

Az ORM legalsó szintjén az adatbázis sémáját php osztályokkal reprezentáljuk. Ezek definiálják a sémát és a viselkedést.

```
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', 7);
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }
   public function setUp()
    {
        parent::setUp();
        $this->hasMany('PhoneNumber', array(
             'local' => 'user_id',
             'foreign' => 'user_id'));
    }
}
```

Ez az osztály a users tábla definícióját adja meg.

Látható, hogy ebben az osztályban tudjuk megadni a kapcsolatokat is. Itt a hasMany metódussal beállítottunk egy 1:N kapcsolatot a telefonszámokhoz. Hogy teljes legyen a model a PhoneNumber osztályban is meg kell adni:

```
class PhoneNumber extends Doctrine_Record
{
    …
   public function setUp()
    {
        parent::setUp();
        $this->hasOne('User', array(
```

```
            'local' => 'user_id',
            'foreign' => 'user_id'));
    }
}
```

Itt hasOne-t használunk hiszen egy telefonszámhoz egy felhasználó tartozhat.
Ha a modell-t jól állítottuk be akkor elkezdhetjük a használni a DQL-t.

Doctrine DQL

A Doctrine Query Language egy objektum lekérdező nyelv ami az összetett lekérdezések írását könnyíti meg. Ráadásul kezeli a táblák közti kapcsolatokat.
Előnyei:

- Objektumokat kapunk vissza nem pedig rekordok halmazát amit aztán ’fetch’-elni kell
- Automatikusan lekérdezi a táblák közti kapcsolatokat amennyiben azok helyesen vannak megadva, így nem kell join-okkal bajlódnunk
- Kvázi adatbázis független
- Több beépített algoritmust tartalmaz amelyek segítik a hatékony lekérdezést

A Doctrine alapértelmezetten ismeri a find és findAll metódusokat. A Doctrine_Core::getTable('*Táblanév*')->find(*elsődleges_kulcs*) az elsődleges kulcs alapján ad vissza egy Doctrine_Record objektumot. A Doctrine_Core::getTable('*Táblanév*')->findAll() pedig a tábla összes rekordjával tér vissza egy Doctrine_Collection-ben.
Lekérdezés DQL-el:

```
$users = Doctrine_Query::create()
    ->select(’u.username as name’)
  ->from('User u')
    ->where(’u.name LIKE ?’, ’%Nyilas%’)$users->execute();
```

A $query változóban egy Doctrine_Collection-t kapunk vissza amely

tartalmazza az összes olyan rekordot amelyben a névnél ’Nyilas’ szerepel.
Az eredményt legegyszerűbb foreach-el végigjárni:

```
foreach ($users as $user)  {
    echo $user->name;
}
```

Itt nyílik lehetőségünk kihasználni, hogy a doctrine automatikusan le tudja kérdezni a kapcsolodó táblákat:

```
foreach ($users as $user) {
    foreach($user->PhoneNumbers as $phonenumber)  {
        echo $phonenumber->phonenumber;
    }
}
```

A $user->PhoneNumbers hatására megnézi, hogy a modellben szerepel-e a kapcsolat és ha igen akkor automatikusan lekérdezi.
A Dcotrine select query természetesen ismeri a groupBy, orderBy, limit utasításokat és van lehetőség kézzel megadni a join-okat.

Kódgenerálás

A symfony keretrendszer igazi előnye abból származik, hogy egy projekt vagy új modul elindulásakor nem kell minden műveletet magunknak végrehajtani. Nem kell például létrehozni a projekt vagy modul könyvtárszerkezetét. A keretrendszer megteszi helyettünk!
Továbbá olyan funkciók generálását is lehetővé teszi, hogy egy alap szinten működő alkalmazást tudunk készíteni anélkül, hogy le kellene kódolni ezeket a műveleteket.
Egy web-alkalmazásban tipikus műveletek egy adatbázis tábla tartalmának vagy egy részének listázása, adatok bekérése a felhasználótól egy form (űrlap) segítségével, rekord hozzáadása egy táblához vagy éppen törlése. Ezekhez az alapműveletekhez mind kódolni kellene, viszont a symfony kódgenerátora ezt is elvégzi ezáltal levéve a terhet a programozó válláról.

Alkalmazás létrehozása

A *symfony generate:project projektnév* parancs sikeres futtatása után létrehoztunk egy új projektet. Első lépésként az adatbázisunkról egy sémát kell hogy készítsünk. Ez a séma fájl a projekt/config/doctrine könyvtárban kell, hogy legyen, schema.yml néven. A kiterjesztéséből is adódóan YAML szintaxist követő kód-ot kell tartalmaznia. Ez szerencsére nem túl bonyolult. A bejegyzéseknek az alábbi struktúrát kell követniük:

```
Programmer:
  connection: adatbázis
  tableName: programmer
  columns:
    programmer_id:
      type: integer(4)
      primary: true
      autoincrement: true
    nev: string(100)
    password: string(32)
```

A Symfony lehetőséget ad, hogy Doctrine segítségével generáljuk ki ezt a séma fájlt, amit az adatbázis feltérképezésével hajt végre. Ezt a következő utasítással hajthatjuk végre:

`symfony doctrine:build-schema`

Ezt óvatosan kell kezelni, mert ha már megvan a séma fájlunk, amit esetleg kiegészítettünk plusz információval ilyenkor felülíródik!

Ha kész a séma fájl akkor kigenerálhatjuk a modellt. Ezt a

`symfony doctrine:build-model`

paranccsal tehetjük meg.

```
webiris:/home/petiokos/webiris/manage2_1_2# ./symfony doctrine:build-model
>> doctrine  generating model classes
```

A generálás után a lib/model/doctrine/base könyvtárban létrejönnek a tábláknak megfelelő osztályok a fentebb említettekhez hasonló struktúrával.

A különbség annyi, hogy az osztáy név Base*Táblanév* alakú.
Ezzel párhuzamosan a lib/model/doctrine könyvtárban létrejön két osztály: *Táblanév* és *Táblanév*Table néven. Az elsőben rekordszintű műveleteket adhatunk meg. Például felül definiálhatjuk a save metódust, hogy egy rekord mentésénél milyen egyéb műveletet hajtson végre. A másodikban tábla szintű műveleteket adhatunk meg. Felülírhatjuk a find és a findAll metódusokat, illetve saját lekérdezéseket írhatunk.

A modell után létrehozhatjuk az alkalmazást. Erre a symfony generate:app *név* paranccsal van lehetőségünk.

```
petiokos@webiris:~/webiris/manage2_1_2$ ./symfony generate:app frontend
>> dir+      /home/petiokos/webiris/manage2_1_2/apps/frontend/lib
>> file+     /home/petiokos/webiris/manage2_...s/frontend/lib/myUser.class.php
>> dir+      /home/petiokos/webiris/manage2_1_2/apps/frontend/i18n
>> dir+      /home/petiokos/webiris/manage2_1_2/apps/frontend/config
>> file+     /home/petiokos/webiris/manage2_...pps/frontend/config/routing.yml
>> file+     /home/petiokos/webiris/manage2_...s/frontend/config/factories.yml
```

A symfony ekkor létrehozza az alkalmazás vázát és minden olyan fájlt ami a futtatásához szükséges lesz. A config mappába az összes konfigurációs YAML fájlt és az i18n, lib, modules, templates könyvtárakat.
Modul létrehozásához az alábbi utasítást kell begépelnünk:

`symfony generate:module` *`alkalmazásnév modulnév`*

```
webiris:/home/petiokos/webiris/manage2_1_2# ./symfony generate:module backend auth
>> dir+      /home/petiokos/webiris/manage2_...ps/backend/modules/auth/actions
>> file+     /home/petiokos/webiris/manage2_.../auth/actions/actions.class.php
>> dir+      /home/petiokos/webiris/manage2_.../backend/modules/auth/templates
>> file+     /home/petiokos/webiris/manage2_...auth/templates/indexSuccess.php
>> file+     /home/petiokos/webiris/manage2_...nal/backend/authActionsTest.php
>> tokens    /home/petiokos/webiris/manage2_...nal/backend/authActionsTest.php
>> tokens    /home/petiokos/webiris/manage2_.../auth/actions/actions.class.php
>> tokens    /home/petiokos/webiris/manage2_...auth/templates/indexSuccess.php
```

Ez létrehozza az *alkalmazásnév* könyvtáron belül a *modulnév* könyvtárat amely 2 továbbit tartalmaz: actions és templates. Az actions mappában létrehozza az actions.class.php-t amelybe az action-jeinket írhatjuk. Alapból az executeIndex action-t teszi bele. A templates mappába pedig elkészíti az executeIndex action-höz tartozó nézetet (view) indexSuccess.php néven. Ez természetesen üres de az action hívásához szükséges, hogy létezzen a neki megfelelő view, különben hibát kapunk. Most már van egy működő alkalmazásunk amelynek van egy darab action-je. Ez persze még nem

jelenít meg semit.

