Software Technology
Szoftvertechnológia

Project Tools
Projekteszközök

Dr. Máté Cserép
ELTE, Faculty of Informatics
2025.

Szoftvertechnológia

ELTE | IK
INFORMATIKAI KAR

Tartalomjegyzék

Software Technology

ELTE | IK
INFORMATIKAI KAR

Contents

Szoftvertechnológia

Projektmenedzsment eszközök

Dr. Cserép Máté
ELTE Informatikai Kar
2025.

Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

Szoftvereszközök

- A fejlesztőcsapat munkáját megfelelő szoftvereszközökkel kell alátámasztani
  - *projektmenedzsment eszközzel* (*project tracking system*), amely támogatja a dokumentálást és a feladatok követését
  - *fejlett tervezőeszközzel* (*case tool*), ahol a fejlesztés folyamata és a felelősség is nyomon követhető
  - *integrált fejlesztőkörnyezettel* (*IDE*)
  - *verziókövető rendszerrel* (*revision control system*), amely lehetővé teszi a programkód változásainak követését
  - *folytonos integrációs* (*continuous integration*) rendszerrel, amely biztosítja a hibák korai kiszűrését

Projektmenedzsment eszközök

Funkcionalitás

- A projektmenedzsment eszköz lehetőséget ad az alábbiakra:
  - fejlesztés ütemtervének, kockázatainak meghatározása
  - fejlesztés egyszerű és folyamatos dokumentálásának lehetősége és generálása
  - feladatok, tevékenységek rögzítése, követése
  - a tesztelés során előfordult hibák rögzítése, a javítási folyamat követése
  - integrált verziókezelés és forráskód böngészés
  - webes vagy grafikus felület, amely biztosítja a könnyű használatot, és bárhonnan való elérést
  - a forráskód áttekintése és vizsgálata (*code review*)

Projektmenedzsment eszközök

Ütemterv és időzítés

- A szoftver lehetőséget ad, hogy a projekt ütemtervét elkészítsük, és azt folyamatosan szem előtt tarthassuk
  - definiálhatunk mérföldköveket (*milestone*), amelyre adott feladatokat el kell végezni
  - a fejlesztők külön-külön láthatják a saját feladataikat, menedzselhetik annak előrehaladását
  - beoszthatjuk a fejlesztési lépések erőforrásait
  - definiálhatunk függőségeket a programrészek között
  - kezelhetjük az egyes fejlesztési lépések időbeli lefolyását, előrevetíthetjük a tervezettől való eltérések hatásait az erőforrásokra, illetve a további fejlesztési időkre

Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

Feladat és hibakövetés

- A rendszerek lehetőséget adnak a tervezők számára feladatok kitűzésére, valamint a tesztelők számára a programban fellelhető hibák jelzésére
  - a feladatok úgynevezett *cédulák* (*ticket, issue*) segítségével írhatóak ki
    - jelölhetnek új funkcionalitást (*feature*), hibát (*bug*), egyéb fejlesztési feladatot (*task*), vagy dokumentációs feladatot (*documentation*)
    - megadható a leírása, felelőse, határideje
    - kommentálhatóak, lezárhatóak, újra kinyithatóak
  - a cédulák biztosítják a fejlesztési és tesztelési folyamat naplózását

Projektmenedzsment eszközök

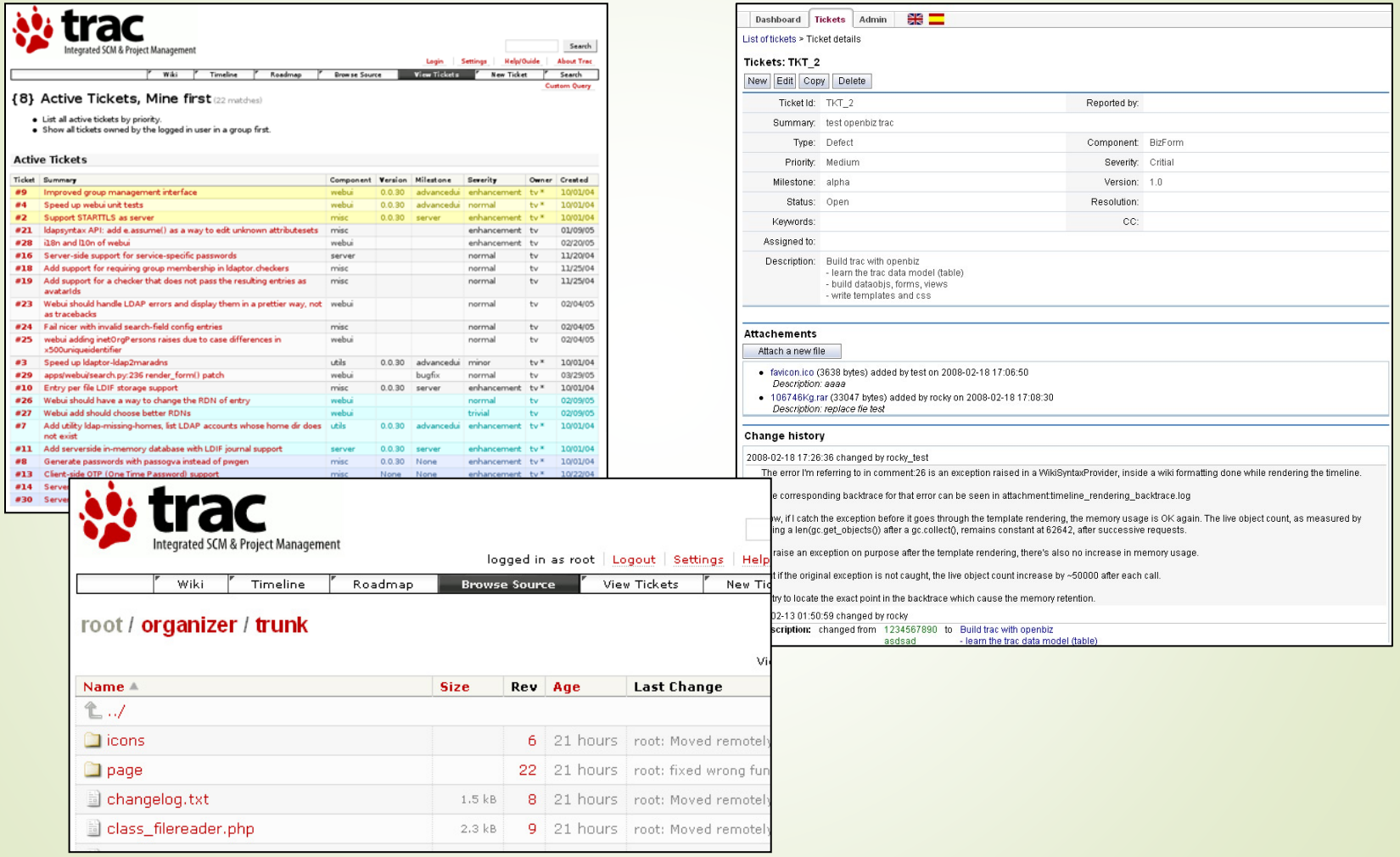
Példák

- Az eszközök felületi része alkalmas webes technológiával, míg az adattárolás adatbázis-motor segítségével valósítják meg
  - a legtöbb eszköz szabad forráskódú, és a projektvezetés ugyanazon eszközzel van menedzselve
- Néhány népszerű projektmenedzser:
  - *Trac*: Python alapú, MySQL/SQLite/PostgreSQL adatbázis háttérrel
  - *Redmine*: Ruby on Rails alapú, MySQL/SQLite/PostgreSQL adatbázis háttérrel
  - *Azure DevOps Services*: a Microsoft megoldása, Git és TFVC támogatással, ASP.NET és MSSQL alapokon (2019-ig *Microsoft Team Foundation Server* néven ismert)
  - *YouTrack*: a JetBrains rendszere, Java alapokon, Xodus adatbázis háttérrel (NoSQL)

Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

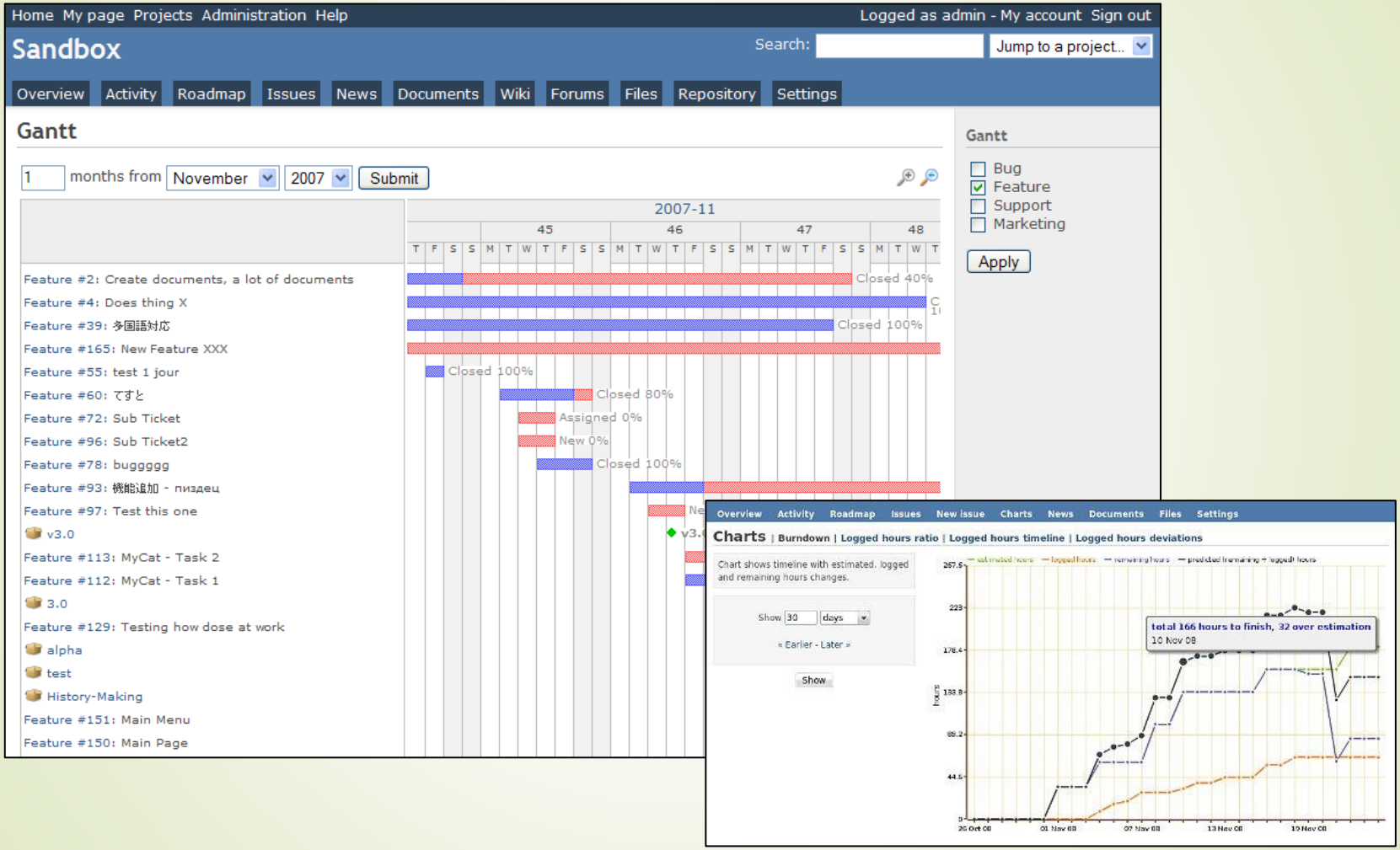
A Trac projektmenedzser



Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

A Redmine projektmenedzser



Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

Projektvezető szolgáltatások

- A *projektvezető szolgáltatás*ok (*project hosting services*) általában rendelkezésre bocsátanak több projektfejlesztő eszközt
  - projektmenedzsment, kód tárolás, kód megtekintés, verziókövetés, kód vizsgálat, dokumentáció (wiki), levelezési lista
  - integrálhatóak projektirányítási eszközökkel, fejlesztőeszközzel
  - garantálják a kód épségét, a folyamatos rendelkezésre állást
  - általában nyílt forráskódú szoftverek esetén ingyenes a szolgáltatás
  - pl.:*GitHub*, *GitLab, SourceForge, Bitbucket, Azure DevOps*

Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

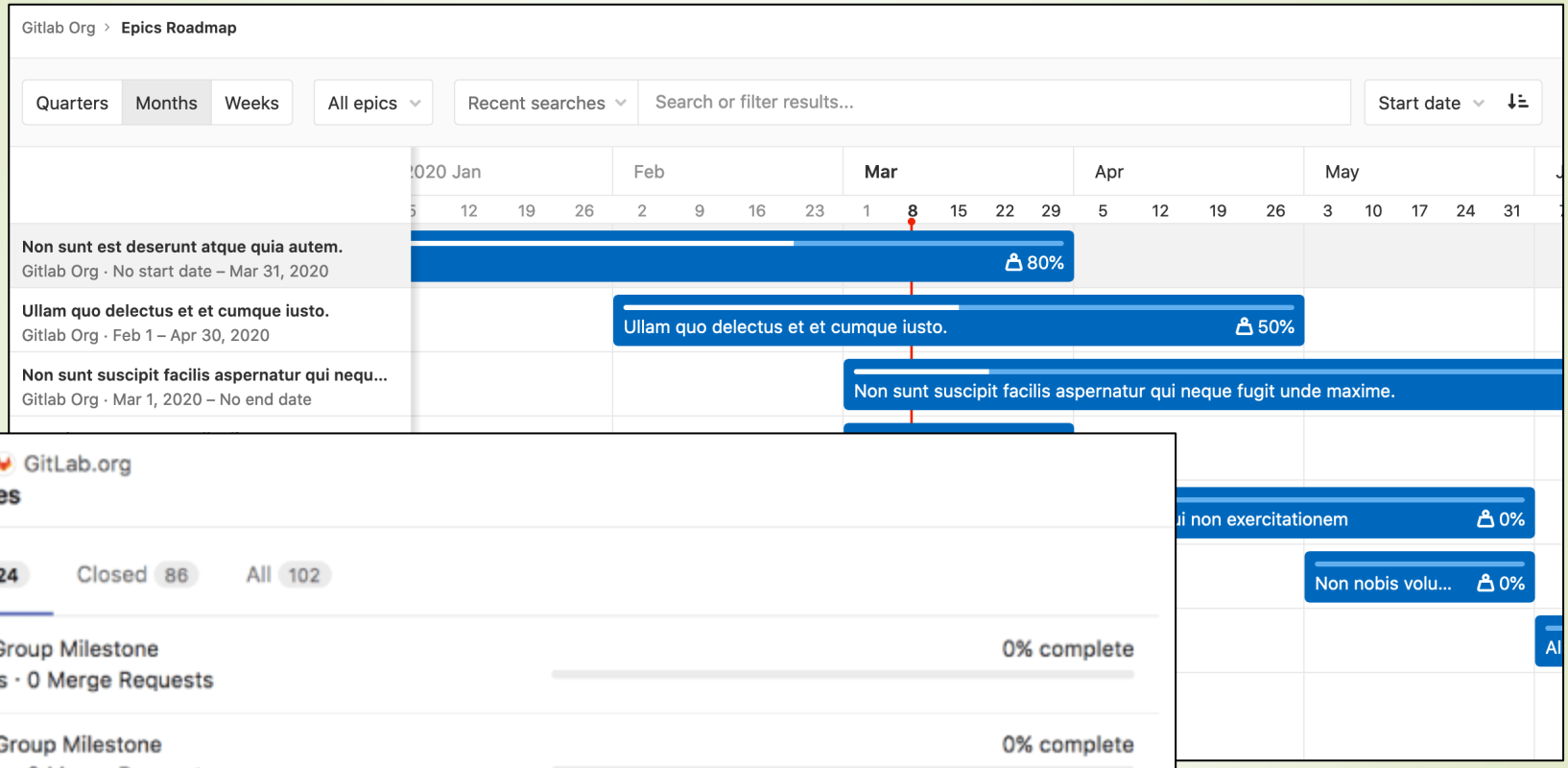
A GitLab projektvezető szolgáltatás

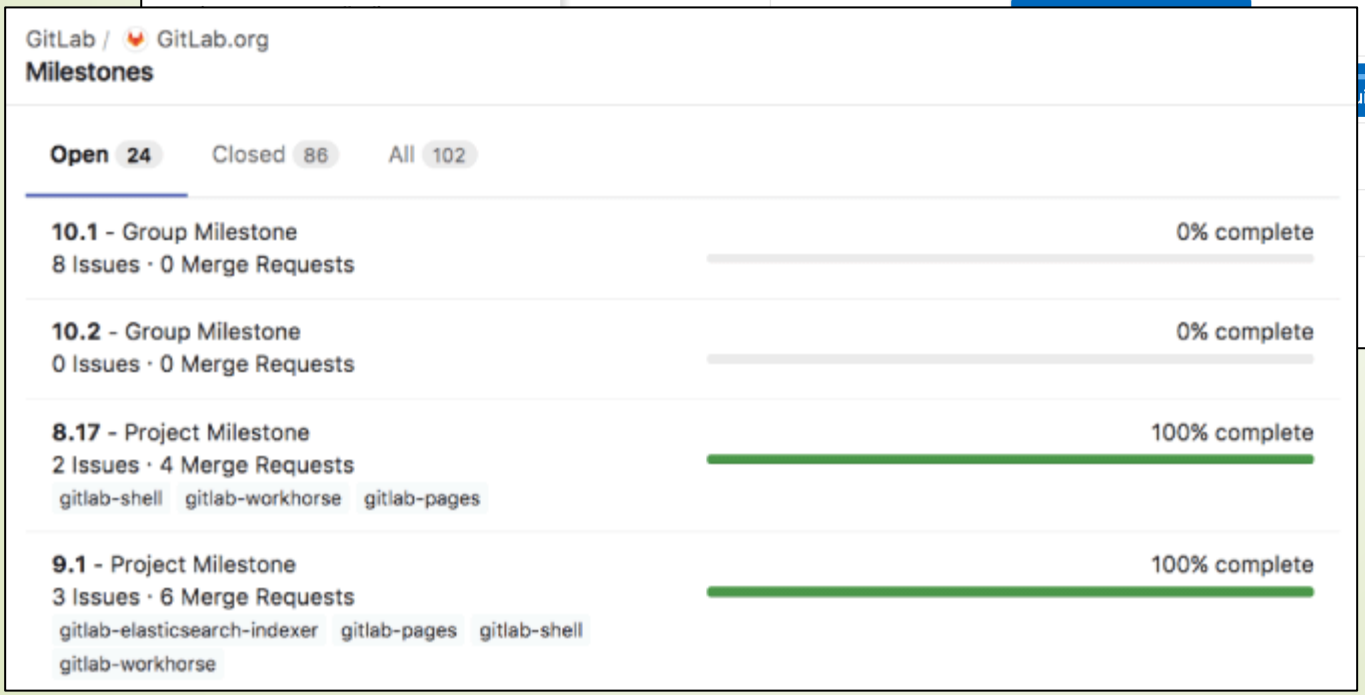
- A GitLab egy webes felülettel rendelkező, számos szoftvereszközt integráló projektvezető szolgáltatás
  - Használható a gitlab.com oldalon elérhető keresztül (SaaS)
    - hasonló a GitHub-hoz
    - ingyenes, korlátlan (privát) projekttel (előfizetés elérhető)
  - GitLab Community Edition
    - saját szerveren kiszolgált (*self-hosted*)
    - megfelelő kisebb és közepes méretű csapatok számára
    - ingyenes, nyílt forráskódú
  - GitLab Enterprise Edition
    - extra funkcionalitásokkal, amelyekre jellemzően nagyobb fejlesztői állomány (100+ fő) esetén van szükség

Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

GitLab - Projektmenedzsment

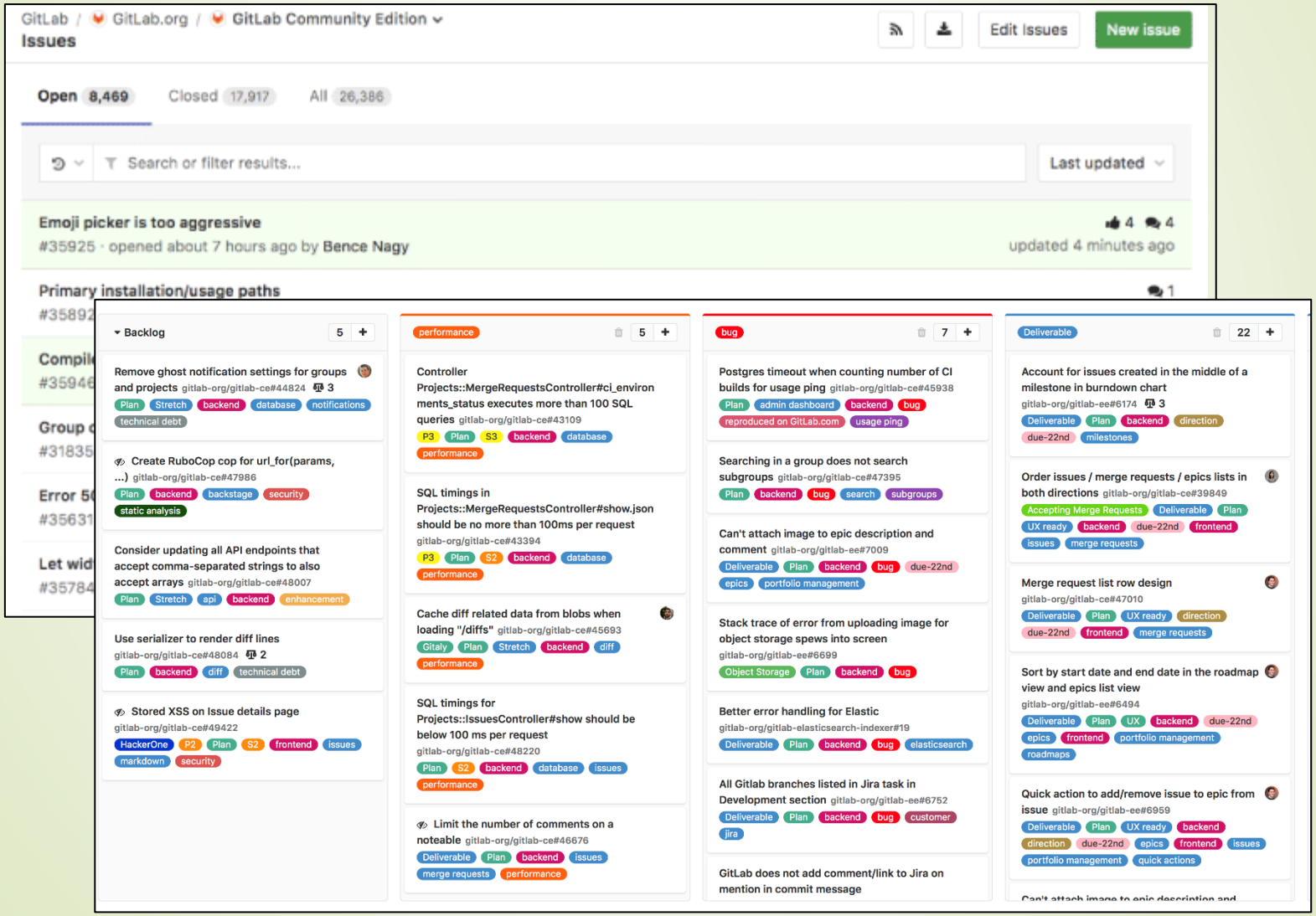




Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

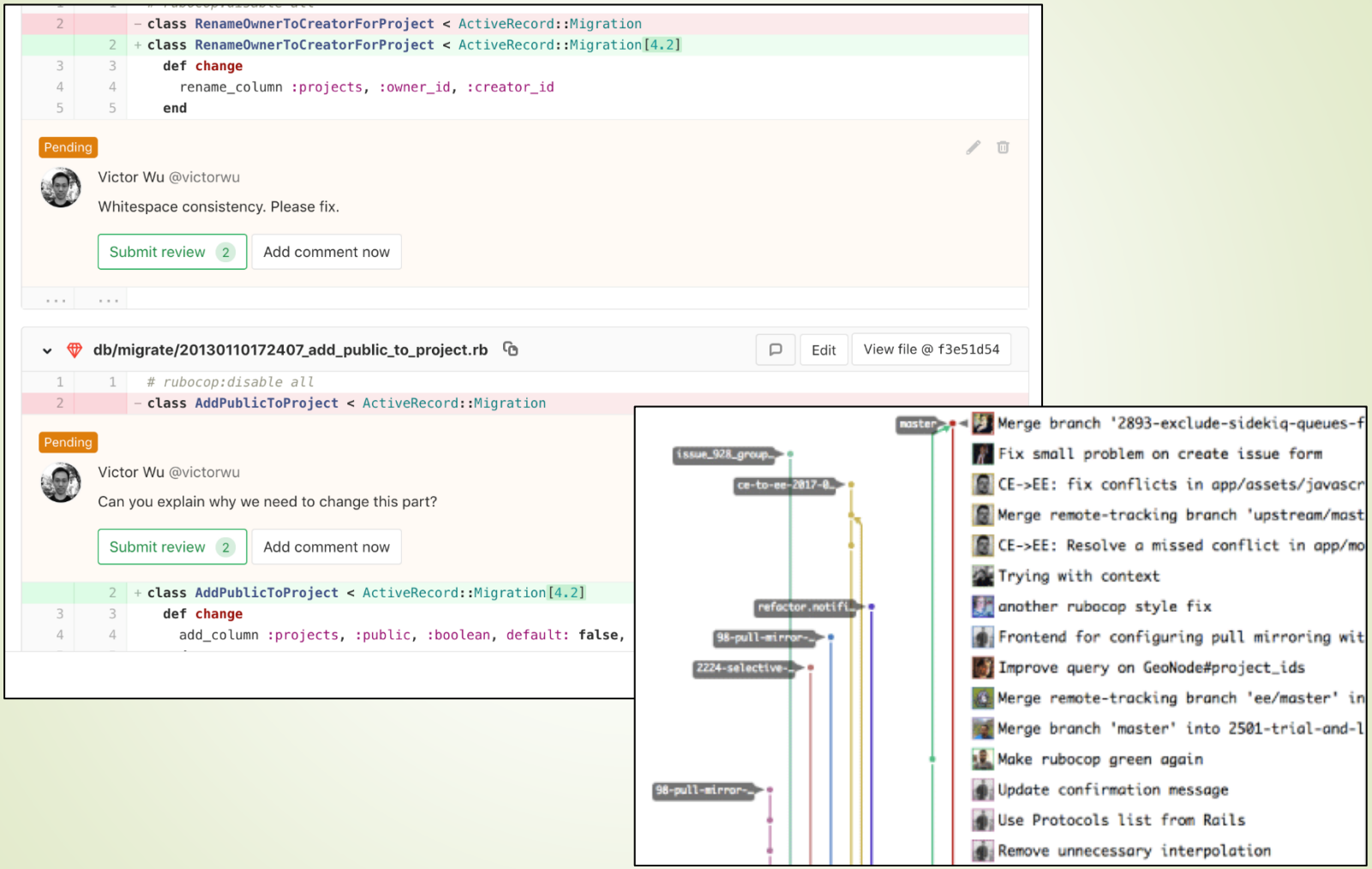
GitLab - Feladatkövetés



Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

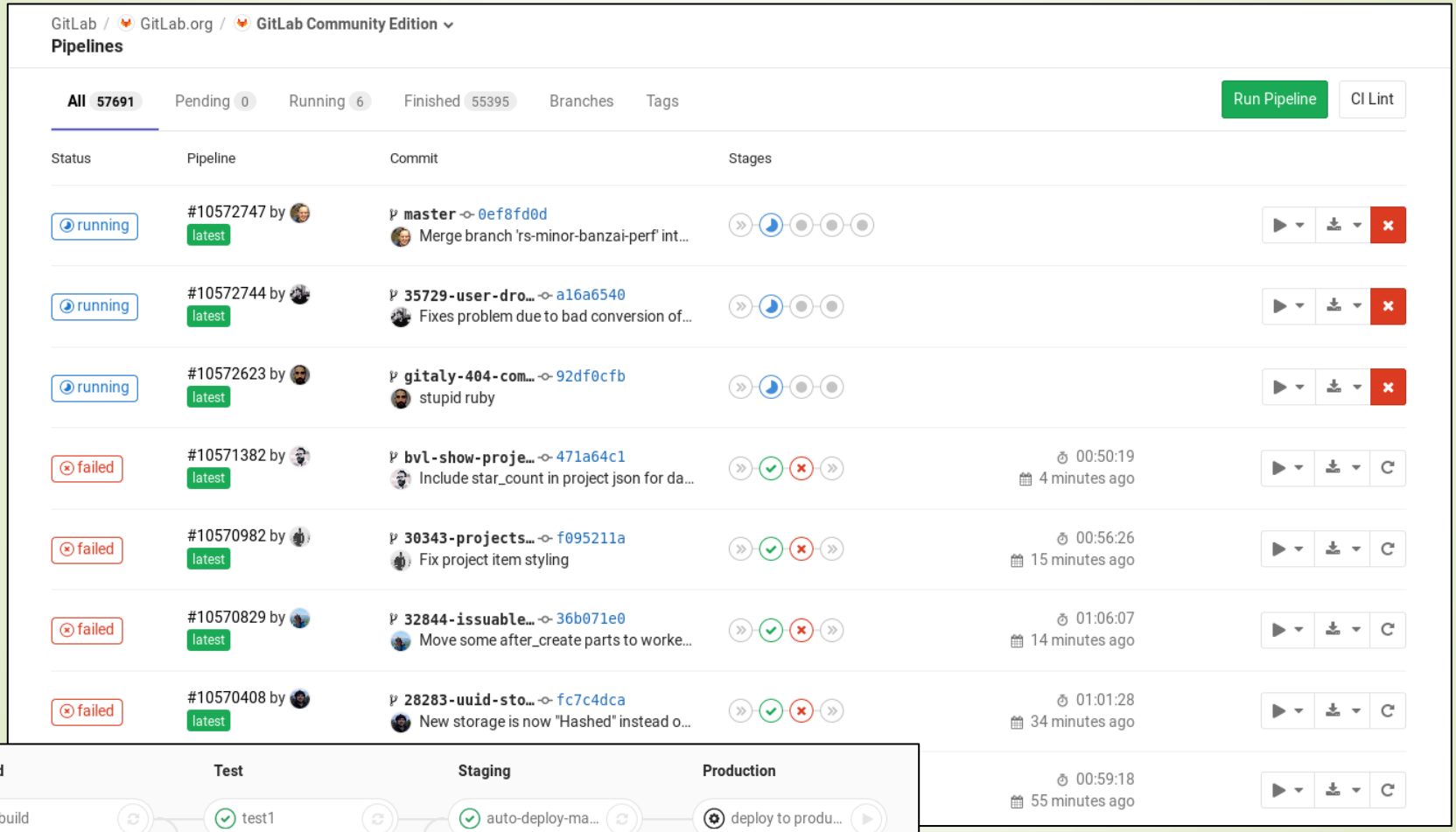
GitLab – Forráskód menedzsment



Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

GitLab – Folytonos integráció



Projektmenedzsment eszközök

ELTE | IK
INFORMATIKAI KAR

Kari GitLab szerver

- A félév folyamán a gyakorlati projekteket a Szoftvertechnológia kurzust kiszolgáló kari GitLab szerveren kell vezetni.
  - Elérhetőség: https://szofttech.inf.elte.hu/
  - Belépés: INF-es azonosítóval (mint a géptermi gépekre)
- A tervezés, a megvalósítás, a dokumentálás és a tesztelés is teljesíthető a GitLabon belül.
  - Wiki oldalak, markdown szintaxis
  - Feladatkövetés
  - Forráskód verziókövetés
  - Folytonos integráció, tesztesetek automatikus futtatása

Szoftvertechnológia

Verziókövető rendszerek

A & B szakirány

Dr. Cserép Máté

ELTE Informatikai Kar

2025.

Verziókövető rendszerek

Történeti háttér

- A szoftverek méretének és komplexitásának növekedésével létrejött szoftverkrízis következményeként megnövekedett:
  - a programok forráskódjának mérete,
  - a szoftverprojektek megvalósításához szükséges idő,
  - és szükséges programozói erőforrás.
- A szoftveripar fejlődésével egyre több alkalmazás készült
  - a fejlesztések életciklusa gyakran nem ért véget a program első publikus verziójának kiadásával,
  - karbantartási és további fejlesztési fázisok követték.

- **A szoftverprojektek méretben, komplexitásban, időben és a résztvevő fejlesztők számában is növekedni kezdtek.**

Verziókövető rendszerek

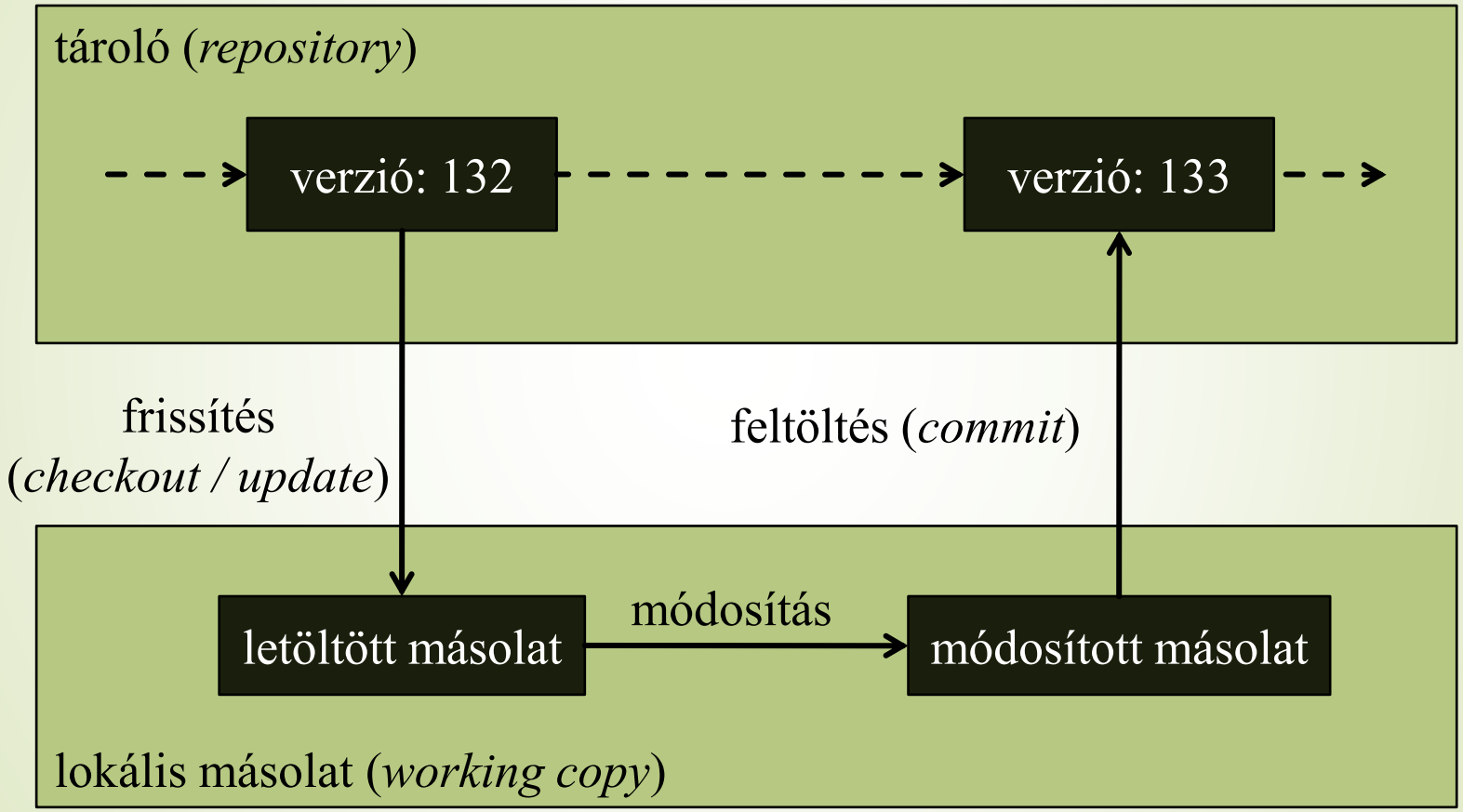
Funkcionalitás

- Mivel az implementáció tehát több lépésben, és sokszor párhuzamosan zajlik, szükséges, hogy az egyes programállapotok, jól követhetőek legyenek, ezt a feladatot a *verziókövető rendszer*ek (*revision control system*) látják el
  - pl. *CVS*, *Apache Subversion* (*SVN*), *Mercurial*, *Git*
  - egy közös tárolóban (*repository*) tartják kódokat
  - ezt a fejlesztők lemásolják egy helyi munkakönyvtárba, és amelyben dolgoznak (*working copy*)
  - a módosításokat visszatöltik a központi tárolóba (*commit*)
  - a munkakönyvtárakat az első létrehozás (*checkout*) után folyamatosan frissíteni kell (*update*)

