

**Budapesti Műszaki és Gazdaságtudományi Egyetem**Villamosmérnöki és Informatikai Kar  
Méréstechnika és Információs Rendszerek Tanszék

Rendszertervezés és -integráció  
Házi feladat

**Verziókövető és konfigurációmenedzsment eszközök összehasonlítása**

*Gungl Szilárd (BNVMXD)*

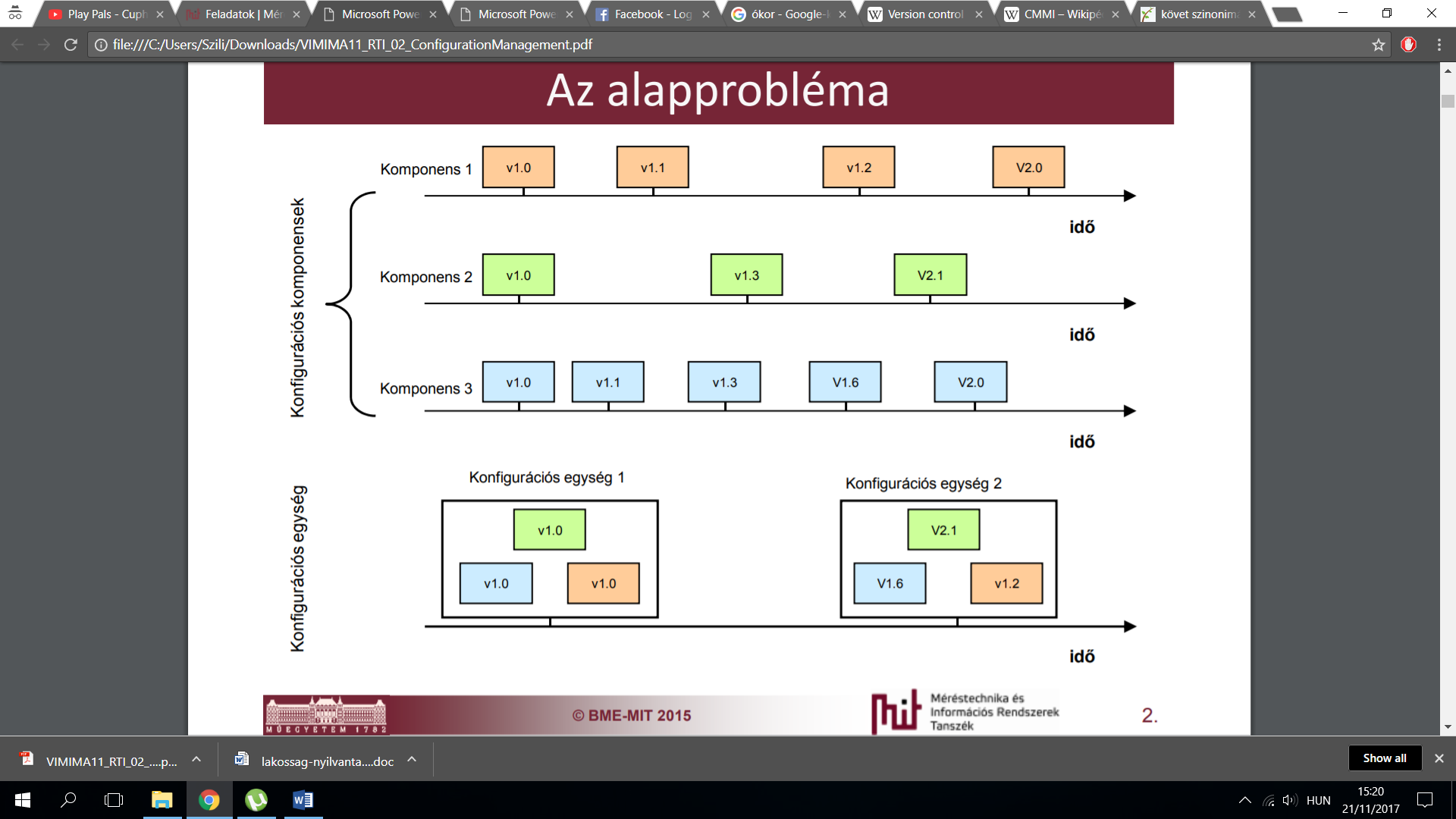
1. Bevezetés

A verziókövetés igénye lényegében már az írás létrejötte óta jelen van. Gondoljunk csak bele: ha például az ókorban módosítani szerettek volna egy törvénykönyvet, vagy valamilyen nyilvántartást, akkor az egységesség érdekében ugyanilyen módon kellet, hogy megváltoztassák a közigazgatási egységek számára kiküldött másolatokat is.

A különböző verziók egységes kezelése a közelmúltban vált kiemelt fontosságúvá. A számítástechnika fejlődésével egyre összetettebb, nagy komplexitású rendszerek jöttek létre, melyek fejlesztését egyre nehezebbé vált átlátni. A helyzetet tovább bonyolítja, hogy az egyes projekteken nagy valószínűséggel többen is dolgoznak egyszerre. A csapat tagjainak a munkáját össze kell szinkronizálni, illetve a fejlesztési folyamatnak nyomon követhetőnek kell lennie, az alkalmazott változtatásokat naplózni kell, hogy később vissza tudjuk keresni.

Ezen igények alapján születtek meg a verzió- vagy konfiguráció menedzsment rendszerek, melyek igyekeznek különböző módszerekkel megoldást nyújtani a fenti problémákra. Ezeket a rendszereket napjainkban elsősorban a számítógépes fejlesztések – tágabb értelembe vett szoftverfejlesztés – során alkalmazzuk.

* 1. Az alapprobléma



1. ábra Egy fejlesztési folyamat időbeli lefolyása

Amint azt az ábra mutatja, egy projekt során az egyes komponensek fejlesztési sebessége eltérő lehet, illetve nem is biztos, hogy egy adott időpontban az éppen legfrissebb elemek kompatibilisek egymással, ezért is kell az egyes verziókat tárolni és elérhetővé tenni. Az egymással működő részegységek halmazát nevezzük konfigurációs egységnek.

A konfigurációs komponensek nem szükségszerűen az elkészült szoftver részei, ez alatt értünk minden olyan eszközt, programot, terméket, mely a szoftver előállításához, leírásához, kezeléséhez szükséges.

Hogy ezeket a komponenseket mennyire „jól” – legalábbis előre jól meghatározottan – kezeljük, arra ad mérőszámot a CMMI modell, mely 5 érettségi szintet definiál:

* *Initial*: kezdeti érettségi szint. Megjósolhatatlan, reaktív működés, a munka elkészül, de gyakran csúszások és anyagi keret túllépés jellemzi.
* *Managed*: projekt szinten szervezett. Tervezés, végrehajtás, mérés és kontroll jellemző.
* *Defined*: előrelátóan cselekvő. Vállalat szintű standardok.
* *Quantitatively Managed*: mérés és ennek megfelelő beavatkozás. Az adatok alapján igyekeznek fejleszteni a problémás részeket.
* *Optimizing*: biztos, flexibilis. Folyamatos fejlődés és javítás a cél. Változás a környezet igényeinek megfelelően.