Tegyük fel, hogy szeretnénk bekérni a programozó adatait. Ehhez írhatnánk egy HTML formot a megfeleló mezőkkel. A Symfony erre is kínál automatizált megoldást.

Formok

A `symfony doctrine:build-form` parancs hatására a lib/form/doctrine/base könyvtárba kigenerálódik az adatbázis összes táblájához egy form.

```
webiris:/home/petiokos/webiris/manage2_1_2# ./symfony doctrine:build-form
>> tokens    /home/petiokos/webiris/manage2_.../doctrine/MessageForm.class.php
>> tokens    /home/petiokos/webiris/manage2_...ine/UsergroupUserForm.class.php
>> tokens    /home/petiokos/webiris/manage2_.../doctrine/SnippetForm.class.php
>> tokens    /home/petiokos/webiris/manage2_...ElementTransitionForm.class.php
>> tokens    /home/petiokos/webiris/manage2_...doctrine/UserUnixForm.class.php
```

Példaként nézzük meg a programmer tábla alapján kigenerált formot.

```
abstract class BaseProgrammerForm extends BaseFormDoctrine
{
  public function setup()
  {
    $this->setWidgets(array(
      'id'           => new sfWidgetFormInputHidden(),
      'name'         => new sfWidgetFormInputText(),
      'passwd'       => new sfWidgetFormInputText()
    ));
    $this->setValidators(array(
      'id'           => new sfValidatorChoice(array('choices' => array($this
->getObject()->get('id')), 'empty_value' => $this->getObject()->get('id'),
'required' => false)),
      'name'         => new sfValidatorString(array('max_length' => 200,
'required' => false)),
      'passwd'       => new sfValidatorString(array('max_length' => 32,
'required' => false))
    ));
    parent::setup();
```

```
  }
  public function getModelName()
  {
    return 'Programmer';
  }
}
```

A mezőket itt widget-eknek nevezzük és a setWidgets segítségével vannak megadva. Ezek típusa a tábla adott oszlopának típusával fog megegyezni. A jelszó mezőt kézzel kell átállítanunk sfWidgetFormInputPassword típusra ahhoz, hogy a szokásos jelszó mező jelenjen meg.

A symfony továbbá kigenerál minden widget-hez egy validátort. Ez amint a neve is mutatja a widget-be írt értéket fogja validálni az adatok elküldésekor. Validátorok egész sorát lehet alkalmazni a különböző típusú mezőkhöz. Sőt, van lehetőségünk ezek tetszőleges kombinálására.

CRUD létrehozása

Egy web-alkalmazásban az adatokat mentünk és kérdezünk le az adatbázisból. Ezeket a műveleteket 4 kategóriába lehet sorolni:

- Rekord létrehozása (**C**reate)
- Rekord lekérdezése (**R**etrieval)
- Rekord módosítása (**U**pdate)
- Rekord törlése (**D**elete)

Példaként nézzük meg az értékelések (grade) listázását. Írnunk kell a modellbe egy lekérdezést ami visszaadja az értékeléseket. Kell egy action ami meghívja a modellből ezt a lekérdezést és átadja a view-nak, amit szintén meg kell írni, hogy meg is jelenjen egy lista. Ez kell ahhoz, hogy lássuk az értékelések listáját. Ha szeretnénk hozzáadni egy új értékelést vagy törölni a listából az mind új modell, controller és view-beli változtatásokat igényel.

Szerencsére a Symfony ebben is segítő kezet nyújt számunkra. Lehetőségünk van a Doctrine segítségével kigeneráltatni egy adminisztrációs

felületet egy modellbeli osztályhoz. Fenti példánkban ez a modellbeli osztály a Grade osztály ami az adatbázis grade tábláját reprezentálja.

```
webiris:/home/petiokos/webiris/manage2_1_2# ./symfony doctrine:generate-admin backend Grade --module="grade"
>> app       Generating admin module "grade" for model "Grade"
>> file+     /home/petiokos/webiris/manage2_...nd/modules/grade/lib/helper.php
>> file+     /home/petiokos/webiris/manage2_...les/grade/lib/configuration.php
>> file+     /home/petiokos/webiris/manage2_...ules/grade/config/generator.yml
>> file+     /home/petiokos/webiris/manage2_...grade/actions/actions.class.php
>> file-     /home/petiokos/webiris/manage2_...les/grade/lib/configuration.php
>> file-     /home/petiokos/webiris/manage2_...nd/modules/grade/lib/helper.php
>> file+     /home/petiokos/webiris/manage2_...al/backend/gradeActionsTest.php
>> tokens    /home/petiokos/webiris/manage2_...al/backend/gradeActionsTest.php
>> tokens    /home/petiokos/webiris/manage2_.../gradeGeneratorHelper.class.php
>> tokens    /home/petiokos/webiris/manage2_...eneratorConfiguration.class.php
>> tokens    /home/petiokos/webiris/manage2_...ules/grade/config/generator.yml
>> tokens    /home/petiokos/webiris/manage2_...grade/actions/actions.class.php
```

A generálás után létrejön a grade modul a szükséges könyvtár struktúrával és osztályokkal. Továbbá létrejön a grade/config könyvtárban egy generator.yml fájl.

generator.yml file:

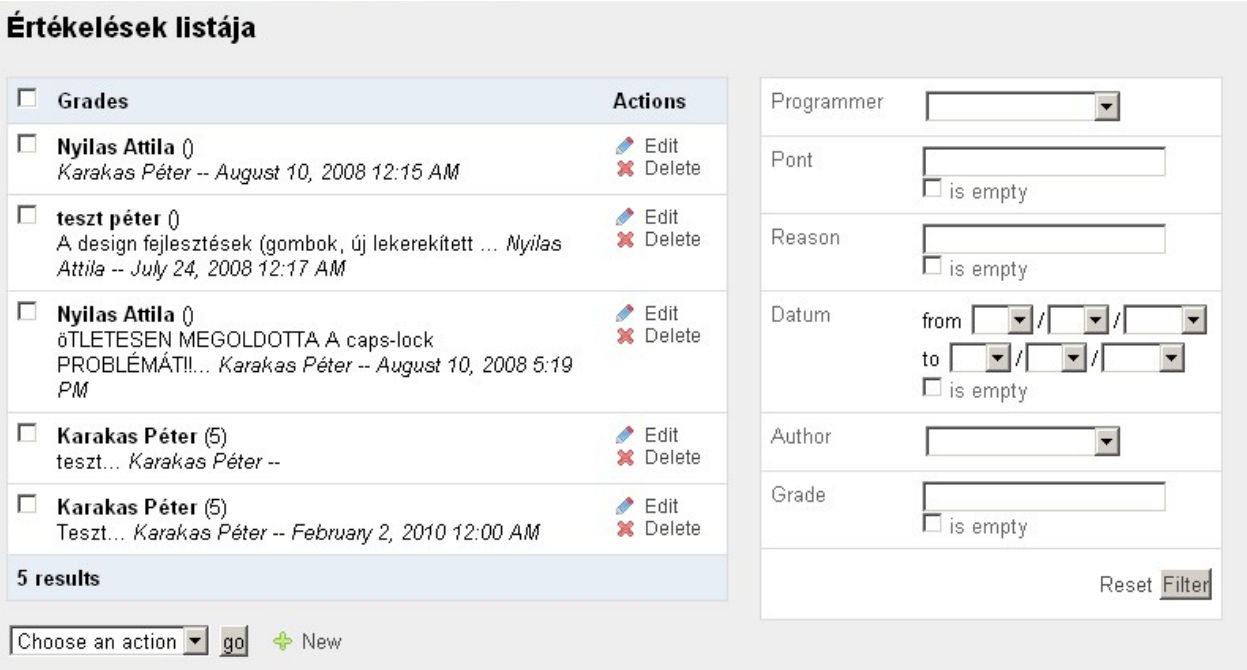
```
generator:
  class: sfDoctrineGenerator
  param:
    model_class:           Grade
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          grade
    with_doctrine_route:   true
    actions_base_class:    sfActions

    config:
      actions: ~
      fields:  ~
      list:
        title:   Értékelések listája
```

```
        display: [ Grades ]
        layout:  stacked
        params:  <b>%%Programmer%%</b> (%%grade%%)<br />%%Detail%%
<em>%%Author%% -- %%datum%%</em>
      filter:  ~
      form:    ~
      edit:
        title:   Értékelése szerkesztése
        fields:
            programmer_id: { params: size=5 }
            Author: { params: size=5 }
      new:     ~
```

Ez a YAML fájl definiálja, hogy melyik adatbázisbeli táblábát kell kezelni ebben a CRUD-ban. Minden módosítás nélkül is használható, mert a model_class-t magától beállítja a rendszer a generálás során. Így listázódnak az értékelések, adahtunk hozzá új értékelést és törölhetünk is. A fenti kód-ban 2 változás látható:

- Listázás
    - A title értelemszerűen a címet jelenti.
    - A display-ben lehet felsorolni, hogy a Grade tábla mely oszlopai jelenjenek meg. Itt a összes oszlopot listázzuk.
    - A layout: stacked hatására nem tesz minden értéket külön oszlopba hanem mindent összefűz egy sorba.
    - A params: Itt adjuk meg, hogy az egyes sorokban milyen szöveg jelenjen meg. Itt HTML elemeket használhatunk. A változó részeket %%-el kell határolni. Itt a Grade osztály adattagjai jelenhetnek meg, vagy get metódusai (get nélkül, pl: getDetail()).
- Edit
    - A title itt a from címét jelenti.
    - A fields paraméterben pedig a programmer_id és az Author mező szélessége van beállítva.

Az eredmény az alábbi képen látható felület.



Az összes értékelés listázva van egy lapozható tábla segítségével. Az alatta található 'new' gomb segítségével adhatunk hozzá új értékelést.
Az értékelések melletti edit gombbal szerkeszthetjük, a delete gombbal pedig törölhetünk egy bejegyzést.

Mindehhez alig kellett kódot írnunk!
A symfony helyettünk kigenerálta az ezekhez szükséges action-öket, nézeteket, formokat, filtereket és módosította a routing beállításokat.

Program dokumentáció

(Nyilas Attila)

A Manage 2 web-alkalmazás azért jött létre, hogy megkönnyítse egyrészről a Webiris kft. adminisztrációs feladatait mint programozók kezelése, értékelése, másrészről pedig a programozóknak nyújt segítséget saját adataik kezelésében. Emiatt a kettősség miatt az alkalmazás 2 részre

van osztva: az egyik a backend ami csak az adminisztrátorok számára elérhető, a másik a frontend ahova a programozók is beléphetnek.

A Frontend

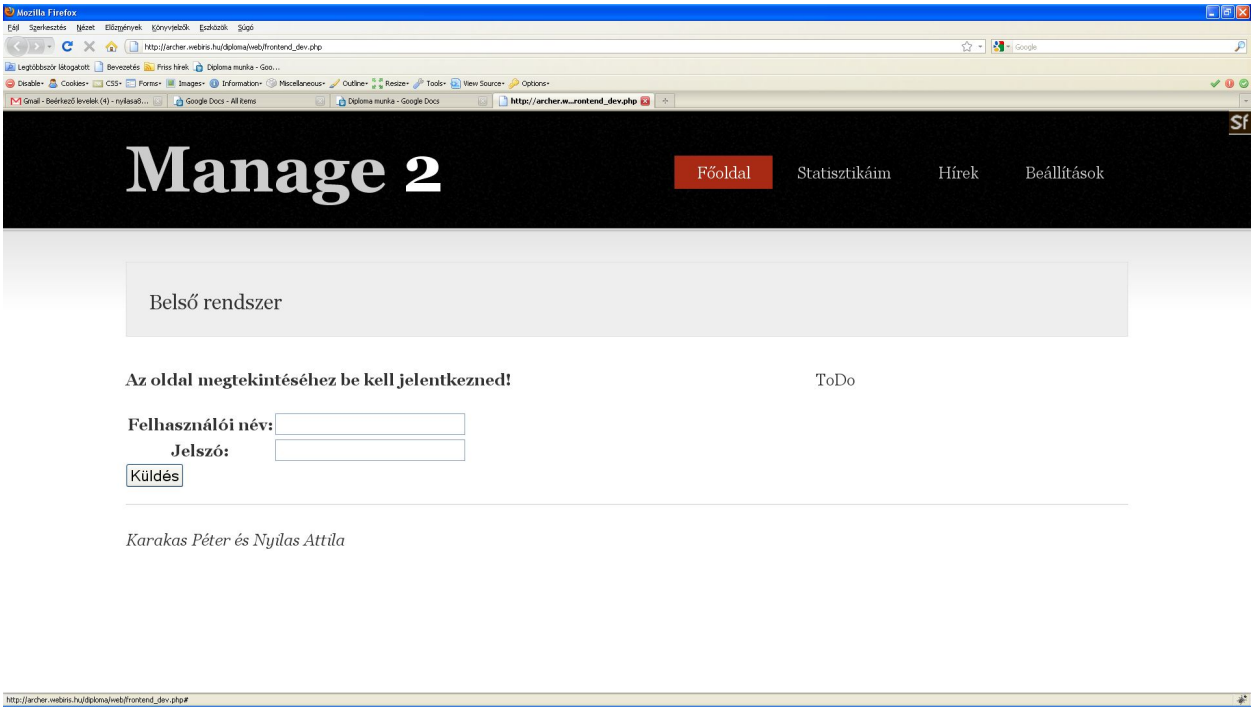
Regisztráció

Új programozó felvételénél egy regisztrációs emailt küld a rendszer amelyben található egy Url. Ez az Url tartalmaz egy, csak ezen programozó számára generált hash-t, amely segytségével hitelesítés nelkül eléri a regisztrációs oldalt. Ez a hash biztosítja, hogy más ne módosíthassa illetéktelenül a dolgozók adatait. Az emailben küldött hivatkozásra kattintva a regisztrációs felületre irányít át a rendszer.

A leendő munkatárs itt megadhatja minden szükséges adatát. Ez a fajta regisztrációs módszer, hogy vele töltetünk ki minden, jól kezeli a bizalmas adatokat. Ebben az esetben ez a jelszó. Itt ő maga adja meg a jelszavát amit a rendszer md5 kódolt formában ment le az adatbázisba. Így ezt rajta kívül senki nem tudhatja.
Ha megadta adatait akkor a rendszer felveszi őt a Gforge-ba és aktívvá is teszi. Ezzel már a Gforge is elérhetővé válik számára.
Utolsó lépésként a választott felhasználónév és jelszava alapján számára is létrejön a .htpasswd állományban egy bejegyzés. Ezzel a Manage 2 rendszerbe is be tud lépni.