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Funkcionalitás



Verziókövető rendszerek

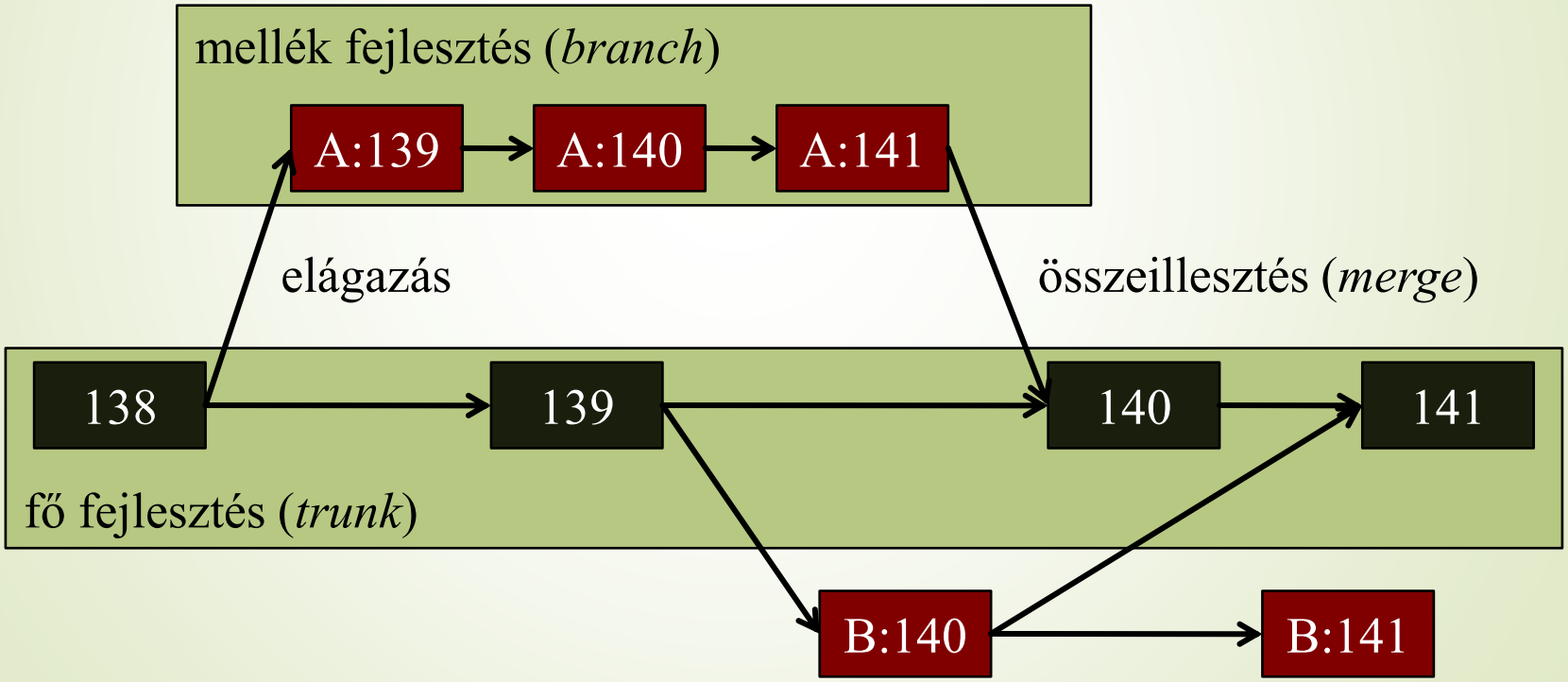
Funkcionalitás

- A verziókövető rendszerek lehetővé teszik:
  - az összes eddig változat (*revision*) eltárolását, illetve annak letöltési lehetőségét
  - a fő fejlesztési vonal (*baseline*, *master* vagy *trunk*) és a legfrissebb változat (*head*) elérését, új változat feltöltését annak dokumentálásával
  - az egyes változatok közötti különbségek nyilvántartását fájlonként és tartalmanként (akár karakterek szintjén)
  - változtatások visszavonását, korábbi változatra visszatérést
  - konfliktust okozó módosítások ellenőrzését, illetve megoldását (*resolve*)

Verziókövető rendszerek

Funkcionalitás

- a folyamat elágazását, és ezáltal újabb fejlesztési folyamatok létrehozását, amelyek a fő vonal mellett futnak (*branch*), valamint az ágak összeillesztését (*merge*)



Verziókövető rendszerek

Funkcionalitás

- az összeillesztés rendszerint utólagos manuális korrekciót igényel
- az összeillesztésnek rendszerint automatikusan illeszti a módosított tartalmakat kódelemzést használva, ez lehet 2 pontos (*two-way*), amikor csak a két módosítást vizsgálja, vagy 3 pontos, amikor az eredeti fájlt is

- programrészek zárolását (*lock*), hogy a konfliktusok kizárhatóak legyenek
- adott verzió, mint pillanatkép (*snapshot*) rögzítése (*tag*), amelyhez a hozzáférés publikus
- feltöltések atomi műveletként történő kezelését (pl. megszakadó feltöltés esetén visszavonás)

Verziókövető rendszerek

Lokális verziókövető rendszerek (1. generáció)

- Forráskód változásainak követése, a szoftver funkcióinak különböző kombinációjával készült kiadások kezelése
  - lokális tároló (de többen is elérhetik pl. *mainframe* esetén)
  - fájl alapú műveletvégzés (1 verzió 1 fájl változásai)
  - konkurenciakezelés kizárólagos zárak által

- Az 1970-es években lefektetésre kerültek az elméleti alapok
  - Source Code Control System (SCCS) – 1972
  - Revision Control System (RCS) - 1982

Verziókövető rendszerek

Centralizált verziókövető rendszerek (2. generáció)

- Több fejlesztő általi párhuzamos szoftverfejlesztés támogatásának előtérbe kerülésre
  - centralizált modellt megtartva, de kliens-szerver architektúra
  - fájlhalmaz alapú műveletek (1 verzió több fájl változásai)
  - konkurenciakezelés jellemzően beküldés előtti egyesítéssel (*merge before commit*)
- Az 1990-es évektől terjedtek el:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
  - SourceSafe, Perforce, Team Foundation Server, stb.
- *Hátrány: a szerver kitüntetett szerepe (pl. meghibásodás), továbbá a verziókezeléshez hálózati kapcsolat szükségeltetik*

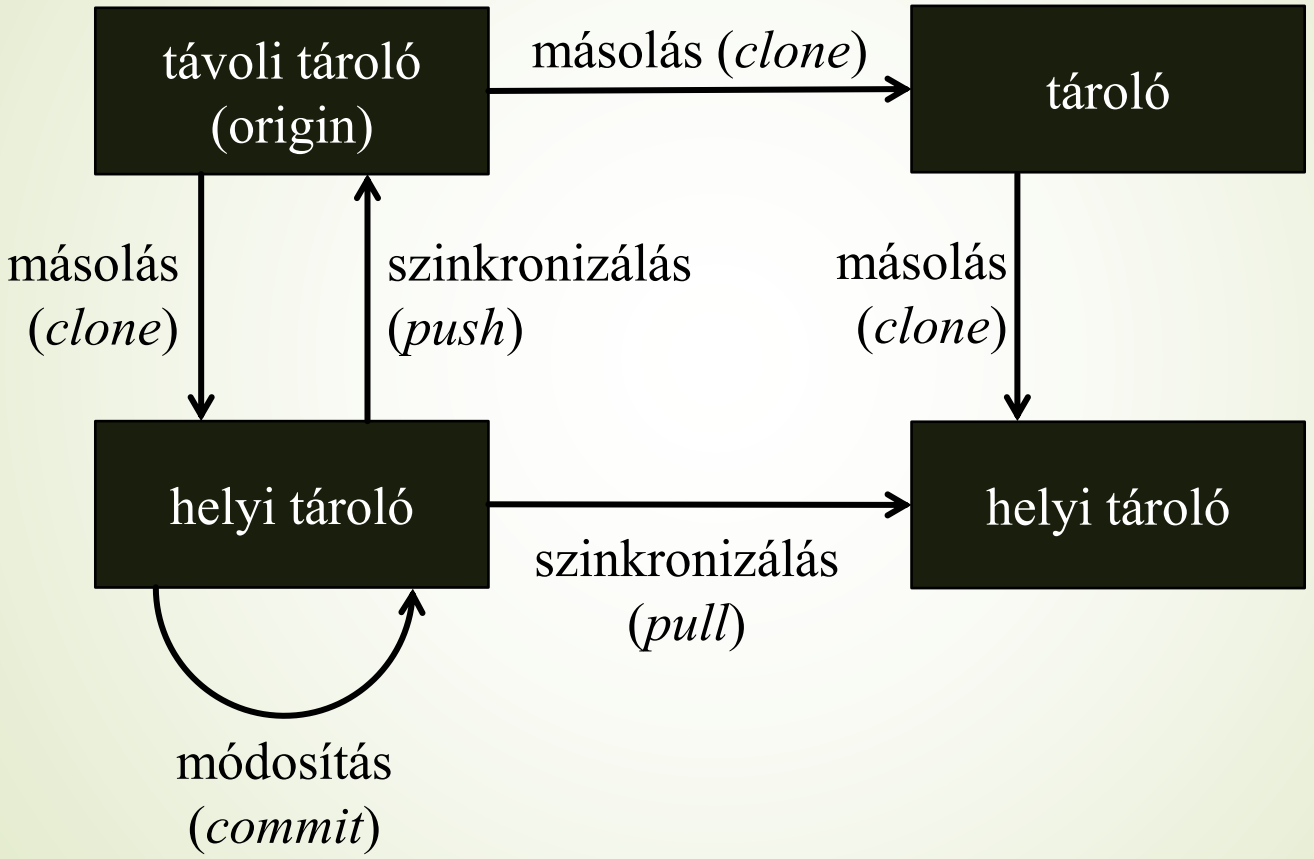
Verziókövető rendszerek

Elosztott verziókövető rendszerek (3. generáció)

- A klasszikus verziókezelő műveletekről leválasztásra kerül a hálózati kommunikáció, azok a felhasználó által kezdeményezhető önálló tevékenységekként jelennek meg
  - decentralizált, elosztott hálózati modell
  - minden kliens rendelkezik a teljes tárolóval és verziótörténettel
  - a revíziókezelő eszköz műveletei lokálisan, a kliens tárolóján történnek
  - a kommunikáció *peer-to-peer* elven történik, de kitüntetett (mindenki által ismert) szerverek felállítására van lehetőség
  - konkurenciakezelés jellemzően beküldés utáni egyesítéssel (*commit before merge*)
- A 2000-es évek első felében jelent meg:
  - Monotone, Darcs, Git, Mercurial, Bazaar, stb.

Verziókövető rendszerek

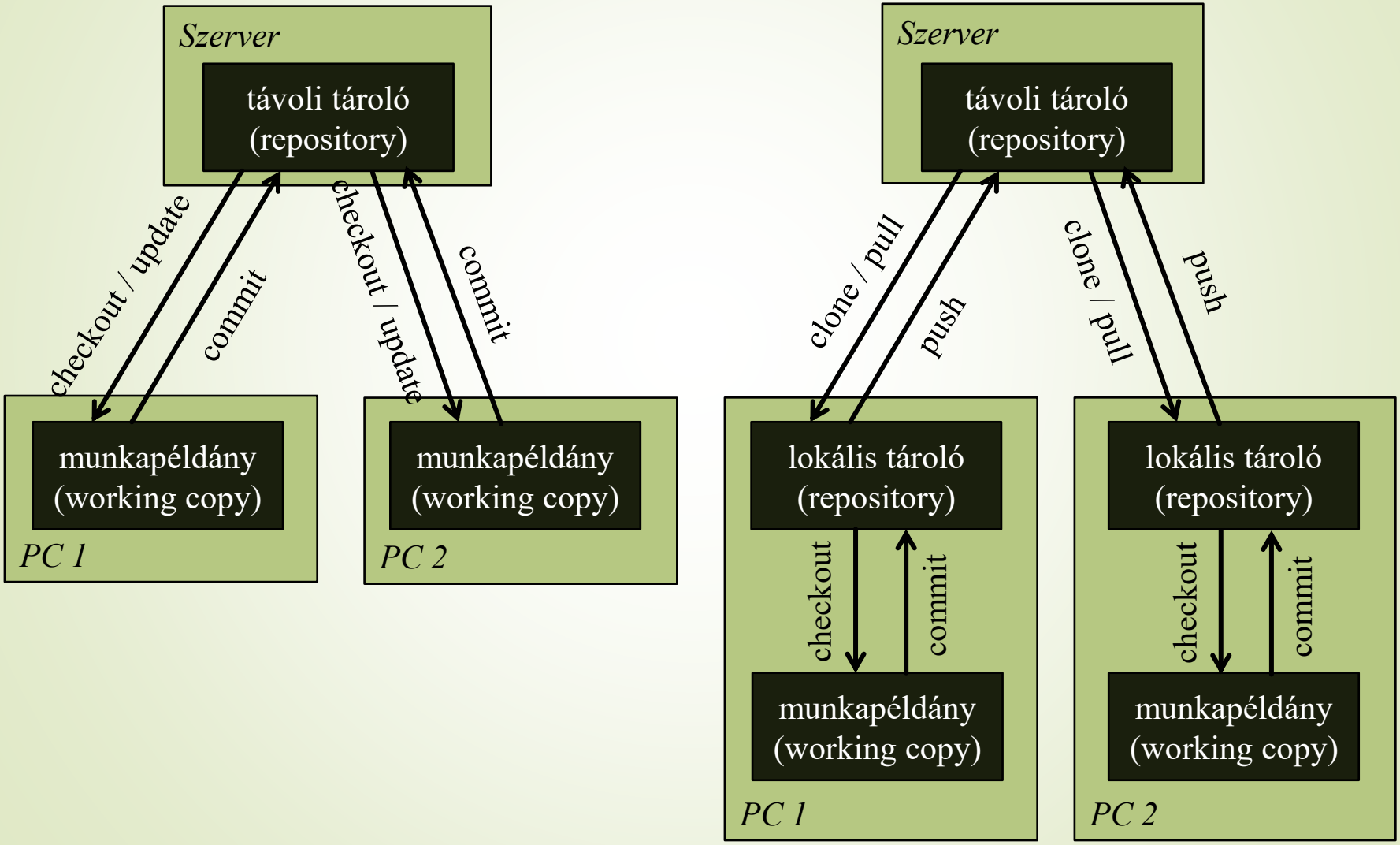
Elosztott verziókövető rendszerek (3. generáció)



Verziókövető rendszerek

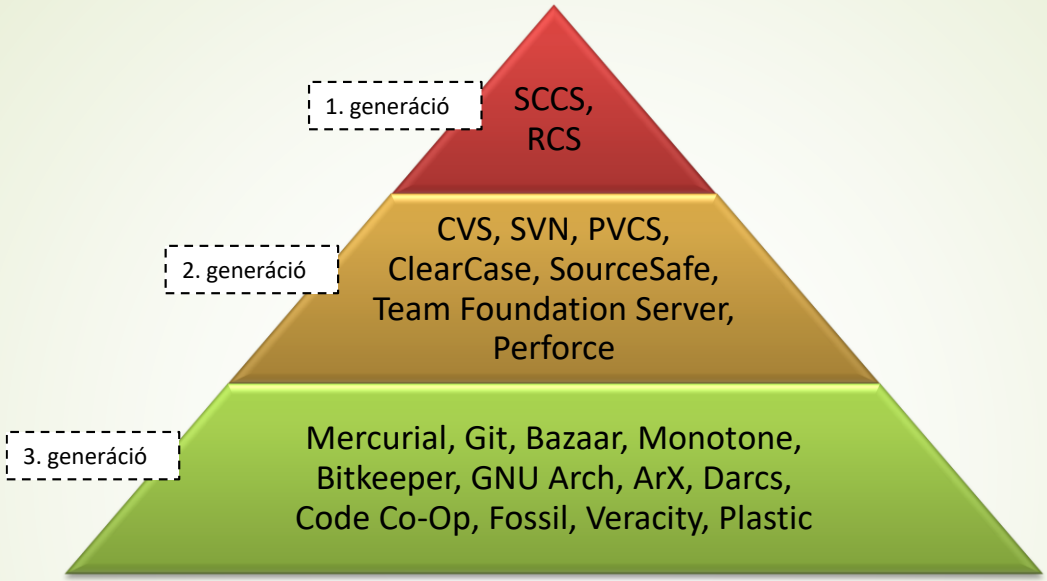
ELTE | IK
INFORMATIKAI KAR

Centralizált és elosztott verziókezelő rendszerek



Verziókövető rendszerek

Generációs modell



Generáció	Hálózati modell	Műveletvégzés	Konkurenciakezelés
Első	Lokális	Fájlonként (non-atomic commits)	Kizórálóagos zárak (exclusive locks)
Második	Központosított	Fájlhalmaz (atomic commits)	Egyesítés beküldés előtt (merge before commit)
Harmadik	Elosztott	Fájlhalmaz (atomic commits)	Beküldés egyesítés előtt (commit before merge)

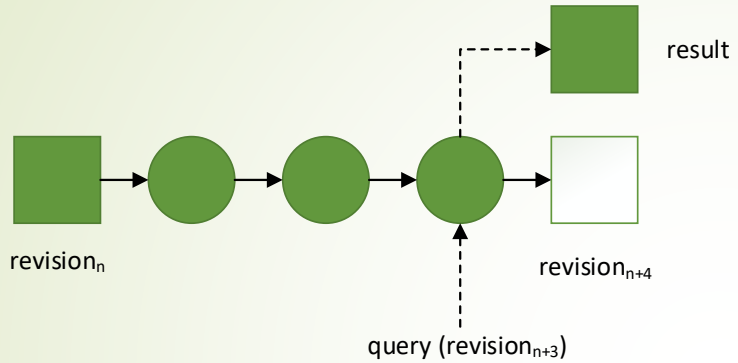
Verziókövető rendszerek

Változások reprezentációja

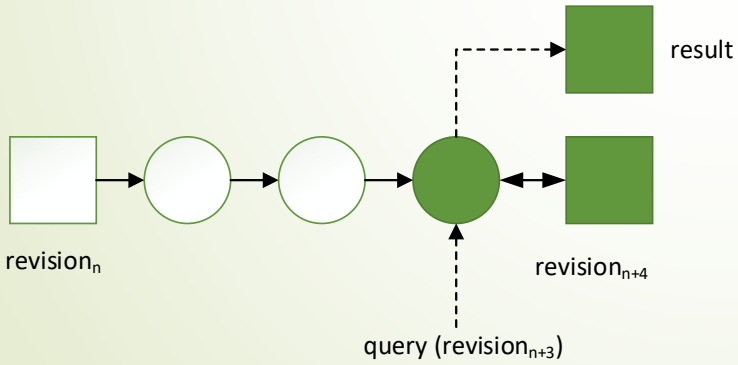
- Kezdetben a teljes revíziók tárolása nem volt lehetséges az adattárolás és adatkezelés jelentős költségei miatt
- Az 1. és 2. generációs verziókezelő eszközök jellemzően ezért csak két egymást követő verzió közötti különbséget, a *változáslistá*t (*changeset*, *delta*) tárolják
  - egyes rendszerek (pl. Mercurial) időnként pillanatfelvételt (*snapshot*) készítenek a teljes tartalomról
- Eleinte (SCCS) a delták a régi verzióból az újat tudták előállítani (*forward deltas*)
- Korán felmerült (RCS), hogy a fordított delták (*reverse deltas*) használata a legújabb verzió pillanatképének tárolásával jobb teljesítményt nyújthat, ugyanis leggyakrabban egy ág legfrissebb állapotát szokták lekérni
  - Kevert megoldás is lehetséges, pl. a fő ágon fordított irányú deltákat, a mellékágakon viszont előre mutató delták

Verziókövető rendszerek

Változások reprezentációja



Előre irányú delták
(*forward deltas*)

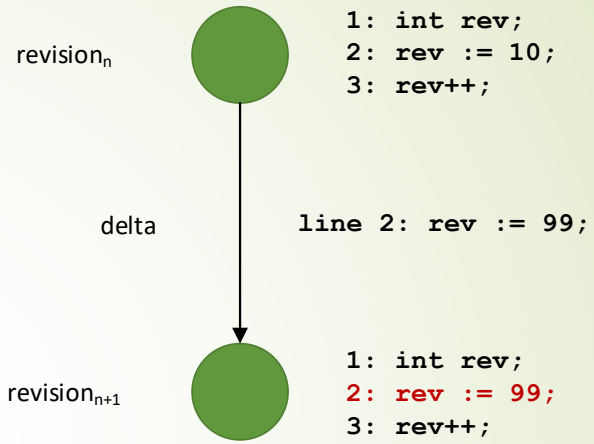


Visszafelé irányú delták
(*reverse deltas*)

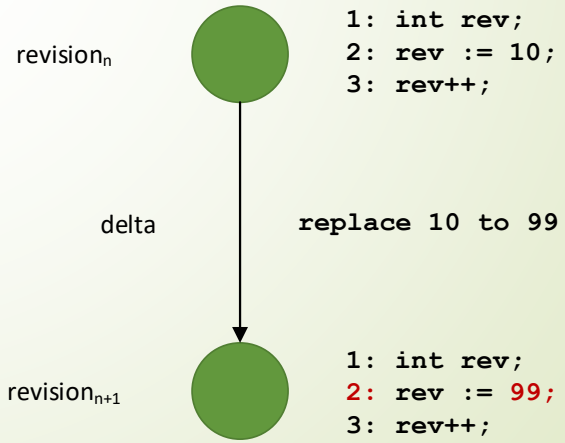
Verziókövető rendszerek

Változások reprezentációja

- Az eltérések meghatározása szöveges fájlok, így programnyelvi forráskódok esetében jellemzően *állapot alapúan* történik
  - a legtöbbször soronkénti összehasonlítással
    - pl. GNU diff
  - struktúrált tartalom esetén az összehasonlítás egysége más is lehet (pl. XML, JSON, UML)
- Bináris adatok (pl. képek) esetén a *művelet alapú* megközelítés is alkalmazható.



Állapot alapú

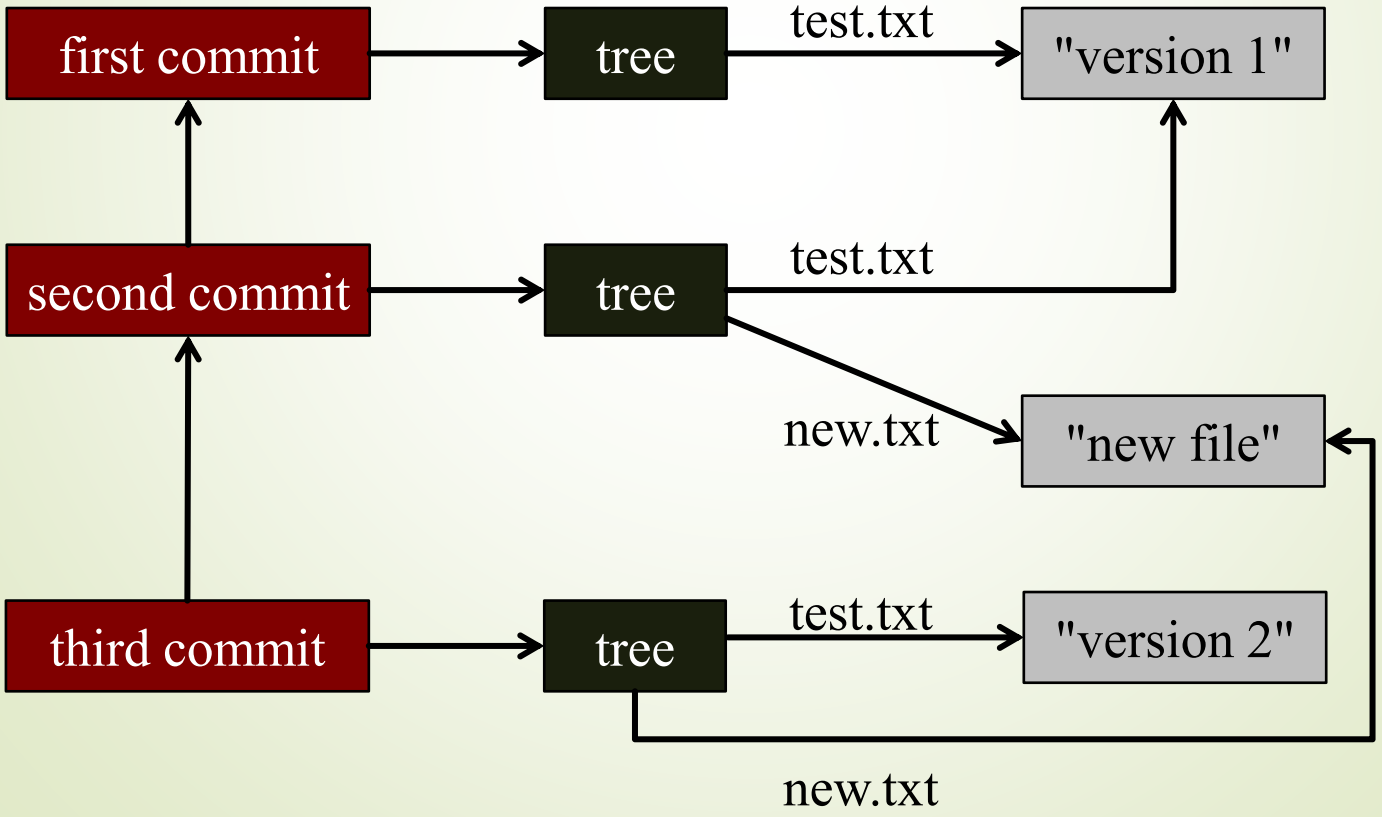


Művelet alapú

Verziókövető rendszerek

Változások reprezentációja

- A 3. generációs verziókövető rendszerek idejére a háttértárak mérete jelentősen megnőtt, míg költségük csökkent
  - Egyes eszközök ezért a megváltozott fájlok teljes tartalmát tárolják a revíziókban, nem csak a módosult tartalomrészt



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git

- A félév folyamán a gyakorlati projektek forráskódját a Git verziókezelő rendszerben fogjuk követni, amelyet integráltan támogat a GitLab projektvezető szolgáltatás.
  - Kari GitLab szerver: https://szofttech.inf.elte.hu/
- A GitLab szerveren lévő távoli tároló (*remote repository*) tartalmát a GitLab webes felületén is böngészhetjük, megtekinthetjük, sőt egyszerűbb módosításokat is végrehajthatunk (ezekből ugyanúgy *commit* lesz).
- A verziókezelést azonban alapvetően egy lokális munkapéldányon (*local repository*) szokás végezni kliens programmal, majd szinkronizálni a távoli tárolóval.
  - Konzolos kliens utasítások
  - Asztali grafikus kliens alkalmazások

Verziókövető rendszerek

Git: telepítés

- A Git verziókezelő rendszer telepítése:
  - Windows, Mac telepítő: https://git-scm.com/downloads
  - Debian/Ubuntu: `apt-get install git`
  - Más UNIX rendszerek: https://git-scm.com/download/linux
- Telepítés után a konzolos Git parancsokkal egyből dolgozhatunk.
  - Windows telepítés esetén célszerű a Git hozzáadását választani a PATH környezeti változóhoz.
  - Grafikus kliens programok külön telepíthetőek.
- Minden Git commithoz hozzárendelésre kerül a szerzője neve és email címe, így ezeket szükséges globálisan beállítanunk, mielőtt először használjuk a Gitet.

```
git config --global user.name "Hallgató Harold"
git config --global user.email hallgato@inf.elte.hu
```

Verziókövető rendszerek

Git: tároló létrehozása

- Egy új lokális tárolót létrehozhatunk üresen:

```
git init
```

- Vagy egy ismert távoli tároló lemásolásával:

```
git clone https://mysite.com/best-project.git
```

- Jellemzően távoli tárolók másolását alkalmazzuk, még akkor is, ha kezdetben üres a projekt.
  - Az így lemásolt távoli tárolóra *origin* néven hivatkozhatunk (alapértelmezetten) majd a későbbiekben, pl. a tárolók szinkronizálásakor.

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: változások követése

- Új fájlokat valamint a fájlok módosításait a `git add` utasítással vonhatjuk verziókezelés alá, az ún. *staging area*-ba helyezve:

```
git add main.cpp
```

  - Konkrét fájl helyett mintát (pl. `*.cpp`) vagy könyvtárat is megadhatunk.

- A munkakönyvtár állapotát a `git status` utasítással ellenőrizhetjük bármikor.

```
git status
> On branch master
> Your branch is up to date with 'origin/master'.
> Changes to be committed:
>   (use "git reset HEAD <file>..." to unstage)
>         new file:   main.cpp
```

Git: módosítások helyi tárolóba küldése

- Amennyiben a kívánt módosításokat a *staging area*-hoz adtunk, annak tartalmát egy új verzióként beküldhetjük a lokális tárolóba, a `git commit` utasítással:

```
git commit -m "Added main program."
> [master d26c7a9] Added main program.
>  1 file changed, 1 insertion(+)
>  create mode 100644 main.cpp
```

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: módosítások változáskövetése

- Új fájlokat is verziókezelés alá vonhatunk:

```
git add rectangle.h

git commit -m "Added Rectangle class."
```

- Egy új verzió új fájlokat és meglévő fájlok módosításait is tartalmazhatja:

```
git add circle.h

git add main.cpp

git commit -m "Added Circle class.
               Modified the main program."
```

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: szinkronizálás távoli tárolóval

- A lokális tárolónkban létrehozott új verziókat szükséges szinkronizálni a távoli tárolóval, hogy a változtatásainkhoz mások is hozzáférhessenek, ezt a `git push` paranccsal tehetjük meg.

```
git push origin master
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 247 bytes | 123.00 KiB/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To /path/to/workspace/folder
   d45172c..80a39a2  master -> master
```

  - Szükséges megadni, hogy melyik távoli tárolóval szeretnénk szinkronizálni, és melyik fejlesztési ágat. Amiről klónoztunk alapértelmezetten az *origin* néven ismert, az ág itt a *master*.
  - Nyomkövető ágak (*tracking branches*) használatakor ezek elhagyhatóak, így az utasítás `git push`-ra egyszerűsödik.
- Fejlesztő társaink beküldött módosításait a saját lokális tárolónkkal és munkapéldányunkkal a `git pull` paranccsal szinkronizálhatjuk.

Git: fejlesztési ágak létrehozása

- Új fejlesztési ágat a `git branch` paranccsal hozhatunk létre.

```
git branch new-branch
```

  - Az új fejlesztési ág abból a verzióból fog elágazni, amelyiket aktuálisan betöltöttük a munkapéldányunkba.

- Fejlesztési ágak között a `git checkout` utasítással válthatunk.

```
git checkout new-branch
```

  - Az alapértelmezett fejlesztési ág neve jellemzően *master*.

- Új fejlesztési ág létrehozása és átváltás egyetlen lépésben:

```
git checkout -b new-branch
```

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: fejlesztési ágak egyesítése

- A fejlesztési ágakon végrehajtott módosításokat az adott funkció elkészülte után szeretnénk a fő fejlesztési ágba visszacsatolni.
  - Az ágak egyesítésére a `git merge` utasítás szolgál.
  - Előbb betöltjük a fő fejlesztési ágat:

```
git checkout master
```

  - Majd a mellék fejlesztési ág módosításait egyesítjük vele:

```
git merge new-branch
```

- Amennyiben ugyanazon fájlok ugyanazon részét időközben mindkét fejlesztési ágon módosítottuk, az egyesítés jellemzően nem kivitelezhető automatikusan, ezt ütközésnek (*merge conflict*) nevezzük.

Verziókövető rendszerek

Git: ütközések feloldása

- Az ütközéseket manuálisan kell feloldanunk az érintett fájlok szerkesztésével.
- Az automatikusan nem egyesíthető részeket a Git forrásfájlokban speciális szintaxisba foglalja:

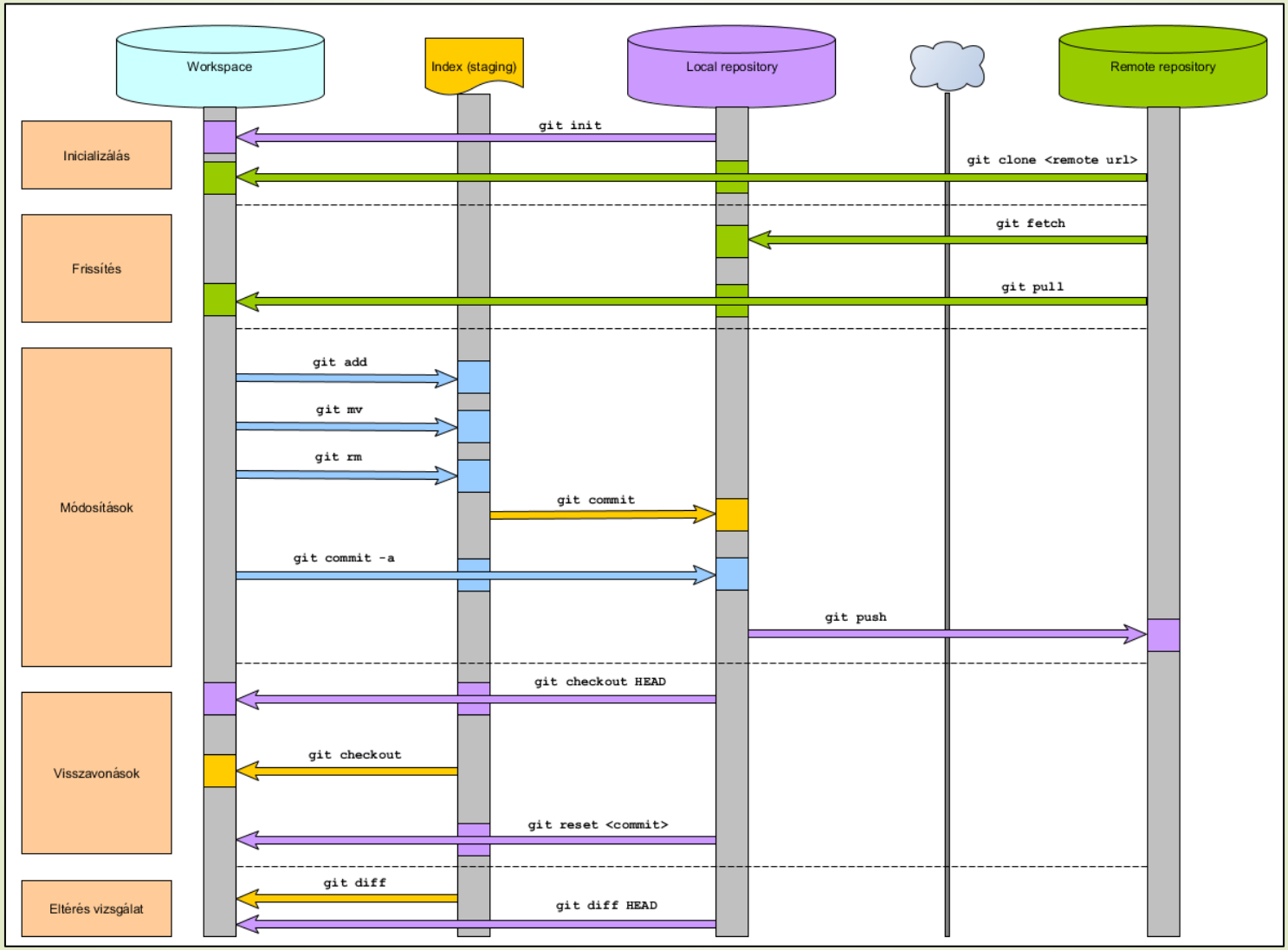
```
<<<<<<< HEAD
Az aktuális, jelen esetben master ágon lévő ütköző
tartalom
=======
Az egyesíteni kívánt, new-branch ágon lévő ütköző
tartalom
>>>>>>> new-branch
```

  - A fejlesztő feladata eldönteni, hogy a két lehetőség közül melyiket kívánja megtartani, esetleg a kettő vegyes megoldását szükséges alkalmazni.
  - A feloldott fájlokat a *staging area*-hoz kell adni (`git add`), majd a változásokat beküldeni (`git commit`).