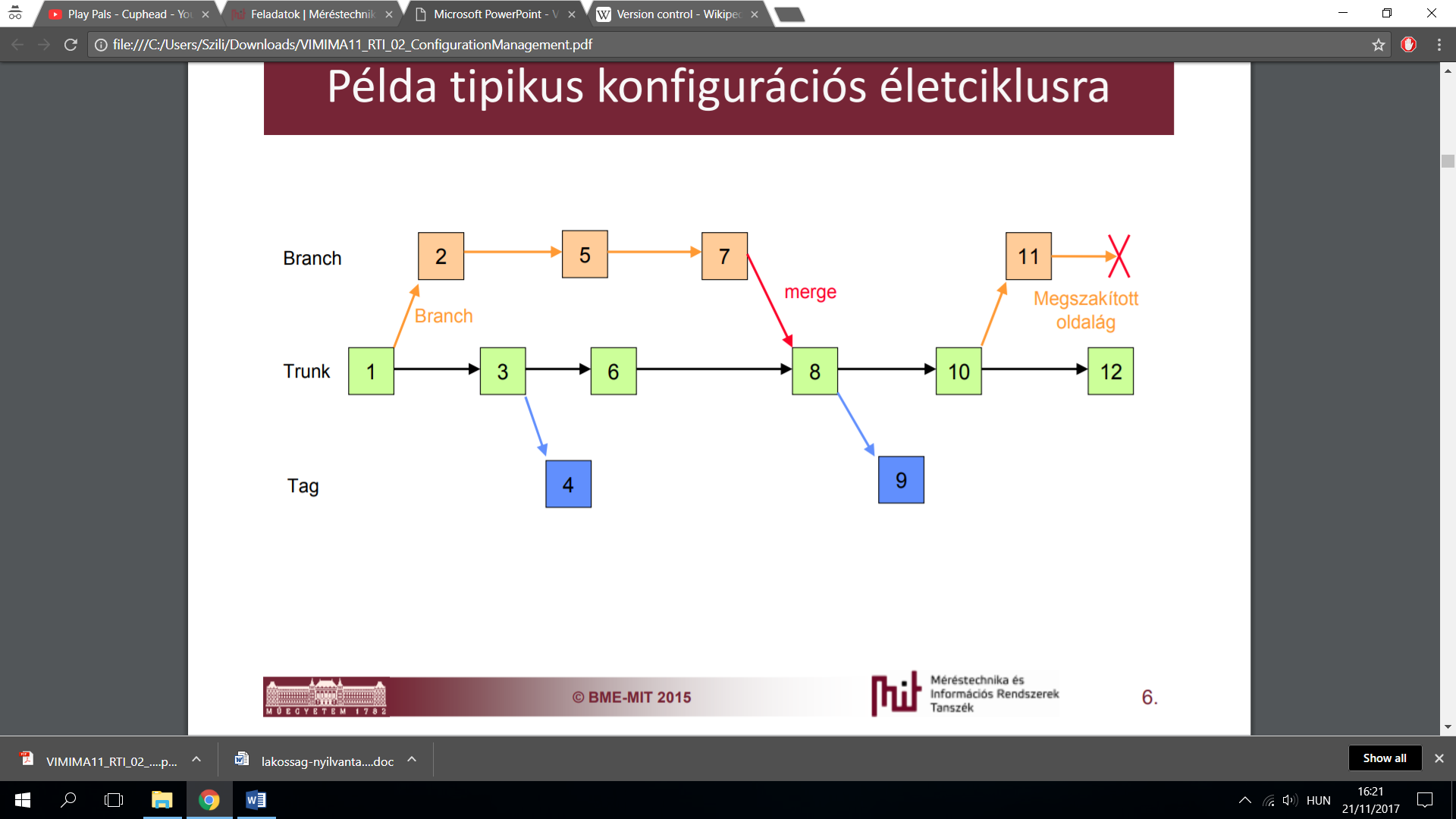
Ezek a szintek leírják, hogy az adott cégre jellemző fejlesztési folyamat mennyire meghatározott, előre definiált lépések szerint történik. A modell ajánlásait követve fejleszteni tudjuk a fejlesztési folyamatot magát, mely a képességek jobb kihasználtságát eredményezi, illetve felfedi a fejlesztési folyamat gyengepontjait, melyek optimalizálhatók.

1. Konfigurációmenedzsment rendszerek

Egyszerűbb esetekben jellemzően egy verziókövető rendszert jelentenek, tehát a konfigurációs komponensek közül a készülő szoftver van a fókuszpontban. Ezen rendszerek legfőbb funkciói:

* A verziókövetés alá helyezett minden állományt, és minden korábbi verziót is vissza lehet keresni.
* Egy állomány két változata közötti eltéréseket meg tudja jeleníteni.
* Kezeli az egyidejű hozzáférést, tehát amikor két személy egyszerre szeretné elérni és módosítani az állományt.
* Elágazásokat hozhatunk létre, ha a fejlesztés két különböző irányba indul el. Emellett, ha például egy fejlesztés alatt levő termékből kiadunk egy verziót, és később ezt vissza szeretnénk keresni, akkor rendelkezésre áll az összes kiadáskori állomány.
  1. Verziókezelés

Ezeknek a rendszereknek az egyik legfőbb feladata az egyes verziók tárolása, illetve a verziók közti változtatások nyilvántartása.



2. ábra Jellemző fejlesztési út

Egy általános fejlesztési folyamatot láthatunk a fenti ábrán. Középen zöld színű vonallal látható a fő irány, avagy trunk (törzs), mely a specifikációban meghatározott irányelveknek megfelelő fejlesztést jelenti a kezdeti verziótól egészen a végtermékig. Narancssárgával jelzett vonal az úgynevezett branch (ág), mely a trunk-tól valamilyen tekintetben jelentősen eltér. Ez általában egy új funkció implementálását jelenti, mely jelentősen megváltoztatja a szoftvert, és így a trunk integritásának megtartása érdekében egy külön ágat nyitnak neki a fejlesztés során. Ezt a különálló ágat aztán tovább fejleszthetik, majd vagy valamilyen módon beépítik a fő verzióba (ez rengeteg munkával járthat hiszen a két ág a párhuzamos fejlesztés során egyre „messzebb kerül egymástól”), vagy ha nem járnak sikerrel a fejlesztés során, az ágat meg is szakíthatják. Az ábra harmadik eleme a tag (címke, elem), mely egy olyan verziója a trunk-nak vagy branch-nek, melyet valamilyen okból meg szeretnénk őrizni, vagy ki szeretnénk tüntetni. Ez az ok általában az, hogy egy release verziót akarunk kiadni (alfa, béta, RC, RTM), tehát a külvilág számára szeretnénk egy működő verziót közzétenni. Ebben a működő verzióban az egyes komponensek azon verzióit rakjuk bele, melyek működőképesek egymással, tehát nem szükségszerűen a legújabb, épen aktuális verziót.

Az egyes verziók között egy „changelog” leírásban adhatjuk meg az általunk végrehajtott változtatásokat, illetve a rendszer maga is meg tudja jeleníteni ezeket könnyen átlátható formában.

* 1. Hozzáférések kezelése

A másik fő feladata a verziókövető rendszereknek, hogy kezeljék az egyidejű hozzáférést. Ez a helyzet akkor áll elő, amikor két (vagy több) személy egyszerre szeretné megnyitni és módosítani az adott állományt. Az egyidejű olvasás komolyabb gondot nem jelente, viszont, ha mindkét személy változtat valamit a fájlon, akkor ők nem fognak tudni egymás változtatásairól, így az állományok mentése és felülírása után az egyik részvevő munkája nem fog belekerülni az új verzióba (bár elveszni azért nem fog, de nem kerülnek bele egységesen az új verzióba).