Bejelentkezés

Az első oldal ami fogadja a felhasználót az a bejelentkező oldal.



Itt meg kell adnia felhasználónevét és jelszavát a hitelesítéshez. Sikeres belépés után egy üdvözlő képernyő fogadja, felül egy menüvel.

Menü

A menü az alábbi pontokat tartalmazza:

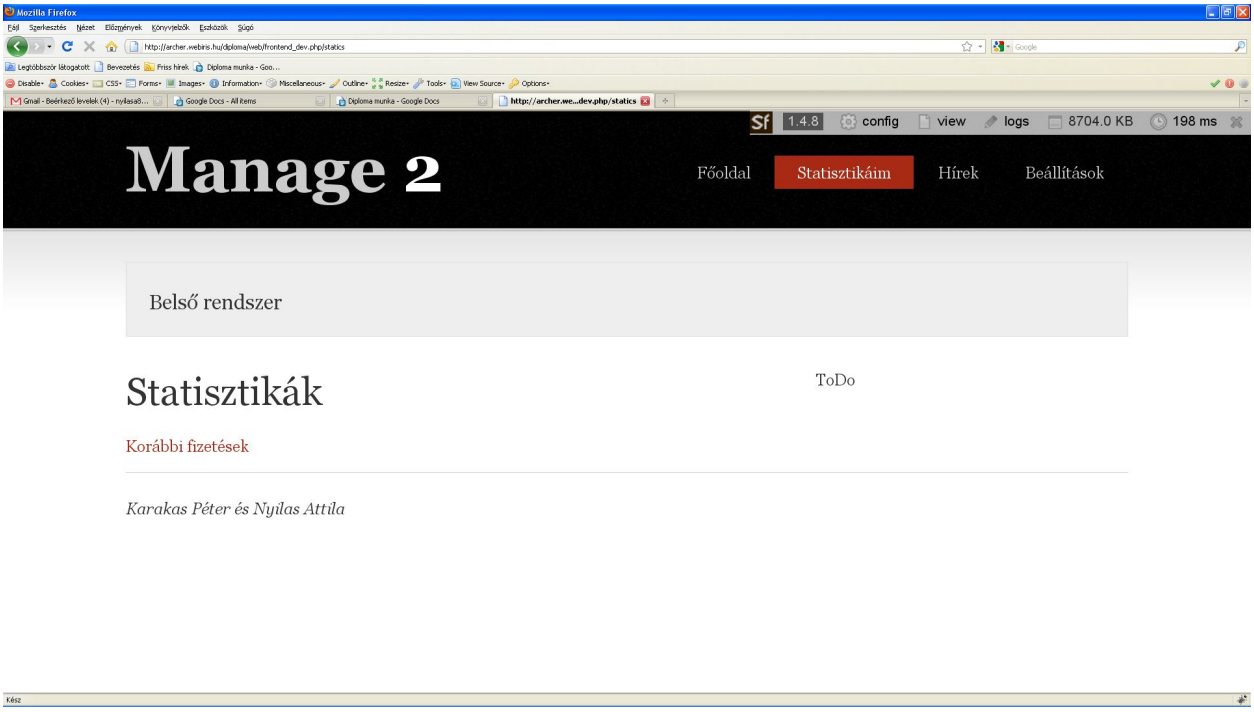
- Főoldal
- Statisztikáim
- Hírek
- Beállítások

Főoldal

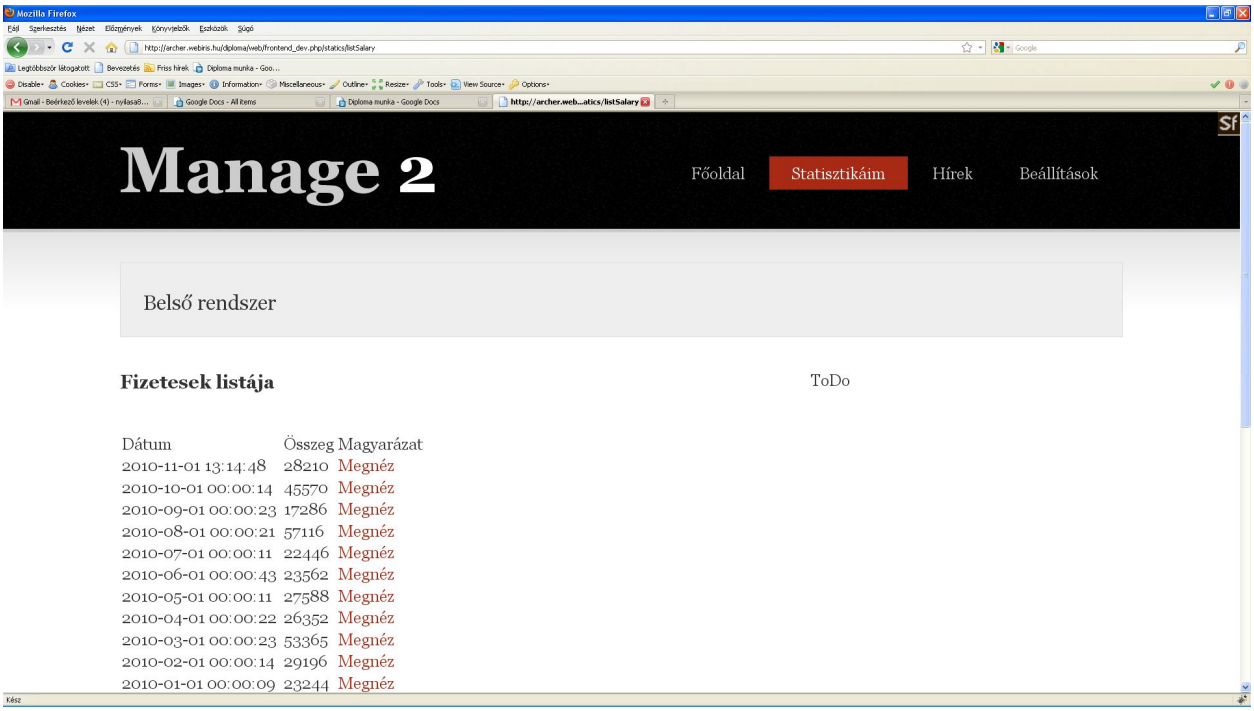
A főoldalra kattintva mindig az alkalmazás mindig visszatér az üfvözlő képernyőre.