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: alapvető konzolos utasítások áttekintése



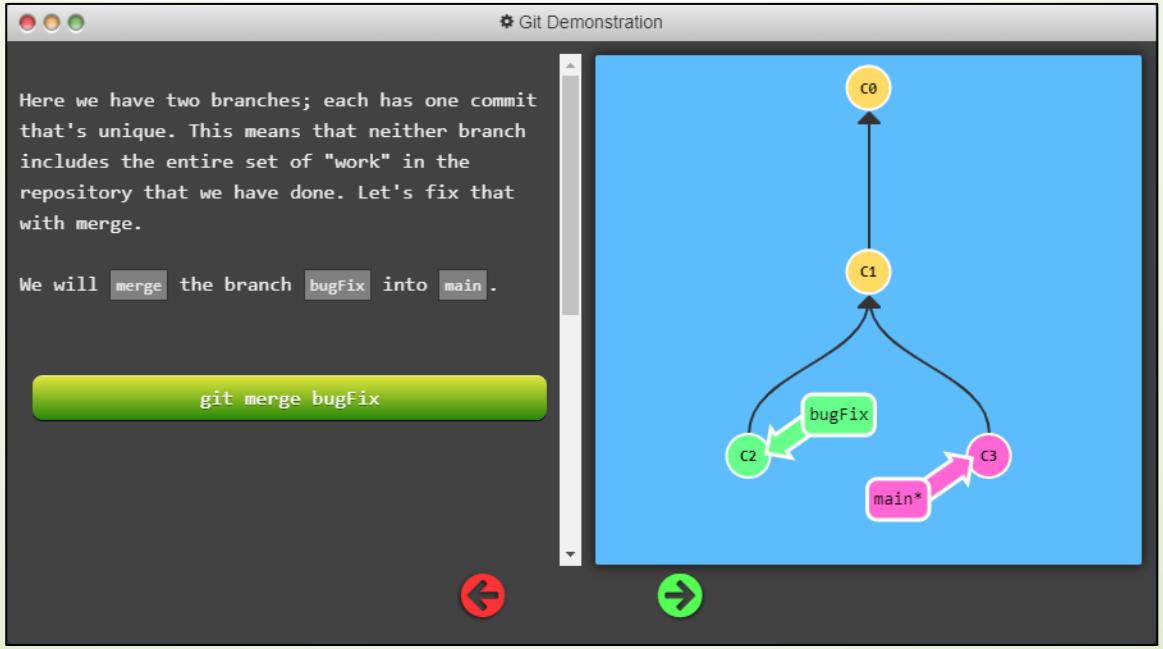
Forrás: Nagy Krisztián (ELTE IK)

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Online eszközök a tanuláshoz

- Több online eszköz is elérhető, amelyekkel vizualizáció mellett, kitűzött feladatok teljesítésével vagy szabadon gyakorolva fejleszthetőek a Git használati ismeretek. Pl.:
  - https://git-school.github.io/visualizing-git/
  - https://learngitbranching.js.org/

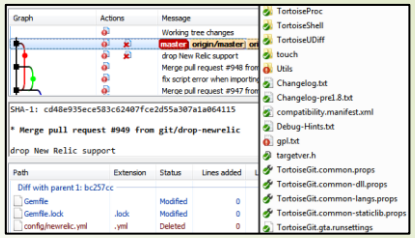


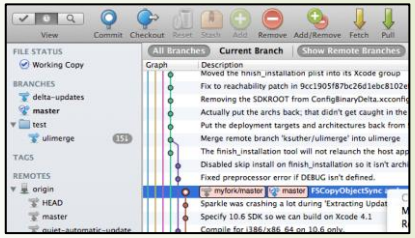
Verziókövető rendszerek

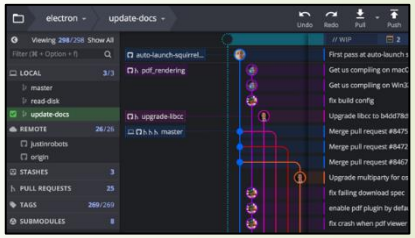
ELTE | IK
INFORMATIKAI KAR

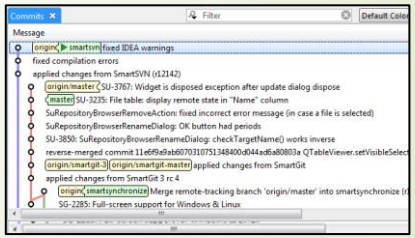
Git GUI kliensek

- TortoiseGit
  - Windows
  - *géptermi gépeken elérhető*
- SourceTree
  - Windows, Mac
- GitKraken
  - Linux, Windows, Mac
- SmartGit
  - Linux, Windows, Mac
- Továbbiak:
  - https://git-scm.com/downloads/guis





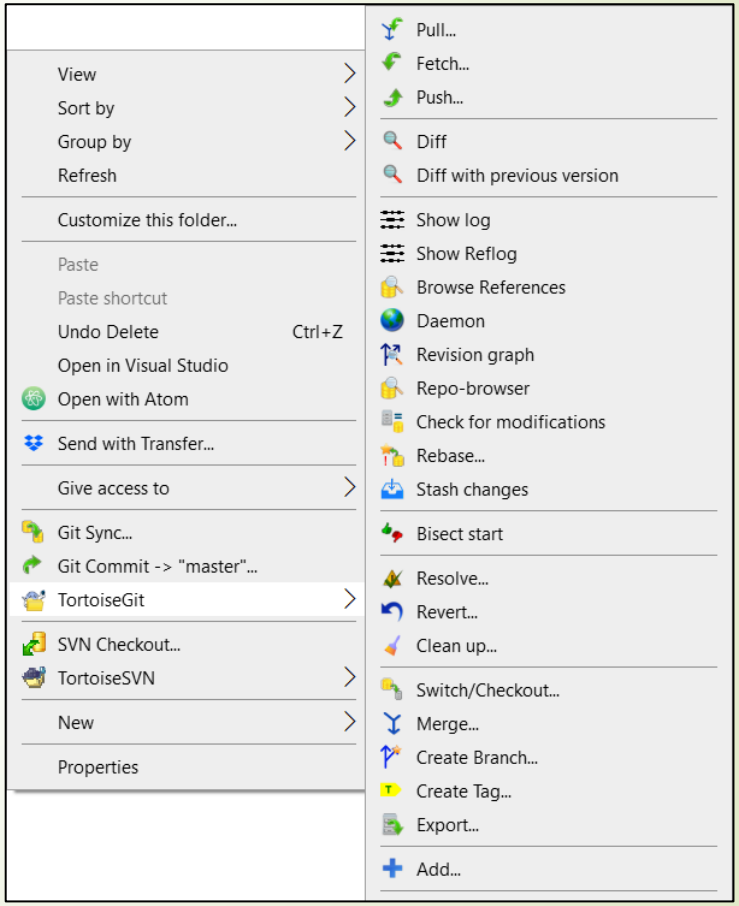




Verziókövető rendszerek

TortoiseGit használata

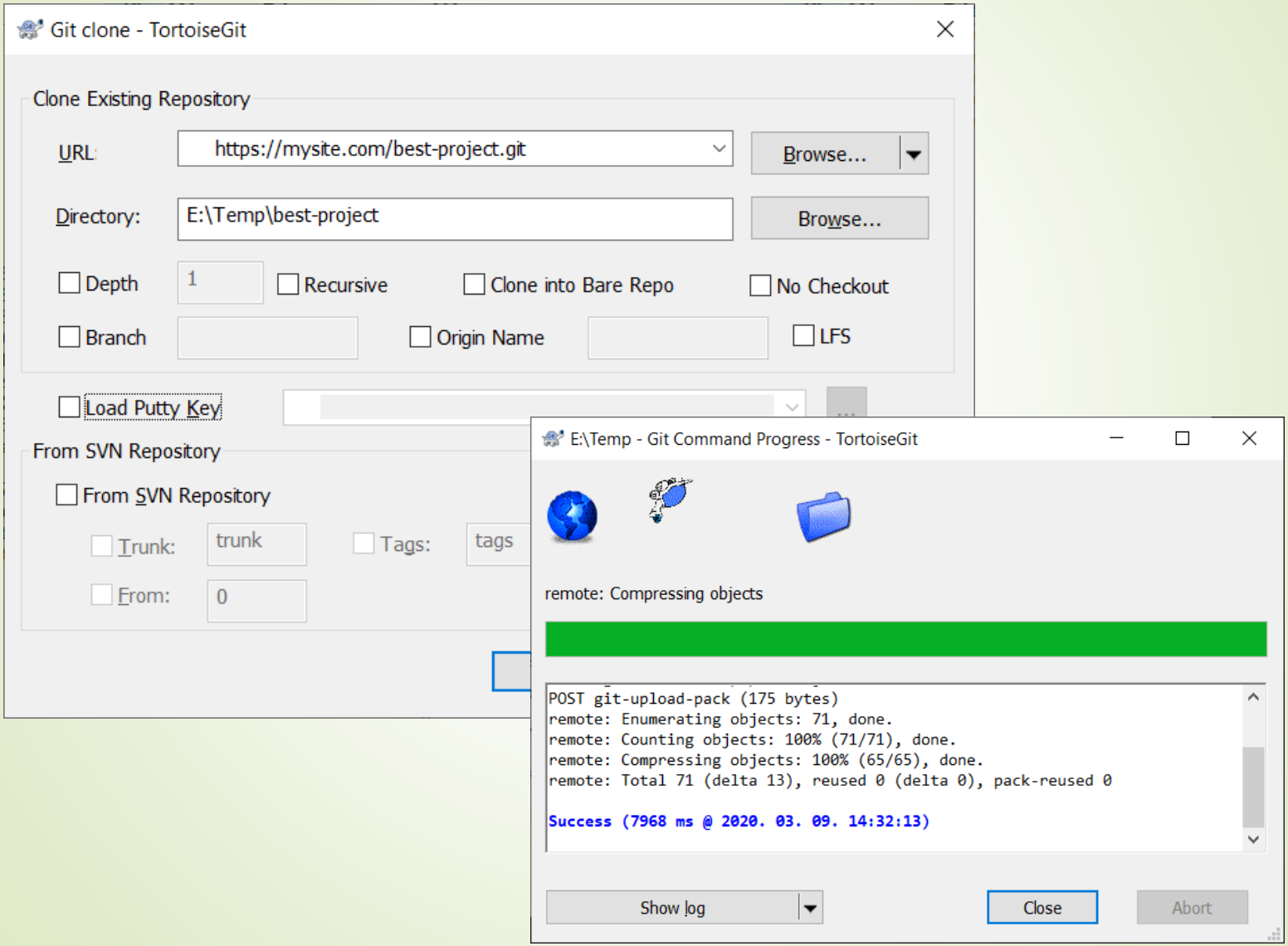
- A TortoiseGit egy Windows rendszekre fejlesztett, ingyenes, asztali grafikus Git kliens
  - Honlap, letöltés: https://tortoisegit.org/
  - Elérhető magyar nyelven is.
  - A Git-et nem tartalmazza, azt külön szükséges telepíteni.
- A TortoiseGit a fájlkezelő alkalmazások (pl. *File Explorer*) jobb egérkattintásra elérhető kontextus menüjébe épül be, innen hívhatóak meg a már megismert utasítások.



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

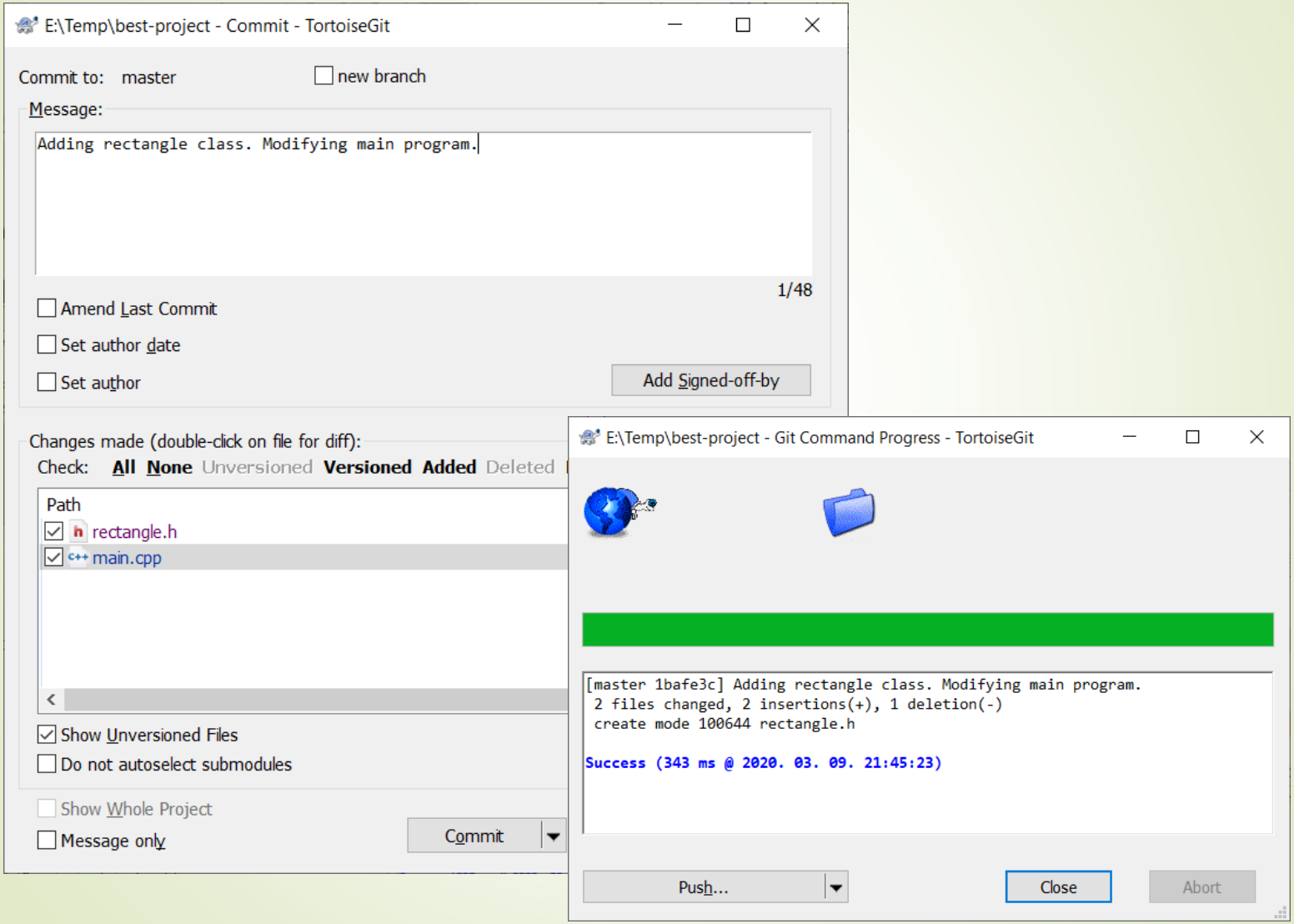
TortoiseGit: klónozás (*clone*)



Verziókövető rendszerek

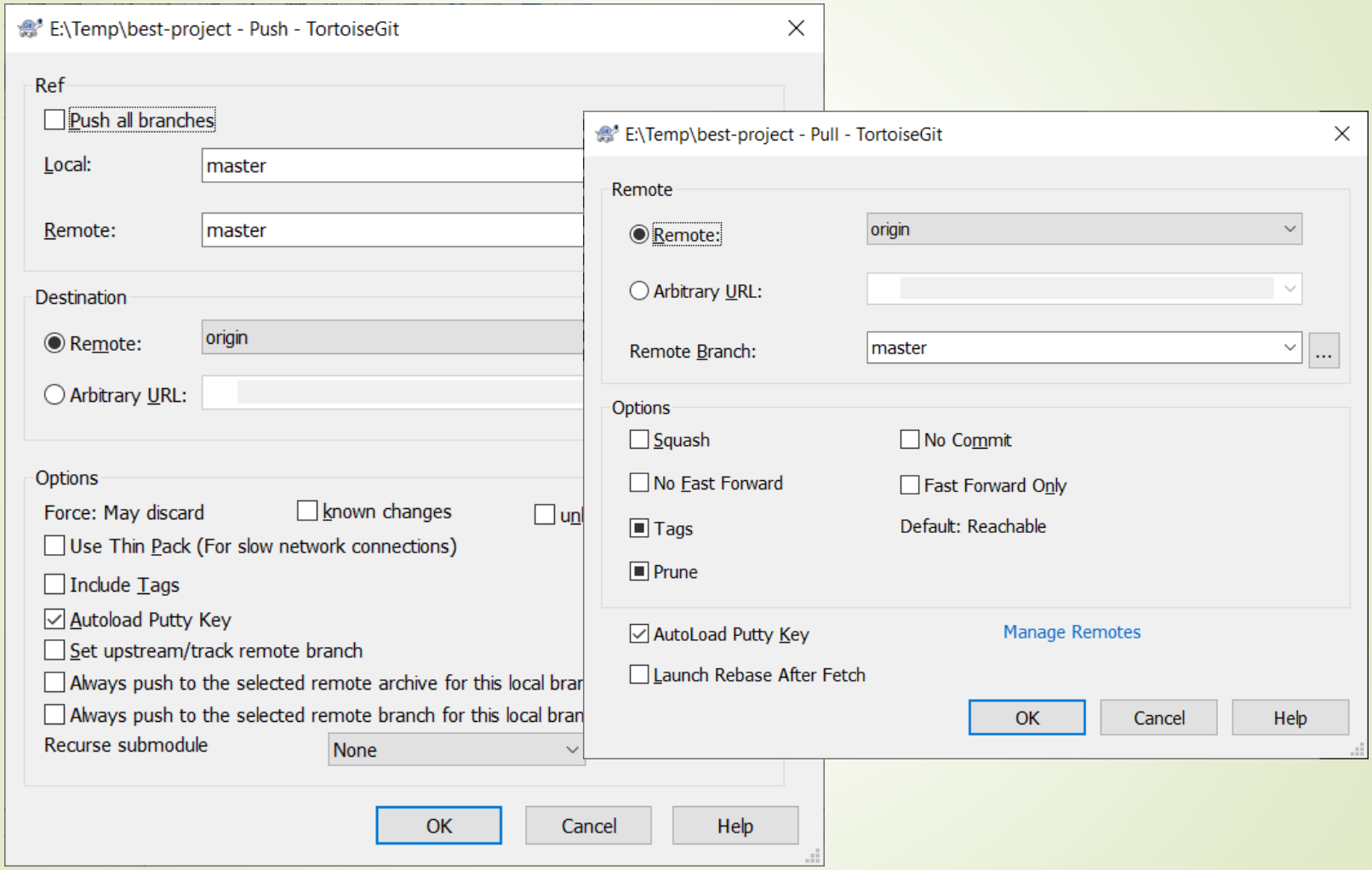
ELTE | IK
INFORMATIKAI KAR

TortoiseGit: új verzió beküldése (*commit*)



Verziókövető rendszerek

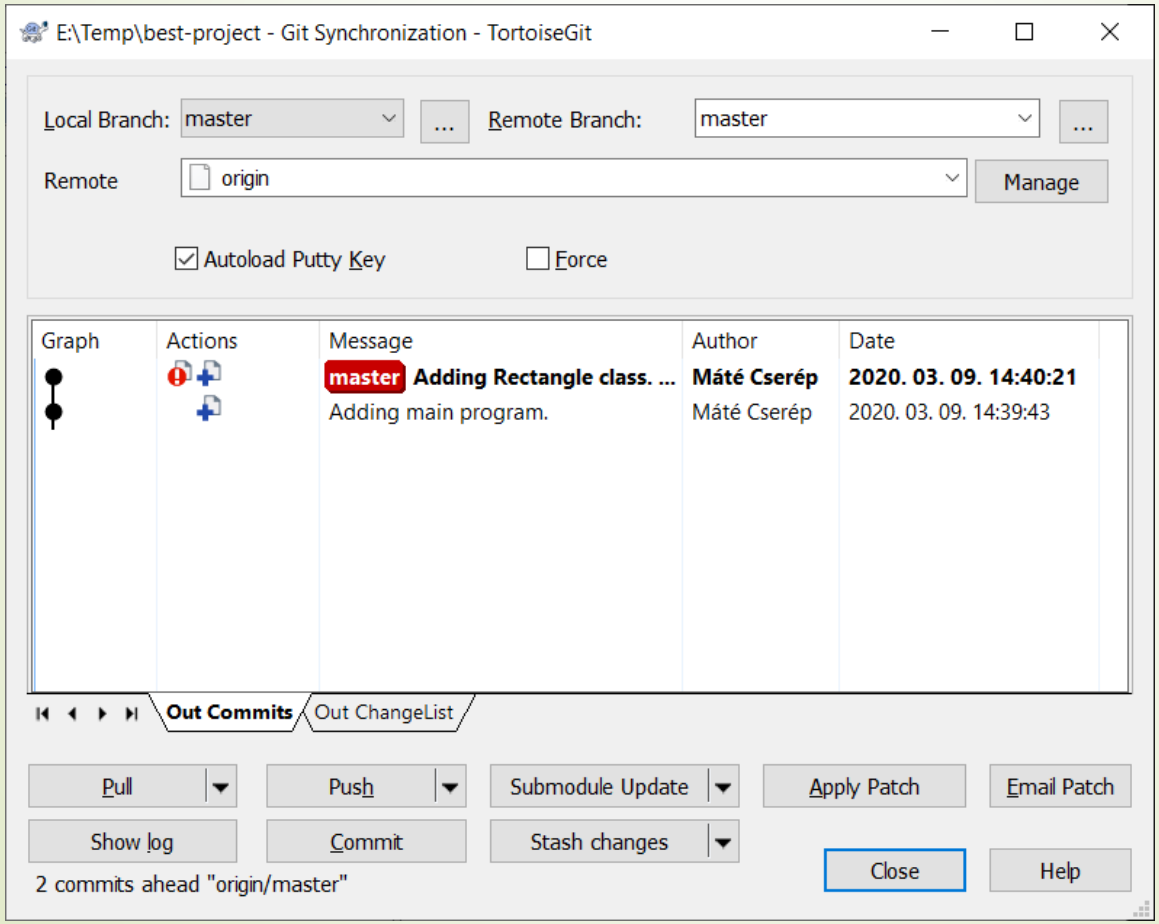
TortoiseGit: szinkronizálás (*push* & *pull*)



Verziókövető rendszerek

TortoiseGit: szinkronizálás (*sync*)

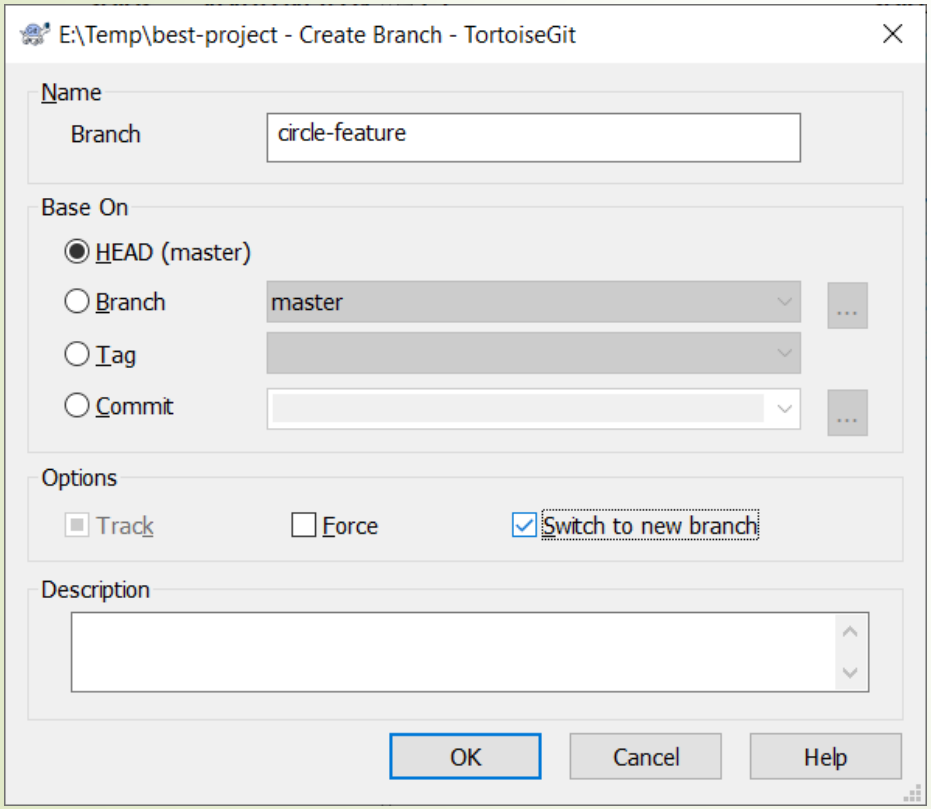
- A *Sync* funkcióval egyszerre érhetjük el a *pull* és a *push* opciókat is egy áttekintő felületen.

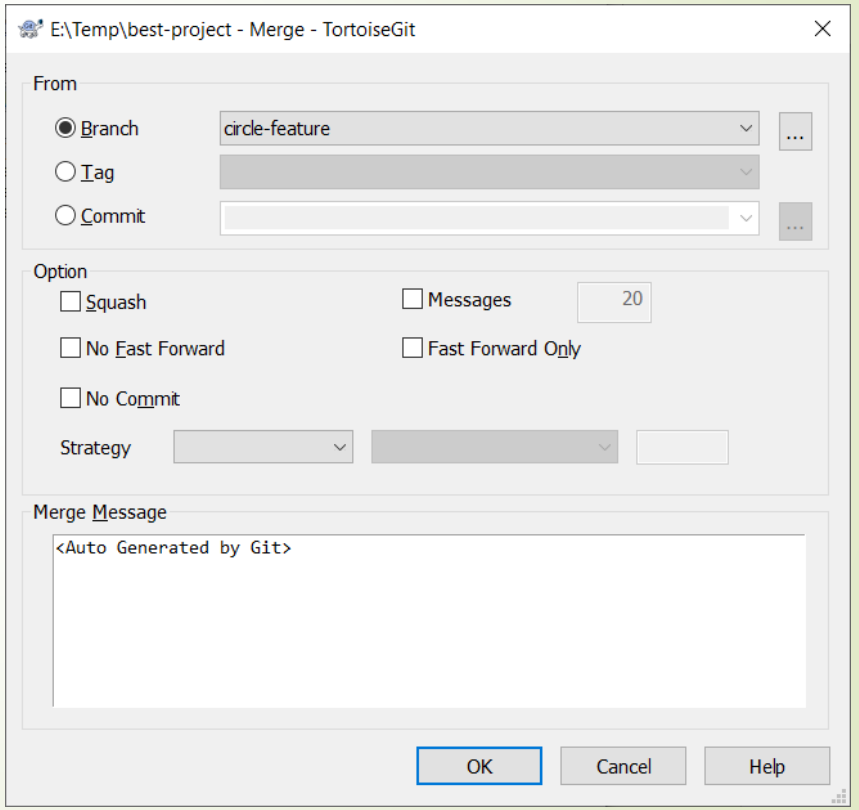


Verziókövető rendszerek

TortoiseGit: fejlesztési ágak (*branch*) kezelése

- Eleinte a grafikus felület jelentősen könnyebbé teheti a fejlesztési ágak létrehozását és összeillesztését.

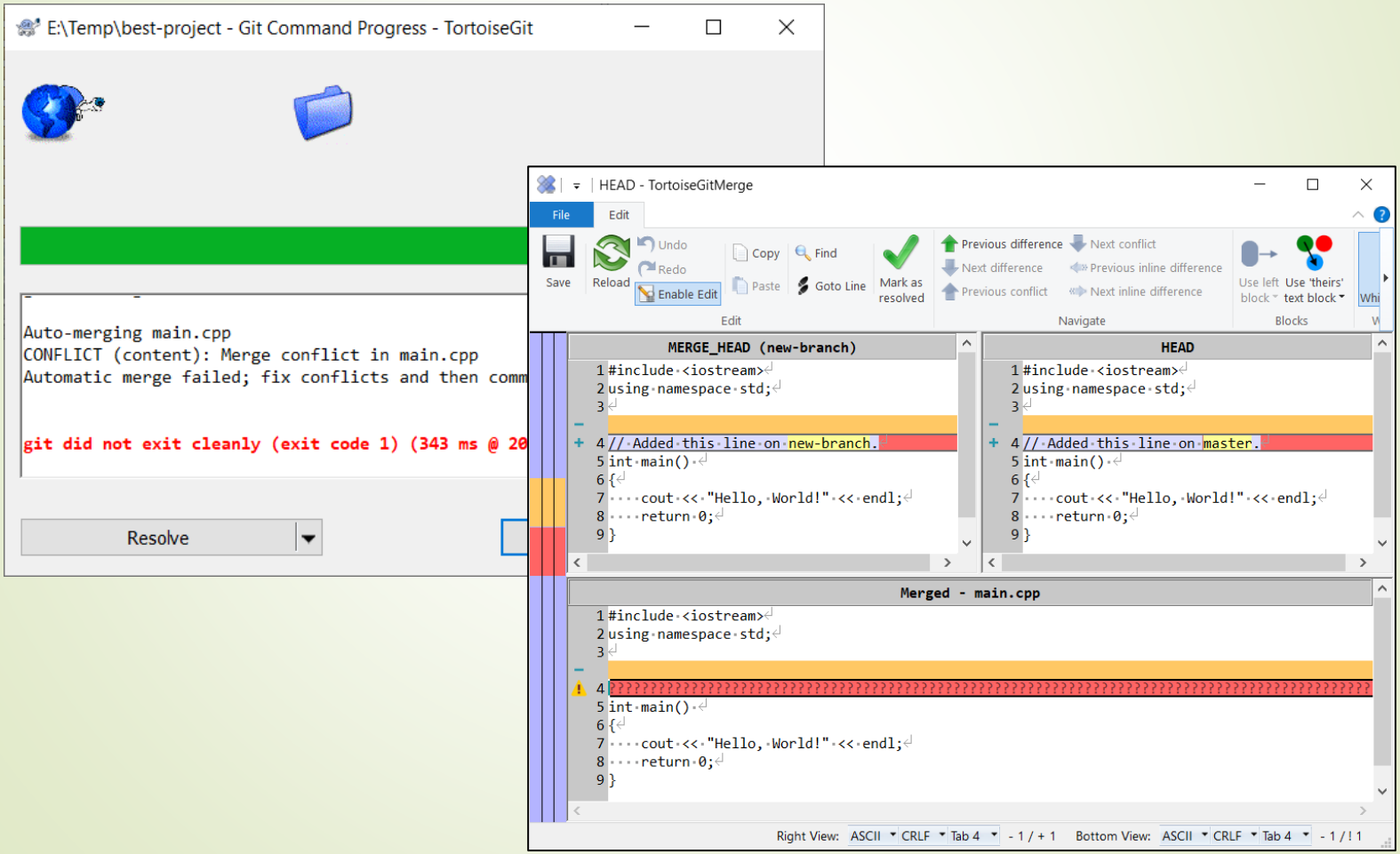




Verziókövető rendszerek

TortoiseGit: ütközések feloldása

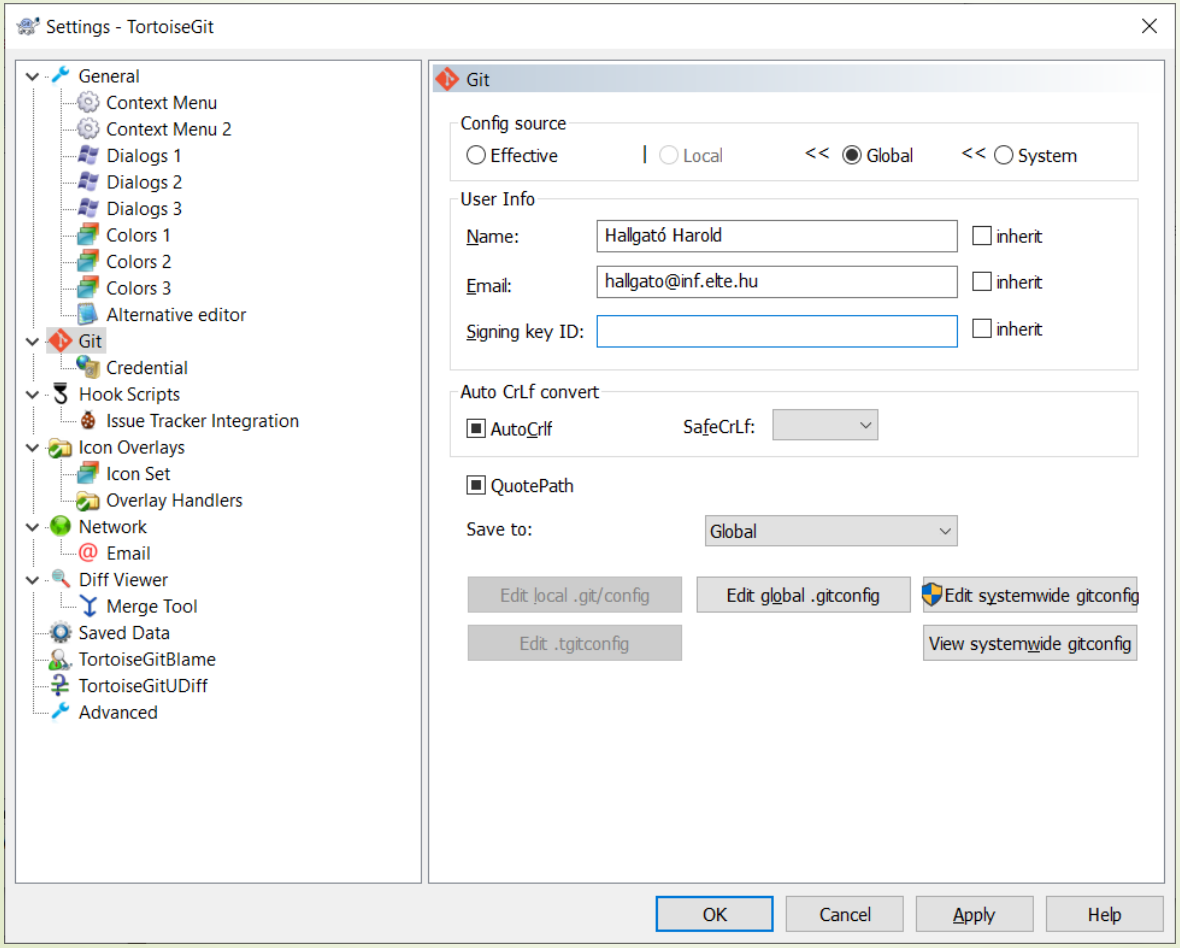
- Különösen igaz ez az egyesítéskor fellépő konfliktusok feloldására, ahol egy összevető nézet segíti a munkát.



Verziókövető rendszerek

TortoiseGit: beállítások

- *TortoiseGit -> Settings* menüpont alatt szerkeszthetjük a beállításokat is, pl. a felhasználó nevét és email címét.

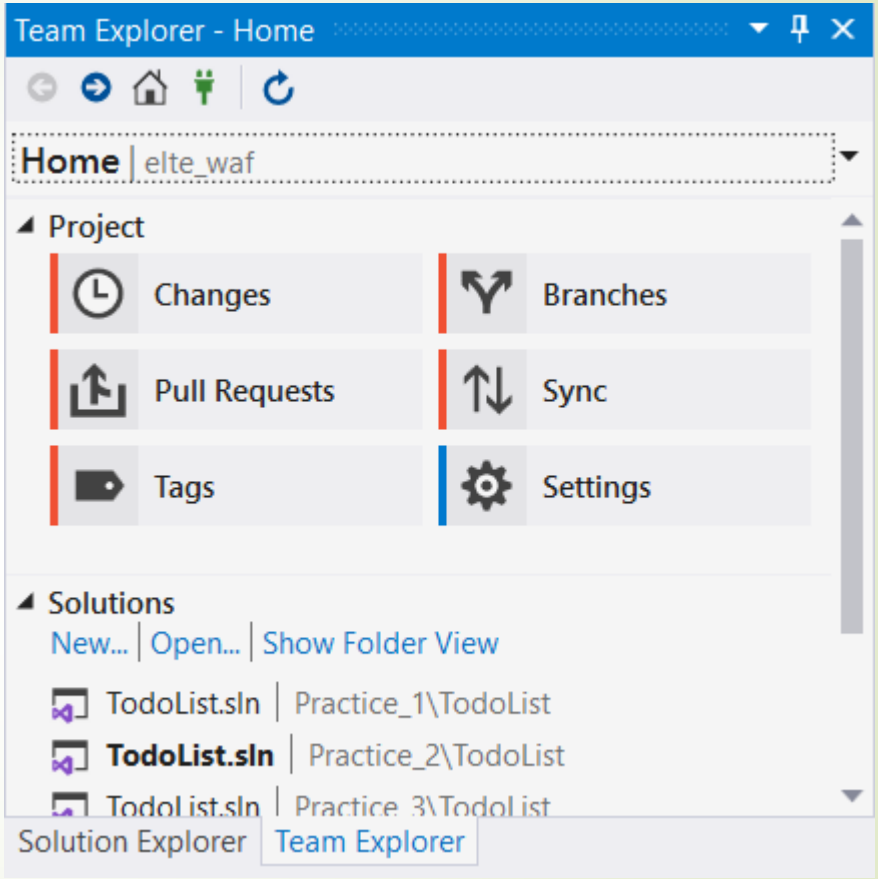


Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Támogatás integrált fejlesztőkörnyezetekben

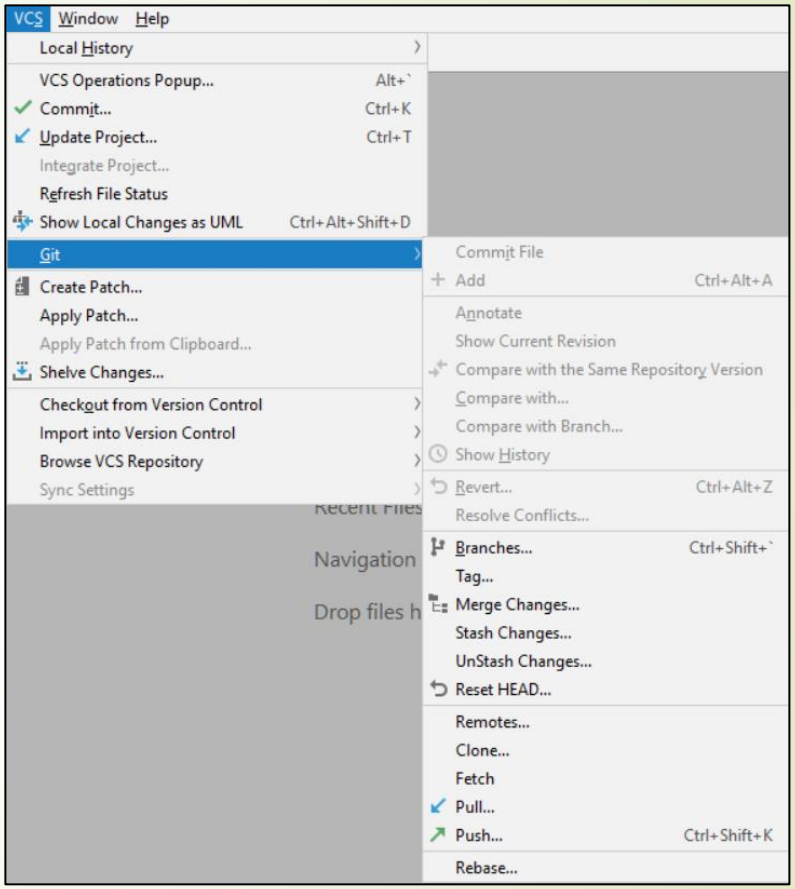
- A legtöbb modern integrált fejlesztőkörnyezet (IDE) beépített támogatást nyújt a verziókezelésre.
  - Részben éppen emiatt nevezzük integrált környezetnek őket.
  - Így nem szükséges különféle alkalmazások kontextusai között váltogatni.
- *Visual Studio* esetében a verziókezelő funkciókat a *Team Explorer* menüpont alatt találjuk.

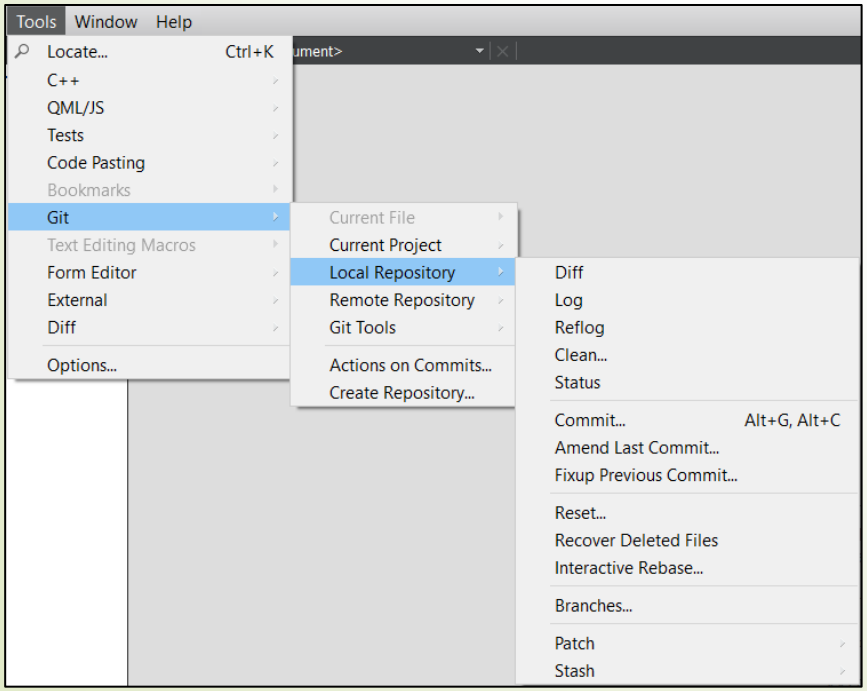


Verziókövető rendszerek

Támogatás integrált fejlesztőkörnyezetekben

- *JetBrains CLion* a funkciókat a VCS (*version control system*) menüpont alatt találjuk.
- *QtCreator* programban ugyanez a *Tools -> Git* menüpont alatt érhető el.





Verziókövető rendszerek

Mely fájlokat érdemes verziókezelés alá vonni?

- A Git verziókezelő rendszer a szöveges állományok, így tipikusan a forráskód fájlok változáskezelésében hatékony. Elsődlegesen a projekt forráskódját érdemes benne elhelyezni.
- Egy általános szoftver projekt esetén nem érdemes verziókezelés alá vonni:
  - fordítás során előálló köztes tárgykódot vagy a végső bináris állományokat, mert újból előállíthatóak, folyamatosan változnak és ütközéseket okoznak.
  - fejlesztő eszközök személyes beállításait (pl. Visual Studio esetén a *.vs/* vagy Netbeans esetén a *nbproject/private/* könyvtárak), amelyek felhasználónként eltérőek.
  - nagy méretű bináris állományokat (pl. videók, nagy méretű képek), amelyek kezelésében a Git nem hatékony. Bár a Git tárolók mérete jól skálázható, egy könnyen kezelhető *repository* mérete az 1-2 GB-os méretet nem haladja meg.