A verziókövető rendszerek erre a problémára is megoldást jelentenek, lehetővé teszik, hogy a felhasználók együtt dolgozzanak, de mégse lépjenek egymás lábára. Két meghatározó szemlélet létezik az egyidejű hozzáférések kezelésére.

* + 1. A Lock–Modify–Unlock megközelítés

Ennek a módszernek a lényege, hogy ha valaki módosítani szeretne egy állományt, akkor azt előtte virtuálisan le kell zárnia, ami lehetővé teszi, hogy egyszerre csak egy ember legyen képes módosítani a fájlt (viszont olvasni tudják eközben többen is). Ezáltal ki van zárva az egyidejű hozzáféréslehetősége. A módosítások elmentését követően pedig a munkát végző személynek kell feloldania a zárat.

Ez a megoldás több szempontból is aggályos. Egyrészt adminisztratív problémákhoz vezet, ha elfelejtjük feloldani a zárat az állományon. Emellett felesleges várakozást okoz, hiszen a fájlon egyszerre csak egy ember tud dolgozni. Fennállhat olyan eset is, hogy két fejlesztő két, egymástól teljesen független részt szeretne módosítani – például két függvényt – viszont azt egyidőben nem tudják megtenni. Ennek a komplementere is elképzelhető: attól még, hogy egy fájlt egyszerre csak egy ember módosíthat, a fájlok közti függőségek fennállása miatt lehetséges, hogy valaki másnak egy másik fájlban végzett módosítása okoz jelentős gondokat.

* + 1. A Copy–Modify–Merge megközelítés

Az előző megoldás legnagyobb gyengesége, hogy nem veszi figyelemben a fájlon belüli, vagy azok közötti függőségeket. A Copy–Modify–Merge megközelítés szerint mindenki, aki használni szeretné a fájlt, kap egy úgynevezett working copy másolatot, melyet tud olvasni, illetve szükség esetén módosítani. Ha időközben átírták az eredeti fájlt (melynek másolatán dolgozunk), akkor a végrehajtott változtatásokról kapunk információt és ezen információk alapján kell összegyúrnunk (merge) az általunk és mások által végzett módosításokat. Tehát az esetlegesen létrejövő konfliktusok kezelése emberi döntéseket igényel, noha a verziókövető rendszer nagyfokú támogatást nyújt hozzá.

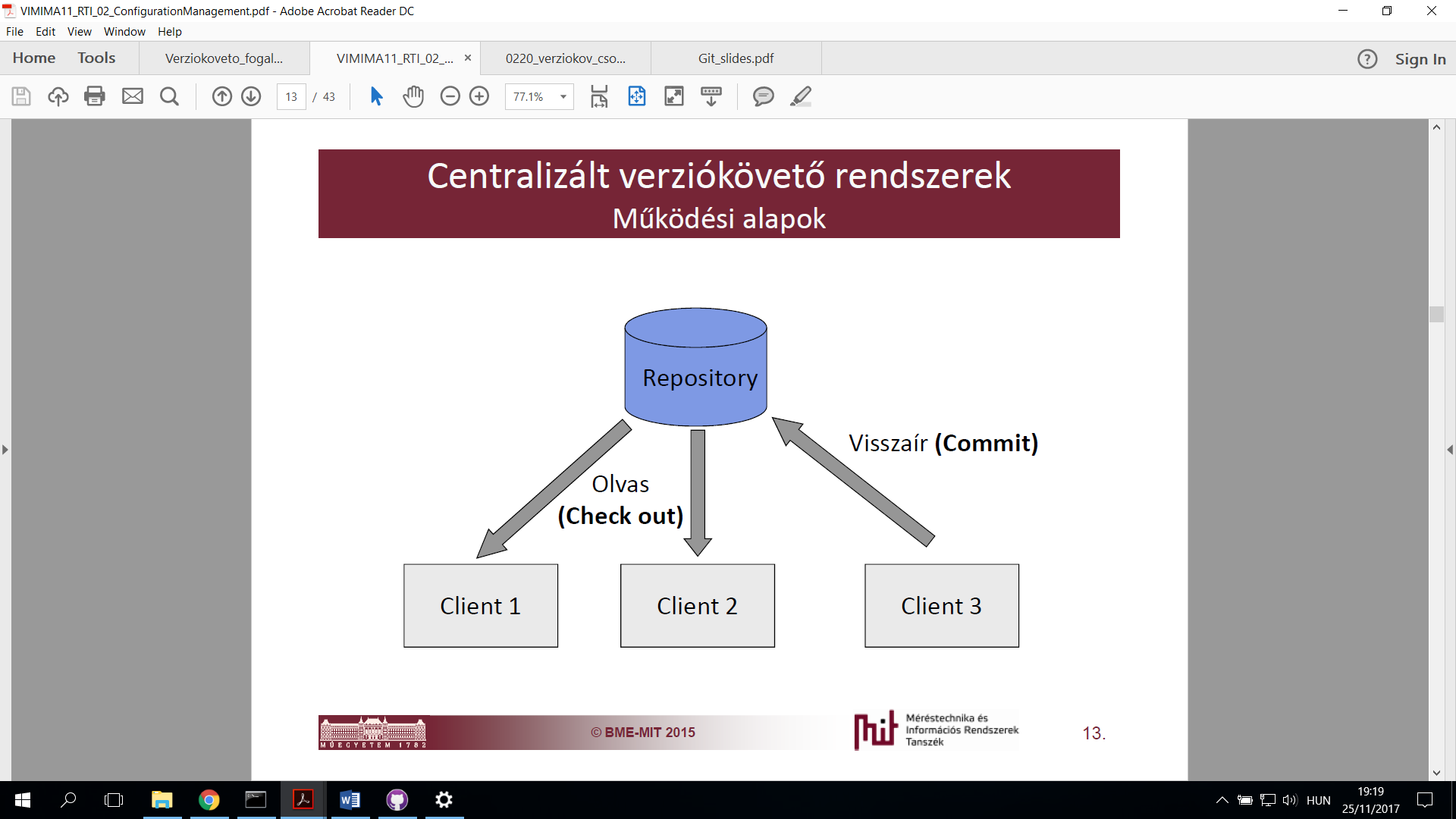
Ezzel a szemlélettel egyszerre többen is dolgozhatnak ugyanazon az állományon, és mivel a konfliktusok létrejöttére számítunk, így azok a mentésnél általában ki is derülnek. Ugyanakkor emberi döntések szükségesek ezek feloldásához, így ez magában hordozza hiba lehetőségét is.

Bizonyos fájloknál (pl.: bináris jellegű, hangfájlok…) a struktúra nem engedi meg a merge használatát, így ezek esetében továbbra is a lezárás-feloldás funkciót kell alkalmaznunk. Ez okozza, hogy a legtöbb verziókövető rendszerben megtalálható minkét funkció.