Statisztikáim

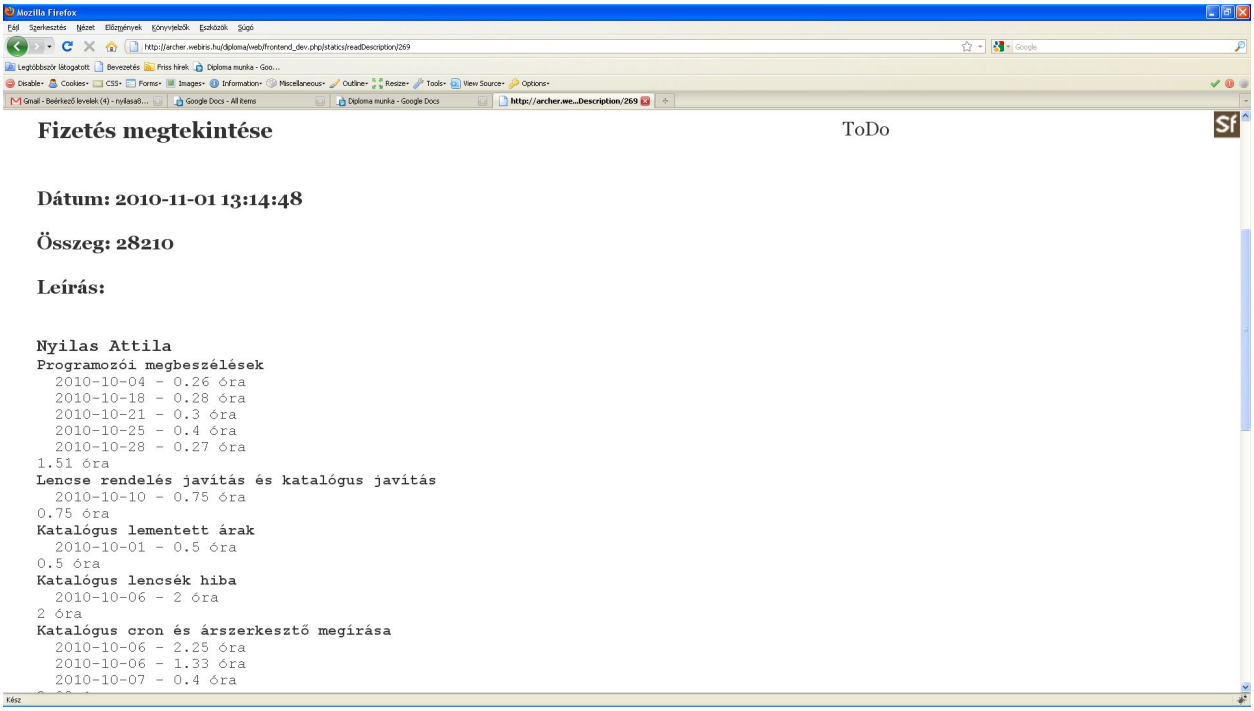
Ezen menüpont alatt a programozó a rá vonatkozó statisztikákat nézhet meg.



Jelen pillanatban a korábbi fizetéseit tekintheti meg.

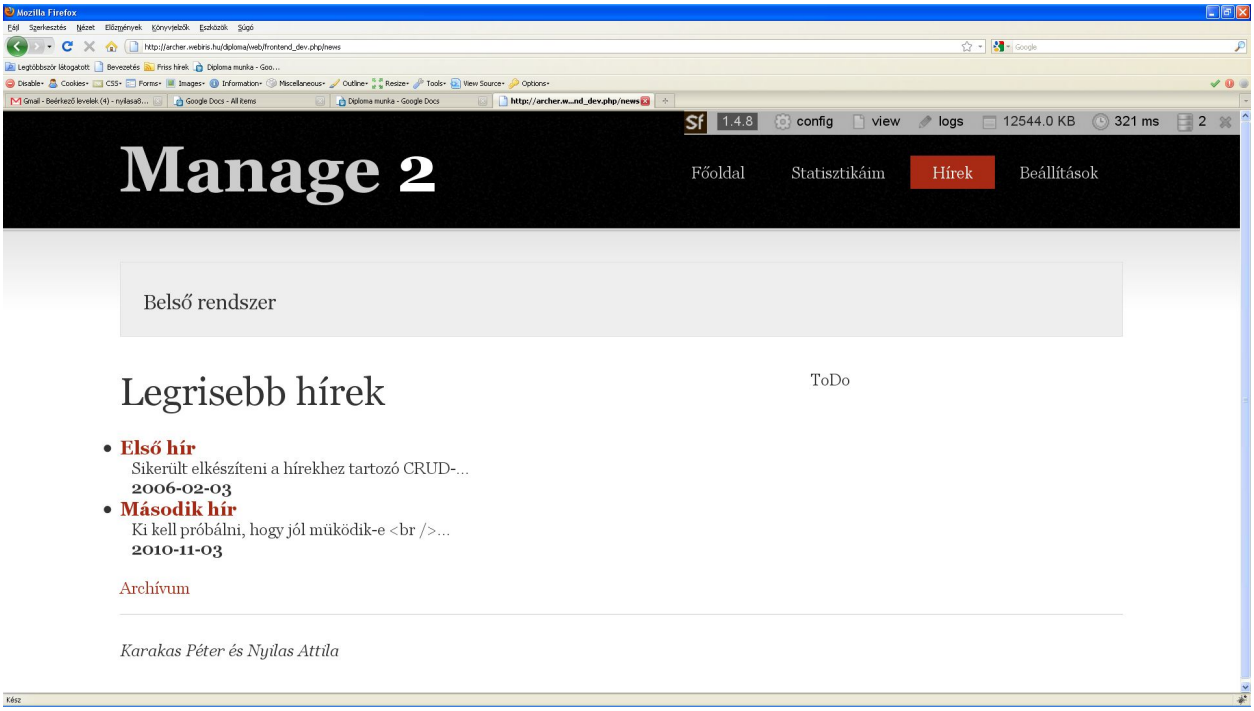


Itt táblázatosan jelenik meg dátum szerint csökkenő sorrendben az összeggel együtt és egy ’megnéz’ link-kel. Erre a linkre kattintva tételes elszámolást kap a ledolgozott óráiról.



Hírek

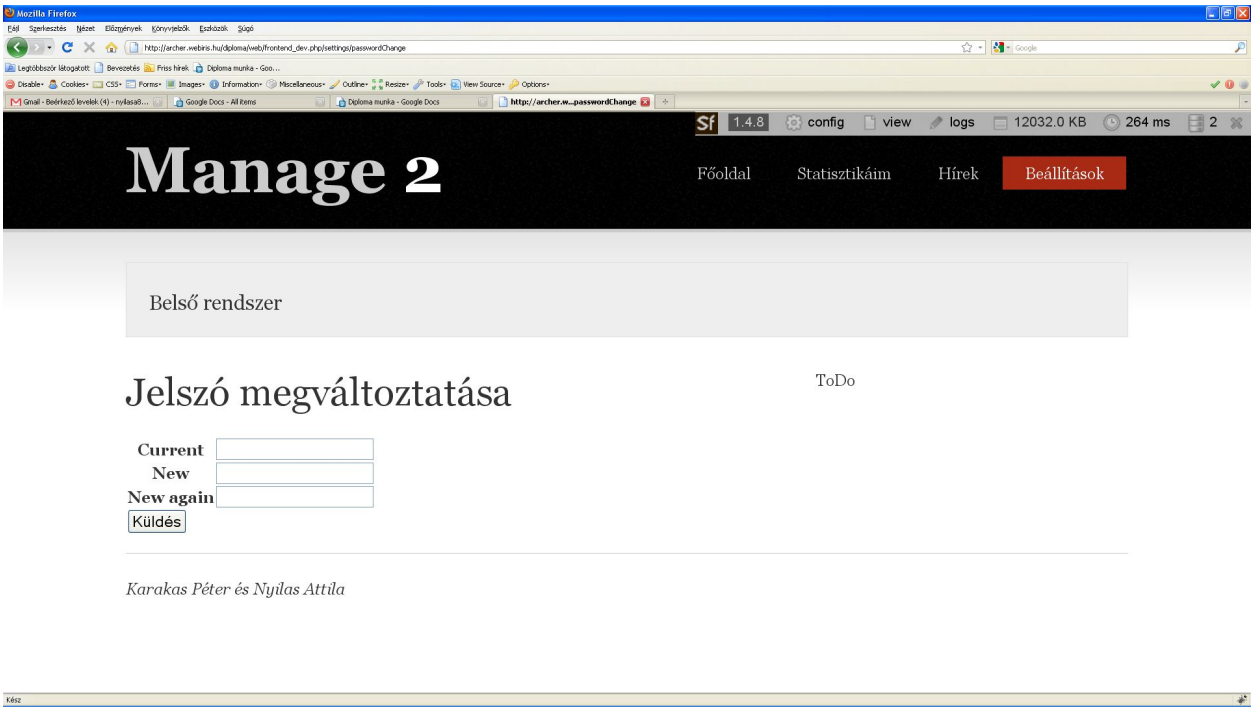
A hírek menüpont alatt az eddig olvasatlan hírek jelennek meg.