Verziókövető rendszerek

Fájlok kivonása a verziókezelés alól

- Verziókezelés alá kizárólag azok a fájlok kerülnek, amelyeket kifejezetten hozzáadunk (`git add`).
- Az esetleges véletlen hozzáadást elkerülendő megjelölhetjük azokat a fájlokat és könyvtárakat, amelyeket mellőzni szeretnénk.
  - A mellőzendő állományokat egy speciális `.gitignore` elnevezésű állományban adhatjuk meg
  - Ezt a fájlt érdemes verziókezelés alá is vonni, hogy a fejlesztők között egységes legyen a beállítás.
- A `.gitignore` minden sorában egy illeszkedési mintát adhatunk meg, hogy mely fájlokat akarjuk kizárni a verziókezelés alól.
  - A beállítás tranzitívan vonatkozik az alkönyvtárakra is, így gyakran elegendő lehet egyetlen `.gitignore` fájl létrehozása a projekt gyökér- könyvtárában.

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: .gitignore minták

Minta	Illeszkedés	Leírás
program.exe	/program.exe /bin/program.exe	Minden **program.exe** fájlra illeszkedés.
/program.exe	/program.exe **de nem:** /bin/program.exe	Az adott könyvtárszinten lévő **program.exe** fájlra illeszkedés.
*.exe	/program.exe /bin/main.exe	Minden **exe** kiterjesztésű fájlra illeszkedés.
bin/	/bin/ /project/bin/ **de nem:** /logs/bin (fájl)	Minden **bin** könyvtárra illeszkedés (de **bin** nevű fájlokra nem!)
/bin/*.exe	/bin/program.exe /bin/main.exe **de nem:** /bin/program.dll /project/bin/program.exe	Az adott könyvtárban lévő **bin** könyvtárban lévő összes **exe** kiterjesztésű fájlra illeszkedés.

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: .gitignore minták

Minta	Illeszkedés	Leírás
*.log !important.log	/application.log /logs/application.log **de nem:** /logs/important.log	Az összes **log** kiterjesztésű fájlra illeszkedés, kivéve, ha a fájl neve **important.log**.
logs/**/*.log	/logs/application.log /logs/runtime/main/01.log /project/logs/deploy.log **de nem:** /runtime.log	Minden olyan **log** kiterjesztésű fájlra illeszkedés, amely egy **logs** könyvtár alatt helyezkedik el (tetszőleges mélységben).

- Számos programozási nyelvhez és IDE-hez érhető el általános esetekre megfelelő `.gitignore` állomány a *GitHub*-on, ezekből vagy ezek kombinációjából jó ötlet kiindulni.
  - URL: https://github.com/github/gitignore

Verziókövető rendszerek

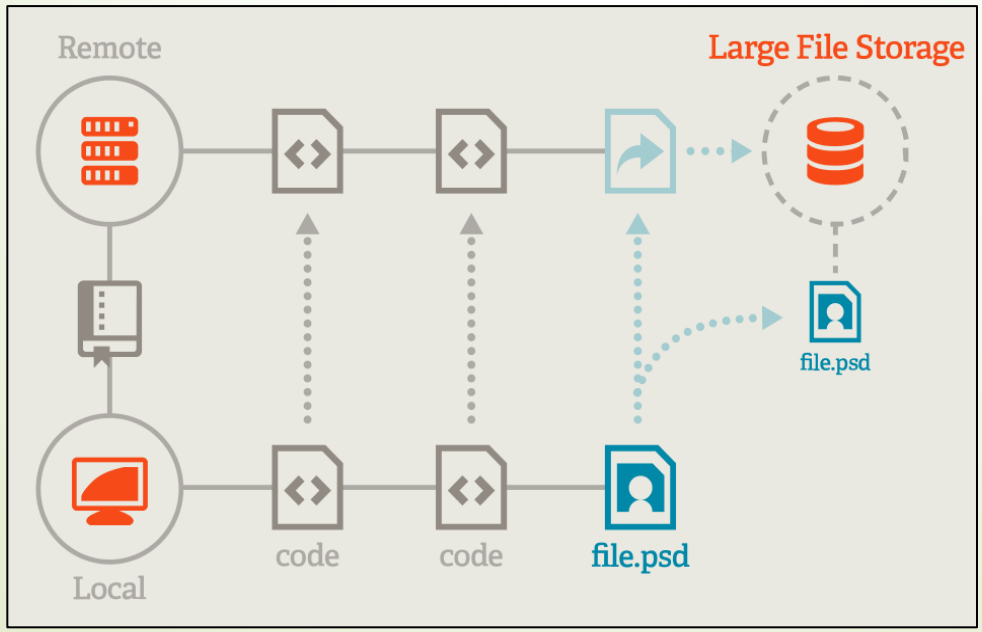
Nagy erőforrás állományok verziókezelése

- A nagy méretű videó, kép és hang erőforrás állományok hatékony kezelése körültekintést igényel.
  - A nagy méretű bináris állományok változásainak kezelésében a Git kevésbé hatékony.
  - Jelentősen megnöveli a tároló helyi másolatának lekéréséhez szükséges hálózati forgalmat (*git clone*).
  - Egy fejlesztőcsapatban a programozóknak nem feltétlenül van szükségük a fejlesztéshez a designerek által készített *assetekre*.
- Ezért a nagy méretű bináris erőforrás állományokat még akkor sem feltétlenül érdemes a Git tárolóban elhelyezni, ha amúgy ritkán változnak.

Verziókövető rendszerek

Git Large File Storage

- A nagy méretű bináris állományok kezelése a *Git Large File Storage* (Git LFS) segítéségével oldható meg.
  - A nagy méretű bináris állományokat egy hivatkozással helyettesíti és magukat a fájlokat egy másik (akár távoli) szerveren tárolja.
  - Így a Git tárolónk mérete kezelhető marad.



Forrás: git-lfs.github.com

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

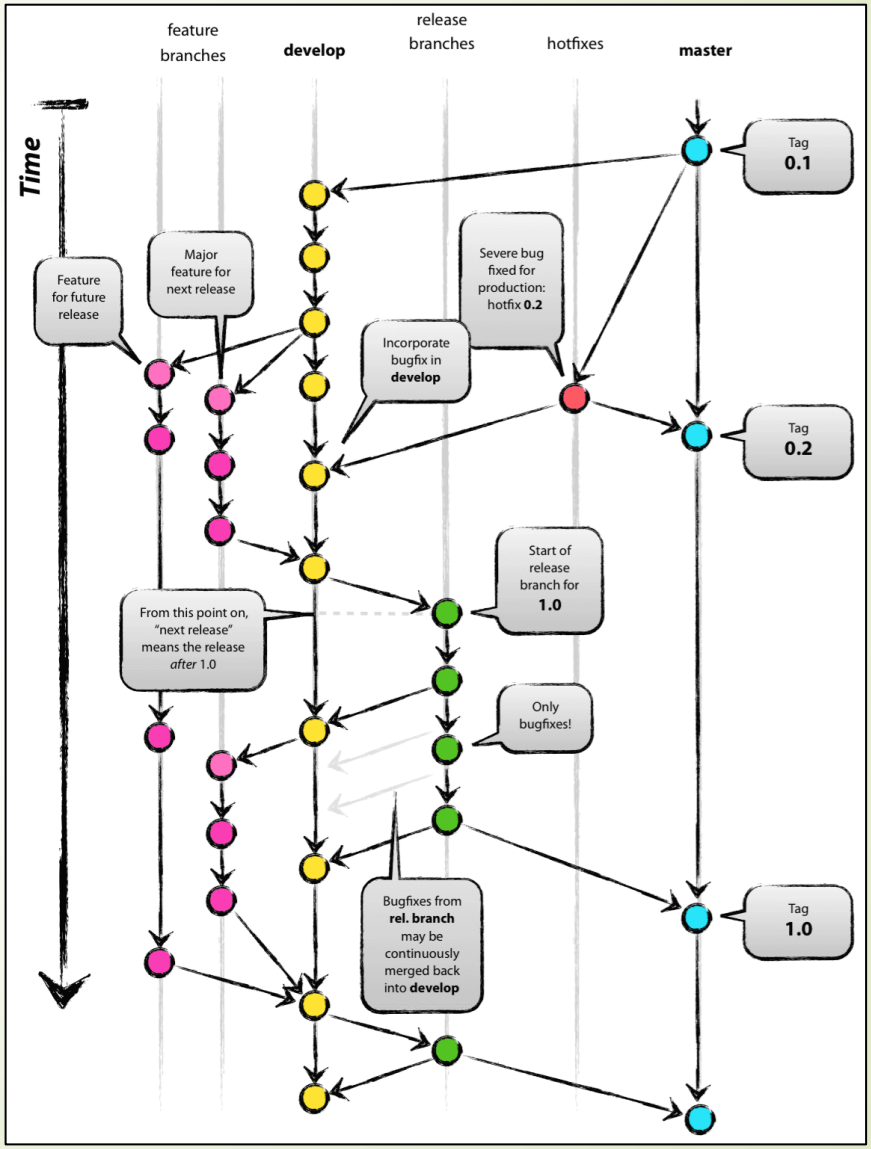
Git Large File Storage

- A szofttech.inf.elte.hu GitLab szerver támogatja a Git LFS-t.
- Használatához csak a kliens gépekre (saját gépetek) is szükséges a Git LFS telepítése.
  - Letöltés: https://git-lfs.github.com/
  - Használat: https://docs.gitlab.com/ee/administration/lfs/manage_large_binaries_with_git_lfs.html

Verziókövető rendszerek

GitFlow feature branching model

- Nagy, több támogatott kiadással rendelkező szoftverekhez ajánlott
- Fő fejlesztési ágak:
  - master
  - develop
- Támogató ágak:
  - feature branches
  - release branches
  - hotfix branches



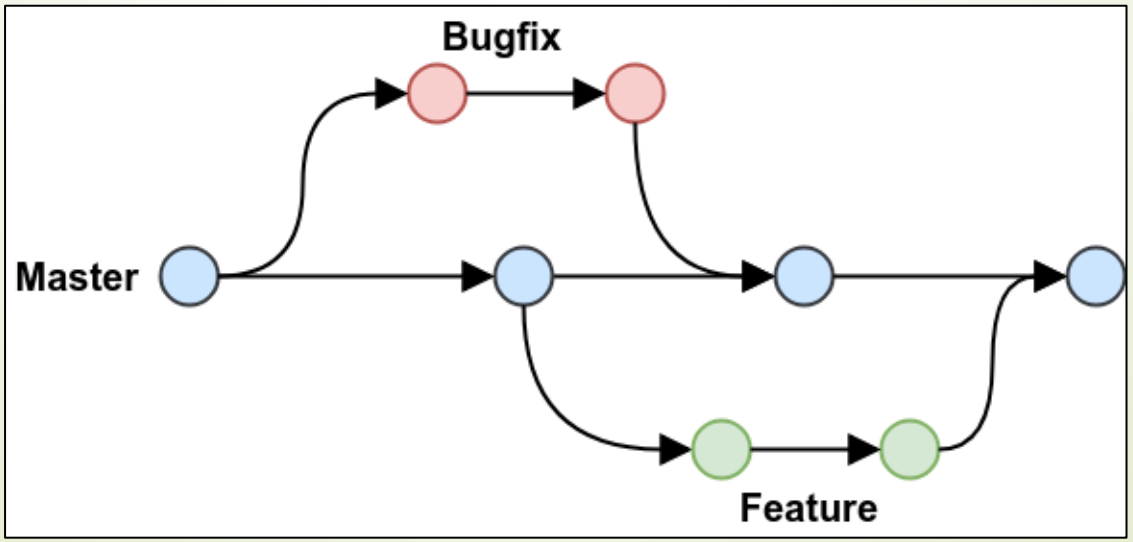
Forrás: nvie.com

Verziókövető rendszerek

GitHub Flow feature branching model

- Jellemzően csak egyetlen támogatott, gyors kiadási ciklussal rendelkező szoftverekhez ajánlott.
  - Ami a fő fejlesztési ágon van, az kiadható!

- Fő fejlesztési ág: *master*
- Támogató fejlesztési ágak: *feature* és *bugfix*

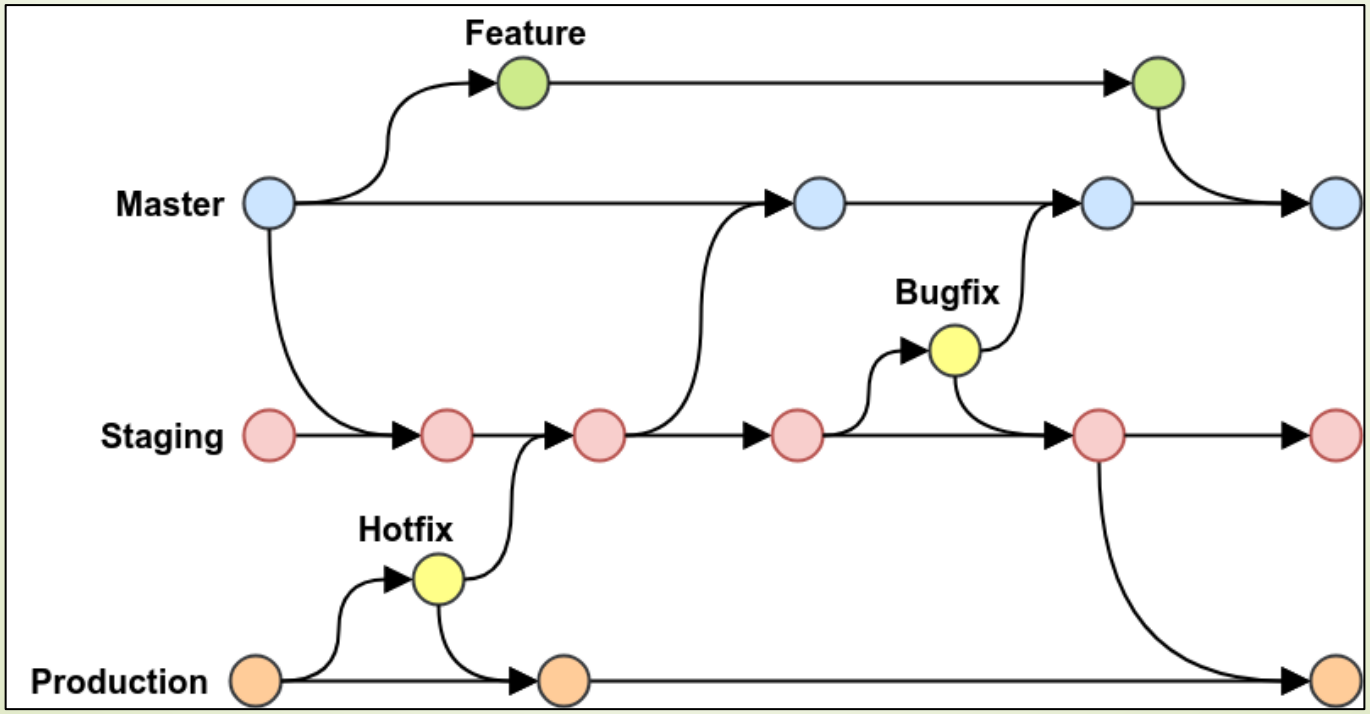


Forrás: blog.programster.org

Verziókövető rendszerek

GitLab Flow feature branching model

- Nem biztos, hogy minden esetben kiadható a szoftver új verziója, ami a fő fejlesztési ágra került.
  - Fő fejlesztési ág: *master*
  - Kiadási ág: *production*
  - Tesztelési (előzetes kiadási) ág: *staging* (opcionális)



Forrás: blog.programster.org

Szoftvertechnológia

Verziókövető rendszerek

C szakirány

Dr. Cserép Máté

ELTE Informatikai Kar

2025.

Verziókövető rendszerek

Történeti háttér

- A szoftverek méretének és komplexitásának növekedésével létrejött szoftverkrízis következményeként megnövekedett:
  - a programok forráskódjának mérete,
  - a szoftverprojektek megvalósításához szükséges idő,
  - és szükséges programozói erőforrás.
- A szoftveripar fejlődésével egyre több alkalmazás készült
  - a fejlesztések életciklusa gyakran nem ért véget a program első publikus verziójának kiadásával,
  - karbantartási és további fejlesztési fázisok követték.

- **A szoftverprojektek méretben, komplexitásban, időben és a résztvevő fejlesztők számában is növekedni kezdtek.**

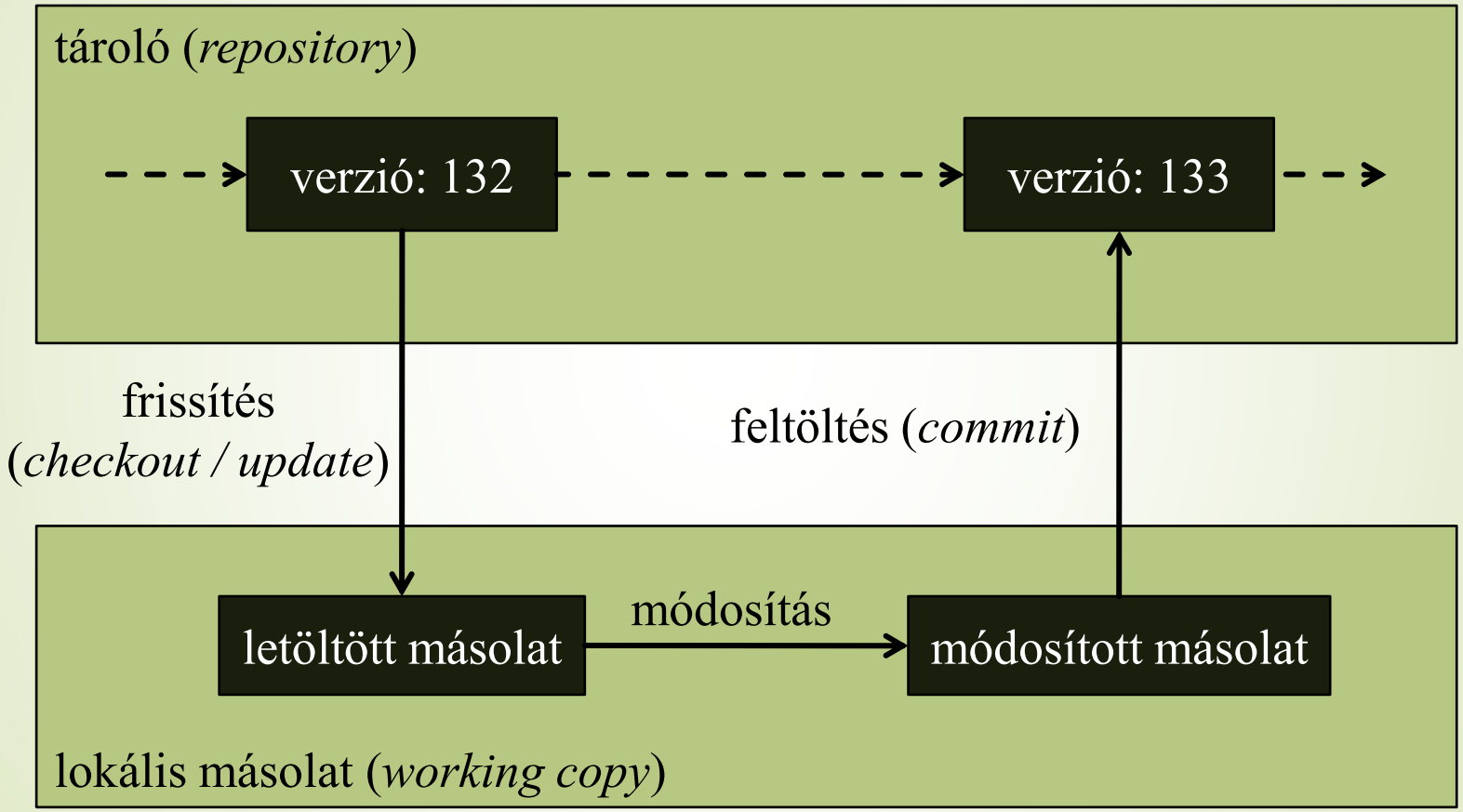
Verziókövető rendszerek

Funkcionalitás

- Mivel az implementáció tehát több lépésben, és sokszor párhuzamosan zajlik, szükséges, hogy az egyes programállapotok, jól követhetőek legyenek, ezt a feladatot a *verziókövető rendszer*ek (*revision control system*) látják el
  - pl. *CVS*, *Apache Subversion* (*SVN*), *Mercurial*, *Git*
  - egy közös tárolóban (*repository*) tartják kódokat
  - ezt a fejlesztők lemásolják egy helyi munkakönyvtárba, és amelyben dolgoznak (*working copy*)
  - a módosításokat visszatöltik a központi tárolóba (*commit*)
  - a munkakönyvtárakat az első létrehozás (*checkout*) után folyamatosan frissíteni kell (*update*)

Verziókövető rendszerek

Funkcionalitás



Verziókövető rendszerek

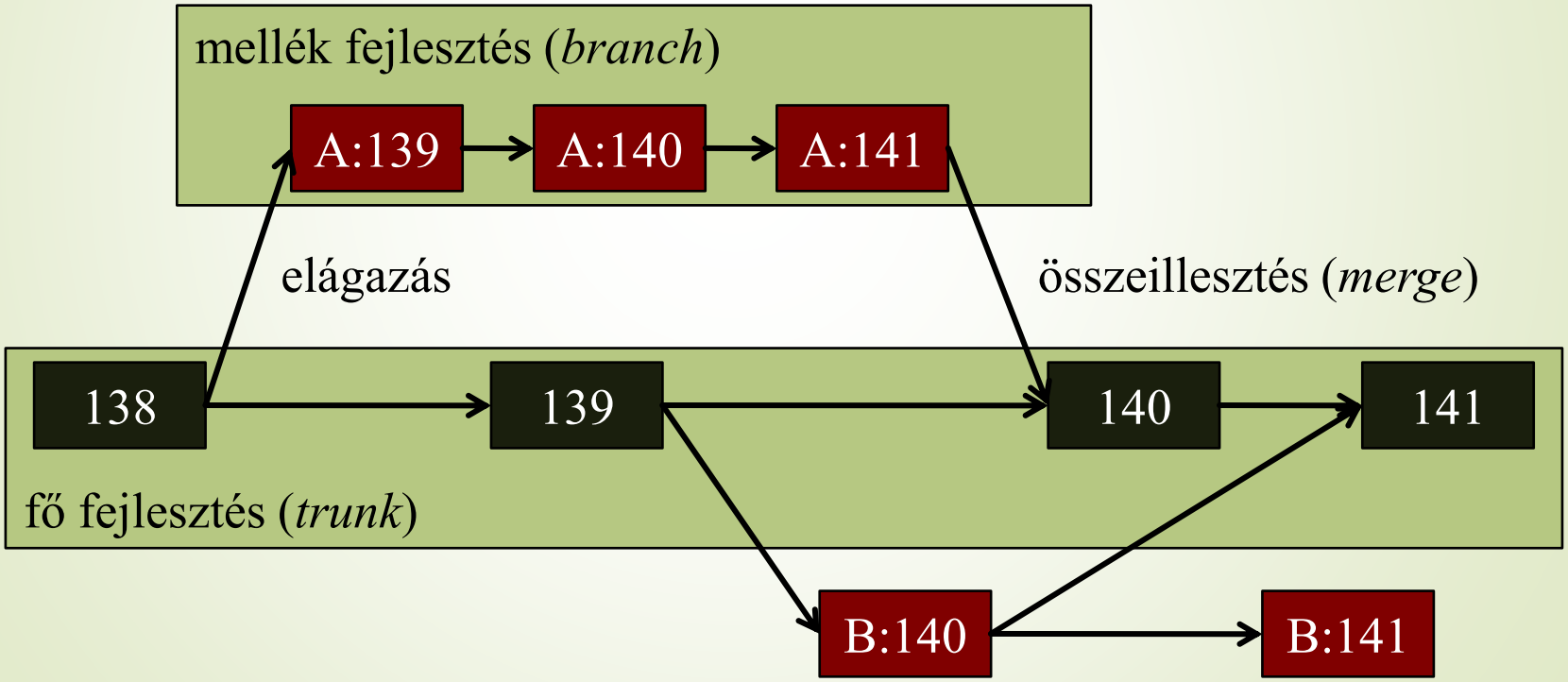
Funkcionalitás

- A verziókövető rendszerek lehetővé teszik:
  - az összes eddig változat (*revision*) eltárolását, illetve annak letöltési lehetőségét
  - a fő fejlesztési vonal (*baseline*, *master* vagy *trunk*) és a legfrissebb változat (*head*) elérését, új változat feltöltését annak dokumentálásával
  - az egyes változatok közötti különbségek nyilvántartását fájlonként és tartalmanként (akár karakterek szintjén)
  - változtatások visszavonását, korábbi változatra visszatérést
  - konfliktust okozó módosítások ellenőrzését, illetve megoldását (*resolve*)

Verziókövető rendszerek

Funkcionalitás

- a folyamat elágazását, és ezáltal újabb fejlesztési folyamatok létrehozását, amelyek a fő vonal mellett futnak (*branch*), valamint az ágak összeillesztését (*merge*)



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Funkcionalitás

- az összeillesztés rendszerint utólagos manuális korrekciót igényel
- az összeillesztésnek rendszerint automatikusan illeszti a módosított tartalmakat kódelemzést használva, ez lehet 2 pontos (*two-way*), amikor csak a két módosítást vizsgálja, vagy 3 pontos, amikor az eredeti fájlt is

- programrészek zárolását (*lock*), hogy a konfliktusok kizárhatóak legyenek
- adott verzió, mint pillanatkép (*snapshot*) rögzítése (*tag*), amelyhez a hozzáférés publikus
- feltöltések atomi műveletként történő kezelését (pl. megszakadó feltöltés esetén visszavonás)

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Lokális verziókövető rendszerek (1. generáció)

- Forráskód változásainak követése, a szoftver funkcióinak különböző kombinációjával készült kiadások kezelése
  - lokális tároló (de többen is elérhetik pl. *mainframe* esetén)
  - fájl alapú műveletvégzés (1 verzió 1 fájl változásai)
  - konkurenciakezelés kizárólagos zárak által

- Az 1970-es években lefektetésre kerültek az elméleti alapok
  - Source Code Control System (SCCS) – 1972
  - Revision Control System (RCS) - 1982

Verziókövető rendszerek

Centralizált verziókövető rendszerek (2. generáció)

- Több fejlesztő általi párhuzamos szoftverfejlesztés támogatásának előtérbe kerülésre
  - centralizált modellt megtartva, de kliens-szerver architektúra
  - fájlhalmaz alapú műveletek (1 verzió több fájl változásai)
  - konkurenciakezelés jellemzően beküldés előtti egyesítéssel (*merge before commit*)
- Az 1990-es évektől terjedtek el:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
  - SourceSafe, Perforce, Team Foundation Server, stb.
- *Hátrány: a szerver kitüntetett szerepe (pl. meghibásodás), továbbá a verziókezeléshez hálózati kapcsolat szükségeltetik*

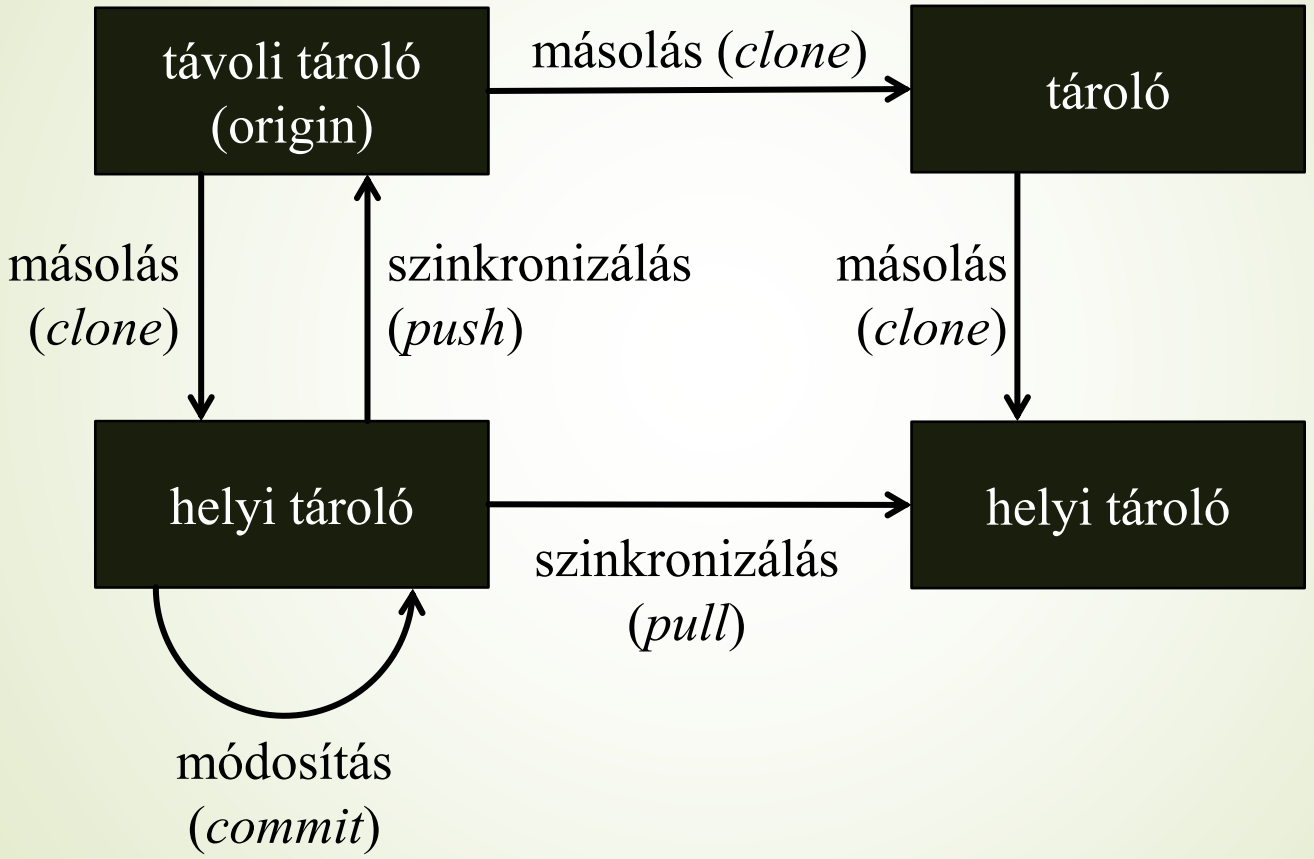
Verziókövető rendszerek

Elosztott verziókövető rendszerek (3. generáció)

- A klasszikus verziókezelő műveletekről leválasztásra kerül a hálózati kommunikáció, azok a felhasználó által kezdeményezhető önálló tevékenységekként jelennek meg
  - decentralizált, elosztott hálózati modell
  - minden kliens rendelkezik a teljes tárolóval és verziótörténettel
  - a revíziókezelő eszköz műveletei lokálisan, a kliens tárolóján történnek
  - a kommunikáció *peer-to-peer* elven történik, de kitüntetett (mindenki által ismert) szerverek felállítására van lehetőség
  - konkurenciakezelés jellemzően beküldés utáni egyesítéssel (*commit before merge*)
- A 2000-es évek első felében jelent meg:
  - Monotone, Darcs, Git, Mercurial, Bazaar, stb.

Verziókövető rendszerek

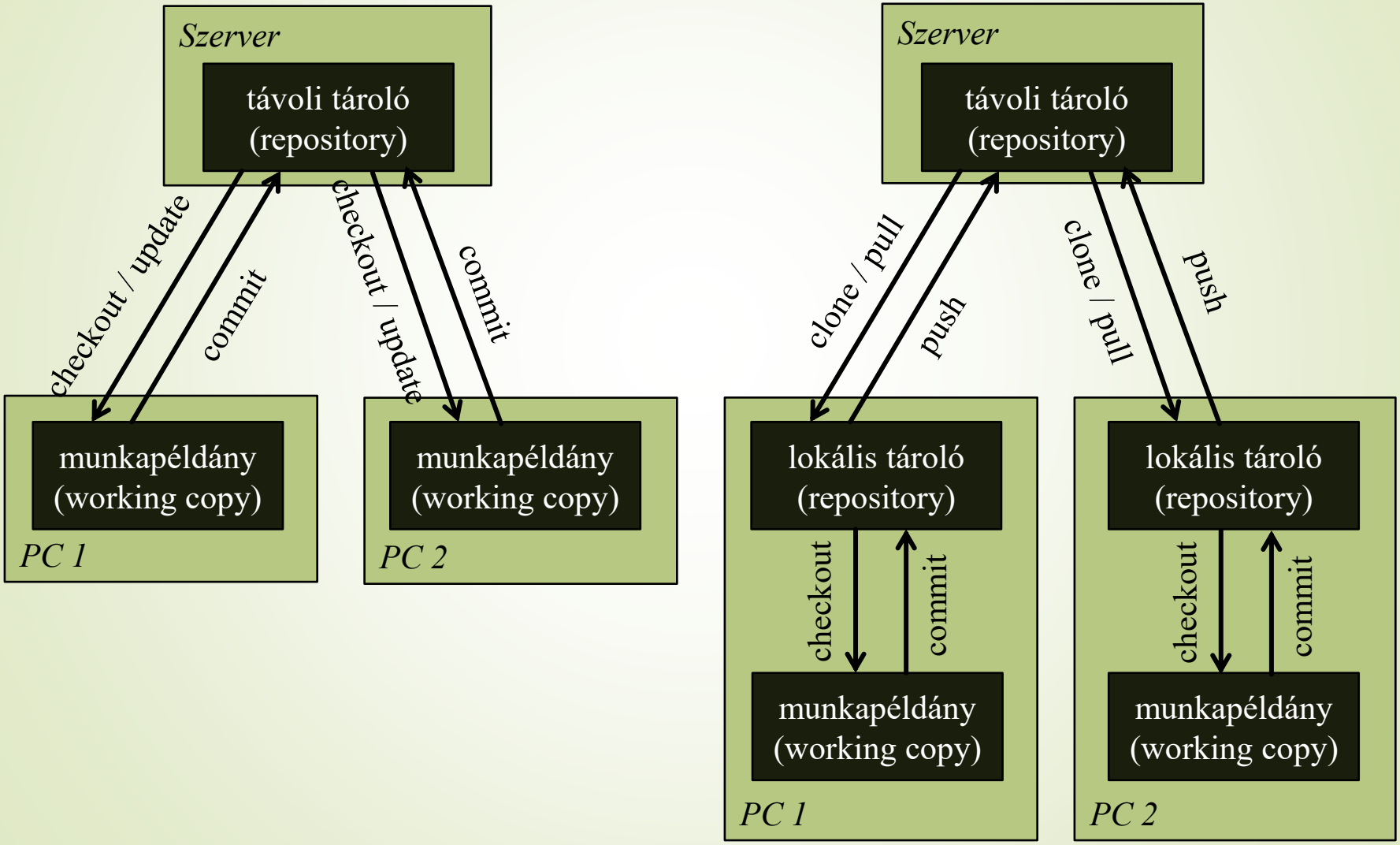
Elosztott verziókövető rendszerek (3. generáció)



Verziókövető rendszerek

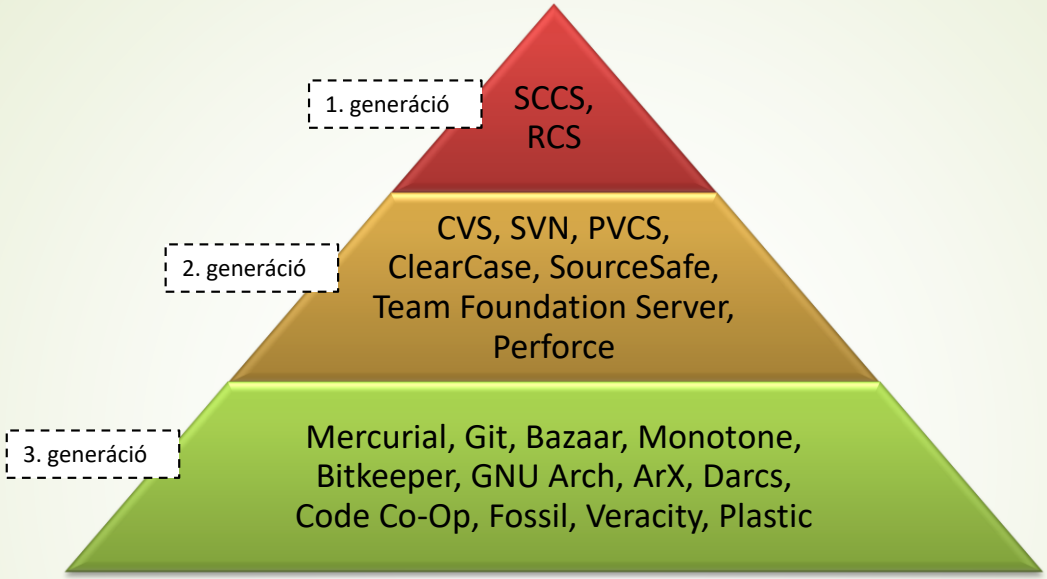
ELTE | IK
INFORMATIKAI KAR

Centralizált és elosztott verziókezelő rendszerek



Verziókövető rendszerek

Generációs modell



Generáció	Hálózati modell	Műveletvégzés	Konkurenciakezelés
Első	Lokális	Fájlonként (non-atomic commits)	Kizárólóagos zárak (exclusive locks)
Második	Központosított	Fájlhalmaz (atomic commits)	Egyesítés beküldés előtt (merge before commit)
Harmadik	Elosztott	Fájlhalmaz (atomic commits)	Beküldés egyesítés előtt (commit before merge)

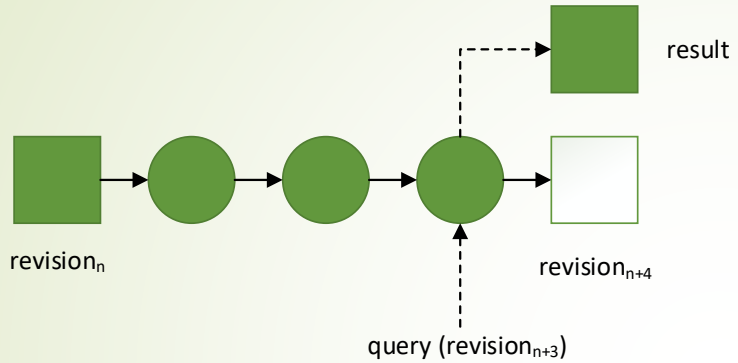
Verziókövető rendszerek

Változások reprezentációja

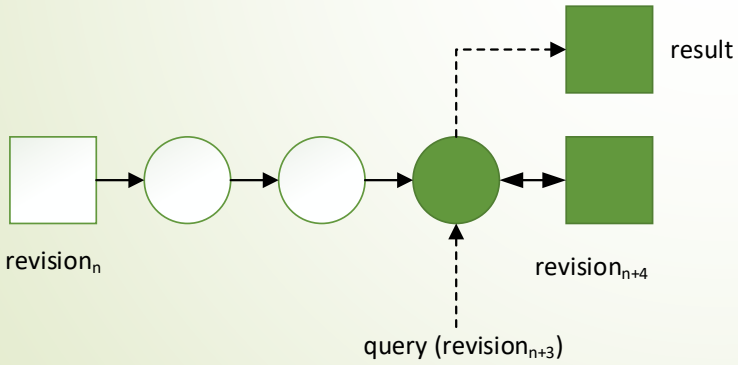
- A teljes revíziók tárolása nem lehetséges az adattárolás és adatkezelés jelentős költségei miatt
- A verziókezelő eszközök ezért csak két egymást követő verzió közötti különbséget, a *változáslistá*t (*changeset*, *delta*) tárolják
  - egyes rendszerek (pl. Mercurial) időnként pillanatfelvételt (*snapshot*) készítenek a teljes tartalomról
- Eleinte (SCCS) a delták a régi verzióból az újat tudták előállítani (*forward deltas*)
- Korán felmerült (RCS), hogy a fordított delták (*reverse deltas*) használata a legújabb verzió pillanatképének tárolásával jobb teljesítményt nyújthat, ugyanis leggyakrabban egy ág legfrissebb állapotát szokták lekérni
  - Kevert megoldás is lehetséges, pl. a fő ágon fordított irányú deltákat, a mellékágakon viszont előre mutató delták

Verziókövető rendszerek

Változások reprezentációja



Előre irányú delták
(*forward deltas*)

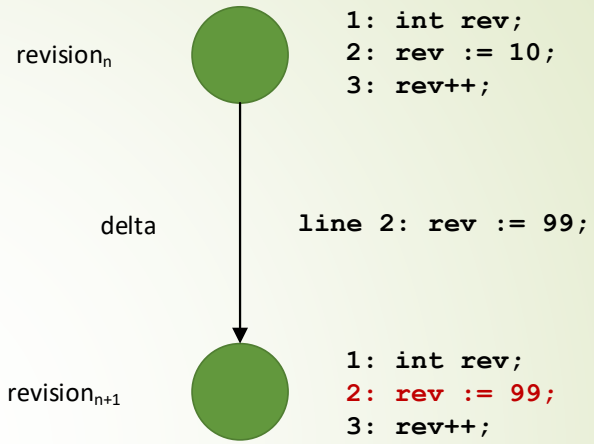


Visszafelé irányú delták
(*reverse deltas*)

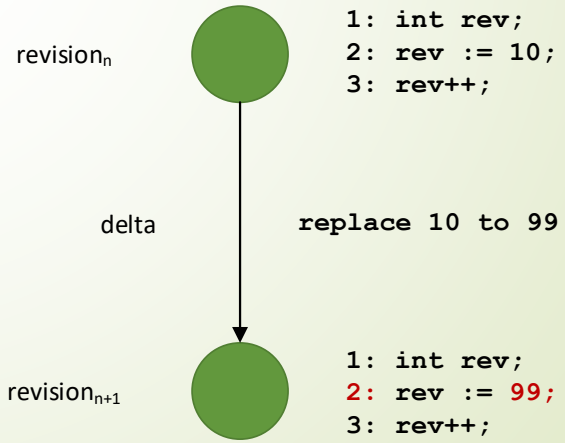
Verziókövető rendszerek

Változások reprezentációja

- Az eltérések meghatározása szöveges fájlok, így programnyelvi forráskódok esetében jellemzően *állapot alapúan* történik
  - a legtöbbször soronkénti összehasonlítással
    - pl. GNU diff
  - struktúrált tartalom esetén az összehasonlítás egysége más is lehet (pl. XML, JSON, UML)
- Bináris adatok (pl. képek) esetén a *művelet alapú* megközelítés is alkalmazható.