* 1. A struktúra

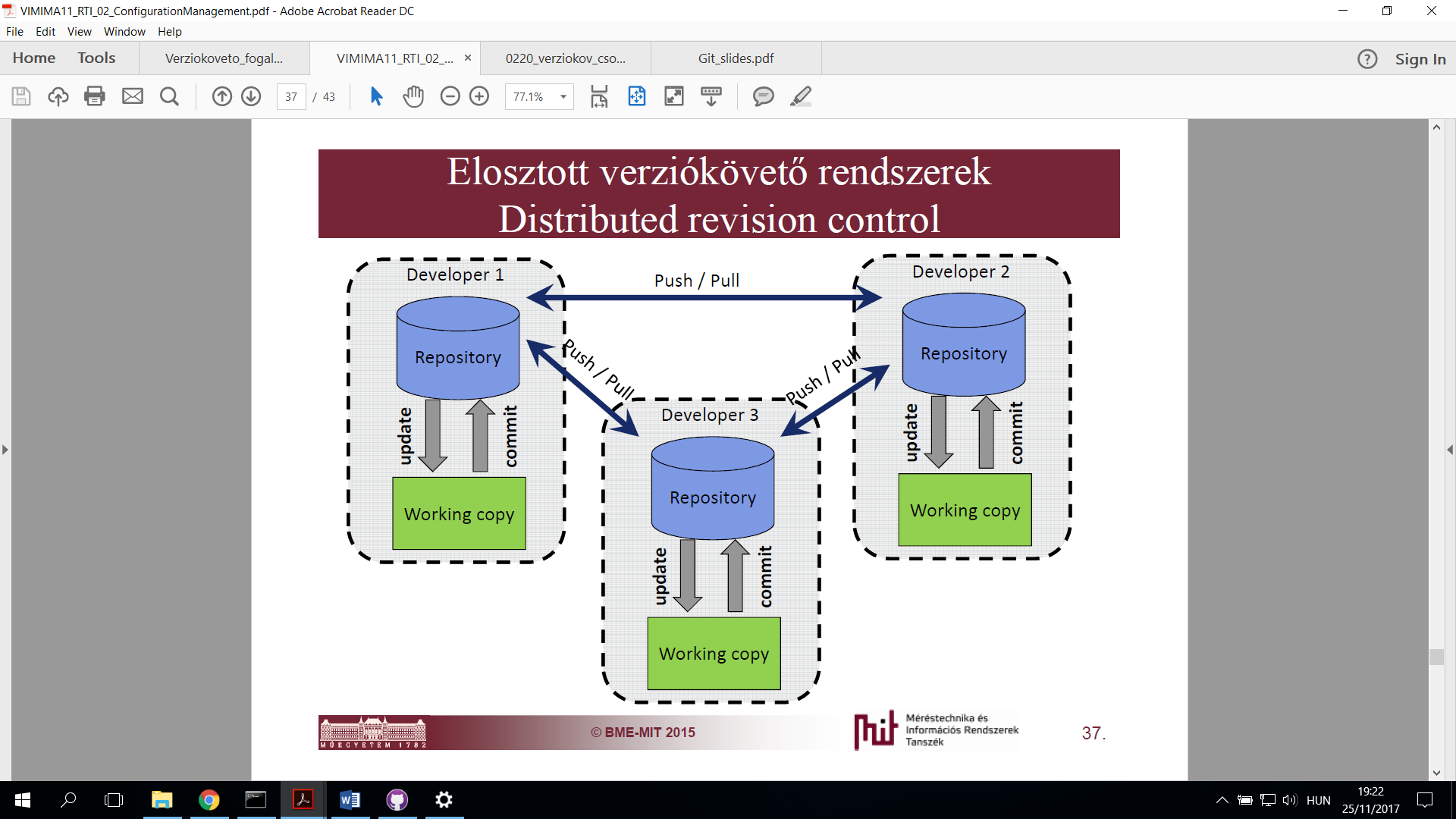
A régebbi verziók és változtatások tárolásának módja fontos kérdés ezekben a rendszerekben. Itt is két fő szemlélet létezik: az egyik a központosított, kontrolált tárolás, a másik pedig az elosztott tárolás.

Központi tárolás esetén egy dedikált szerver végzi a verziók és az azok közötti változtatások nyilvántartását. Visszakereshető a file állapot egy adott időpontban, hogy ki és mikor végzett változtatásokat, illetve ezekről statisztikákat is készíthetünk. Ezt a rendszert követi az általam vizsgált TortoiseSVN verziókövető rendszer interfész is.



3. ábra Cetralizált rendszer

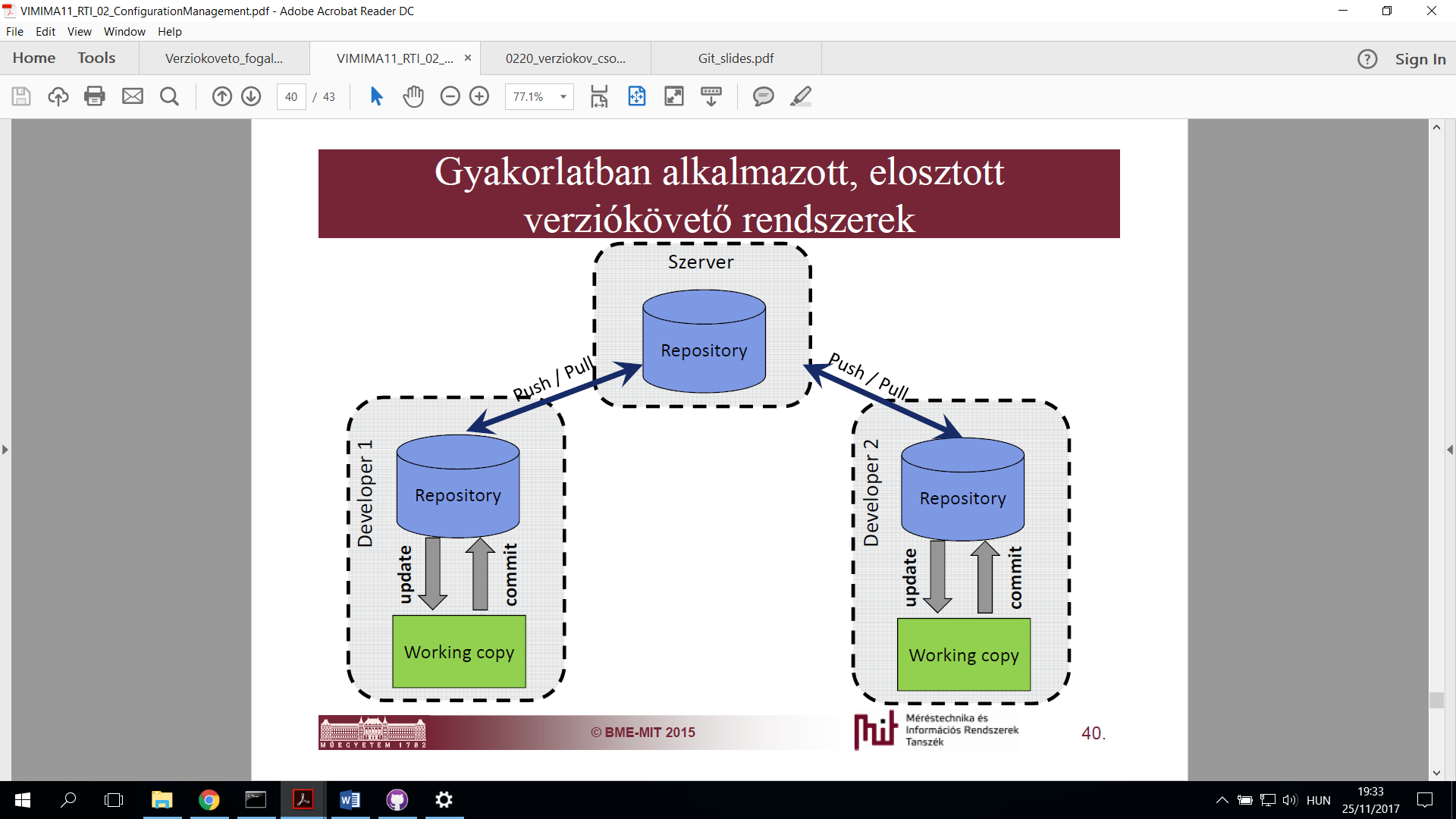
Elosztott rendszerek esetén lényegében minden fejlesztő rendelkezik egy teljes másolattal, az egész projektről. Ebből adódóan a változtatások gyorsan, helyben elvégezhetők, a rendszer offline is működőképes (szemben a központi tárolással, ahol a szervert mindig el kell tudni érni), valamint kevesebb a menedzselésre fordított energia is. Ilyen elosztott rendszer például a Git.