A hír címe alatt a tartalom első 50 karaktere jelenik meg. Amennyiben megnyitja valamelyik hírt, úgy a teljes tartalmát megtekintheti. Ebben a pillanatban a rendszer olvasottnak állítja be, így a hírek menüpont alatt legközelebb már nem fog megjelenni. Az olvasott hírek megtekintésére szolgál az Archívum. Ennek felépítése megegyezik az olvasatlanéval, de itt az olvasás hatására természetesen nem történik semmi, továbbra is az archívumban marad.

Beállítások

A programozó itt módosíthatja a jelszavát, mégpedig a régi és az új kétszeri megadásával.

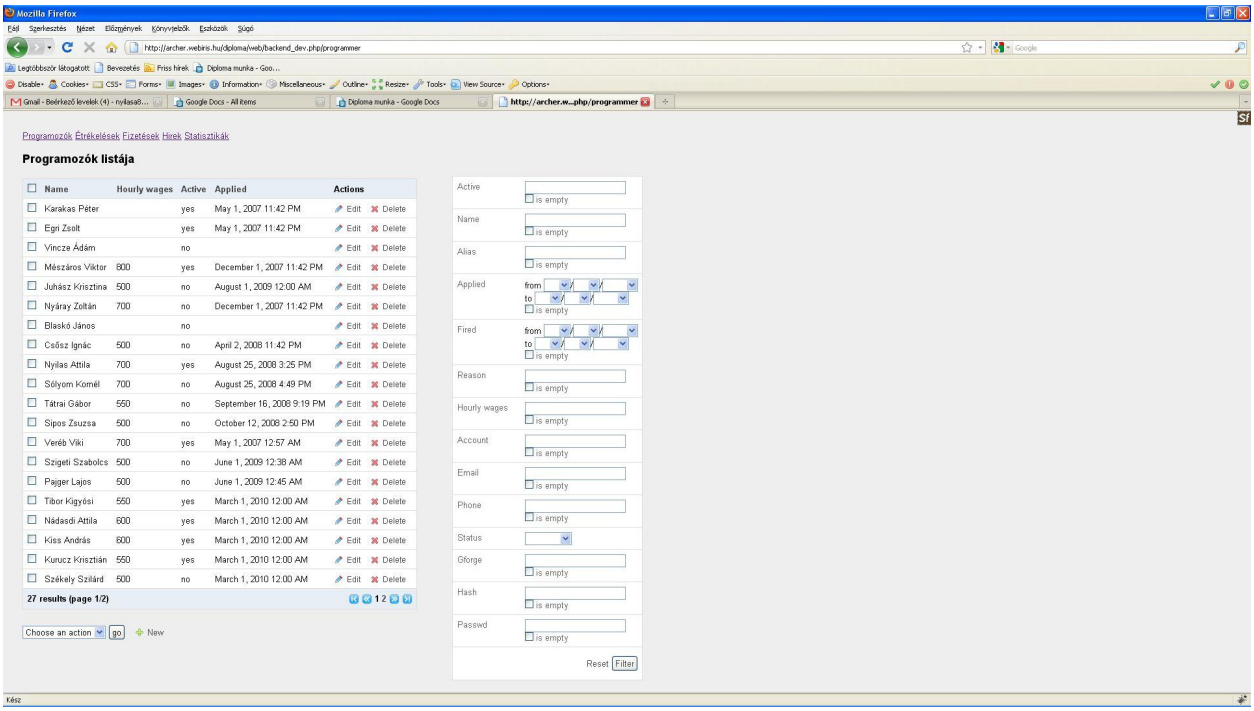


A Backend

Az admin felületen is egy üdvözlő képernyő fogadja a felhasználót. Itt is egy menü segíti a navigációt.
Menü:

- Programozók
- Értékelések
- Fizetések
- Hírek
- Statisztikák

Programozók



A programozók felület bal oldalán táblázatosan jelennek meg a programozók, legfontosabb adataikkal.

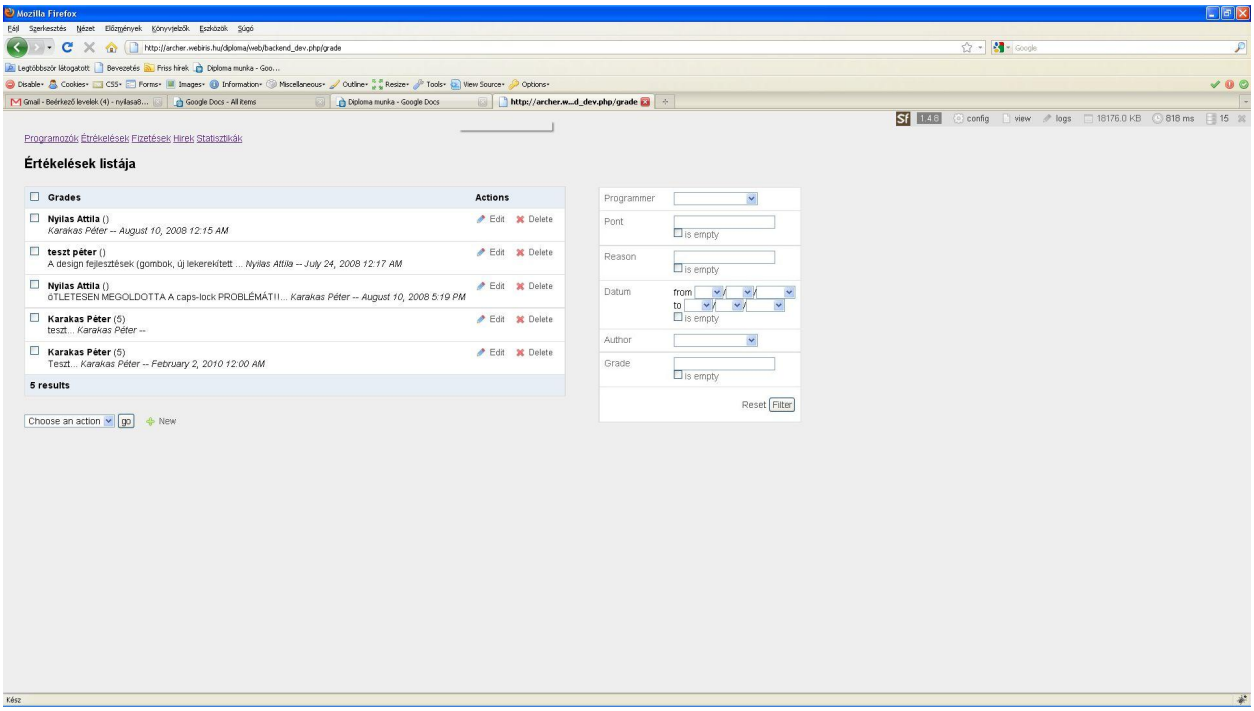
Megjelenik mindegyik mellett egy edit gomb amellyel módosítani lehet egy programozót, illetve egy delete amellyel törölni.

A táblázat alján található a new gomb amellyel új programozót lehet hozzáadni. Itt meg kell adni az új programozó nevét, email címét és órabérét. A felvesz gombra kattintva a rendszer egy email-t küld a megadott címre. Ez az email tartalmazza azt url-t amelyre kattintva az új dolgozó elvégezheti a regisztrációt. Ez az url egy generált hash-t tartalmaz, hogy kizárólag ő tudjon hozzáférni a rendszerhez addig amíg nem végezte el a regisztrációt.

A felület jobb oldalán pedig egy szűrő található, ahol a listát szűkíthetjük különböző feltételekkel.