Állapot alapú

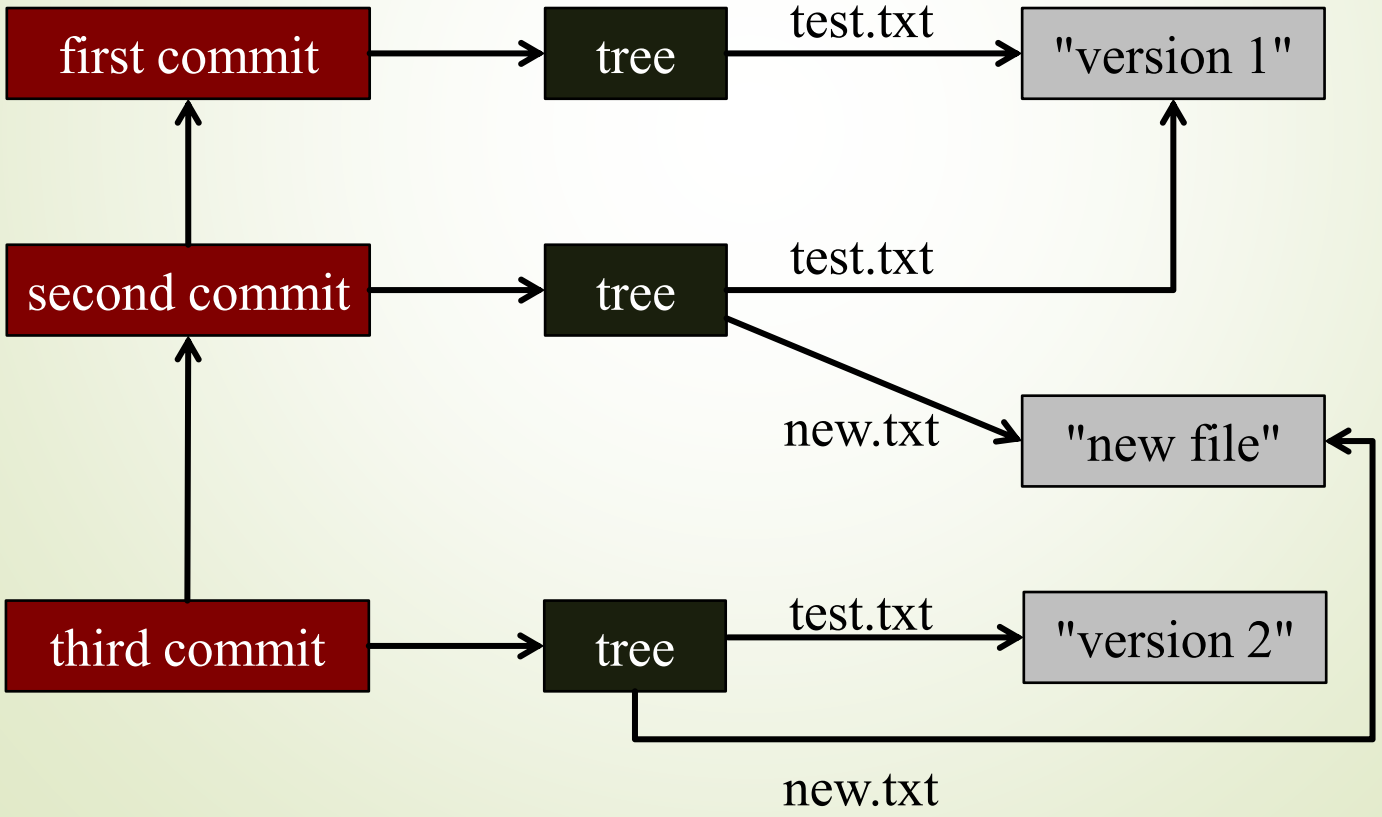


Művelet alapú

Verziókövető rendszerek

Változások reprezentációja

- A 3. generációs verziókövető rendszerek idejére a háttértárak mérete jelentősen megnőtt, míg költségük csökkent
  - Egyes eszközök ezért a megváltozott fájlok teljes tartalmát tárolják a revíziókban, nem csak a módosult tartalomrészt



Verziókövető rendszerek

Git

- A félév folyamán a gyakorlati projektek forráskódját a Git verziókezelő rendszerben fogjuk követni, amelyet integráltan támogat a GitLab projektvezető szolgáltatás.
  - Kari GitLab szerver: https://szofttech.inf.elte.hu/
- A GitLab szerveren lévő távoli tároló (*remote repository*) tartalmát a GitLab webes felületén is böngészhetjük, megtekinthetjük, sőt egyszerűbb módosításokat is végrehajthatunk (ezekből ugyanúgy *commit* lesz).
- A verziókezelést azonban alapvetően egy lokális munkapéldányon (*local repository*) szokás végezni kliens programmal, majd szinkronizálni a távoli tárolóval.
  - Konzolos kliens utasítások
  - Asztali grafikus kliens alkalmazások

Verziókövető rendszerek

Git: telepítés

- A Git verziókezelő rendszer telepítése:
  - Windows, Mac telepítő: https://git-scm.com/downloads
  - Debian/Ubuntu: `apt-get install git`
  - Más UNIX rendszerek: https://git-scm.com/download/linux
- Telepítés után a konzolos Git parancsokkal egyből dolgozhatunk.
  - Windows telepítés esetén célszerű a Git hozzáadását választani a PATH környezeti változóhoz.
  - Grafikus kliens programok külön telepíthetőek.
- Minden Git commithoz hozzárendelésre kerül a szerzője neve és email címe, így ezeket szükséges globálisan beállítanunk, mielőtt először használjuk a Gitet.

```
git config --global user.name "Hallgató Harold"
git config --global user.email hallgato@inf.elte.hu
```

Git: tároló létrehozása

- Egy új lokális tárolót létrehozhatunk üresen:

```
git init
```

- Vagy egy ismert távoli tároló lemásolásával:

```
git clone https://mysite.com/best-project.git
```

- Jellemzően távoli tárolók másolását alkalmazzuk, még akkor is, ha kezdetben üres a projekt.
  - Az így lemásolt távoli tárolóra *origin* néven hivatkozhatunk (alapértelmezetten) majd a későbbiekben, pl. a tárolók szinkronizálásakor.

Verziókövető rendszerek

Git: változások követése

- Új fájlokat valamint a fájlok módosításait a `git add` utasítással vonhatjuk verziókezelés alá, az ún. *staging area*-ba helyezve:

```
git add Main.java
```

  - Konkrét fájl helyett mintát (pl. `*.java`) vagy könyvtárat is megadhatunk.

- A munkakönyvtár állapotát a `git status` utasítással ellenőrizhetjük bármikor.

```
git status
> On branch master
> Your branch is up to date with 'origin/master'.
> Changes to be committed:
>   (use "git reset HEAD <file>..." to unstage)
>         new file:   Main.java
```

Git: módosítások helyi tárolóba küldése

- Amennyiben a kívánt módosításokat a *staging area*-hoz adtunk, annak tartalmát egy új verzióként beküldhetjük a lokális tárolóba, a `git commit` utasítással:

```
git commit -m "Added main program."
> [master d26c7a9] Added main program.
>  1 file changed, 1 insertion(+)
>  create mode 100644 Main.java
```

Verziókövető rendszerek

Git: módosítások változáskövetése

- Új fájlokat is verziókezelés alá vonhatunk:

```
git add Rectangle.java

git commit -m "Added Rectangle class."
```

- Egy új verzió új fájlokat és meglévő fájlok módosításait is tartalmazhatja:

```
git add Circle.java

git add Main.java

git commit -m "Added Circle class.
               Modified the main program."
```

Verziókövető rendszerek

Git: szinkronizálás távoli tárolóval

- A lokális tárolónkban létrehozott új verziókat szükséges szinkronizálni a távoli tárolóval, hogy a változtatásainkhoz mások is hozzáférhessenek, ezt a `git push` paranccsal tehetjük meg.

```
git push origin master
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 247 bytes | 123.00 KiB/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To /path/to/workspace/folder
   d45172c..80a39a2  master -> master
```

  - Szükséges megadni, hogy melyik távoli tárolóval szeretnénk szinkronizálni, és melyik fejlesztési ágat. Amiről klónoztunk alapértelmezetten az *origin* néven ismert, az ág itt a *master*.
  - Nyomkövető ágak (*tracking branches*) használatakor ezek elhagyhatóak, így az utasítás `git push`-ra egyszerűsödik.
- Fejlesztő társaink beküldött módosításait a saját lokális tárolónkkal és munkapéldányunkkal a `git pull` paranccsal szinkronizálhatjuk.

Git: fejlesztési ágak létrehozása

- Új fejlesztési ágat a `git branch` paranccsal hozhatunk létre.

```
git branch new-branch
```

  - Az új fejlesztési ág abból a verzióból fog elágazni, amelyiket aktuálisan betöltöttük a munkapéldányunkba.
- Fejlesztési ágak között a `git checkout` utasítással válthatunk.

```
git checkout new-branch
```

  - Az alapértelmezett fejlesztési ág neve jellemzően *master*.
- Új fejlesztési ág létrehozása és átváltás egyetlen lépésben:

```
git checkout -b new-branch
```

Verziókövető rendszerek

Git: fejlesztési ágak egyesítése

- A fejlesztési ágakon végrehajtott módosításokat az adott funkció elkészülte után szeretnénk a fő fejlesztési ágba visszacsatolni.
  - Az ágak egyesítésére a `git merge` utasítás szolgál.
  - Előbb betöltjük a fő fejlesztési ágat:

```
git checkout master
```

  - Majd a mellék fejlesztési ág módosításait egyesítjük vele:

```
git merge new-branch
```

- Amennyiben ugyanazon fájlok ugyanazon részét időközben mindkét fejlesztési ágon módosítottuk, az egyesítés jellemzően nem kivitelezhető automatikusan, ezt ütközésnek (*merge conflict*) nevezzük.

Verziókövető rendszerek

Git: ütközések feloldása

- Az ütközéseket manuálisan kell feloldanunk az érintett fájlok szerkesztésével.
- Az automatikusan nem egyesíthető részeket a Git forrásfájlokban speciális szintaxisba foglalja:

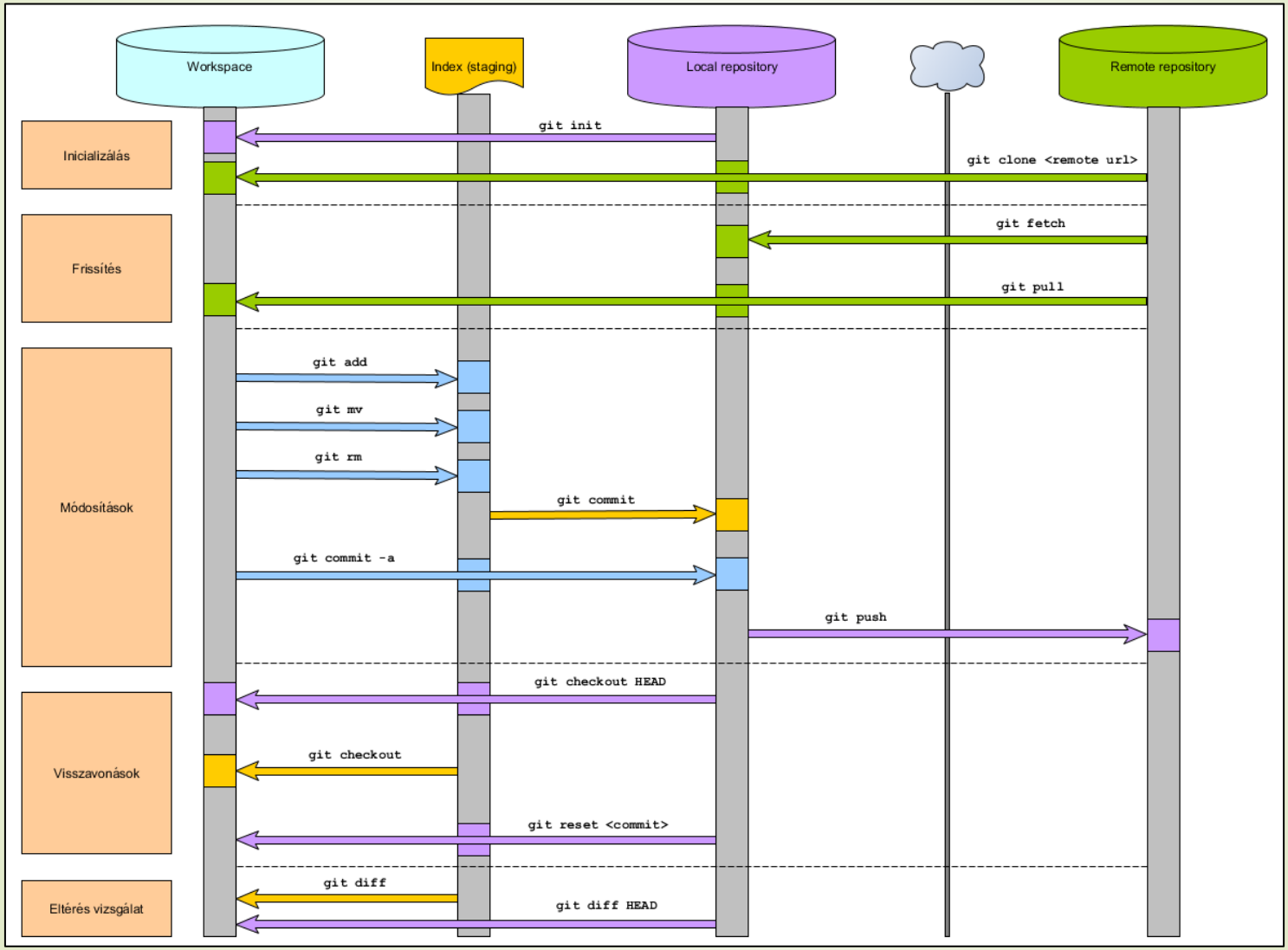
```
<<<<<<< HEAD
Az aktuális, jelen esetben master ágon lévő ütköző
tartalom
=======
Az egyesíteni kívánt, new-branch ágon lévő ütköző
tartalom
>>>>>>> new-branch
```

  - A fejlesztő feladata eldönteni, hogy a két lehetőség közül melyiket kívánja megtartani, esetleg a kettő vegyes megoldását szükséges alkalmazni.
  - A feloldott fájlokat a *staging area*-hoz kell adni (`git add`), majd a változásokat beküldeni (`git commit`).

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: alapvető konzolos utasítások áttekintése



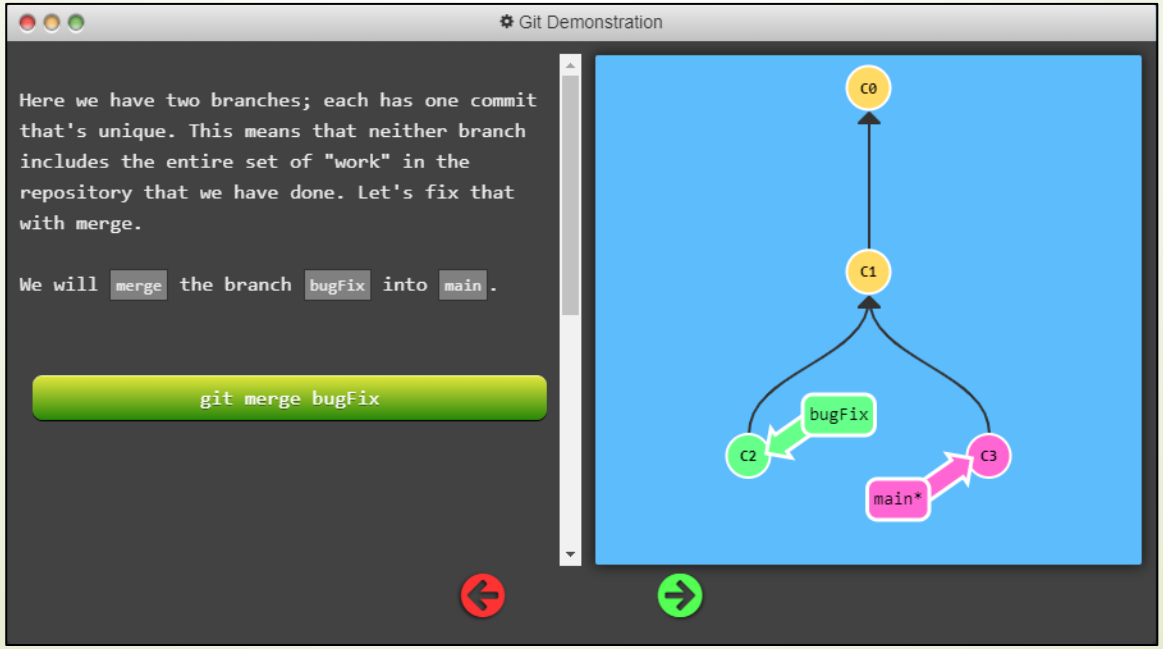
Forrás: Nagy Krisztián (ELTE IK)

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Online eszközök a tanuláshoz

- Több online eszköz is elérhető, amelyekkel vizualizáció mellett, kitűzött feladatok teljesítésével vagy szabadon gyakorolva fejleszthetőek a Git használati ismeretek. Pl.:
  - https://git-school.github.io/visualizing-git/
  - https://learngitbranching.js.org/

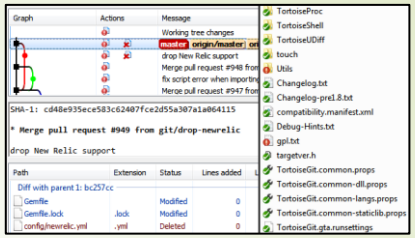


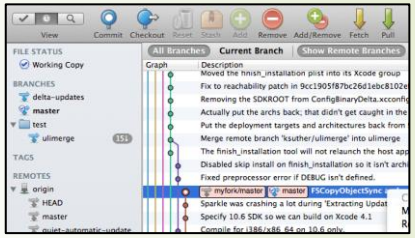
Verziókövető rendszerek

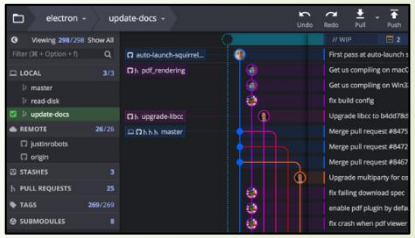
ELTE | IK
INFORMATIKAI KAR

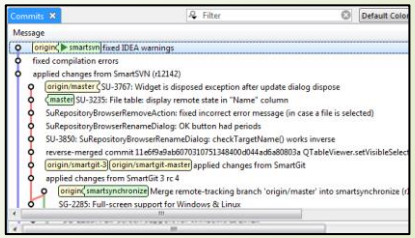
Git GUI kliensek

- TortoiseGit
  - Windows
  - *géptermi gépeken elérhető*
- SourceTree
  - Windows, Mac
- GitKraken
  - Linux, Windows, Mac
- SmartGit
  - Linux, Windows, Mac
- Továbbiak:
  - https://git-scm.com/downloads/guis





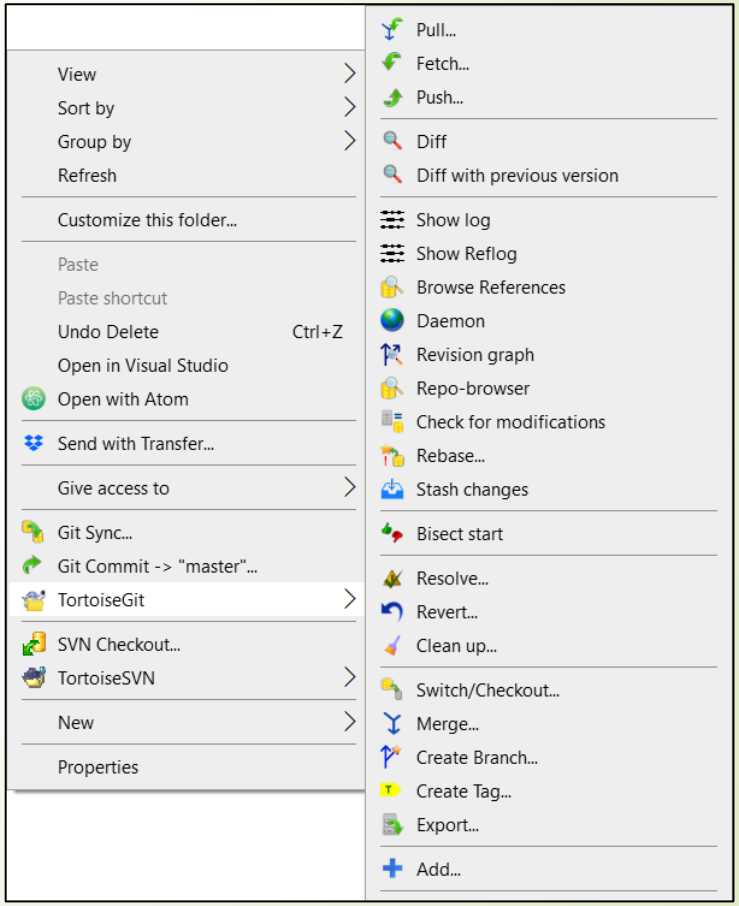




Verziókövető rendszerek

TortoiseGit használata

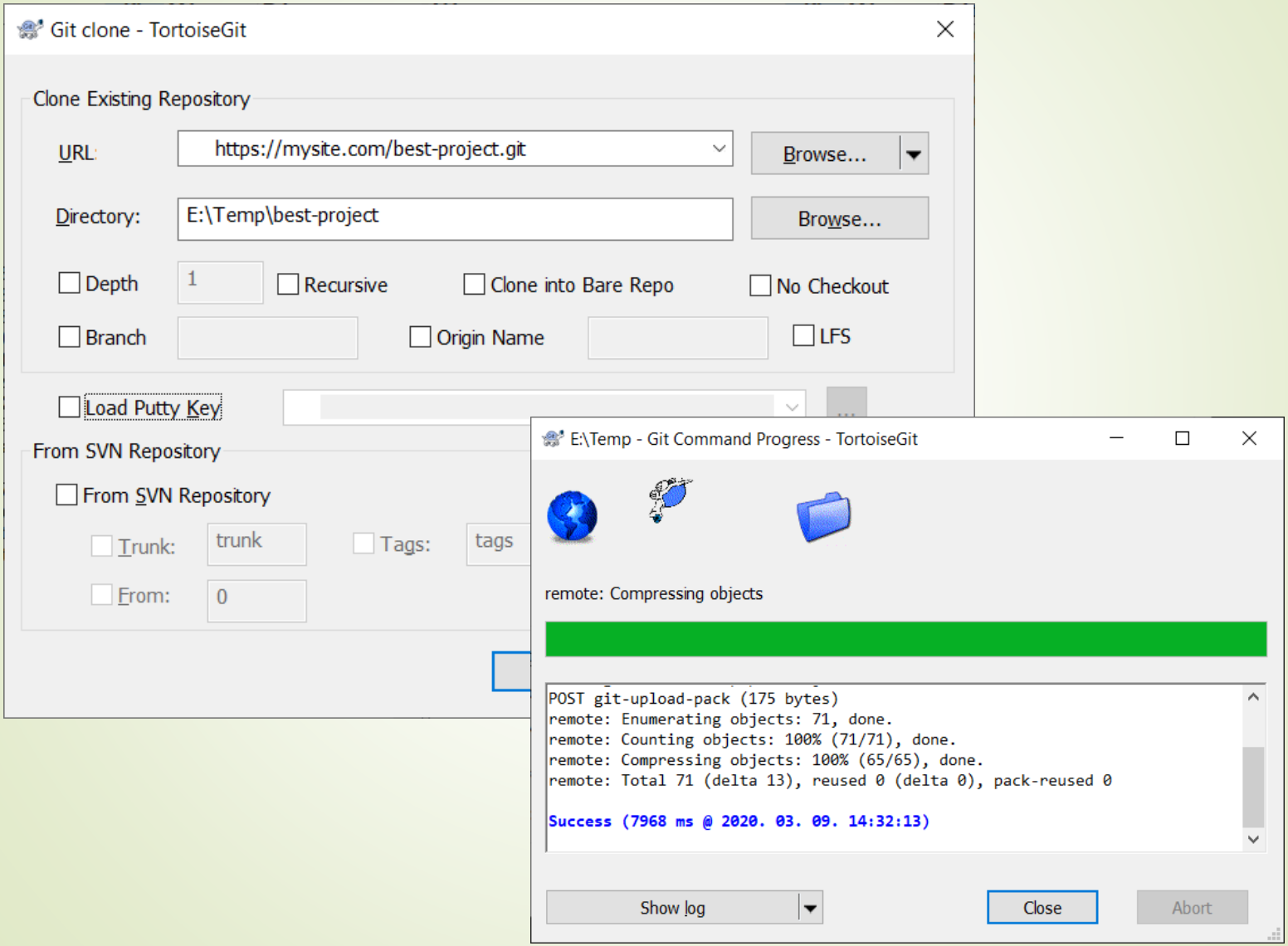
- A TortoiseGit egy Windows rendszekre fejlesztett, ingyenes, asztali grafikus Git kliens
  - Honlap, letöltés: https://tortoisegit.org/
  - Elérhető magyar nyelven is.
  - A Git-et nem tartalmazza, azt külön szükséges telepíteni.
- A TortoiseGit a fájlkezelő alkalmazások (pl. *File Explorer*) jobb egérkattintásra elérhető kontextus menüjébe épül be, innen hívhatóak meg a már megismert utasítások.



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

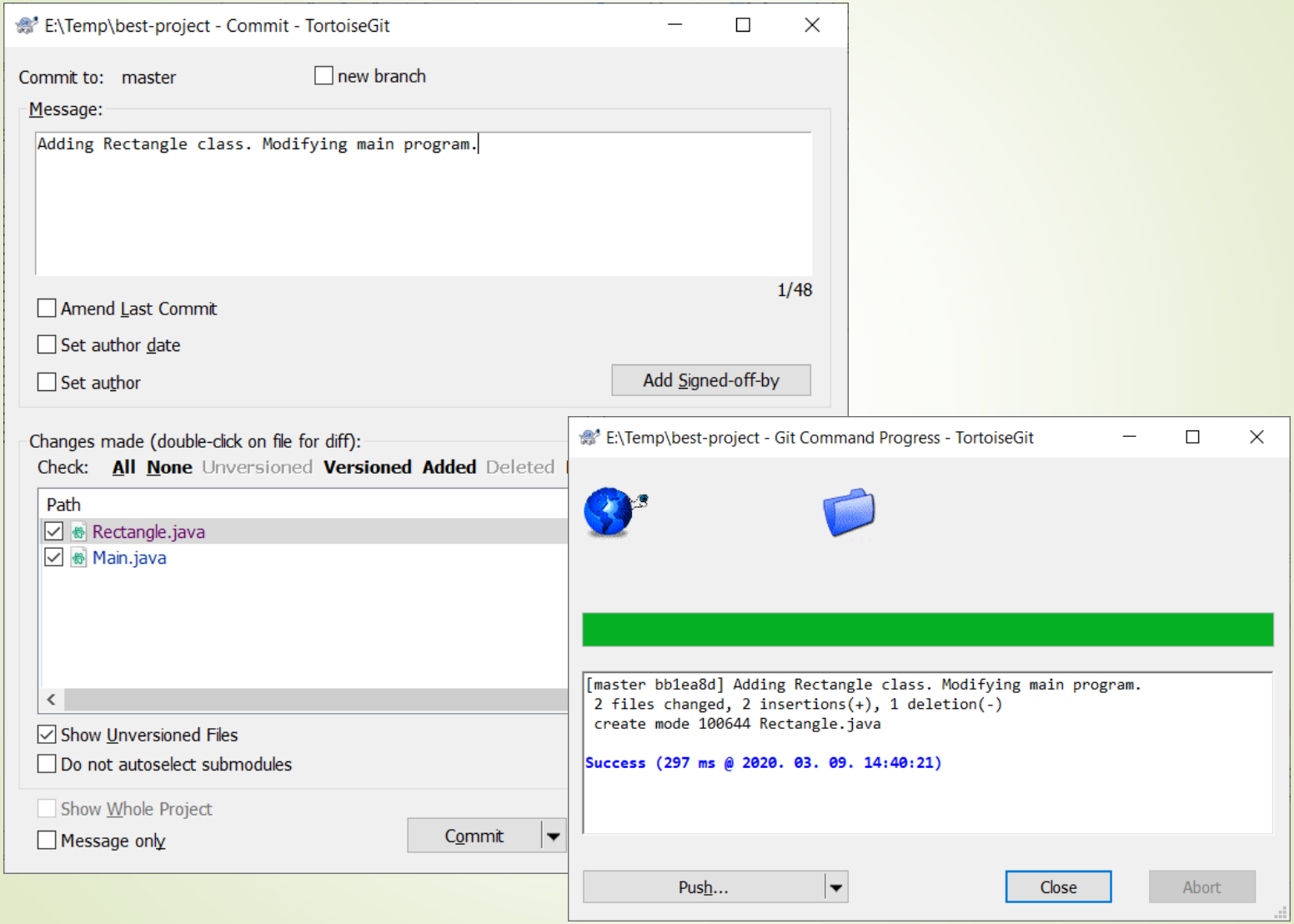
TortoiseGit: klónozás (*clone*)



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

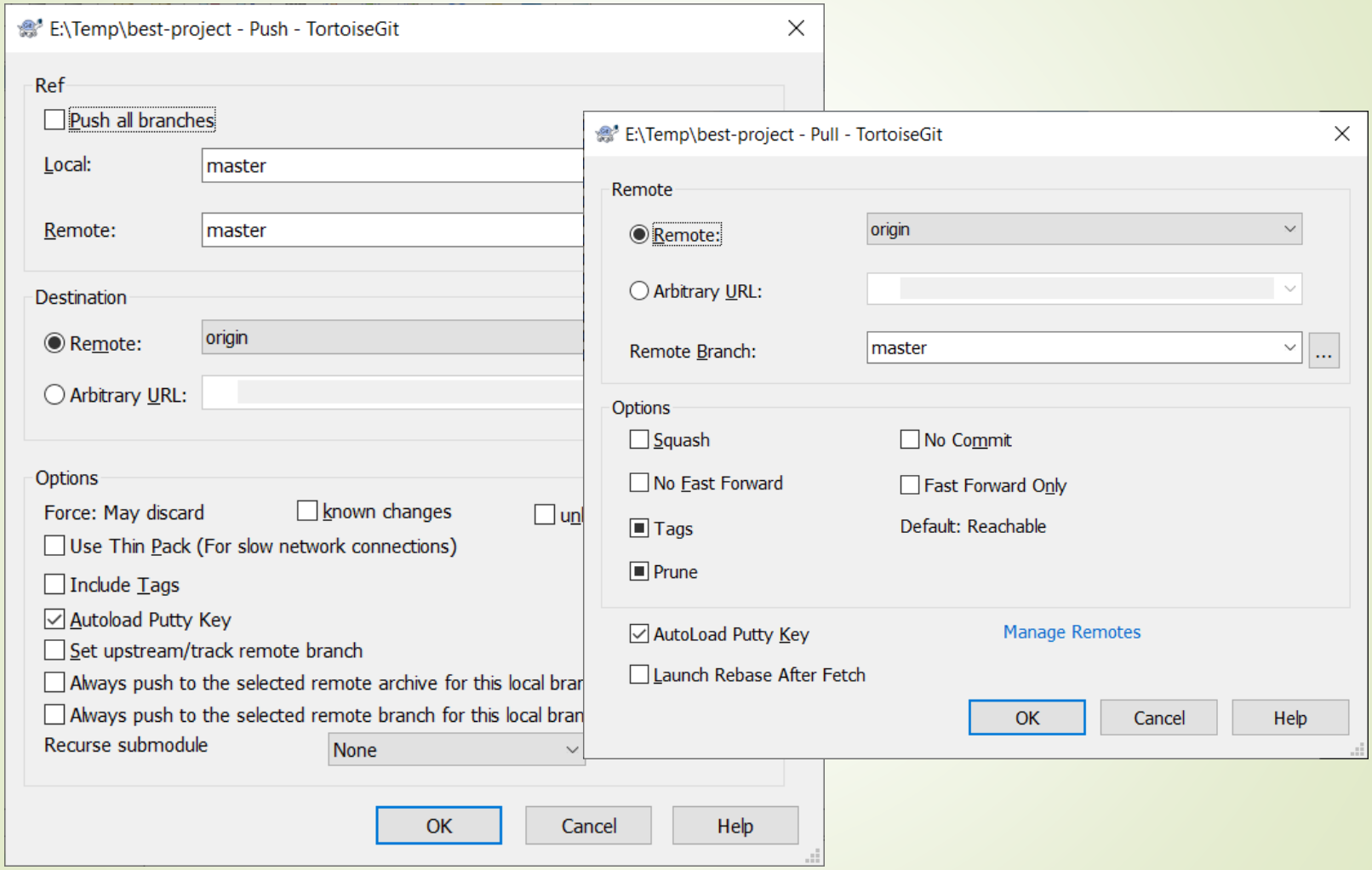
TortoiseGit: új verzió beküldése (*commit*)



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

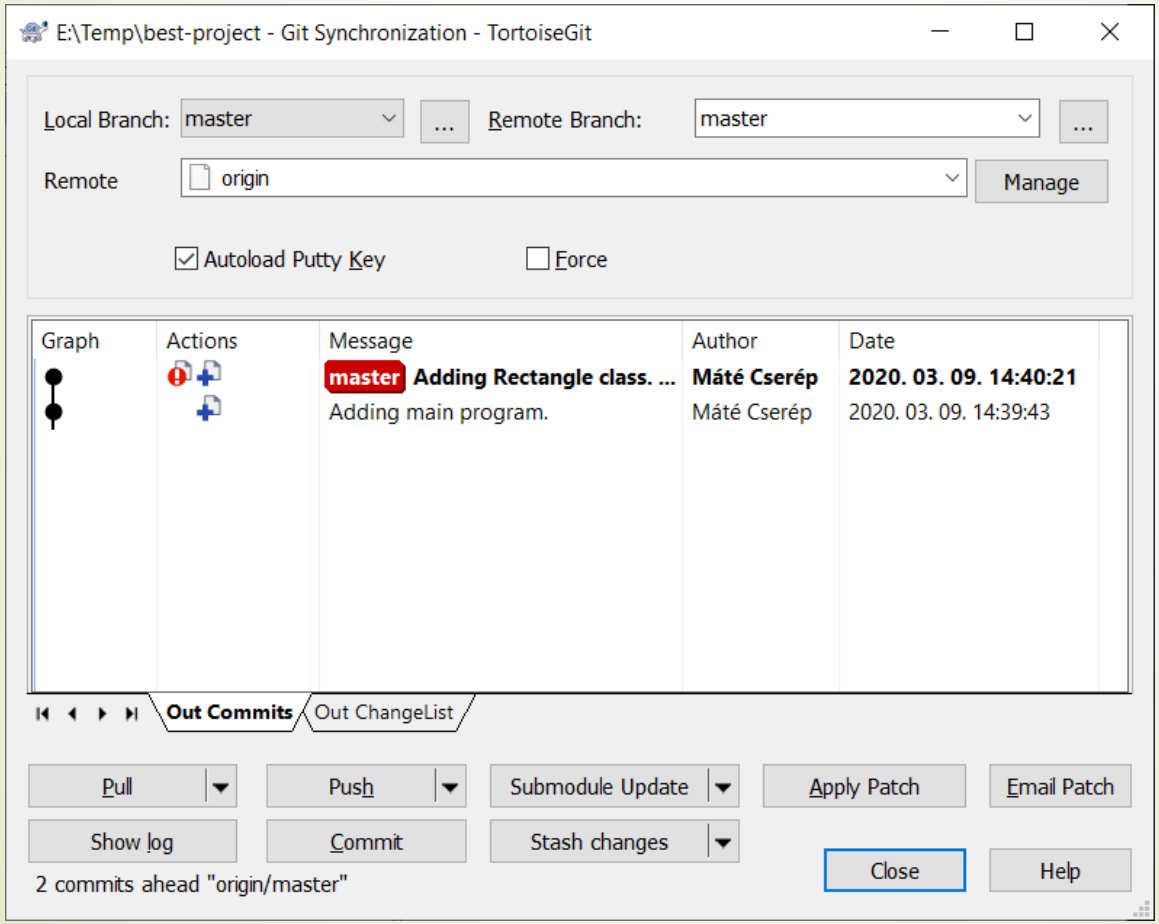
TortoiseGit: szinkronizálás (*push* & *pull*)



Verziókövető rendszerek

TortoiseGit: szinkronizálás (*sync*)

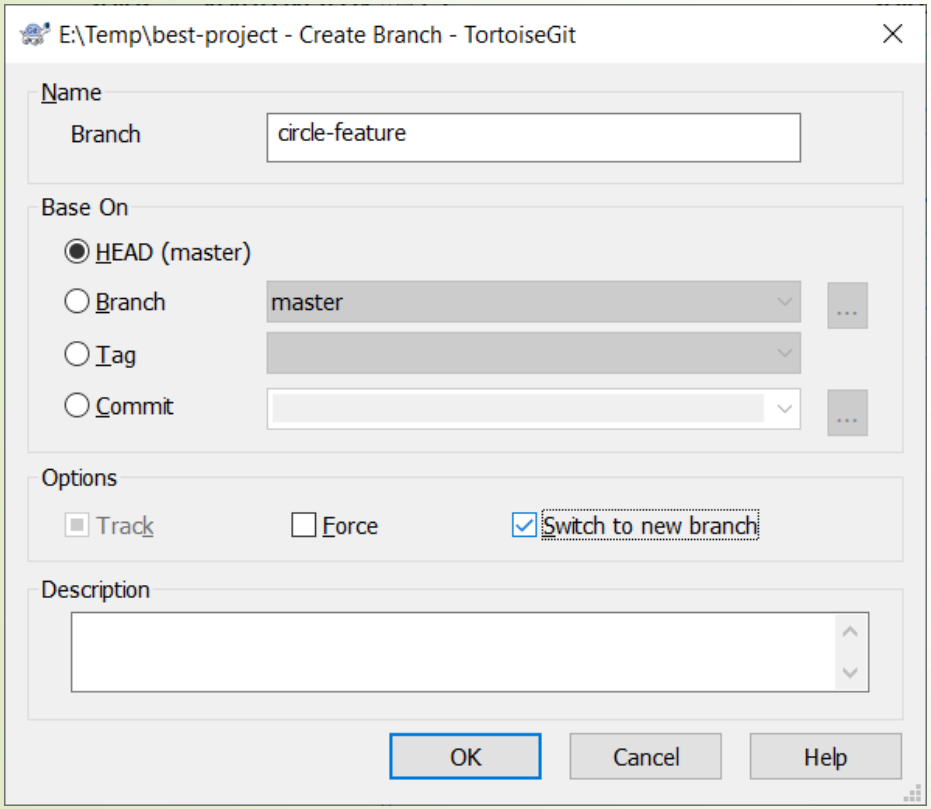
- A *Sync* funkcióval egyszerre érhetjük el a *pull* és a *push* opciókat is egy áttekintő felületen.