4. ábra Elosztott rendszer

A gyakorlatban azonban ebben az esetben is szükség van egy központi tároló egységre. Ennek több oka is van. Egyrészt szükség van egy backup szerverre. Másrészt az elosztott rendszerekben lényegében mindenki egy saját verzióval rendelkezik – ami tekinthető önálló branch-nek – így az egyes verziók nem egyenértékűek. Ebből következik az is, hogy nem nagyon tudjuk meghatározni, hogy mi a jelenlegi verzió, illetve egyáltalán verzió számot se tudunk megadni.

A gyakorlatban a központi tárolóval rendelkező elosztott rendszer terjedt el, mely hierarchikusan épül fel. A központi tárolóban levő állományokat „leklónozzák” az egyes felhasználók és mint elosztott rendszer használják azt. Majd ha a változtatásokat érvényre akarják juttatni a központi verzió esetében is, akkor vagy „feltolják” ezeket saját állomány esetén, vagy kérvényezniük kell, hogy a tulajdonos ténylegesen „behúzza” az általuk végrehajtott változtatásokat. Ez a struktúra az alábbi képen látható.



5. ábra Gyakorlatban használt elosztott rendszer

Ezt a struktúrát követi az általam vizsgált másik, elosztott verziókövető rendszer a GitHub.

1. GitHub

A GitHub a Git elosztott verziókövető rendszerre épülő alkalmazás, mely egy központi szerver-szolgáltatással lett kiegészítve. Az így kapott hibrid struktúra látható az 5. ábra illusztrációján. A központi szerver egyfajta backup funkciót lát el, illetve egységesíti a fejlesztés menetét, azáltal, hogy központilag tárolja az eredeti állományt. Így lehetőségünk nyílik release-ről, tag-ről és egyáltalán valamilyen értelemben verzióról beszélni.

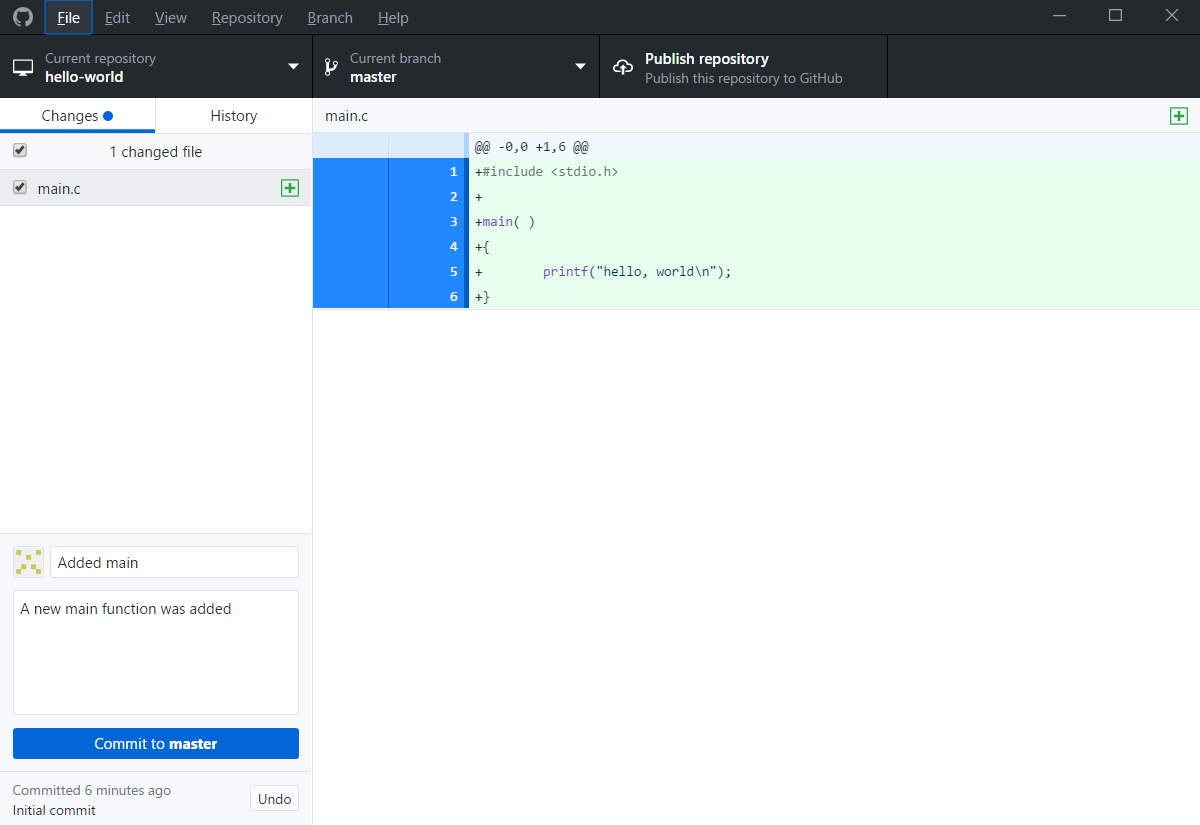
A projekteket úgynevezett repository-ban tároljuk el. Ez lényegében egy strukturált tárolási lehetőséget biztosít mindenféle projekthez tartozó fájl számára. Ezt a repository-t létrehozhatjuk hagyományos módon, lokálisan, a saját gépünkön, vagy közvetlenül a szerveren is. Ha a saját gépünkön hoztuk létre, felmerülhet az igény, hogy ezt feltöltsük a központi szerverre. Ezt a push művelettel tehetjük meg. Ellenkező irányban a pull segítségével másolhatjuk le a repository-t (vagy clone, mely lényegében ugyanazt eredményezi).

A GitHub használatához különböző kliens oldali interfészek állnak rendelkezésünkre, mint például a TortoiseGit, vagy a GitHub Desktop. Mivel a másik bemutatott program a TortoiseSVN volt, így inkább a közelmúltban megjelent GitHub Desktop-ot mutatom be részletesebben.