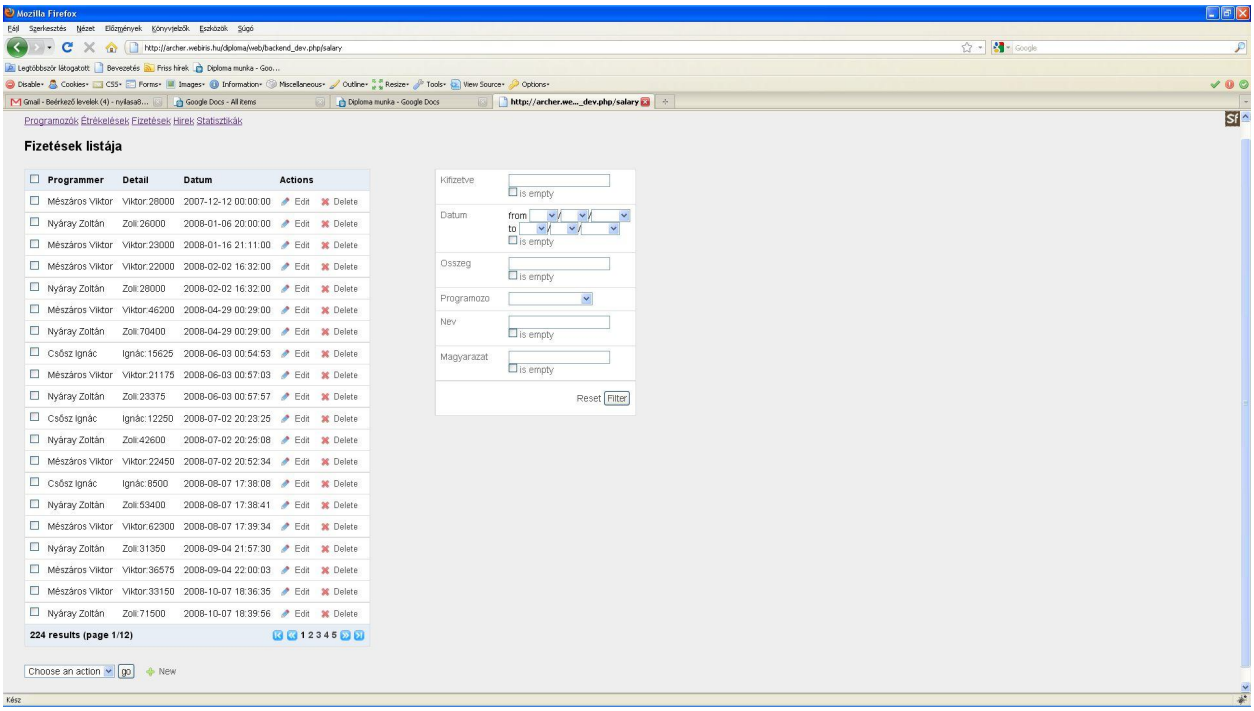
Értékelések

Ezen fül alatt a programozókról készíthetünk szöveges értékelést.



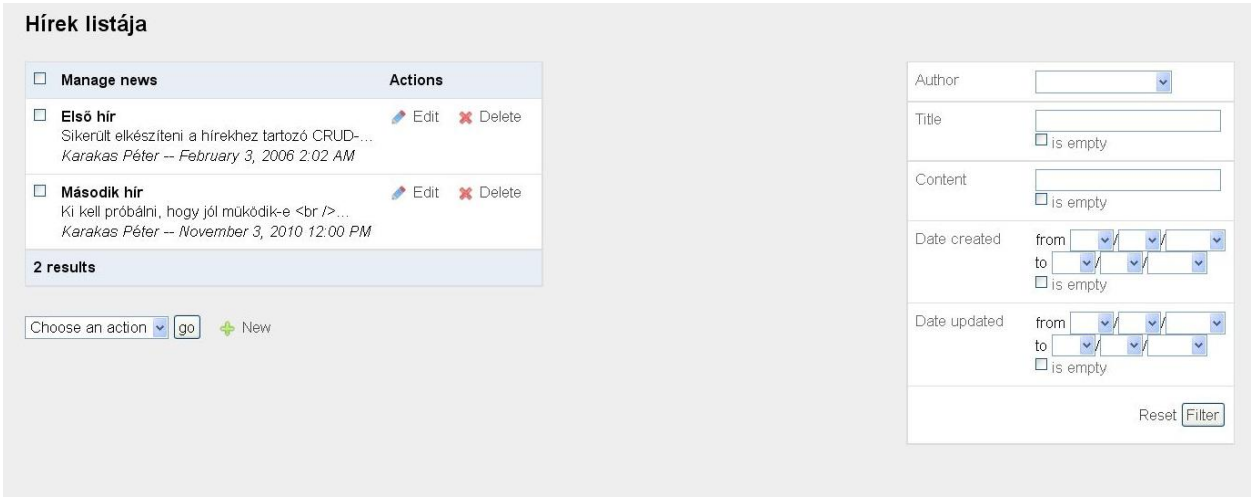
Fizetések

A rendszer által kigenerált fizetések jelennek meg táblázatos formában. Itt is lehet egy bejegyzést szerkeszteni és törölni is. Új hozzáadására is van lehetőség, ha valamiért a generáló folyamat nem működne.



Hírek

Itt jelennek meg a hírek. Szerkeszthetjük őket, adhatunk hozzá újat és törölhetünk is.



Statisztikák

A statisztikához szükséges adatokat a gforge rendszerből kérdezi le. A gforge PostgreSQL-t használ, de ezt a Doctrine teljesen elfedi a programozó elől. A dinamikus táblákat és dátum választó mezőt a JQuery JavaScript-es keretrendszer segítségével jelenítettük meg.

A grafikonok megjelenítését a Fusion Chart nevű eszközzel lehetséges. Ez csupán egy XML-t vár a megjelenítendő adatokról.

Ledolgozott órák

Itt lehet megtekinteni a programozók munka óráit heti vagy havi bontásban. Grafikonon illetve táblázatos formában jelenik meg. A grafikonon a kiválasztott programozók jelennek meg. A dátum megadásával kiválasztható, hogy az adott hét vagy az adott hónap statisztikája jelenjen meg. Például, ha dátumnak 2010-10-26 adunk meg és heti bontást akkor 2010-10-25-től 2010-10-31-ig jeleníti meg a ledolgozott órákat.

Ez a felület rendkívül hasznos megbeszélések alatt, amikor visszamenőleg megnézzük az előző heti teljesítményt.

Nyilas Attila

Show 10 entries Search:

Ticket	Óraszám	Dátum
Margó készletáramlás	1	2010-08-16
Margó készletáramlás	1.5	2010-08-19
Margó készletáramlás	1.5	2010-08-21
Margó készletáramlás	0.75	2010-08-22
Programozói megbeszélések	1	2010-08-16
Programozói megbeszélések	0.33	2010-08-19
Ár és dátum kiirás kimutatásokban	1.5	2010-08-20
Ár és dátum kiirás kimutatásokban	0.5	2010-08-21
Ár és dátum kiirás kimutatásokban	0.75	2010-08-22
Értékesités modul tesztelés	2.4	2010-08-22
Ticket	**Óraszám**	**Dátum**

Showing 1 to 10 of 10 entries

Összesen: 11.23 óra.

Nádasdi Attila

Show 10 entries Search:

Ticket	Óraszám	Dátum
_ar_kiir-t lecserélni	1.5	2010-08-21
Beszerzés többmyenvűsítése	2	2010-08-17
Dolgozó segítség	2	2010-08-19
Dátumok többnyelvűsítése	1.6	2010-08-16
Leltár többnyelvűsítése	1.5	2010-08-17
Programozói megbeszélések	1	2010-08-16
Programozói megbeszélések	0.67	2010-08-17
Programozói megbeszélések	0.84	2010-08-19
Raktárkezelés többnyelvűsítése	1.7	2010-08-18
Raktárkezelés többnyelvűsítése	2	2010-08-19
Ticket	**Óraszám**	**Dátum**

Showing 1 to 10 of 12 entries

Összesen: 17.31 óra.