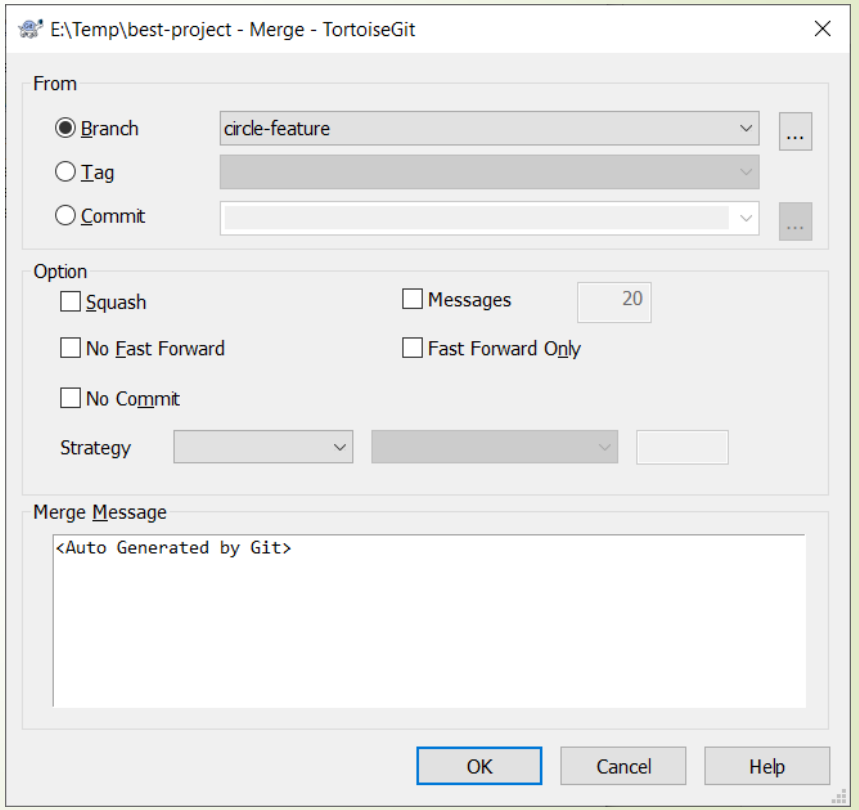


Verziókövető rendszerek

TortoiseGit: fejlesztési ágak (*branch*) kezelése

- Eleinte a grafikus felület jelentősen könnyebbé teheti a fejlesztési ágak létrehozását és összeillesztését.

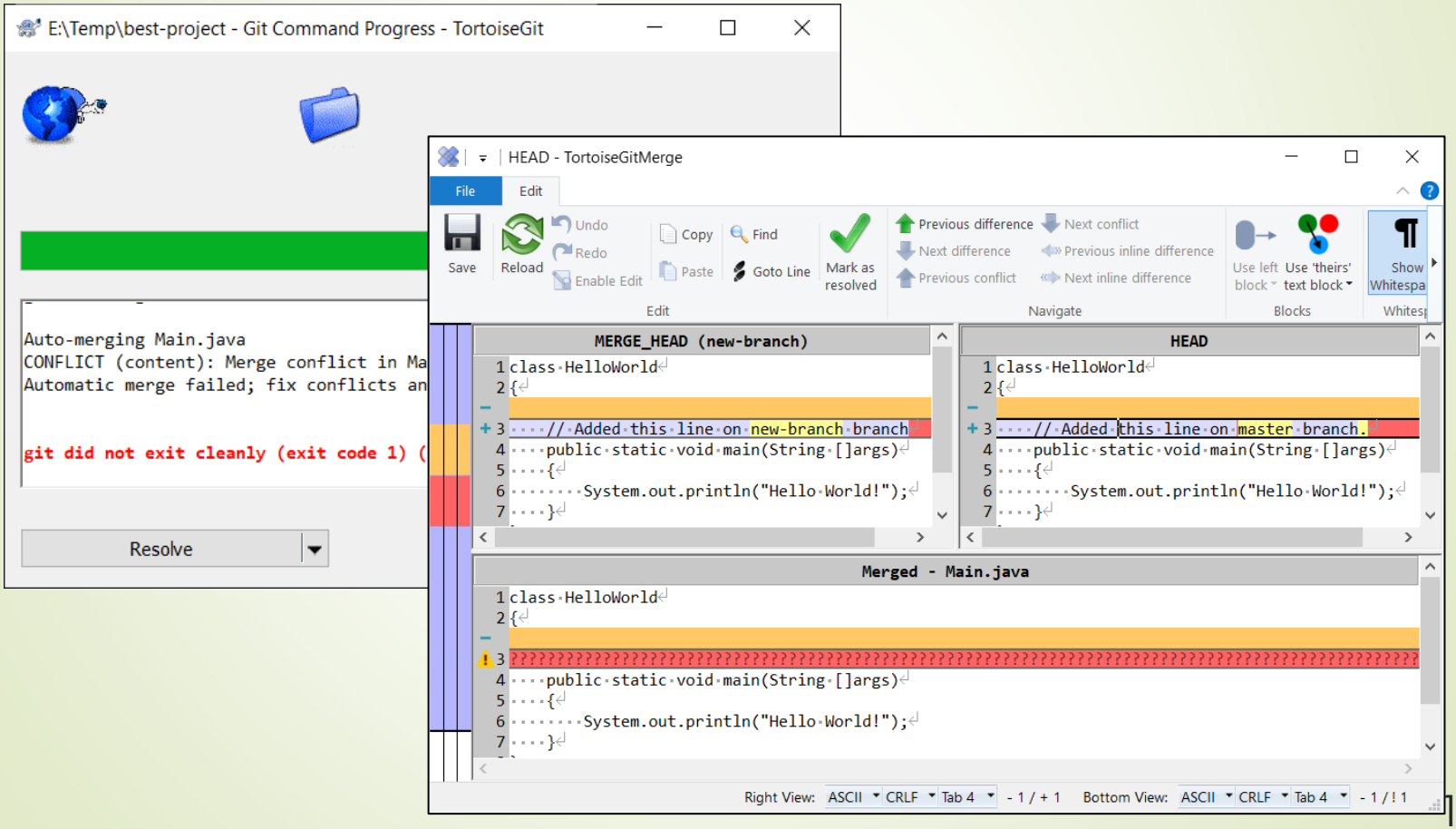




Verziókövető rendszerek

TortoiseGit: ütközések feloldása

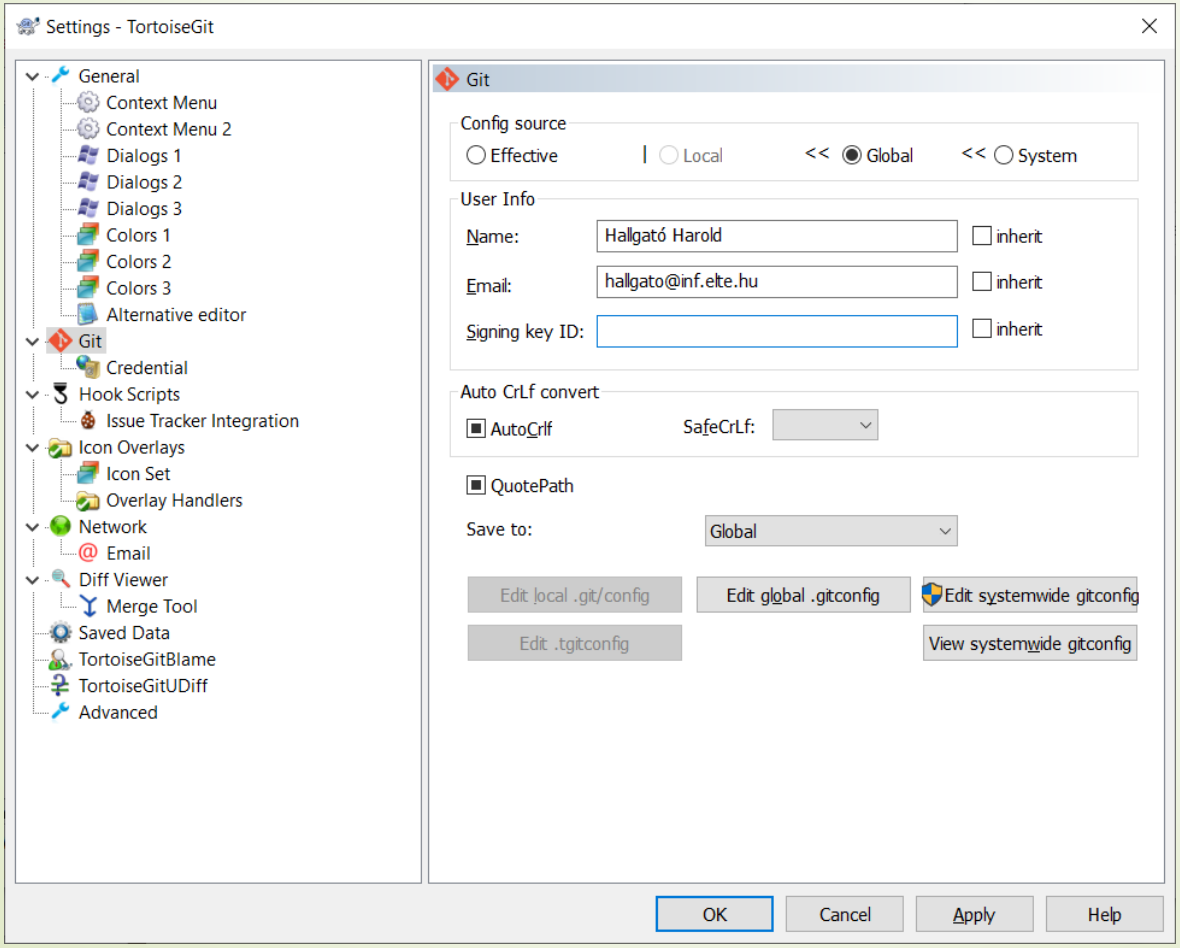
- Különösen igaz ez az egyesítéskor fellépő konfliktusok feloldására, ahol egy összevető nézet segíti a munkát.



Verziókövető rendszerek

TortoiseGit: beállítások

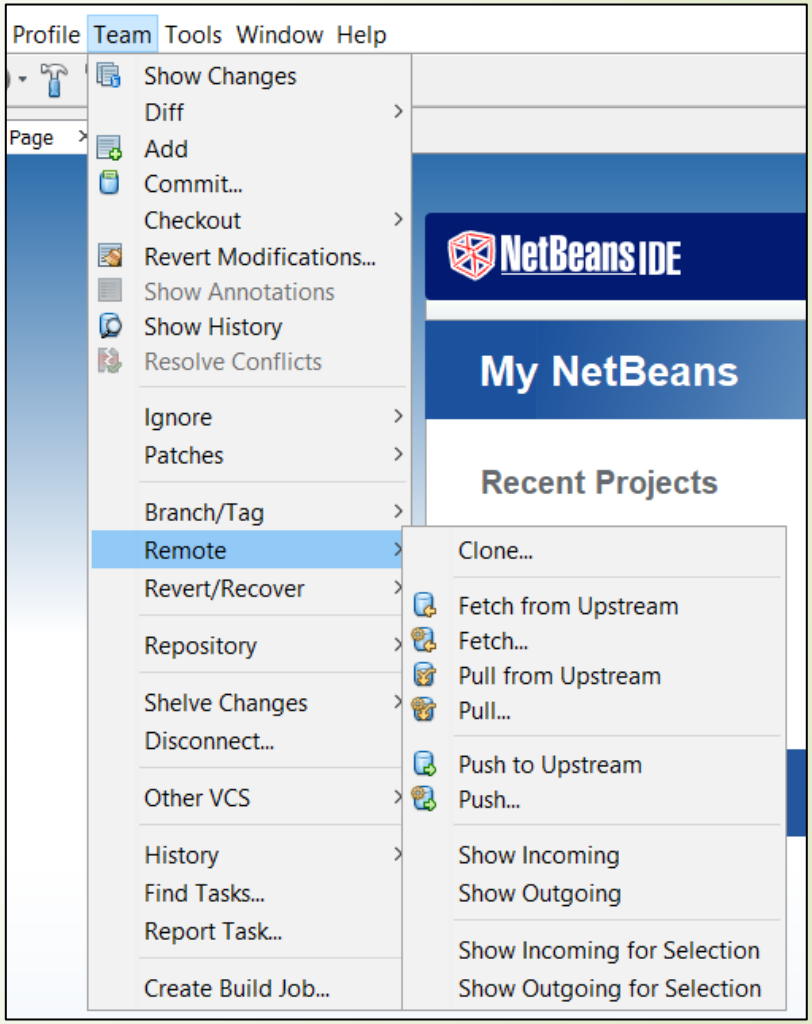
- *TortoiseGit -> Settings* menüpont alatt szerkeszthetjük a beállításokat is, pl. a felhasználó nevét és email címét.



Verziókövető rendszerek

Támogatás integrált fejlesztőkörnyezetekben

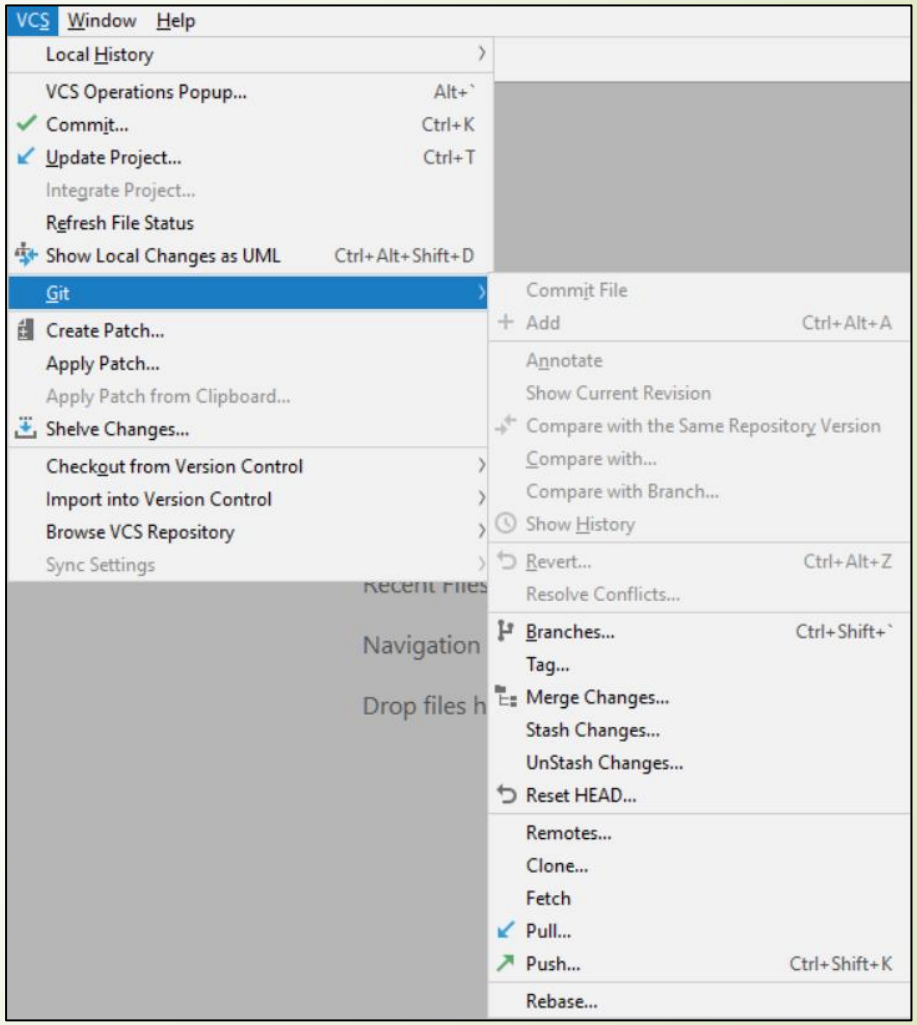
- A legtöbb modern integrált fejlesztőkörnyezet (IDE) beépített támogatást nyújt a verziókezelésre.
  - Részben éppen emiatt nevezzük integrált környezetnek őket.
  - Így nem szükséges különféle alkalmazások kontextusai között váltogatni.
- *NetBeans* esetében a verziókezelő funkciókat a *Team* menüpont alatt találjuk.



Verziókövető rendszerek

Támogatás integrált fejlesztőkörnyezetekben

- *IntelliJ IDEA* esetében ezeket a funkciókat a VCS (*version control system*) menüpont alatt találjuk.



Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Mely fájlokat érdemes verziókezelés alá vonni?

- A Git verziókezelő rendszer a szöveges állományok, így tipikusan a forráskód fájlok változáskezelésében hatékony. Elsődlegesen a projekt forráskódját érdemes benne elhelyezni.
- Egy általános szoftver projekt esetén nem érdemes verziókezelés alá vonni:
  - fordítás során előálló köztes tárgykódot vagy a végső bináris állományokat, mert újból előállíthatóak, folyamatosan változnak és ütközéseket okoznak.
  - fejlesztő eszközök személyes beállításait (pl. Visual Studio esetén a *.vs/* vagy Netbeans esetén a *nbproject/private/* könyvtárak), amelyek felhasználónként eltérőek.
  - nagy méretű bináris állományokat (pl. videók, nagy méretű képek), amelyek kezelésében a Git nem hatékony. Bár a Git tárolók mérete jól skálázható, egy könnyen kezelhető *repository* mérete az 1-2 GB-os méretet nem haladja meg.

Verziókövető rendszerek

Fájlok kivonása a verziókezelés alól

- Verziókezelés alá kizárólag azok a fájlok kerülnek, amelyeket kifejezetten hozzáadunk (`git add`).
- Az esetleges véletlen hozzáadást elkerülendő megjelölhetjük azokat a fájlokat és könyvtárakat, amelyeket mellőzni szeretnénk.
  - A mellőzendő állományokat egy speciális `.gitignore` elnevezésű állományban adhatjuk meg
  - Ezt a fájlt érdemes verziókezelés alá is vonni, hogy a fejlesztők között egységes legyen a beállítás.
- A `.gitignore` minden sorában egy illeszkedési mintát adhatunk meg, hogy mely fájlokat akarjuk kizárni a verziókezelés alól.
  - A beállítás tranzitívan vonatkozik az alkönyvtárakra is, így gyakran elegendő lehet egyetlen `.gitignore` fájl létrehozása a projekt gyökér- könyvtárában.

Verziókövető rendszerek

Git: .gitignore minták

Minta	Illeszkedés	Leírás
program.jar	/program.jar /bin/program. jar	Minden **program.jar** fájlra illeszkedés.
/program.jar	/program.jar **de nem:** /bin/program.jar	Az adott könyvtárszinten lévő **program.jar** fájlra illeszkedés.
*.jar	/program.jar /bin/main.jar	Minden **jar** kiterjesztésű fájlra illeszkedés.
bin/	/bin/ /project/bin/ **de nem:** /logs/bin (fájl)	Minden **bin** könyvtárra illeszkedés (de **bin** nevű fájlokra nem!)
/bin/*.jar	/bin/program.jar /bin/main.jar **de nem:** /bin/program.class /project/bin/program.jar	Az adott könyvtárban lévő **bin** könyvtárban lévő összes **jar** kiterjesztésű fájlra illeszkedés.

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

Git: .gitignore minták

Minta	Illeszkedés	Leírás
*.log !important.log	/application.log /logs/application.log **de nem:** /logs/important.log	Az összes **log** kiterjesztésű fájlra illeszkedés, kivéve, ha a fájl neve **important.log**.
logs/**/*.log	/logs/application.log /logs/runtime/main/01.log /project/logs/deploy.log **de nem:** /runtime.log	Minden olyan **log** kiterjesztésű fájlra illeszkedés, amely egy **logs** könyvtár alatt helyezkedik el (tetszőleges mélységben).

- Számos programozási nyelvhez és IDE-hez érhető el általános esetekre megfelelő `.gitignore` állomány a *GitHub*-on, ezekből vagy ezek kombinációjából jó ötlet kiindulni.
  - URL: https://github.com/github/gitignore

Verziókövető rendszerek

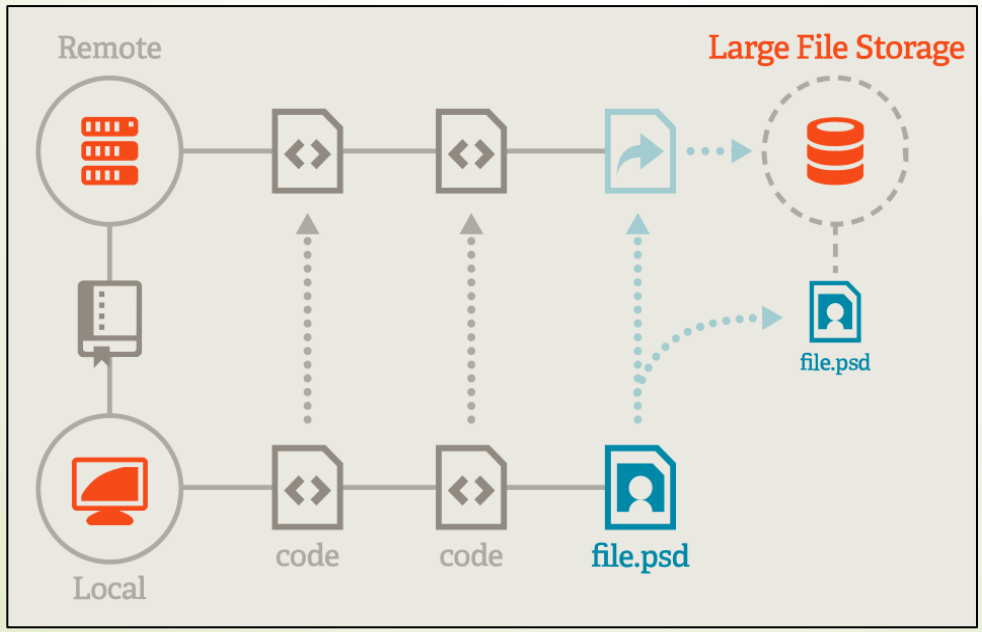
Nagy erőforrás állományok verziókezelése

- A nagy méretű videó, kép és hang erőforrás állományok hatékony kezelése körültekintést igényel.
  - A nagy méretű bináris állományok változásainak kezelésében a Git kevésbé hatékony.
  - Jelentősen megnöveli a tároló helyi másolatának lekéréséhez szükséges hálózati forgalmat (*git clone*).
  - Egy fejlesztőcsapatban a programozóknak nem feltétlenül van szükségük a fejlesztéshez a designerek által készített *assetekre*.
- Ezért a nagy méretű bináris erőforrás állományokat még akkor sem feltétlenül érdemes a Git tárolóban elhelyezni, ha amúgy ritkán változnak.

Verziókövető rendszerek

Git Large File Storage

- A nagy méretű bináris állományok kezelése a *Git Large File Storage* (Git LFS) segítéségével oldható meg.
  - A nagy méretű bináris állományokat egy hivatkozással helyettesíti és magukat a fájlokat egy másik (akár távoli) szerveren tárolja.
  - Így a Git tárolónk mérete kezelhető marad.



Forrás: git-lfs.github.com

Verziókövető rendszerek

ELTE | IK
INFORMATIKAI KAR

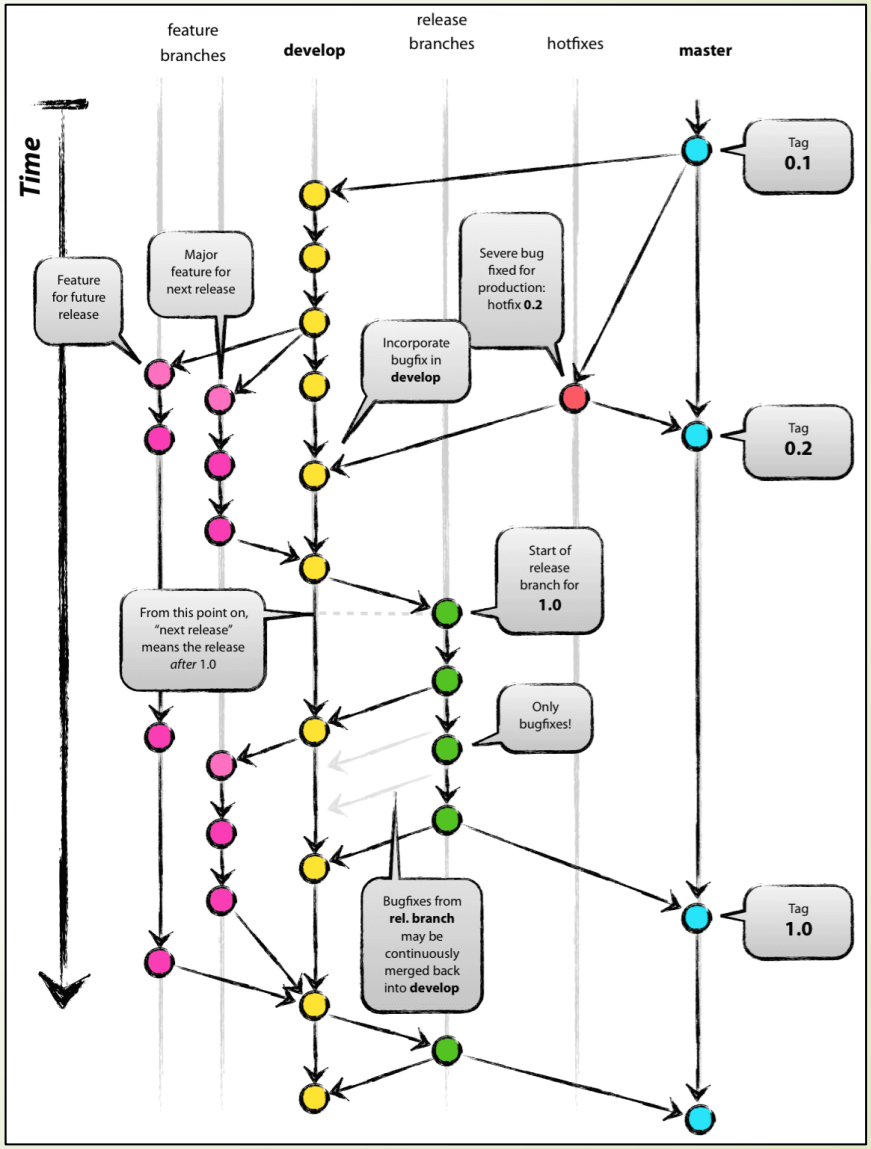
Git Large File Storage

- A szofttech.inf.elte.hu GitLab szerver támogatja a Git LFS-t.
- Használatához csak a kliens gépekre (saját gépetek) is szükséges a Git LFS telepítése.
  - Letöltés: https://git-lfs.github.com/
  - Használat: https://docs.gitlab.com/ee/administration/lfs/manage_large_binaries_with_git_lfs.html

Verziókövető rendszerek

GitFlow feature branching model

- Nagy, több támogatott kiadással rendelkező szoftverekhez ajánlott
- Fő fejlesztési ágak:
  - master
  - develop
- Támogató ágak:
  - feature branches
  - release branches
  - hotfix branches



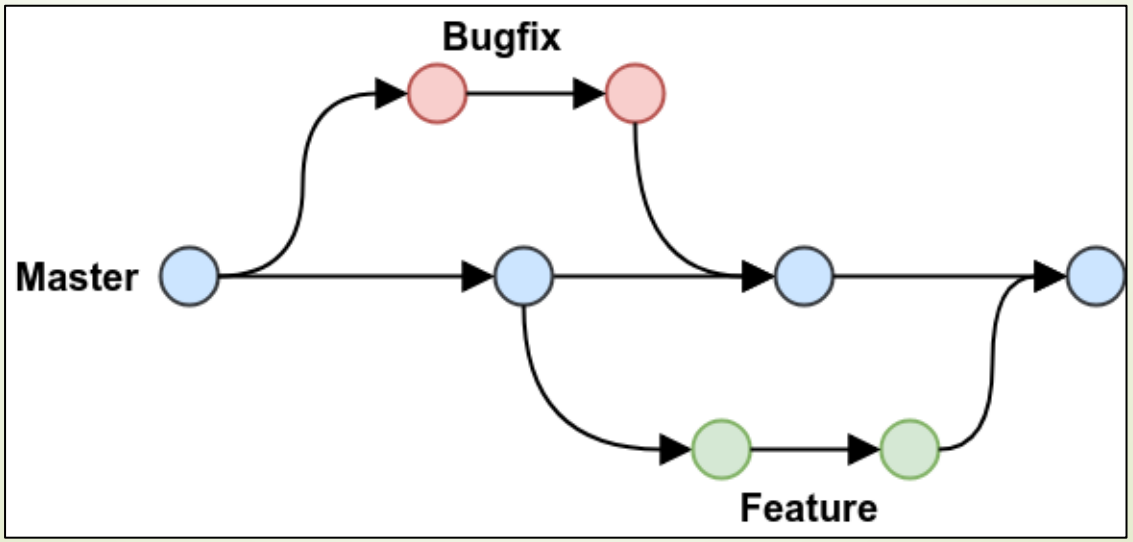
Forrás: nvie.com

Verziókövető rendszerek

GitHub Flow feature branching model

- Jellemzően csak egyetlen támogatott, gyors kiadási ciklussal rendelkező szoftverekhez ajánlott.
  - Ami a fő fejlesztési ágon van, az kiadható!

- Fő fejlesztési ág: *master*
- Támogató fejlesztési ágak: *feature* és *bugfix*

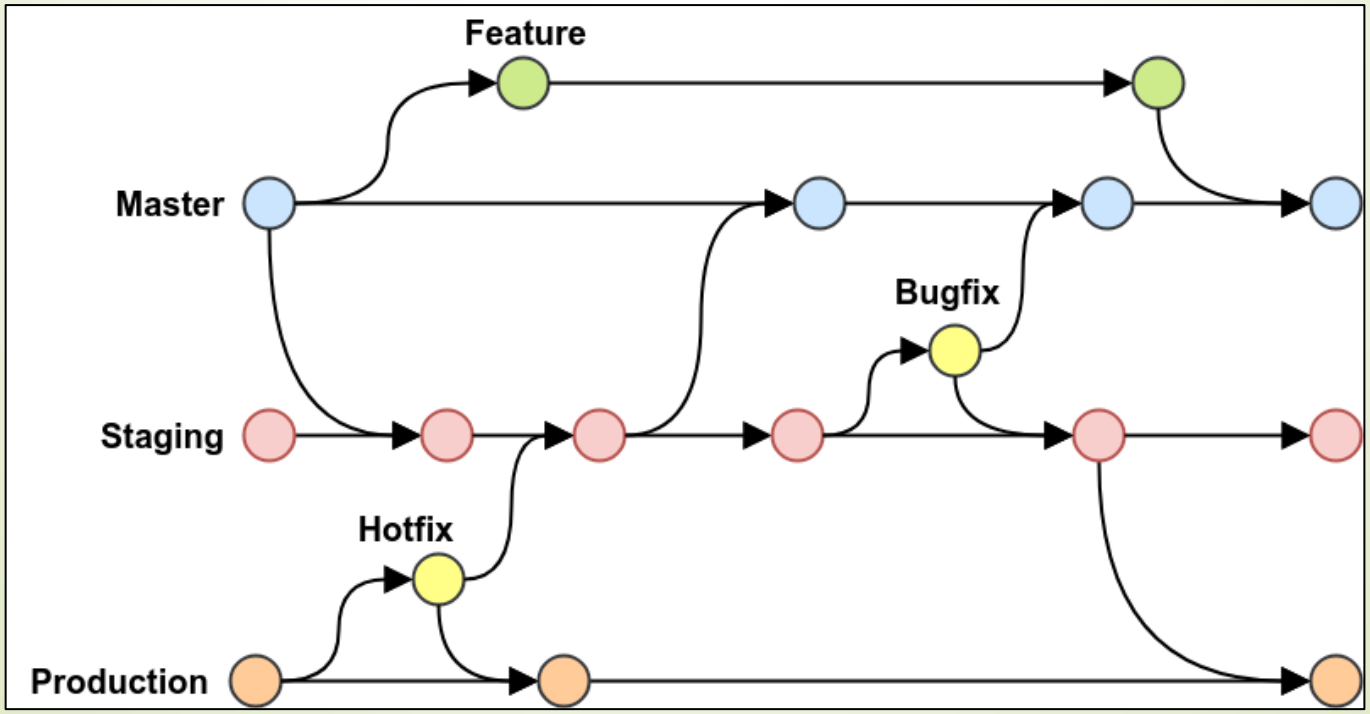


Forrás: blog.programster.org

Verziókövető rendszerek

GitLab Flow feature branching model

- Nem biztos, hogy minden esetben kiadható a szoftver új verziója, ami a fő fejlesztési ágra került.
  - Fő fejlesztési ág: *master*
  - Kiadási ág: *production*
  - Tesztelési (előzetes kiadási) ág: *staging* (opcionális)



Forrás: blog.programster.org

Szoftvertechnológia

Build rendszerek

A & B szakirány

Dr. Cserép Máté

ELTE Informatikai Kar

2025.

Build rendszerek

ELTE | IK
INFORMATIKAI KAR

„The problem of handling the compilation of a project has been encountered well before you were born. That's why there's a single method to do it today, being a consensus since several decades.

Ha! Ha! Just kidding!"

Julien Jorge

Build rendszerek

C++ programok fordítása

- Hogyan tudunk konzolos eszközökkel lefordítani egy C++ programot (pl. GNU g++ fordítóval)?

```
g++ -c -o foo.o foo.cpp \
    -O2 -std=c++17 -pedantic -I./include/


... további fordítási egységek ...


g++ -c -o main.o main.cpp \
    -O2 -std=c++17 -pedantic -I./include/
g++ -o program.exe  foo.o ...  main.o
```

C++ programok fordítása

- Mint látjuk, már egy egyszerűbb, néhány forrás fájlból álló program esetén sem egyszerű a megoldás.

- **Problémák:**
  - A programok manuális fordítása már kisebb programoknál is könnyen nehezen kezelhetővé válik.
  - Nagyobb programoknál nem követhető mely fordítási egységek újrafordítása szükséges.
    - A teljes alkalmazás újrafordítása valós vállalati alkalmazásoknál hosszabb időt is igényelhet.

Build rendszerek

Fordító eszközök kategorizálása

- Önálló (*standalone*)
  - Projekt automatizált fordítása megadott utasítások és/vagy szabályok alapján.
  - Pl. Make, NMake, Scons, Jom, BJam, Ninja
- Fejlesztő környezetebe integrált (*integrated*)
  - Az IDE-vel együtt használható build rendszer
  - Pl. Visual Studio, Xcode, Eclipse
- Generátorok (*generators*)
  - Olyan generátorok, amelyekkel más, *standalone* build rendszerek forrásfájljai állíthatóak elő, automatizáltan.
  - CMake, Autotools, GYP

Build rendszerek – Make

Telepítés

- GNU Make
  - https://www.gnu.org/software/make/
- Telepítés:
  - Windows alatt célszerű a MinGW-t (*Minimalist GNU for Windows*) telepíteni, majd ennek *bin* könyvtárát a *PATH* környezeti változóhoz adni.
  - UNIX rendszerekre a *package repository* tartalmazza.
    - Debian/Ubuntu: `apt-get install make`
  - Jellemzően fejlesztőkörnyezetekkel együtt is telepítésre kerül (pl. JetBrains CLion, Qt Creator).