* 1. GitHub Desktop

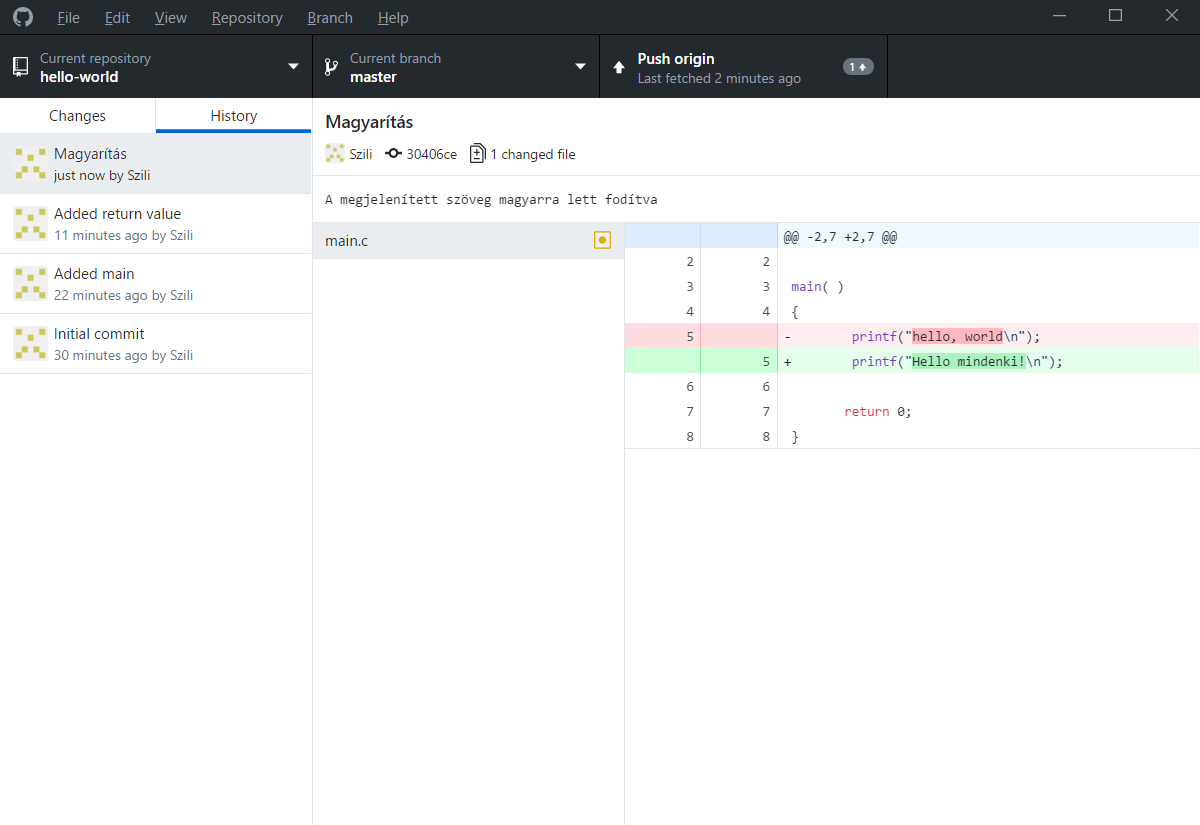
Egy új repository létrehozása a szerver és kliens oldalon (a központi tárhelyhez viszonyítva) teljesen ugyanúgy történik. Megadhatjuk a nevet, egy rövid jellemzést, illetve egy hosszabb leírást az egész projektről – ez alapértelmezésben a README fájl.

Ezt követően különböző állományokat adhatunk hozzá a projektünkhöz (kód, ábra, hangfelvétel, leírás…), bemásolva őket a repository könyvtárába. Ezeket a változtatásokat észlelni fogja a program. Ahhoz, hogy a projekt feltöltésre kerüljön a központi szerverre, először publikálni kell azt a „publish repository” opció segítségével. Itt meg kell adni, hogy hova szeretnénk publikálni (GitHub szerver), illetve, hogy a kód privát, vagy mindenki számára szabadon elérhető-e. Ezeket beállítva a továbbiakban már csak egy gombnyomással automatikusan megtörténik a feltöltés.



6. ábra GitHub Desktop felülete

Ezt követően a fájlokon végzett változtatásokat a commit művelet segítségével tudjuk érvényre juttatni. A művelet során egy rövid leírásban meg kell adnunk, hogy pontosan mit változtattunk az előző verzióhoz képest. A változtatások sorát a history fül alatt tudjuk megtekinteni. Ezt követően a commit-olt változtatásokat a push művelet segítségével tudjuk feltölteni a szerverre.



7. ábra A változtatások megtekintése

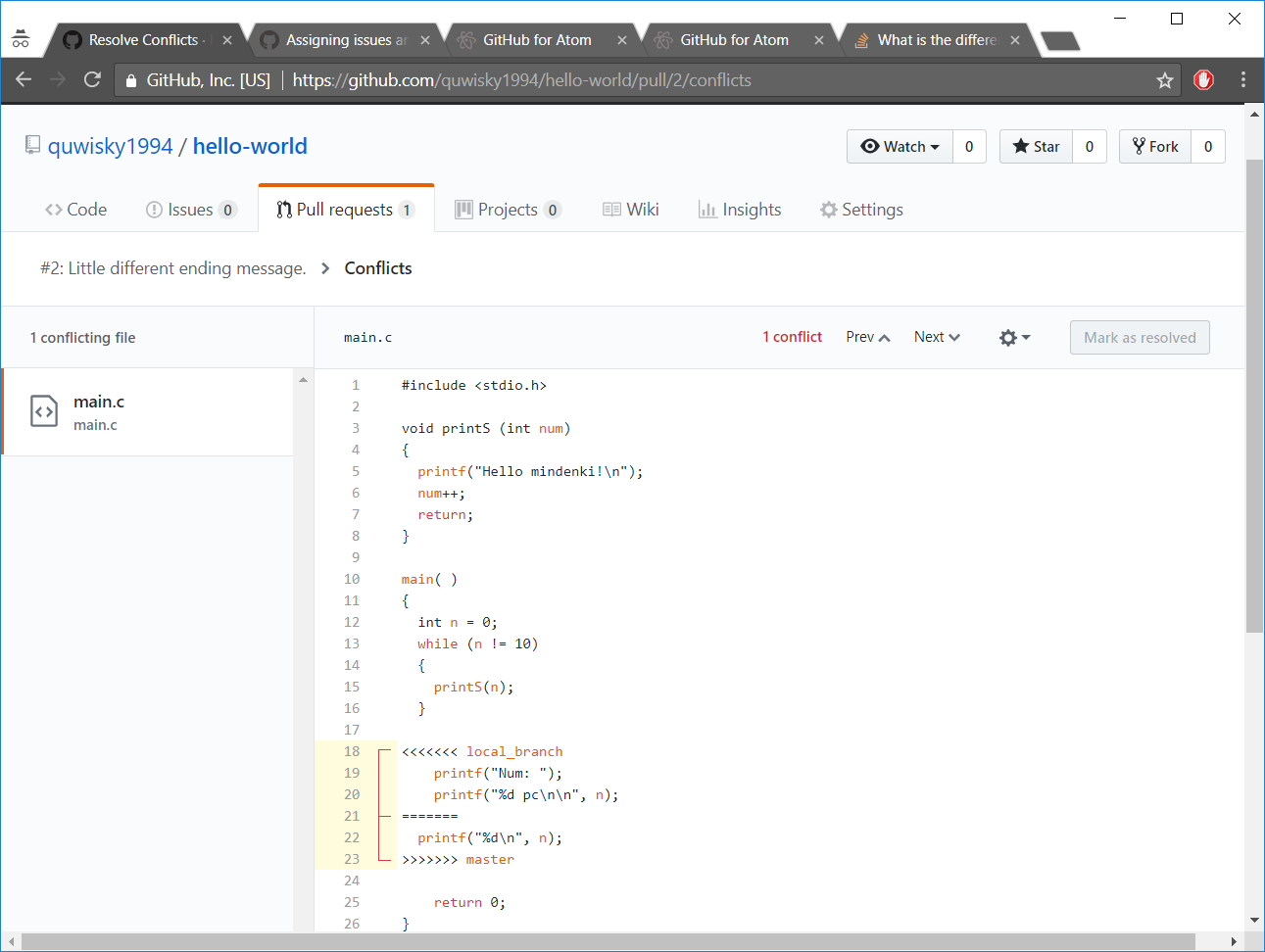
Ha a szerverről szeretnénk egy repository-t lementeni, azt a fájl, clone repository… opcióval tehetjük meg. A repository kiválasztása esetén választhatunk saját projektet, vagy valaki más projektjét GitHub felhasználónév és projektnév, vagy akár URL alapján is. A program automatikusan generál nekünk egy könyvtárszerkezetet, amelybe a projektet lementi.