Monte Carlo

A Monte Carlo szimulációt mérnökök, fizikusok előszeretettel használják. Segítségével meghatározható a görbe alatti terület, vagy egy test tömegközéppontja. Azonban mi sprint időbecslésére használjuk.

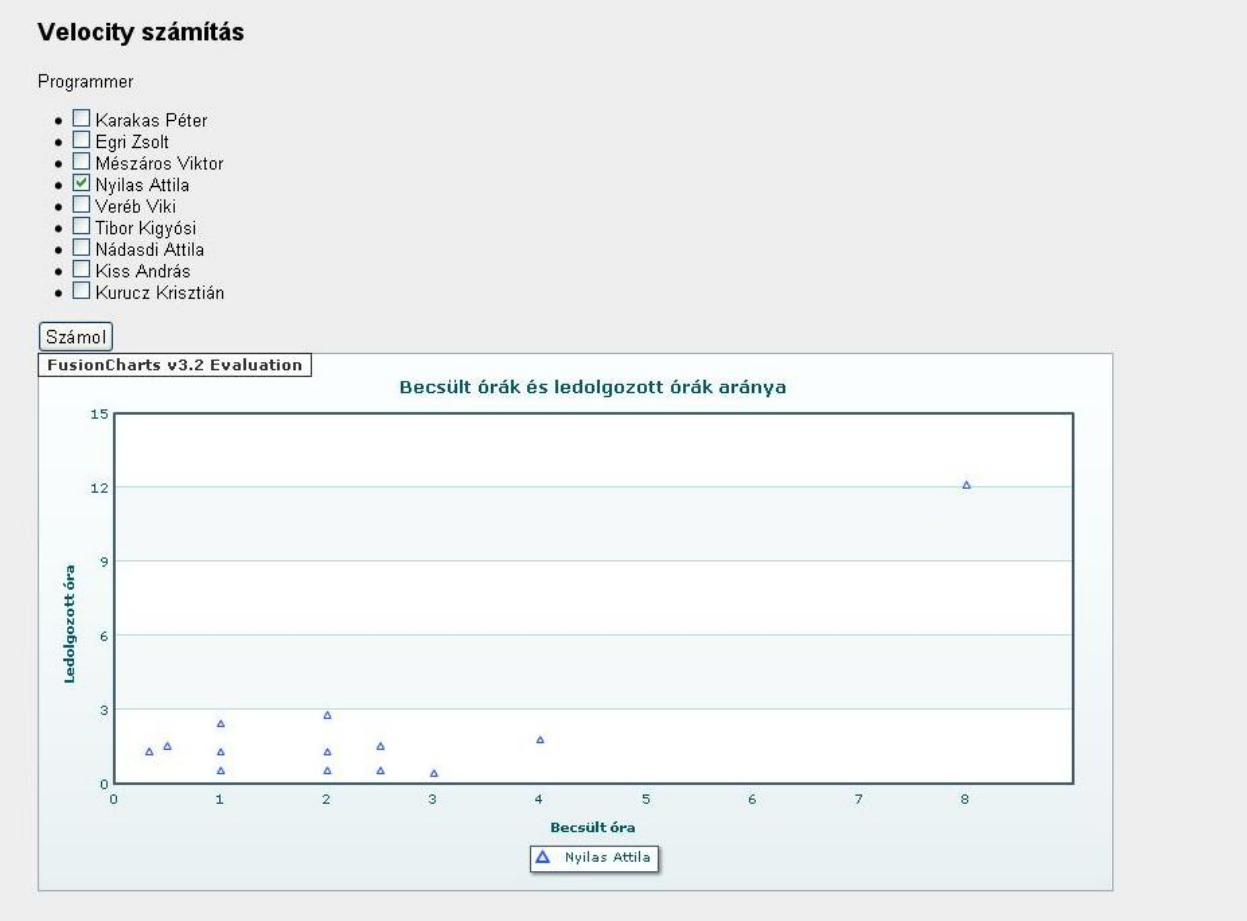
A szimuláció lényege, hogy megadjuk, hogy egy-egy programozó hány órára becsüli a rábízott feladatát. A példán Nyilas Attila 10 óra, Kígyósi Tibor 15 óra. A korábbi becslésekre alapozva megadott számú szimulációt készítünk (a korábbi becslések a gforge rendszerben vannak, minden hibajegyet meg kell becsülnie a programozónak, hogy hány órába fog kerülni). Ezeket nevezzük velocity-nek. Egy programozóhoz több velocity érték is tartozik. Ezekből véletlenszerűen választ ki egy-egy értéket a szimuláció során. Így kapunk egy lehetséges óraszámot, amikor elkészül a sprint. Minél több szimulációt futtattunk annál pontosabb értéket kaphatunk.

A grafikon a szükséges óraszámok valószínűségét mutatja. A felosztás finomítható a "Felosztás részletessége" értékkel. Ez adja, meg hogy a grafikonon az időintervallumok hány órásak legyenek.

Programozót hozzáadni a "+" jellel elvenni pedig a "-" jellel lehet. Az "X" törli az összes felvett programozót.

Velocity

A kiválasztott programozóknak jeleníti meg a velocity-ét. Ez hibajegyenként mutatja, hogy mennyi votl a becsült és mennyi volt a ledolgozott óraszám. Az egyes programozókhoz tartozó értékpárokat egyedi jellel jelöli. Ezt egy véletlen szám generátor dönti el, hogy milyen színű és milyen formájú legyen. A képen kék háromszögek. Ha ezekre rávisszük az egeret, akkor további információkat tudhatunk meg (velocity értéke, hibajegy neve).



Felhasznált irodalom

Internetes tartalmak:

1. http://www.tavmunkainfo.hu/miatavmunka.htm - 2010.11.01
2. http://www.tavmunkainfo.hu/EU99.htm - 2010.11.02

3. *http://www.tavmunkainfo.hu/elonyok.htm#6 - 2010.11.04*
4. *http://www.tavmunkainfo.hu/hatranyok.htm - 2010.11.04*
5. *http://hu.wikipedia.org/wiki/Ciklomatikus_komplexit%C3%A1s - 2010.10.25*
6. *Veráb Viktória szakdolgozata: A jövő munkamódszere: a távmunka. 31 oldal*
7. *http://www.joelonsoftware.com/items/2007/10/26.html - 2010.10.20*
8. *http://ikon.inf.elte.hu/wiki/index.php?title=MVC - 2010.11.07*
9. *(http://hu.wikipedia.org/wiki/Modell-n%C3%A9zet-vez%C3%A9rl%C5%91* - 2010-11-07
10. Scrum: ebben a fejezetben a forrásként megjelölt oldalon található cikket használtuk fel. Itt főleg a megfogalmazáson módosítottunk, de túlnyomó része megegyezik a cikkével. Azért választottuk az idézést, mert ez egy nagyon jól összeszedett és megírt cikk, így jobban be tudja mutatni ezt a témát, mintha saját szavainkkal írtuk volna le. Forrás: *http://webisztan.blog.hu/2010/05/17/megertik_es_atelik_a_munkajukat_a_scrum_modszerrol_1_resz* - 2010-10-29
11. Extrém programozás*:* hasonlóan a scrum-hoz ez a fejezet is lefedi a forráskánt megadott cikket, az előző bejegyzésben megadott okok miatt.
    http://www.valodi.hu/agile - 2010-11-03
12. *http://www.symfony-project.org/gentle-introduction/1_4/en/* - 2010-10-10
13. *http://www.doctrine-project.org/projects/orm/1.2/docs/manual/en - 2010-10-19*
14. *http://www.slideshare.net/rob_knight/object-relational-mapping-in-php - 2010-10-24*