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Jellemzők

- A fordítást célokon (*target*) és szabályokon (*rule*) keresztül definiáljuk.
- A szabályok függőségeket (*dependency*) és utasításokat (*command*) tartalmazhatnak.
- A szabályrendszert az ún. *Makefile* tartalmazza.
- Futtatás: **`make`**
  - Amennyiben eltérő a *Makefile* neve:
    `make -f MyMakefile`
- Példa Makefile:

```
program: foo.cpp main.cpp
    g++ -o program foo.cpp main.cpp -std=c++17
```

függőség

szabály

cél

utasítás

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Szabályok

- A *Makefile* szabályok a következő módon épülnek fel:
  - Cél (*target*): a szabály által előállítandó, gyakran (de nem feltétlenül) bináris állomány.
  - Függőségek (*dependencies*): amennyiben a cél nem létezik vagy ha a függőségek valamelyike frissül, a célt újra kell generálni.
    - A függőségek frissülése a fájlok időbélyege alapján eldönthető.
  - Utasítások (*commands*): a cél előállításához szükséges végrehajtandó utasítások.

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Változók

- A *Makefile*ban definiálhatunk a szabályok mellett változókat is:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic

program: foo.cpp main.cpp
  $(CXX) -o program foo.cpp main.cpp $(CFLAGS)
```

Build rendszerek – Make

Több fordítási egység kezelése

- További szabályokkal több fordítási egységet is kezelhetünk:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic

foo.o: foo.cpp foo.h
  $(CXX) -c -o foo.o foo.cpp $(CFLAGS)

main.o: main.cpp
  $(CXX) -c -o main.o main.cpp $(CFLAGS)

program: foo.o main.o
  $(CXX) -o program foo.o main.o $(CFLAGS)
```

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Automatikus változók

- Az automatikus változók segítségével szabályainkat minták alapján általánosíthatjuk.
- Fontosabb automatikus változók:
  - $@: a cél (fájl)neve
  - $<: az első függőség neve
  - $^: az összes függőség neve
  - $?: a célnál frissebb függőségek nevei
  - $(@D): a cél nevének könyvtárrésze
  - $(@F): a cél nevének fájlrésze

Build rendszerek – Make

Automatikus változók

- Minta szabály (*pattern rule*) az object fájlok fordítására:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic
DEPS=foo.h

%.o: %.cpp $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)

program: foo.o main.o
  $(CXX) -o program foo.o main.o $(CFLAGS)
```

Build rendszerek – Make

Automatikus változók

- Object fájlok kiemelése (és a redundancia elkerülése):

```
CXX=g++
CFLAGS=-std=c++17 -pedantic
DEPS=foo.h
OBJ=foo.o main.o

%.o: %.c $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)

program: $(OBJ)
  $(CXX) -o $@ $^ $(CFLAGS)
```

Build rendszerek – Make

Projekt könyvtár struktúra kezelése

- A forrás és fordítási könyvtárszerkezet kezelése (pl. *include*, *obj*):

```
IDIR=../include
ODIR=obj
CXX=g++
CFLAGS=-std=c++17 -pedantic -I$(IDIR)
_DEPS=foo.h
DEPS=$(patsubst %, $(IDIR)/%, $(_DEPS))
_OBJ=foo.o main.o
OBJ = $(patsubst %, $(ODIR)/%, $(_OBJ))

$(ODIR)/%.o: %.cpp $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)
$(ODIR)/program: $(OBJ)
  $(CXX) -o $@ $^ $(CFLAGS)
```

../include/foo.h

obj/foo.o obj/main.o

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Hamis célok

- A Makefile célok alapvetően a szabály eredményeként előálló állományt jelölik meg. Nézzünk példát, ahol ez nem teljesül:

```
clean:
	rm -f $(ODIR)/*.o $(ODIR)/program
```

- Amennyiben létezik *clean* nevű állomány, a *make clean* utasítás nem fogja végrehajtani az utasításokat. (Nincsenek függőségek.)
- Az ilyen célokat hamis (*phony*) céloknak jelöljük. Tipikusan pl.:

```
.PHONY: clean
clean:
	rm -f $(ODIR)/*.o $(ODIR)/program

.PHONY: all
all: $(ODIR)/program
```

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Modulok

- A GNU Make támogatja a modularizált projekteket. Ilyenkor minden alkönyvtár saját *Makefile*t tartalmaz, amelynek végrehajtását a szülő könyvtár *Makefile*ja kezdeményezi:

```
.PHONY: all

all: program

  $(MAKE) -C module_a
  $(MAKE) -C module_b
  $(CXX) -o program $(CFLAGS) \
    module_a/obj/modul_a.o \
    module_b/obj/modul_b.o
```

Build rendszerek – Make

ELTE | IK
INFORMATIKAI KAR

Példa

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp

```
# Programs & tools (Linux)
#CXX     = g++
#RM      = rm -f
#MKDIR   = mkdir -p
#CP      = cp -f

# Programs & tools (Windows)
CXX     = g++
RM      = rmdir /s /q
MKDIR   = mkdir
CP      = copy /y

# Flags & options
CFLAGS  = -O2 -std=c++17 -pedantic
INCLUDE = -I./include/

# Output
OBJDIR  = build
MAIN    = thesisgenarator.exe
TARGET  = $(OBJDIR)/$(MAIN)

_OBJ = GeneratorModel.o main.o
OBJ = $(patsubst %, $(OBJDIR)/%, $(_OBJ))

# Make targets
.PHONY: all
all: $(TARGET)

$(OBJDIR):
  $(MKDIR) $(OBJDIR)

$(OBJDIR)/GeneratorModel.o: GeneratorModel.cpp GeneratorModel.h
  $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)

$(OBJDIR)/main.o: main.cpp GeneratorModel.h
  $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)

$(TARGET): $(OBJDIR) $(OBJ)
  $(CXX) -o $@ $(OBJ) $(CFLAGS)

.PHONY: clean
clean:
  $(RM) $(OBJDIR)
```

Build rendszerek – CMake

Telepítés

- CMake
  - https://cmake.org/
- Platformfüggetlen és fordítófüggetlen generátor alacsonyabb szintű fordító eszközökhöz.
  - Pl. GNU Make, MSVC, Ninja.
- Telepítés:
  - A hivatalos weboldalon elérhetőek a binárisok és telepítők Windows, Linux és Mac operációs rendszerekre is. https://cmake.org/download/
  - UNIX alapú operációs rendszerekre jellemzően *package repository*-ból is telepíthető.
    - Debian/Ubuntu: `apt-get install cmake`

Build rendszerek – CMake

ELTE | IK
INFORMATIKAI KAR

Használat

- A konfigurációt a *CMakeLists.txt* állományban adjuk meg.
- Futtatás: **`cmake <paraméterek>`**
- Példa CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)

project(HelloWorld)

add_executable(HelloWorld main.cpp foo.cpp)
```

Minimálisan szükséges CMake verzió

Projekt neve

Új végrehajtható bináris cél hozzáadása:
HelloWorld bináris készítése:
main.cpp és a *foo.cpp* forrásfájlokból

Build rendszerek – CMake

Változók

- Definiálhatók változók és feloldhatjuk a `${name}` szintaxissal:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)

set(SRCS \
  main.cpp \
  foo.cpp \
  foo.h)

add_executable(HelloWorld ${SRCS})
```

Build rendszerek – CMake

C++ fordító paraméterezése

- Testre szabhatjuk a C++ fordító paraméterezését:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)
add_executable(HelloWorld main.cpp foo.cpp)

target_compile_features(
  HelloWorld PRIVATE cxx_std_17)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)
```

GNU GCC specifikus kapcsolók

A láthatósági szabályozókra később térünk vissza

Programkönyvtárak fordítása

- Statikus vagy dinamikus programkönyvtár készítése:

```
cmake_minimum_required (VERSION 3.16)
project(MyStack)

add_library(my_stack_static STATIC mystack.cpp)
add_library(my_stack_dynamic SHARED mystack.cpp)
```

  - Statikus könyvtárak Windows alatt `.lib`, UNIX operációs rendszerek alatt `.a` kiterjesztéssel rendelkeznek.
  - Dinamikus könyvtárak Windows alatt `.dll`, UNIX operációs rendszerek alatt `.so` kiterjesztéssel rendelkeznek.

Build rendszerek – CMake

Include útvonal konfigurálása

- Bővítsük az *include* útvonalat az *include* könyvtárral:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)


target_compile_features(
  HelloWorld PRIVATE cxx_std_17)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)


include_directories("include")


add_executable(HelloWorld main.cpp foo.cpp)
```

Build rendszerek – CMake

Fordítási célok szerkesztése (*linkelése*)

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)

add_executable(HelloWorld
  main.cpp foo.cpp mystack.h)
add_library(MyStack STATIC mystack.cpp mystack.h)
target_compile_options(
  MyStack PRIVATE -O2 -pedantic)

target_link_libraries(HelloWorld MyStack)
```

- Ilyenkor van szerepe a korábban említett láthatósági szabályozóknak.

Fordítási célok szerkesztése (*linkelése*)

- A következő láthatósági szabályozók érhetőek el:
  - **PRIVATE**: a kapcsolók csak az adott fordítási célra vonatkoznak. A példában így a `-O2 -pedantic` kapcsolók a `HelloWorld` cél fordítására már nem érvényesek.
  - **PUBLIC**: a kapcsolók az adott fordítási célt feldolgozó további célokra is érvényesek.
  - **INTERFACE**: a kapcsolók az adott fordítási célra nem, de az adott fordítási célt feldolgozó további célokra érvényesek.

Build rendszerek – CMake

Csomagok

- A CMake számos használt külső függőségeket kezel csomagként és tudja azt a programunkhoz fordítani és szerkeszteni.

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

find_package(SomePackage)
include_directories(...)
link_directories(...)
link_libraries(...)
```

- Elegendő lehet a célhoz linkelni:

```
target_link_directories(mytarget ...)
target_link_libraries(mytarget ...)
```

Build rendszerek – CMake

Csomagok

- Példa az OpenCV könyvtár használatára:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

find_package(OpenCV REQUIRED)
link_libraries(${OpenCV_LIBS})
```

Build rendszerek – CMake

Csomagok

- Példa a Boost könyvtár használatára:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

set(Boost_USE_STATIC_LIBS ON)
set(Boost_USE_STATIC ON)
find_package(Boost REQUIRED
  COMPONENTS filesystem log program_options)
include_directories(${Boost_INCLUDE_DIRS})
link_libraries(${Boost_LIBRARIES})
```

Build rendszerek – CMake

Csomagok

- Példa a Qt könyvtár használatára:

```
cmake_minimum_required (VERSION 3.16)
project(QtHelloWorld)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTOUIC ON)

find_package(Qt5Widgets CONFIG REQUIRED)

set(SRCS mainwindow.ui mainwindow.cpp main.cpp)
add_executable(helloworld WIN32 ${SRCS})
target_link_libraries(helloworld Qt5::Widgets)
```

Build rendszerek – CMake

ELTE | IK
INFORMATIKAI KAR

Modulok

- A CMake támogatja a modularizált projekteket. Ilyenkor minden alkönyvtár saját *CMakeLists.txt* fájlt tartalmaz, a szülő könyvtár hivatkozza a kezelendő alkönyvtárakat:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

... Konfiguráció ...

add_subdirectory(module_a)
add_subdirectory(module_b)
```

Build rendszerek – CMake

ELTE | IK INFORMATIKAI KAR

Generátorok

- Jellemzően egy külön *build könyvtárba* szeretnénk fordítani:

  `mkdir build && cd build`

  `cmake ../` vagy `cmake ../src`

- Több féle generátor közül választhatunk:
  - GNU Makefile használata:
    `cmake -G "Unix Makefiles" ../`
  - Microsoft Visual Studio:
    `cmake -G "Visual Studio 17 2022" -A x64 ../`
  - Xcode:
    `cmake -G Xcode ../`
- Majd fordítjuk a programot:
  - Makefile: `make`
  - MSVC: `msbuild MyProgram.sln`

Build rendszerek – CMake

Kihelyezés

- Megadhatunk egy telepítési könyvtárat, ahova a végső binárisokat át szeretnénk másolni:

```
cmake -DCMAKE_INSTALL_PREFIX=../install ../src
```

- Példa CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

set(SRCS main.cpp foo.cpp foo.h)
add_executable(HelloWorld ${SRCS})
install(TARGETS HelloWorld
  DESTINATION ${CMAKE_INSTALL_PREFIX})
```

- Makefiles: **`make install`**
- MSVC: **`msbuild INSTALL.vcxproj`**

Build rendszerek – CMake

ELTE | IK
INFORMATIKAI KAR

Példa

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp
- CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)
project(ThesisGenerator)
include_directories(./)
add_executable(ThesisGenerator
    GeneratorModel.cpp
    GeneratorModel.h
    main.cpp)
target_compile_features(ThesisGenerator PUBLIC cxx_std_17)
install(TARGETS ThesisGenerator
    DESTINATION ${CMAKE_INSTALL_PREFIX})
```

Build rendszerek – Qt

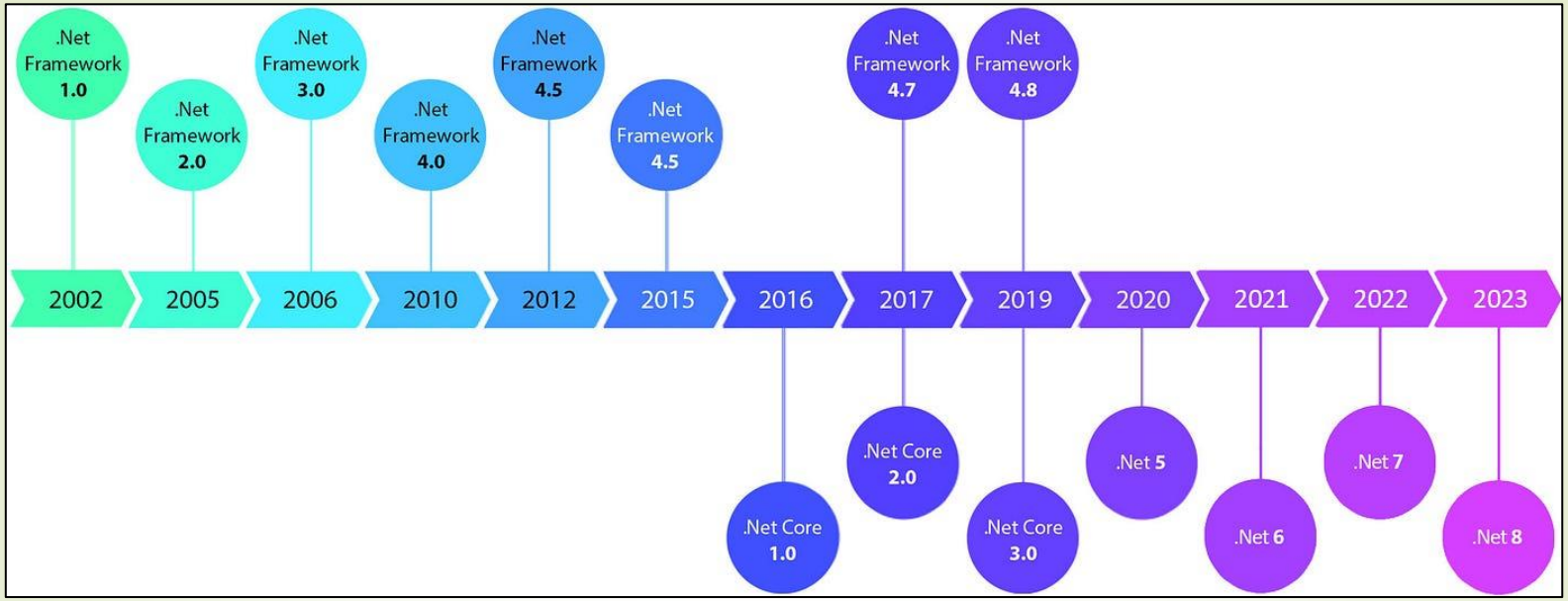
ELTE | IK
INFORMATIKAI KAR

Használat

- A Qt saját build rendszere a *Makefile*ra épít.
- Egy Qt projekt létrehozása és fordítása 3 lépésből áll:
  - `qmake -project`: új Qt projekt fájl (.pro fájl) létrehozása. Ezt a projekt fájlt manuálisan vagy IDE-vel szerkesztjük, nyilvántartja a projekt fájljait és konfigurációit.
  - `qmake`: *Makefile* generálása a projekt fájl alapján.
  - `make`: projekt fordítása GNU Make által.
- Ezt a folyamatot egy IDE, pl. a *Qt Creator* tudja gombnyomásra automatizálni.
- Minta projekt: Qt-s tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-qt
- A Qt már hivatalosan is támogatja a CMake-t is (pl. Qt Creator).

Build rendszerek – .NET

A .NET Framework keretrendszer



- *Problémák a .NET Framework keretrendszerrel:*
  - Windows-központú megközelítés
  - Monolitikus, nem megfelelően modularizált felépítés
  - Zárt forráskód (közreműködés szempontjából)

Build rendszerek – .NET

A .NET Core keretrendszer

- *A .NET Core ezekre nyújt megoldást:*
  - Platformfüggetlen (Windows, Linux, macOS) és nyílt forráskódú (https://github.com/dotnet/core)
    - Grafikus felülettel rendelkező alkalmazásokhoz is használható (.NET Core 3 óta), *desktop pack*-ek segítségével, de ekkor platformfüggő lehet a program
  - Modularizált felépítés, csak az alkalmazáshoz szükséges komponenseknek kell jelen lennie
    - Hasznos pl. microservicek készítéséhez, konténerek használatakor (pl. Docker), beágyazott rendszerekben
  - Magas teljesítmény és skálázhatóság (a .NET Core és az ASP.NET Core teljesítménye jelentősen jobb)

Build rendszerek – .NET

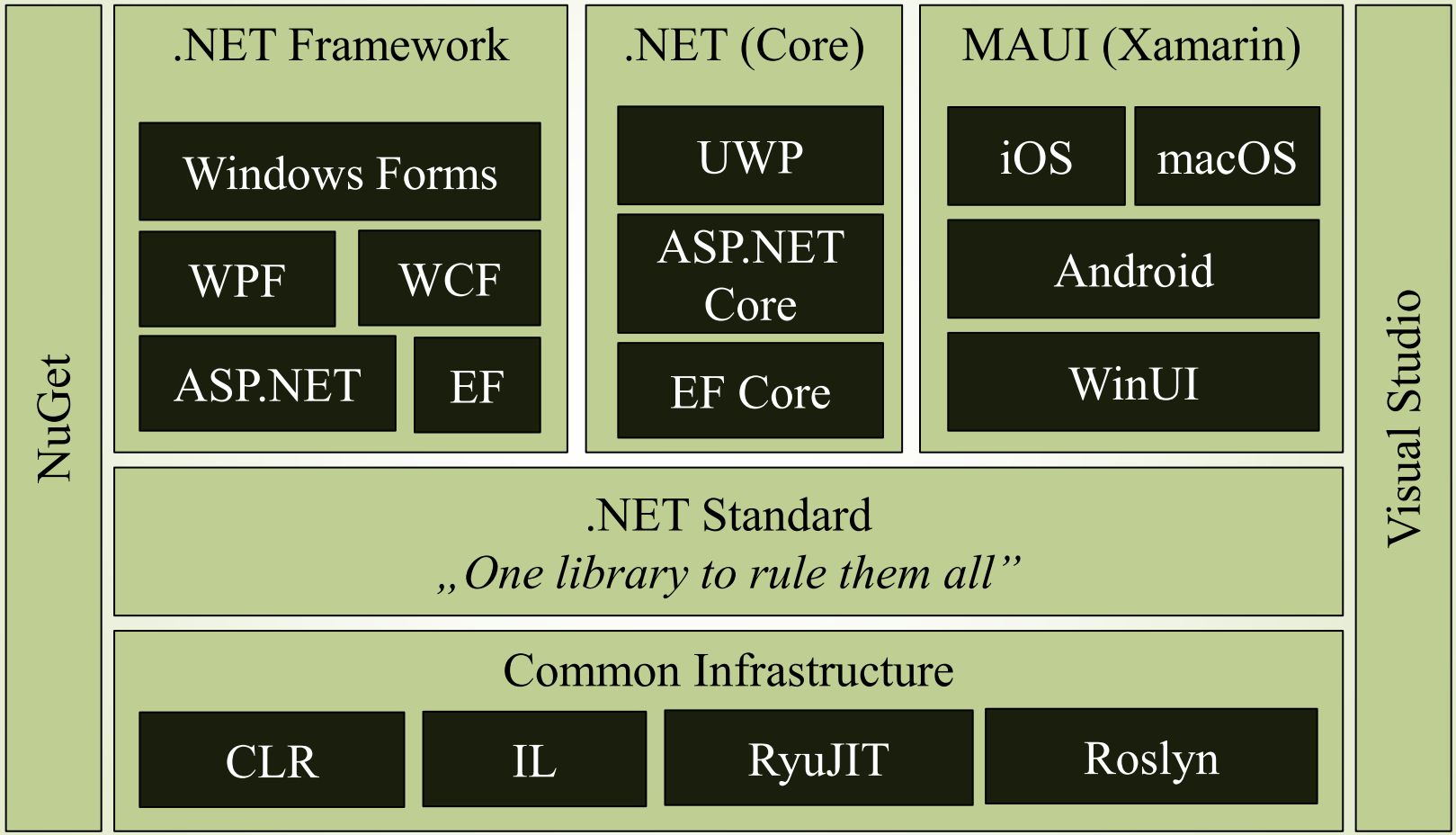
ELTE | IK
INFORMATIKAI KAR

A .NET Standard

- *Felmerülő problémák:*
  - Különböző keretrendszerben írt alkalmazások integrálása
  - Általános felhasználható programkönyvtárak fejlesztése
- *.NET Standard:*
  - Közös, megosztott API az egyes keretrendszer BCL-ek (Base Class Library) felett
  - Felváltja a PCL (Portable Class Library) projekteket

Build rendszerek – .NET

A .NET Standard



Build rendszerek – .NET

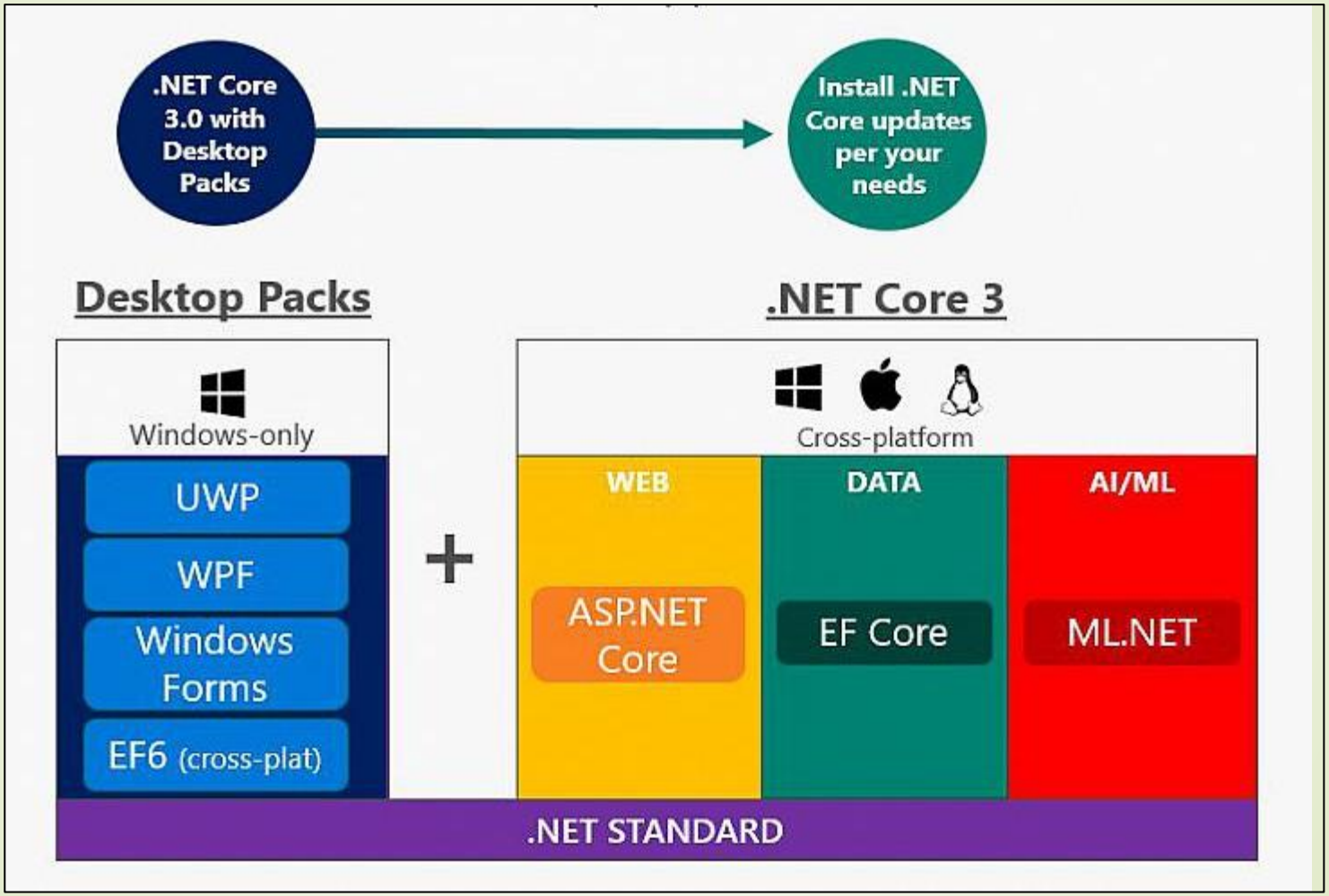
A .NET Standard

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	*N/A*
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	*N/A*
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2021.2

Forrás: https://docs.microsoft.com/en-us/dotnet/standard/net-standard

Build rendszerek – .NET

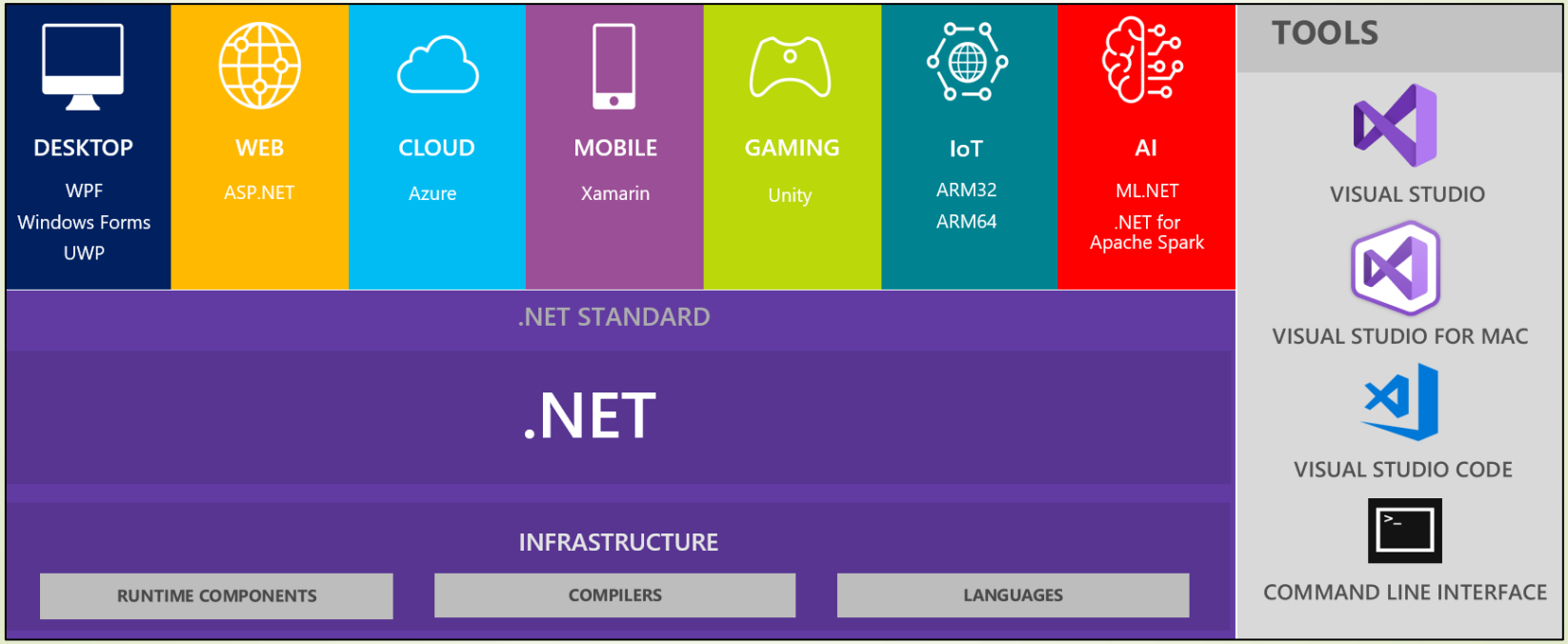
Grafikus alkalmazások .NET Core 3+ keretrendszerrel



Build rendszerek – .NET

A .NET keretrendszer jövője

- A .NET Framework-ből 4.8 az utolsó verzió
- A .NET Core 3.1 után a következő verzió a .NET 5 nevet kapta, a verziózásból adódó félreértések csökkentése érdekében.
  - Jelenleg a .NET 8 az aktuális LTS (*Long Term Support*) kiadás és a .NET 9 elérhető rövidebb támogatási élettartam mellett.



Build rendszerek – .NET

Fordítás .NET Framework keretrendszer alatt

- A Microsoft saját build eszköze Visual Studio projektekre.
  - A *.sln* solution és a *.csproj/.vcxproj* projekt fájlok alapján fordítja le az alkalmazást.
  - A projekt állományok XML formátumúak, a projekthez tartozó fájlok mellett fordítási konfigurációk és egyéb beállítások is megadhatók bennük.
- Például:

```
msbuild SolutionFile.sln /t:Build /p:Configuration=Release
```

  - a *SolutionFile.sln* fájlban adott solutionra
  - a *Build* target végrehajtása
  - a *Release* konfiguráció szerint
- Az MSBuild Mono projekt alatti implementációja az *XBuild*.

Build rendszerek – .NET

Fordítás .NET Core keretrendszer alatt

- A platformfüggetlen .NET Core keretrendszer alatt a *dotnet* konzolos utasítással érhetjük el az SDK-t.
  - Háttérben az azóta cross-platform vált *MSBuild*-et használja.
- A projekteket változatlanul solutionökre (.sln fájl) és projektekre (*.csproj/.vcxproj* fájlok) osztjuk, amelyek XML formátumúak.
- Telepítés Debian/Ubuntu: `apt-get install dotnet`
- A .NET Core build rendszerének használatára példa:
  - NuGet csomagok visszaállítása: `dotnet nuget restore`
  - Fordítás: `dotnet build path/to/SolutionFile.sln`
    Kurrens könyvtárra: `dotnet build`
  - Fordítás a *Release* konfiguráció szerint:
    `dotnet build --configuration Release`
  - Futtatatás: `dotnet run path/to/SolutionFile.sln`
    Már előállított binárisra: `dotnet path/to/Binary.exe`

Build rendszerek – .NET

Fordítás .NET Core keretrendszer alatt

- A .NET alkalmazás fordítási kimenete nem feltétlenül tartalmaz minden, a futtatásához szükséges állományt.
  - Kihelyezés (publikálás) alatt azt értjük, hogy a projekt összes fordított bináris állományát, a konfigurációs fájlokat és az összes függőséget is egy helyre másoljuk.
    - Legegyszerűbb formában egy kimeneti könyvtárba, webes alkalmazásnál egyből akár egy webszerverre kihelyezve.
  - Elvégezhető Visual Studioból vagy a `dotnet publish -c Release -o out_dir` paranccsal
  - Ezt követően futtatható is a webalkalmazás, amelyhez az SDK már nem, csak a .NET Runtime szükséges.
    - A `-s` kapcsolóval *self-contained* kihelyezés is lehetséges, amely tartalmazza a .NET Runtime szükséges elemeit is.

Build rendszerek – .NET

Javasolt felépítés a gyakorlati projektekhez

- Model projekt: felületfüggetlen, így egy .NET Standard könyvtárként implementálható. Függőségként használható lesz .NET Core vagy más keretrendszerű projektekben is.
- View projekt: specifikus (WinForms, WPF vagy Avalonia UI) projekt a .NET keretrendszer 8-as LTS verziójával. *Desktop pack* használatával megvalósítható, de ekkor platform specifikus (Windows) lesz a termék.
- Test projekt: .NET 8 projekt, így a folyamatos integráció (CI) Linux operációs rendszeren is elvégezhető lesz. Használhatjuk az *MSTest*, az *NUnit* vagy x*Unit* keretrendszert.

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-net

Szoftvertechnológia

Build rendszerek

C szakirány

Dr. Cserép Máté

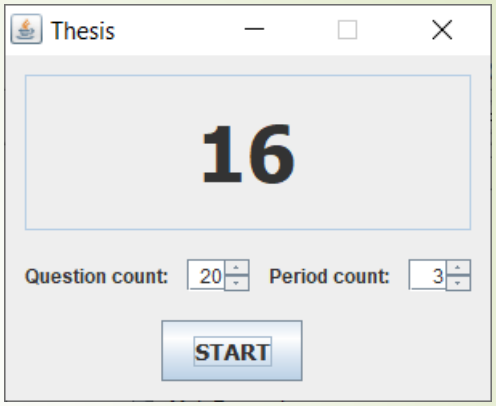
ELTE Informatikai Kar

2025.

Build rendszerek

Minta alkalmazás

- Tekintsünk egy minta Java alkalmazást.
- A *ThesisGenerator* alkalmazás szóbeli vizsgákhoz képes tételszámot generálni.
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-java
- Modell-Nézet architektúra:
  - **`thesisGenerator.model`** csomag: Felhasználói felülettől független üzleti logika
  - **`thesisGenerator.view`** csomag: Swing alapú felhasználói felület
  - **`thesisGenerator`** csomag: Főprogram



Build rendszerek

Java programok fordítása

```
mkdir dist
javac -d dist
  src\thesisGenerator\*.java
  src\thesisGenerator\model\*.java
  src\thesisGenerator\view\*.java


cd dist
jar -cfe thesis-generator.jar
  thesisGenerator.ThesisGenerator
  thesisGenerator\*.class
  thesisGenerator\model\*.class
  thesisGenerator\view\*.class


java -jar thesis-generator.jar
```

Java programok fordítása

- Mint látjuk, már egy egyszerűbb, néhány forrás fájlból álló program esetén sem egyszerű a megoldás.

- **Problémák:**
  - A programok manuális fordítása már kisebb programoknál is könnyen nehezen kezelhetővé válik.
  - Nagyobb programoknál nem követhető mely fordítási egységek újrafordítása szükséges.
    - A teljes alkalmazás újrafordítása valós vállalati alkalmazásoknál hosszabb időt is igényelhet.

Build rendszerek

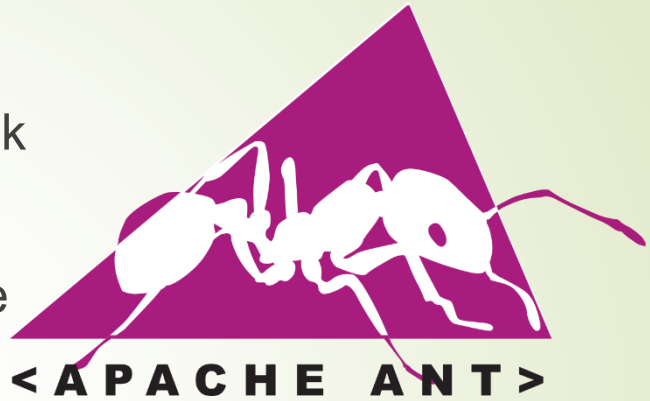
Elvárások a build rendszerekkel szemben

- A kód lefordítása
  - A fordítási célok (*target*) függőségeinek (*dependency*) kezelése
- A binárisok csomagolása
  - Többféle kiadási (*release*) opció támogatása
- Automatizált tesztek végrehajtása
- A binárisok kihelyezése (*deployment*) a teszt szerverre
- Kód átmásolása egyik helyről a másikra

- Csomagkezelő rendszerek (*package repository*) kezelése

Build rendszerek - Ant

Jellemzők



- Imperatív megközelítés
- Tipikusan Java projektekhez használják
- XML alapú build fájl
  - Alapértelmezetten *build.xml* a neve
- Hivatalos weboldal és tutorial:
  - https://ant.apache.org/
  - https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- Telepítés:
  - Windowsra a hivatalos weboldalról letölthető a telepítő.
  - UNIX rendszerekre a *package repository* tartalmazza.
    - Debian/Ubuntu: `apt-get install ant`
  - Egy IDE-vel együtt, pl. a NetBeans telepíti az Ant-ot.

Build rendszerek - Ant

Build fájl (build.xml)

- A *build.xml* egy projekt gyökérelemet tartalmaz, amely beállítja:
  - A projekt nevét
  - Az alapértelmezett célt (lásd később)
  - A projekt alapkönyvtárát, jellemzően az aktuális könyvtárat.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="projname"
         default="deftarget"
         basedir=".">
   ...
</project>
```

Build rendszerek - Ant

Cél definiálása

- A projekt elemen belül többféle típusú elem definiálható, a legfontosabbak a célok (*targets*):

```
<project ...>
  <target name="compile">
    ...
  </target>
</project>
```

  - A *target*-ek parancssorban is futtathatók
    `ant` **`compile`**

Könyvtár létrehozása

- Cél létrehozása, amely a *classes* könyvtárat hozza létre a fordítandó *.class* fájloknak:

```
<project ...>
  <target name="prepare">
    <mkdir dir="classes"/>
  </target>
</project>
```

- Parancssor:
  `ant` **`prepare`**

Célok közötti függőségek

- Definiáljunk egy célt, amely a *src* könyvtárban található összes Java forrásfájlt lefordítja, a kimenetet pedig a *classes* mappába helyezi.
  - Állítsuk be, hogy a *compile* cél függjön a *prepare* céltól!

```
<project ...>
  <target name="compile" depends="prepare">
    <javac srcdir="src" destdir="classes"/>
  </target>
</project>
```

  - Parancssor:
    `ant` **`compile`**

Tisztítási cél (*cleanup target*)

- Adjunk hozzá egy tisztítási célt, amely eltávolítja az összes lefordított bináris fájlt.

```
<project ...>
  <target name="clean">
    <delete>
      <fileset dir="classes" includes="*"/>
    </delete>
    <delete dir="classes"/>
  </target>
</project>
```

  - A fájlok külön törlése nem szükséges, a classes könyvtár törlése rekurzívan eltávolítja annak tartalmát.
  - Parancssor:
    `ant` **`clean`**

Tisztítási cél (*cleanup target*)

- A `failonerror` attribútummal beállítható, hogy a cél sikertelensége esetén a teljes folyamat megszakadjon-e:

```
<project ...>
  <target name="clean" failonerror="false">
    <delete dir="classes"/>
  </target>
</project>
```

- Parancssor:
  `ant` **`clean`**

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

Változók

- Egy projekten belül változókat (*properties*) is definiálhatunk.
  - A változók kulcs-érték párokat alkotnak.
  - A `${name}` szintaxis használatával futásidőben értékelődnek ki.