Több résztvevős projektek esetén a fejlesztés menete általában az, hogy készítünk egy saját branch-et, mely egy teljesértékű másolata a fő ágnak (master branch) – ezt snapshot-nak, avagy pillanatképnek is nevezik. Ezután ezt a saját verziót tudjuk fejleszteni a többi résztvevőtől függetlenül. Ha szeretnénk a saját módosításainkat érvényre juttatni a fő verzióban is, akkor egy pull request műveletet kell indítani. Ez tulajdonképen egy kérvény a repository tulajdonosa felé, hogy az általunk készített verziót „húzza be” és gyúrja össze (merge) a fő verzióval.

A pull requestben meg kell adni, hogy miért szeretnénk a mi verziónkat beküldeni, illetve, hogy mik a végrehajtott változtatások. Nem feltétlenül a repository tulajdonosának kell „elbírálni” a változtatásokat, meghívhat más felhasználókat is, hogy értékeljék azt, és segítsenek a probléma megoldásában (reviewers, assignees).

A GitHub rendszere magas fokon támogatja a konfliktusok felderítését, és ha nem érzékel semmilyen ütközést, akkor minden további nélkül egyetlen gombnyomással hozzá adhatjuk a változtatásokat az adott ághoz (nyilván nem csak a főágban tudjuk ezt megtenni). Ha esetleg valamilyen konfliktus lépne fel, akkor azt emberi döntéssel kell, hogy megoldjuk a conflicts oldalon. Itt láthatjuk a konfliktust szülő kódrészletet kiemelve. Ha ezt megszüntettük a merge művelet már gond nélkül végrehajtható.

A pull request indítása alapértelmezett esetben a desktop interfészen nem hajtható végre. Ehhez szükségünk van az online felületet használni, és ott indítani a kérést.



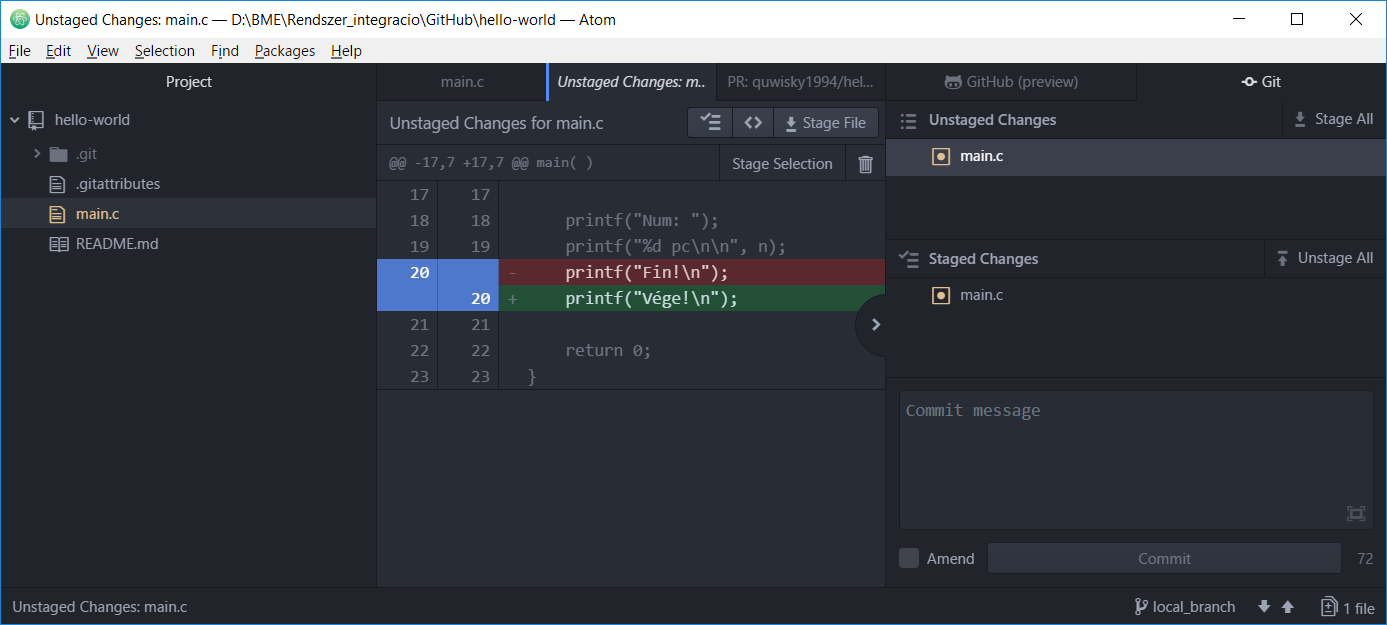
8. ábra A konfliktus megszűntetése GitHub-on

* 1. GitHub Desktop & Atom

A GitHub interfésze alapvetően kódszerkesztővel nem rendelkezik. Az Atom nevezetű nyílt forráskódú editorral kiegészítve viszont a rendszert ily módon is tudjuk használni. A program alapértelmezett verziója teljes körben támogatja a Git és GitHub használatát, annak minden funkciójával együtt. Emellett a nyílt forráskódú szerkesztőt a nagyfokú konfigurálhatóságának köszönhetően rengeteg egyéb, számunkra fontos attribútummal kiegészíthetjük. Többek közt ilyen például a git-blame, melynek segítségével azonosíthatjuk, hogy az adott kódrészlet mely felehasználó melyik commit-jából származik.

Az Atom segítségével lehetőségünk van a pull request-eket is kezelni, illetve a konfliktusokat feloldani (ehhez ugyan egyelőre még manuálisan kell a pull request-eket tartalmazó GitHub URL-t beilleszteni).

Kiegészítő funkció még a stage lehetőség, mely röviden annyit jelent, hogy a elvégzett változtatások csak akkor kerülnek bele a kódunkból a commit-ba, hogy ha staged-ként be vannak jelölve.



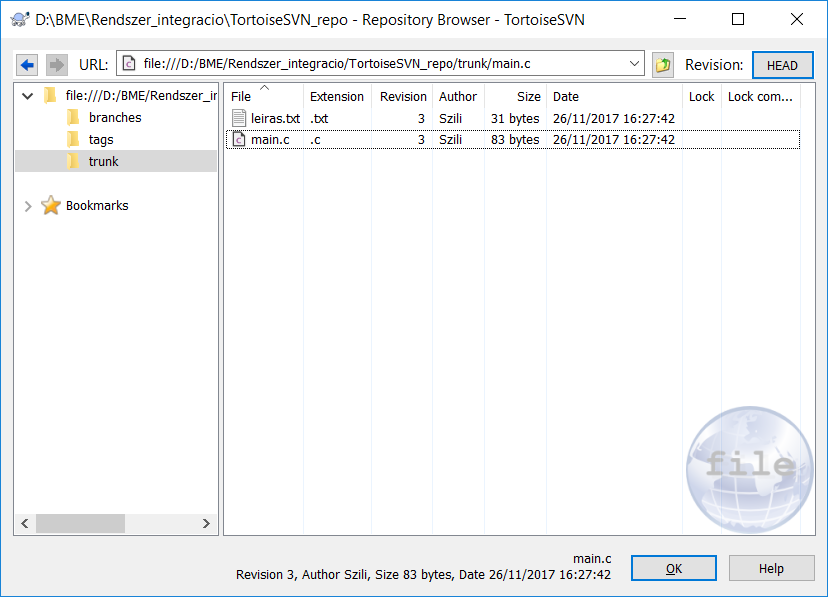
9. ábra Az Atom felülete (változtatások kezelése, stage bellítása)

1. TortoiseSVN

A TortoiseSVN egy nyílt forráskódú windows kliens az Apache centralizált verziókövető rendszerhez. A centralizáltság azt jelenti, hogy az állományok tárolása egy biztonságos, központi szerveren történik. A házi feladat keretében nem hoztam létre egy külön szervert, hanem demonstrációs jelleggel csupán a saját gépemen készítettem egy repository-t, melyhez teljes hozzáférésem volt és ki tudtam próbálni a második fejezetben említett, centralizált rendszerekre jellemző funkciókat.

A TortoiseSVN egyik legjobb tulajdonsága, hogy a telepítéssel beépül a windows shell-be (fájl explorer), ezzel rengeteg funkciót nagymértékben megkönnyít. Például egy lokális repository létrehozása a windows intézőben egyetlen jobb klikk – create repository segítségével megtörténhet. Emellett egyszerűen használható grafikus interfésze van, illetve saját ikon készlettel is rendelkezik, melyeket a fájlok eredeti ikonjain jelenít meg, ezáltal indikálva az adott állomány státuszát (csak olvasható, módosított, lezárt…).

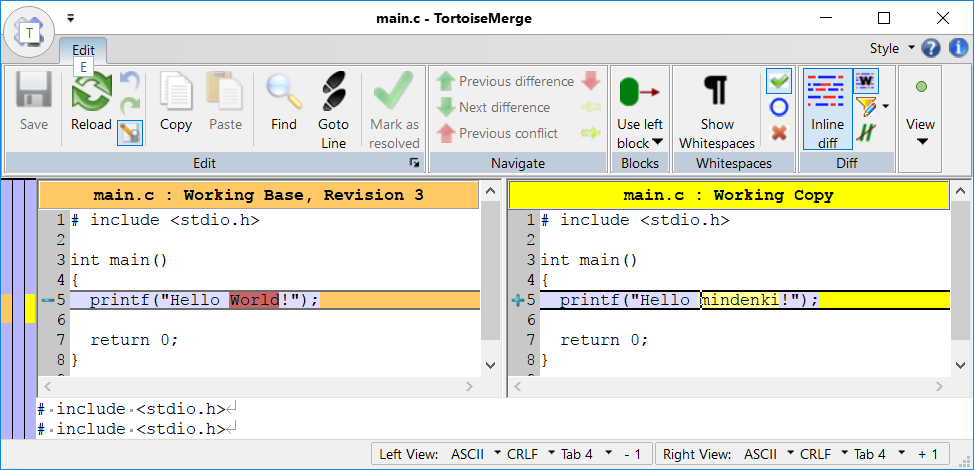
Egy lokális repository létrehozását követően néhány fájlt importáltam be, hogy ki tudjam próbálni az alap funkciókat. A repository-t és annak fájljait a repo-browser segítségével tudjuk megvizsgálni. Ahogy az az alábbi képen is látható, a program segítségével generáltam az alapértelmezett könyvtárszerkezetet (trunk, tags, branch). Jobb oldalt lehetőségünk van kiválasztani, hogy melyik revíziót szeretnénk megtekinteni. Mivel itt központilag tároljuk az állományokat, van értelme explicit módon verzióról – revízióról – beszélni. A head revízió mindig az aktuálisat jelöli.



10. ábra TortoiseSVN repo-browser

Ez után, hogy dolgozhassunk az adott állományokon kérnünk kell egy saját másolatot, úgynevezett working copy-t. Ezt a jobb klikk, checktout opcióval tehetjük meg. Ezt elvégezve a mappában megjelenik a repository könyvtárszerkezete, a lekért revíziónak megfelelő állományokkal. Az ikonok mellett zöld pipa látható, mely normál állapotot jelent (win10 esetén lehetséges, hogy az ikonok nem jelennek meg, ekkor a regedit segítségével a „HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\explorer\ShellIconOverlayIdentifiers” útvonalon található TortoiseSVN bejegyzéseket kell kiegészíteni a nevek elején 3 darab space karakterrel, majd újraindítás után minden rendben látszódik).

Ha a working copy állományt megváltoztatjuk, az ikonok pirosra váltanak, ezzel jelezve, hogy lokálisan módosították őket. A változtatásokat a jobb klikk – Diff opcióval jeleníthetjük meg.

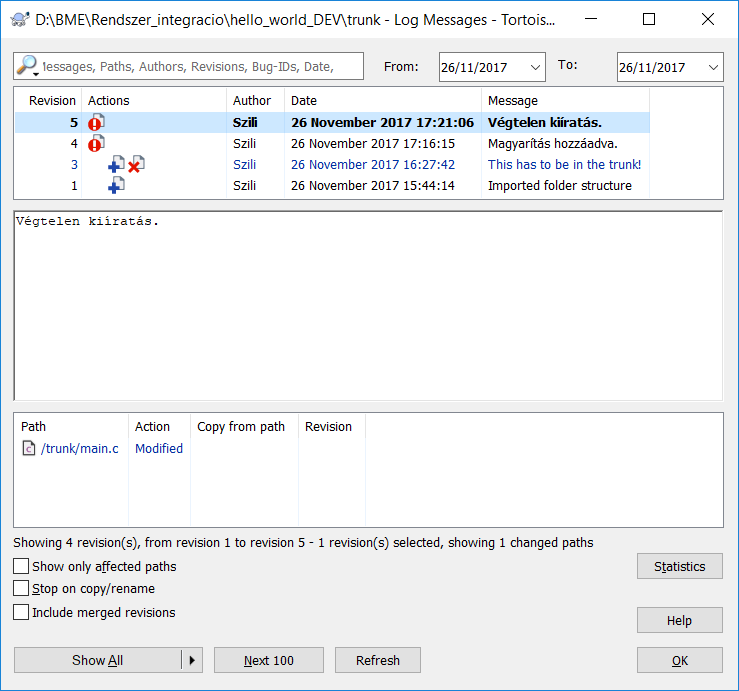


11. ábra Változtatások megjelenítése

A kapott ablak neve TortoiseMerge. Ennek az az oka, hogy ha egyidejű hozzáférés lép fel – tehát miközben valaki dolgozik a fájlon, addig más azt megváltoztatja (commit) – akkor ugyanebben az ablakban kell feloldanunk a változtatásokból kialakuló konfliktusokat.

Hogy a változtatásokat érvényre juttathassuk, commit-olni kell az állományokat, melyet szintén jobb klikk – SVN commit… segítségével tehetünk meg. Ennek hatására (mivel időközben más nem módosította az állományokat) zöld pipák jelennek meg a fájlok ikonjai felett.

A TortoiseSVN egyik leghasznosabb funkciója a Show log, mely a repository életében bekövetkező minden változást elment: visszakereshető, hogy ki, mikor, mit commit-olt, milyen változtatások történtek, és ezekhez milyen leírások tartoznak. A változtatások naplózása mellett mindenféle statisztikai adatokat is megjeleníthetünk.



12. ábra TortoiseSVN naplózási funkciója