```
<project ...>
  <property name="jarname"
            value="filename.jar" />
  ...
</project>
```

  - Vannak beépített változók is, például a `${basedir}` a projekt alapkönyvtára.
    - https://ant.apache.org/manual/properties.html

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

Csomagolás

```
<project ...>
  <target name="jar" depends="compile">
    <jar destfile="${jarname}">
      <fileset dir="classes">
        <include name="*.class"/>
      </fileset>
      <manifest>
        <attribute name="Main-Class" value="Main"/>
      </manifest>
    </jar>
  </target>
</project>
```

- **Megjegyzés:** Fontos a belépési pont megadása a *manifest*-ben!

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

Cél: komplex példa

```
<target name="compile" depends="prepare,init">
  <javac destdir="build/classes" debug="on">
    <src path="src1/java"/>
    <src path="src2/java"/>
    <include name="**/*.java"/>
    <exclude name="com/comp/xyz/applet/*.java"/>
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </javac>
</target>
```

Build rendszerek - Ant

Kihelyezés (fájlműveletek)

- A végleges binárisok másolása a célhelyre:

```
<target name="install" depends="jar">
  <mkdir dir="build/war/WEB-INF/lib"/>
  <copy todir="build/war/WEB-INF/lib">
    <fileset dir="lib">
      <include name="*.jar"/>
      <exclude name="servlet-api.jar"/>
      <exclude name="catalina-ant.jar"/>
      <exclude name="el-api.jar"/>
    </fileset>
  </copy>
</target>
```

  - Parancssor:
    `ant` **`install`**

JVM futtatása

- Meghatározhatunk egy célt a lefordított és csomagolt JAR fájl futtatására:

```
<target name="run" depends="jar">
  <java jar="${jarname}" fork="true" />
</target>
```

  - A `fork` attribútum biztosítja, hogy a feladat külön folyamatban és külön Java virtuális gépben (JVM) fusson.
  - Parancssor:
    `ant` **`run`**

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

JVM futtatása

- Bonyolultabb futtatási példa:

```
<target name="run">
  <java classname="com.comp.foo.TestClient"
        jvmargs="-Xdebug server=y,suspend=n">
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </java>
</target>
```

  - Parancssor:
    `ant` **`run`**

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

API dokumentáció generálása

```
<target name="doc">
  <tstamp>
    <format property="timestamp" pattern="d.M.yyyy"
            locale="en"/>
  </tstamp>
  <mkdir dir="doc"/>
  <javadoc sourcepath="src" destdir="doc"
           windowtitle="Projekt dokumentáció">
    <header>Nagyon fontos projekt</header>
    <footer>Javadocs fordítva ${timestamp}></footer>
  </javadoc>
</target>
```

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

Unit tesztek végrehajtása

```
<target name="test" depends="compile">
  <mkdir dir="report"/>
  <junit printsummary="yes" haltonfailure="no">
    <classpath location="build/classes" />
    <classpath location="build/test/classes" />
    <classpath location="lib/junit-4.12.jar" />
    <classpath location="lib/hamcrest-core-1.3.jar" />

    <formatter type="plain" />
    <test name="com.comp.foo.ServerTest"
          haltonfailure="no" todir="report" />
  </junit>
</target>
```

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

Unit tesztek végrehajtása

```
<target name="test" depends="compile">
  <mkdir dir="report"/>
  <junit printsummary="yes" haltonfailure="no">
    <classpath location="build/classes" />
    <classpath location="build/test/classes" />
    <classpath location="lib/junit-4.12.jar" />
    <classpath location="lib/hamcrest-core-1.3.jar" />

    <formatter type="plain" />
    <batchtest fork="yes" todir="report">
      <fileset dir="test">
          <include name="**/*Test.java" />
      </fileset>
    </batchtest>
  </junit>
</target>
```

Build rendszerek - Ant

ELTE | IK
INFORMATIKAI KAR

A ThesisGenerator alkalmazás teljes build.xml fájlja

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="ThesisGenerator" default="jar" basedir=".">
    <target name="clean">
        <delete dir="build"/>
    </target>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/ThesisGenerator.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="thesisGenerator.ThesisGenerator"/>
            </manifest>
        </jar>
    </target>
    <target name="run" depends="jar">
        <java jar="build/jar/ThesisGenerator.jar" fork="true"/>
    </target>
    <target name="doc" depends="compile">
      <tstamp>
        <format property="timestamp" pattern="d.M.yyyy" locale="en"/>
      </tstamp>
      <mkdir dir="doc"/>
      <javadoc sourcepath="src" destdir="doc" windowtitle="Thesis Generator">
        <header>Thesis Generator</header>
        <footer>Javadocs compiled ${timestamp}></footer>
        <fileset dir="src/" includes="**/*.java" />
      </javadoc>
    </target>
    <target name="test" depends="compile">
      <mkdir dir="report"/>
      <junit printsummary="yes" haltonfailure="no">
        <classpath location="lib/junit-4.12.jar" />
        <classpath location="lib/hamcrest-core-1.3.jar" />
        <classpath location="build/classes" />
        <classpath location="build/test/classes" />
        <formatter type="xml" />
        <formatter type="plain" />
        <batchtest fork="yes" todir="report">
          <fileset dir="test">
              <include name="**/*Test.java" />
          </fileset>
        </batchtest>
      </junit>
    </target>
</project>
```

Build rendszerek - Ant

NetBeans

- Alapértelmezetten a NetBeans az Ant-et használja build rendszerként.
  - *a build.xml* a projekt gyökérkönyvtárában található.
  - hivatkozik a *nbproject/build-impl.xml* fájlra, amelyet a NetBeans automatikusan generál, ezt ne módosítsuk manuálisan
  - *a build.xml* fájlban definiálhatunk *hook* célokat, amelyeket a NetBeans build folyamata automatikusan meghív:
    - *-pre-init*, *-post-init*, *-pre-compile*, *-post-compile*, *-pre-jar*, *-post-jar*, *-post-clean*, stb.

Build rendszerek - Maven

Funkciók

- Szoftverprojekt-menedzsment eszköz
- Projekt fordítása, tesztek futtatása, függőségek kezelése, dokumentáció
- Csomagkezelés: függőségek automatikus letöltése
- A build folyamat deklaratív megadása
- Fix, előre definiált könyvtárstruktúra és konvenciók
- Elsősorban Java-hoz, de pluginek segítségével más programozási nyelveket is kezelhet
- XML-alapú build fájl
  - Alapértelmezett neve: *pom.xml*
- Hivatalos weboldal és egy ajánlott oktatóanyag:
  - http://maven.apache.org/
  - https://www.baeldung.com/maven

Build rendszerek - Maven

Telepítés

- Önálló telepítés
  - Binárisok letölthetők a hivatalos weboldalról
  - Egyszerűen ki kell csomagolni egy tetszőleges helyre
  - A `MAVEN_HOME` környezeti változót erre a könyvtárra kell állítani
  - A `JAVA_HOME` környezeti változónak a JDK telepítési könyvtárára kell mutatnia
  - A `MAVEN_HOME\bin` könyvtárat hozzá kell adni a `PATH` változóhoz
- Telepítés a UNIX *package repository*-ból
  - Általában elérhető
  - Debian/Ubuntu: `apt-get install maven`
- Gyakran az IDE-kkel együtt is telepítésre kerül (pl. NetBeans, IntelliJ)

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Projekt

- Project Object Model (POM)
- A projekt egyedileg azonosított: a projekt csoportja (*group*), artifact azonosító (*artifact id)* és verzió (*version*) alapján. Ezt a hármat gyakran GAV rövidítéssel jelölik.
- A projekt több modulra is osztható, amelyek önállóan kezelhetők

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Project Object Model (pom.xml)

- A Project Object Model (pom.xml) tartalmazza a projekt minden fontos információját:
  - azonosítók: groupId, artifactId, version
  - hogyan van build-elve a projekt
  - a build eredménye
  - tesztesetek
  - a projekt függőségei

Project Object Model (pom.xml)

- A pom.xml fájl gyökéreleme egyben `project` elem is.
- A project elemen belül kötelezően megadandó elemek:
  - `modelVersion`: a POM specifikáció verziója
  - `groupId`: a projektet létrehozó cég vagy csoport egyedi azonosítója, Java package névválasztási szabályok szerint. Ez azt jelenti, hogy fordított tartománynévvel kezdődik, pl. *hu.elte.inf*
  - `artifactId`: a projekt egyedi neve
  - `version`: a projekt verziója
  - `packaging`: a csomagolás típusa (alapértelmezett: jar, egyéb lehetőségek: pom, maven-plugin, ejb, war, ear, rar, par)

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Project Object Model (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.software</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
</project>
```

Build rendszerek - Maven

Könyvtárstruktúra

- Egy Maven projekt könyvtárstruktúrája előírt konvenciókon alapul.
  - Az alapértelmezett könyvtárelrendezés felüldefiniálható projektleírókkal, de ez nem gyakori és nem javasolt.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Könyvtárstruktúra felüldefiniálása

```
<project>
   ...
   <build>
     <directory>target</directory>
     <outputDirectory>classes</outputDirectory>
     <finalName>${project.artifactId}-${project.version}</finalName>
     <testOutputDirectory>test-classes</testOutputDirectory>
     <sourceDirectory>src/main/java</sourceDirectory>
     <scriptSourceDirectory>src/main/scripts
     </scriptSourceDirectory>
     <testSourceDirectory>/src/test/java</testSourceDirectory>
     ...
   </build>
</project>
```

Életciklus fázisok

- A Maven build rendszer egy meghatározott életciklust (*lifecycle*) követ, amely különböző fázisokból áll. A legfontosabb fázisok az alapértelmezett életciklusban:
  - **validate:** ellenőrzi, hogy a projekt helyesen van-e definiálva és minden szükséges információ elérhető-e
  - **compile:** lefordítja a projekt forráskódját
  - **test:** teszteli a lefordított kódot egy megfelelő egységtesztelési keretrendszerrel. Ezek a tesztek nem igényelhetik a kód csomagolását vagy kihelyezését
  - **package:** becsomagolja a lefordított kódot egy terjeszthető formátumba, pl. JAR
  - **integration-test:** szükség esetén feldolgozza és kihelyezi a csomagot egy tesztelési környezetbe, ahol további integrációs tesztek futtathatók

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Életciklus fázisok

- **verify:** ellenőrzések lefuttatása, hogy a csomag érvényes és megfelel a minőségi kritériumoknak
- **install:** a csomag telepítése a helyi tárolóba, hogy más helyi projektek függőségként felhasználhassák
- **deploy:** integrációs vagy kiadási környezetben történik, a végső csomagot átmásolja a távoli tárolóba, hogy más fejlesztők és projektek is hozzáférhessenek

- További részletek: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
- Parancssor: `mvn install`
  - Az *install* fázisig hajtja végre a fázisokat az alapértelmezett életciklusban, de a *deploy* fázist nem tartalmazza.

Életciklus fázisok

- Az alapértelmezett életciklus mellett léteznek más életciklusok is, például a clean életciklus, amely a korábban lefordított binárisok eltávolítására szolgál egy projektből.
  - Ennek az életciklusnak 3 fázisa van:
    pre-clean, clean, post-clean.
- Parancssor: `mvn clean install`
  - Végrehajtja a *clean*, majd az *install* fázist, valamint az összes azt megelőző fázist.
  - Végső soron ez eltávolítja és újraépíti az összes binárist.

Célok (*goals*)

- A fordítási fázisok egy vagy több célból (*goal*) állnak
- A cél egy olyan feladat, amely a projekt fordításához vagy kezeléséhez kapcsolódik
  - A célok sorrendjét a fázis kötése (*binding*) határozza meg
  - Sok fázis csak egy célt tartalmaz
  - Például a `compile` fázis a `compiler:compile` célt tartalmazza
- Nemcsak fázisokat, hanem célokat is megadhatunk a Maven parancsban, így csak az adott cél kerül végrehajtársra az előző fázisok és céljaik nélkül.
  - Például: `mvn compiler:compile`
- Egyedi fázisok és célok is definiálhatók (a pom.xml fájlban)

Build rendszerek - Maven

Tárolók (*repositories*)

- Egy tároló a build eredményeit (*artifacts*) és a függőségeket tartalmazza.
  - Az alapértelmezett helyi tároló a fejlesztő home könyvtárában található:
    ~/.m2/repository
- Ha egy *artifact* elérhető a helyi tárolóban, a Maven azt használja.
- Ellenkező esetben a központi tárolóból tölti le, majd eltárolja a helyi tárolóban.
  - Az első build során a hálózati forgalom és a build idő jelentősen megnőhet.
  - Az alapértelmezett központi tároló a Maven Central: https://repo.maven.apache.org
  - A Maven beállítható, hogy mely tárolókat és *mirror*-okat használjon a *~/.m2/settings.xml* fájlban.
    - Vállalatok gyakran saját belső központi tárolókat üzemeltetnek.

Build folyamat eredménye

- A fordítás során létrejön a *target* könyvtár, amely a fordítás során újonnan generált fájlokat tartalmazza
  - output, pl.: *my-app-1.0-SNAPSHOT.jar*
  - *classes* könyvtár: a fordítás során létrejött osztályfájlok (*class files*), de a tesztosztályok nélkül
  - *test-classes*: teszt forrásokból készült osztályok
  - *maven-archiver/pom.properties*: a projekt GAV azonosítóját leíró fájl
  - *surefire-reports*: a tesztelési jelentések

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Projekt hierarchiák

- A Maven támogatja az almodulos projekteket:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>parent-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <!-- subprojects -->
  <modules>
    <module>first-child-app</module>
    <module>second-child-app</module>
  </modules>
</project>
```

Projekt hierarchiák

- Az almodul projektek szintén hivatkoznak a szülőjükre:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>parent-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.mycompany.app</groupId>
  <artifactId>first-child-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  ...
</project>
```

Build rendszerek - Maven

Pluginek

- A Maven alapfunkciói korlátozottak, de a Maven egy bővíthető (*pluginable*) keretrendszer.
- Számos különböző plugin elérhető
  - például: C++, LaTeX, ant build, javadoc stb.
  - A hivatalos pluginek listája: https://maven.apache.org/plugins/
  - Léteznek harmadik féltől származó pluginek is, és saját pluginok is készíthetőek

Plugin példa: Javadoc

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

- Dokumentáció generálása: `mvn javadoc:javadoc` vagy `mvn:site`

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Függőségek

- Azokat a külső könyvtárakat, amelyeket a projekt használ, függőségeknek nevezzük (*dependencies*).
- A Maven függőségkezelési funkciója biztosítja a könyvtárak automatikus letöltését egy központi tárolóból.

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- A GAV egyértelműen azonosítja a szükséges *artifact*ot
- A *scope* határozza meg, hogyan használjuk a függőséget

A függőségek *scope*-jai

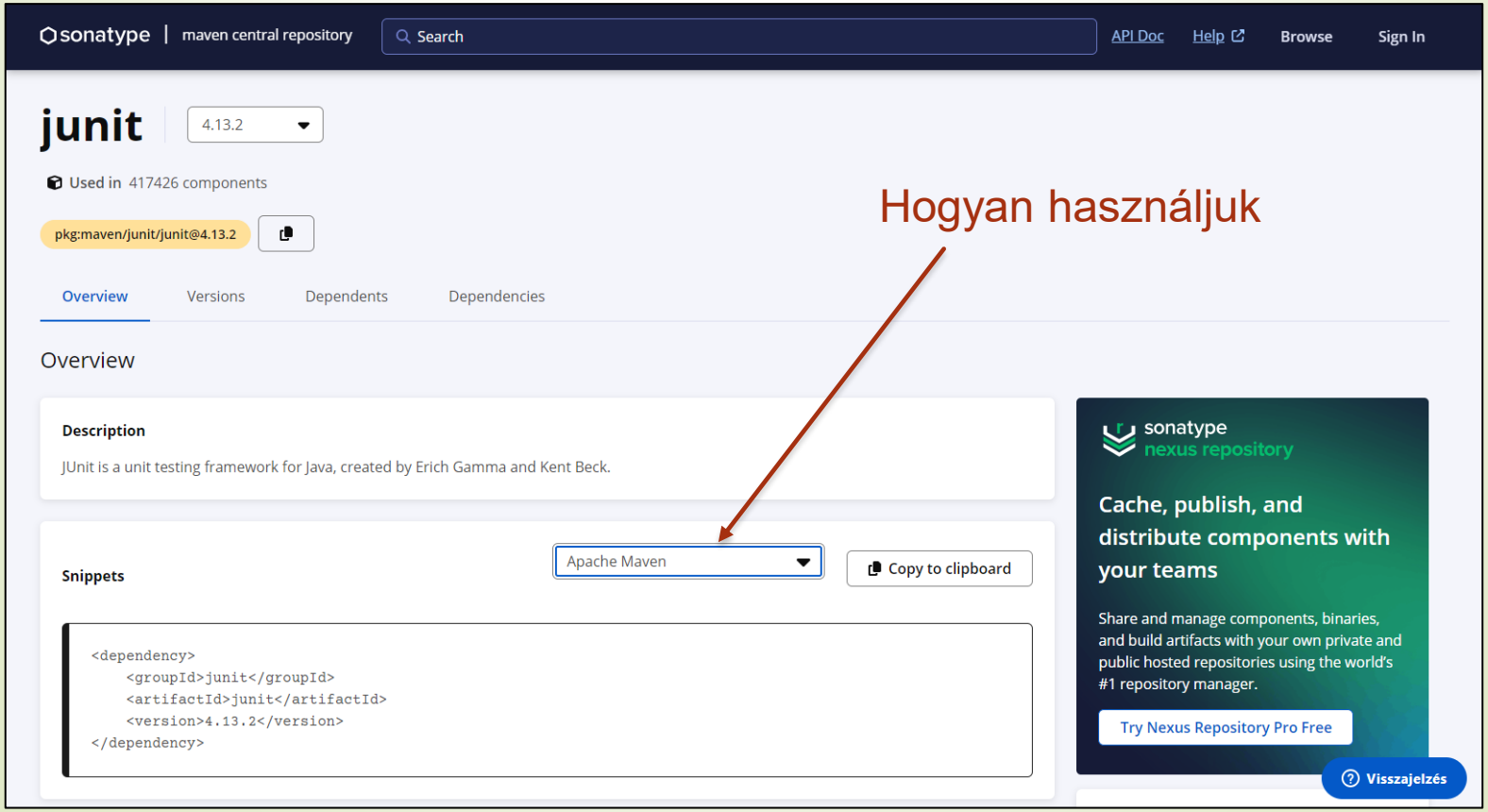
- A legfontosabb *scope* típusok:
  - **compile:** Alapértelmezett, ha nincs megadva. A fordításhoz szükséges függőségek.
  - **runtime:** Futásidőben szükséges függőségek, de a fordítás során nem szükségesek.
  - **test:** Csak teszteléshez szükséges függőségek, nem részei az éles alkalmazásnak.

Build rendszerek - Maven

Függőségek keresése

- A Maven Central tárolóban elérhető könyvtárak böngészhetők és kereshetők:
  - https://central.sonatype.com/

Új alapértelmezett URL és felület:
https://central.sonatype.com/



Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Teljes pom.xml fájl a *ThesisGenerator* alkalmazáshoz

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>hu.elte.inf</groupId>
    <artifactId>thesis-generator</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>15</maven.compiler.source>
        <maven.compiler.target>15</maven.compiler.target>
        <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
        <maven-project-info-reports-plugin.version>3.0.0</maven-project-info-reports-plugin.version>
        <maven-javadoc-plugin.version>3.2.0</maven-javadoc-plugin.version>
        <junit.version>4.13.2</junit.version>
    </properties>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>${maven-jar-plugin.version}</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <mainClass>thesisGenerator.ThesisGenerator</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Build rendszerek - Maven

ELTE | IK
INFORMATIKAI KAR

Teljes pom.xml fájl a *ThesisGenerator* alkalmazáshoz

```
38      <reporting>
39          <plugins>
40              <plugin>
41                  <groupId>org.apache.maven.plugins</groupId>
42                  <artifactId>maven-project-info-reports-plugin</artifactId>
43                  <version>${maven-project-info-reports-plugin.version}</version>
44              </plugin>
45              <plugin>
46                  <groupId>org.apache.maven.plugins</groupId>
47                  <artifactId>maven-javadoc-plugin</artifactId>
48                  <version>${maven-javadoc-plugin.version}</version>
49                  <configuration>
50                      <doctitle>My API for ${project.name} ${project.version}</doctitle>
51                      <windowtitle>My API for ${project.name} ${project.version}</windowtitle>
52                      <show>public</show>
53                  </configuration>
54              </plugin>
55          </plugins>
56      </reporting>
57      <dependencies>
58          <dependency>
59              <groupId>junit</groupId>
60              <artifactId>junit</artifactId>
61              <version>${junit.version}</version>
62              <scope>test</scope>
63          </dependency>
64      </dependencies>
65  </project>
```

Build rendszerek - Gradle

Funkciók

- Egyre népszerűbb build automatizációs rendszer [1] [2]
- Célja az Ant és a Maven legjobb koncepcióinak egyesítése
- Támogatja az inkrementális buildeket
  - Jelentős teljesítménynövekedést nyújt nagyobb vállalati projektek esetén
- Konfiguráció XML helyett Groovy vagy Kotlin-alapú domain specifikus nyelvvel (DSL)
- Hivatalos weboldal: https://gradle.org/
- Oktatóanyag: https://docs.gradle.org/current/userguide/getting_started_eng.html

[1] https://www.baeldung.com/java-in-2019

[2] https://www.jetbrains.com/lp/devecosystem-2023/java/

Build rendszerek - Gradle

build.gradle

- A Gradle-ben a build rendszer a *build.gradle* fájlban van definiálva Groovy nyelven.
  - Kotlin használata esetén: *build.gradle.kts*
  - Alapértelmezett használathoz a konvenció szerint elnevezés használatát kövessük.
- Hogyan hozzunk létre *build.gradle* fájlt?
  1. Írhatjuk kézzel a nulláról
  2. A `gradle init` paranccsal automatikusan
  3. Használhatjuk preferált IDE-nket a generálására

Build rendszerek - Gradle

ELTE | IK
INFORMATIKAI KAR

Feladatok (*tasks*)

- A Gradle-ben a buildek egy vagy több projektből állnak, és minden projekt egy vagy több feladatból (*task*) áll.
  - Egy *task* egyetlen műveletet végez, pl. osztályok fordítása, archívumok létrehozása vagy publikálása.
  - Egyszerű *task* definíció:

```
task hello {
  doLast {
    println "Hello World"
  }
}
```

Viselkedés hozzáadása meglévő taskokhoz

- Meglévő taskok kiegészíthetők további műveletekkel, prefixálással vagy suffixálással.

```
task hello {
  doLast {
    println "Hello World"
  }
}

hello.doFirst {
  println "First Line"
}

hello.doLast {
  println "Last Line"
}
```

Task függőségek

- Függőségi kapcsolat definiálható taskok között a *dependsOn* argumentummal:

```
task init {
  doLast {
    ...
  }
}

task compile(dependsOn: init) {
  doLast {
    ...
  }
}
```

Build rendszerek - Gradle

Pluginek

- A meglévő taskok listázhatók a `gradle tasks` paranccsal.
- A build rendszer konfigurációjának egyszerűsítésére a Gradle számos beépített plugint kínál, amelyek előre definiált taskokat tartalmaznak.
  - A java plugin használatával elérhetővé válnak taskok általános Java-alapú alkalmazások fordítására, tesztelésére, csomagolására.

```
plugins {
  id "java"
}
```

  - Néhány parancs: `gradle assemble`, `gradle build`, `gradle jar`, `gradle test`, `gradle clean`

Pluginek

- Az *application* plugin (amely a *java* plugint is magába foglalja) lehetővé teszi futtatható build létrehozását.

```
plugins {
  id "application"
}
```

- A manifest belépési pontját még meg kell adni:

```
application {
  mainClass = "project.MainClass"
}
```

Build rendszerek - Gradle

Függőségek

- A Gradle rugalmas függőségkezelő rendszert támogat.
- Sok konvenciót és megoldást átvett a Maven-ből, például a Maven Central használatának lehetőségét a függőségek forrásaként.

```
repositories {
  mavenCentral()
}
```

- A függőségeket különböző konfigurációkba csoportosítjuk, pl. `implementation`, `testImplementation` vagy `runtimeOnly`, amelyek egymást is bővíthetik.

```
dependencies {
  testImplementation "junit:junit:4.13.2"
}
```

Build rendszerek - Gradle

Teljes build.gradle fájl a *ThesisGenerator* alkalmazáshoz

```
1. plugins {
2.     // Apply the application plugin. Implies the java plugin.
3.     id 'application'
4. }
5. application {
6.     // Define the main class for the application.
7.     mainClass = 'thesisGenerator.ThesisGenerator'
8. }
9. repositories {
10.    // Use the Maven Central Repository for dependencies.
11.    mavenCentral()
12. }
13. dependencies {
14.    // Use JUnit test framework.
15.    testImplementation 'junit:junit:4.13.2'
16. }
17. // Set source directories. Advised to use the conventional directory layout in general.
18. sourceSets {
19.    main.java.srcDirs = ['src']
20.    test.java.srcDirs = ['test']
21. }
22. // Set build directory.
23. project.buildDir = 'build-gradle'
24.
```

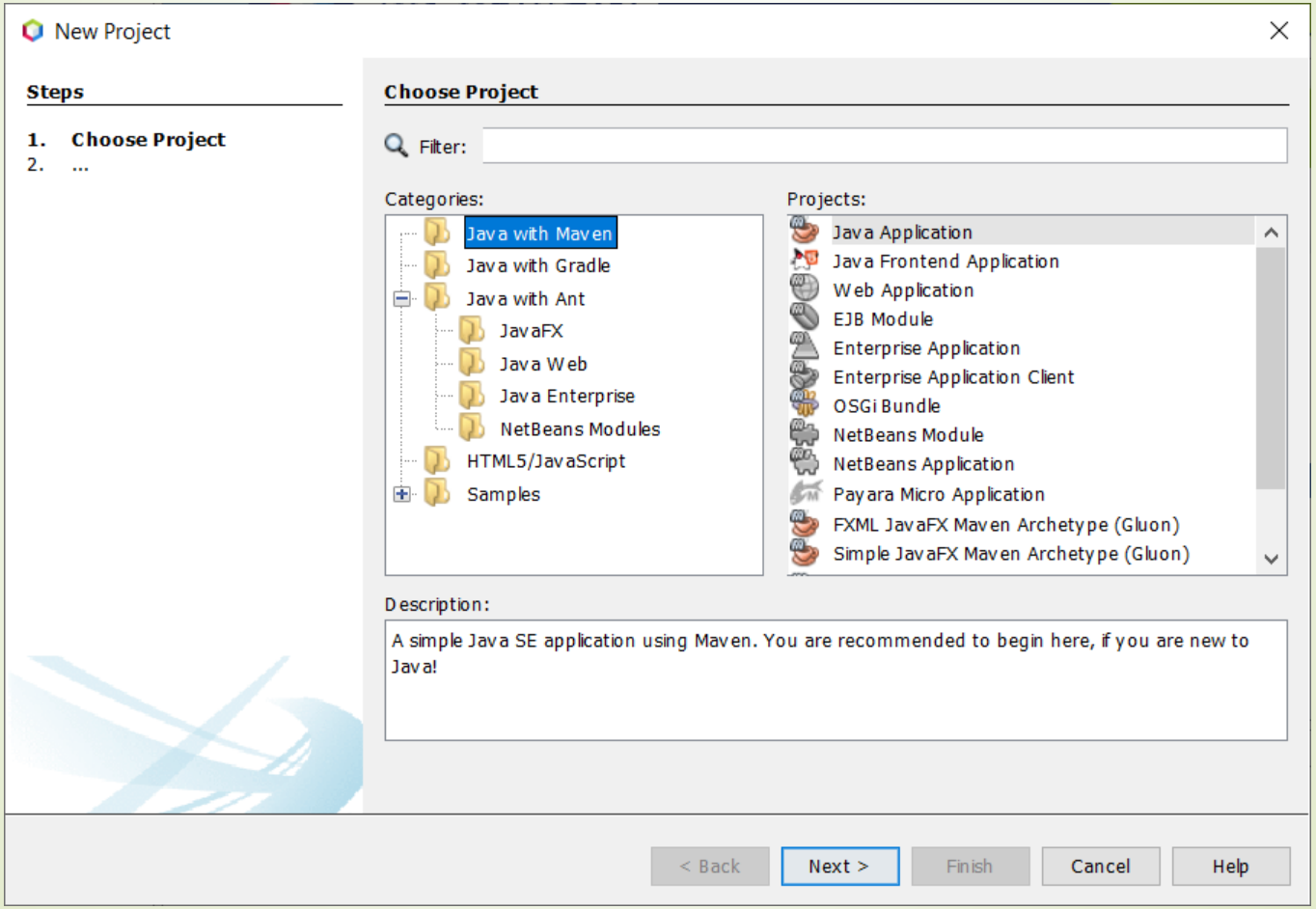
Gradle Wrapper

- Projekt létrehozásakor hozzáadható a *Gradle Wrapper* (gradlew). (Vagy később a `gradle wrapper` paranccsal.)
  - A Gradle Wrapper egy script, amely egy adott verziójú Gradle-t indít el, szükség esetén letölti azt.
- A Gradle build-ek futtatásának ajánlott módja a Gradle Wrapper használata. Előnyei:
  - A projekt létrehozása után nem kell manuálisan Gradle-t telepíteni a fejlesztőkörnyezetbe.
  - A projekt fejlesztői (és CI rendszerek) között rögzített Gradle verziót biztosít.
  - A projekt egyszerűen frissíthető új Gradle verzióra a Wrapper definíció módosításával.
- Verziókezelés: a `gradlew` Bash és Batch script, valamint a `gradle` könyvtár legyenek verziókezelve, a letöltött binárisokat tartalmazó `.gradle` könyvtár viszont ne.

Build rendszerek – IDE integráció

NetBeans

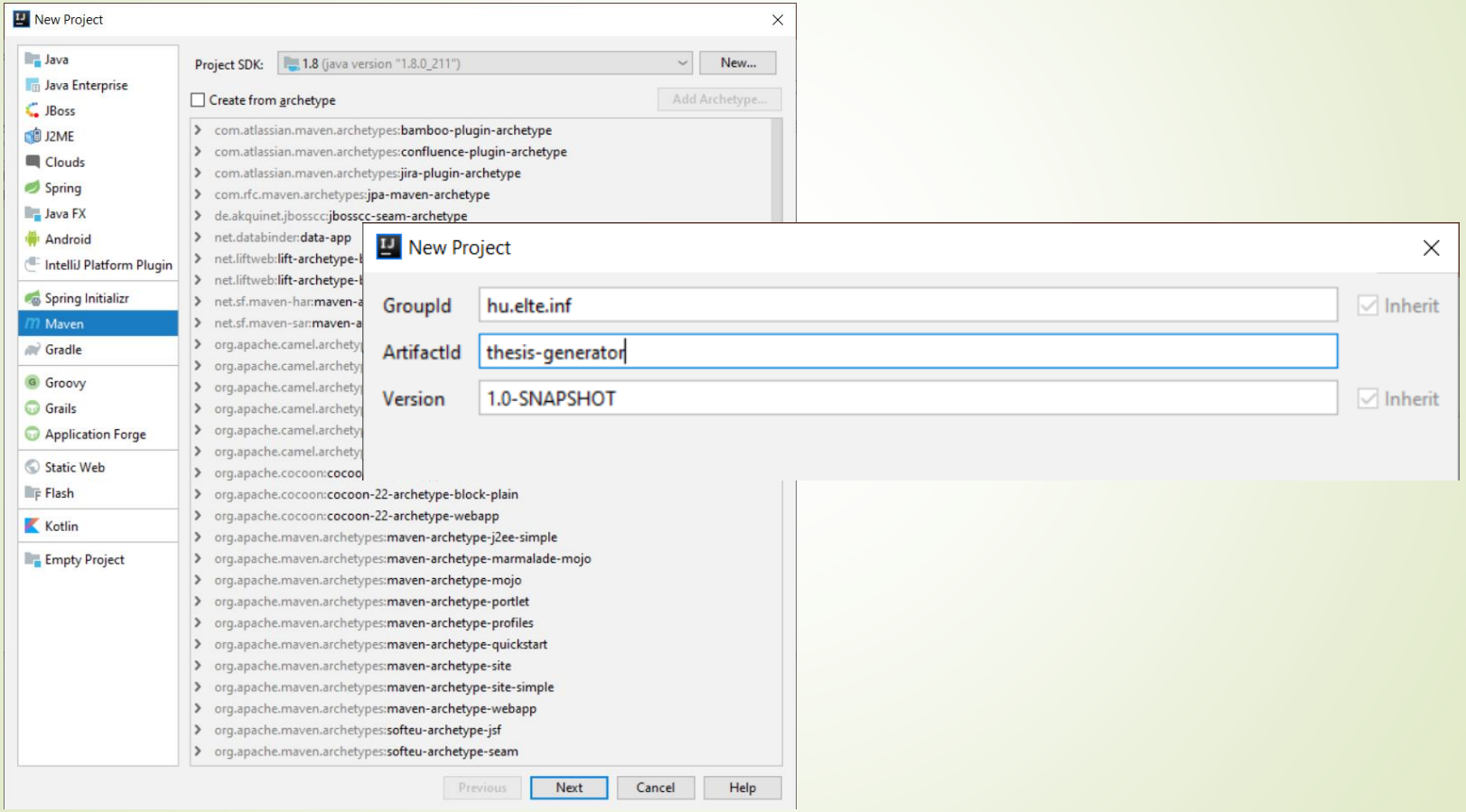
- A NetBeans támogatja az Ant, Maven és Gradle projekteket is.



Build rendszerek – IDE integráció

IntelliJ

- Az IntelliJ IDEA szoros integrációt kínál Maven és Gradle projektekhez.



- *Ant build fájl generálása is támogatott: Build -> Generate Ant Build*

Szoftvertechnológia

Folytonos integráció és kiadás

Continuous integration & delivery

A & B szakirány

Dr. Cserép Máté

ELTE Informatikai Kar

2025.

Folytonos integráció és kiadás

Folyamatos integráció

- A *folytonos integráció* (*continuous integration*, *CI*) egy olyan gyakorlati módszer, amely lehetővé teszi a programkódok ellenőrzésének és tesztelésének felgyorsítását
  - célja a lehetséges hibák, integrációs problémák azonnali, automatizált kiszűrése, visszajelzés a fejlesztőnek
  - a programkódok verziókezelő rendszer segítségével egy központi tárhelyre kerülnek, naponta többször
  - a tárhely tartalma minden módosítást követően automatikusan fordításra kerül (*build automation*), a fordítással pedig a lekódolt tesztek is végrehajtódnak
  - az így ellenőrzött kódot további tesztelés követheti

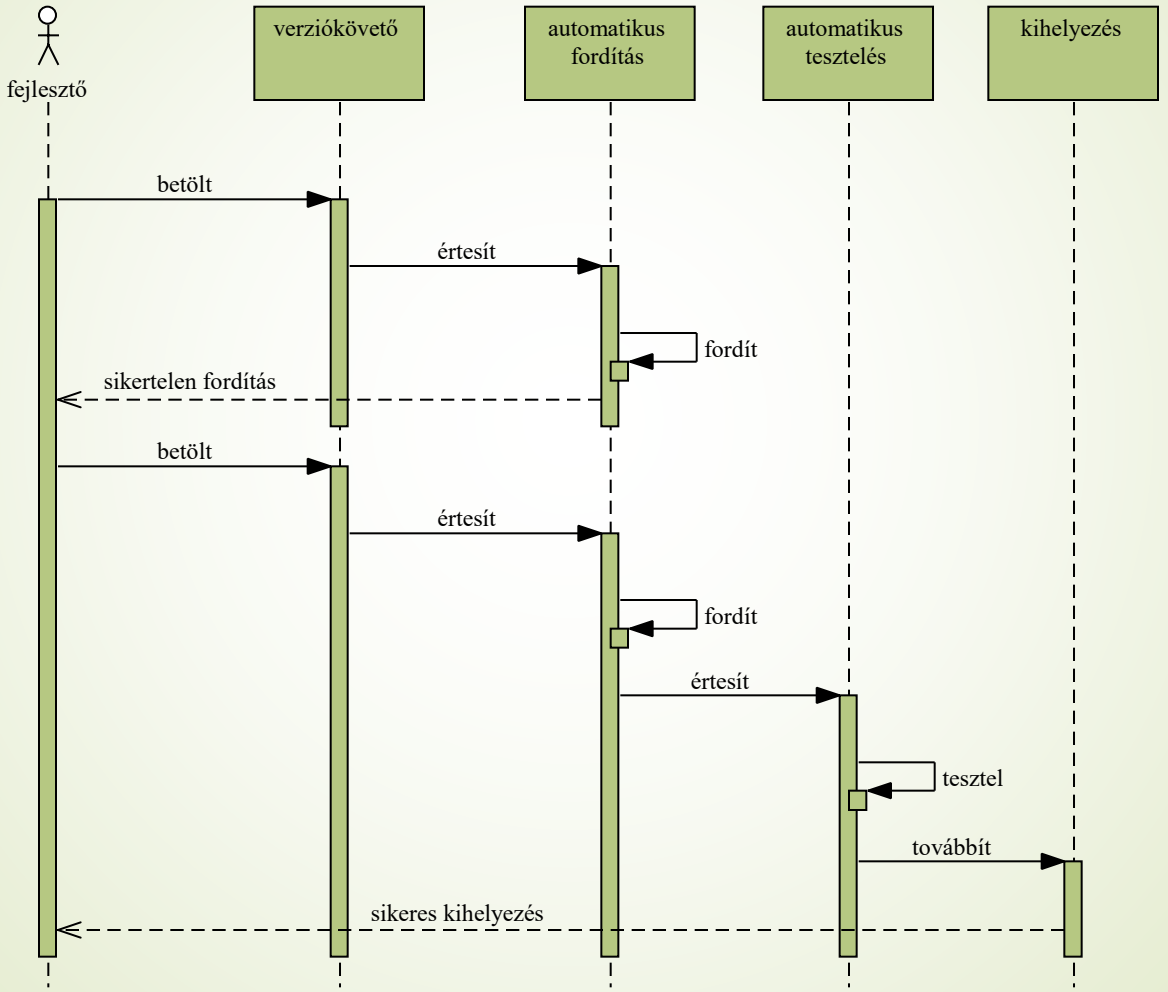
Folytonos integráció és kiadás

Folyamatos teljesítés

- Az agilis szoftverfejlesztés (*agile software development*) célja a gyors alkalmazásfejlesztés megvalósítása, inkrementális alapon
  - a szoftver folyamatos fejlesztés és kiadás alatt áll (*continuous delivery*), a sebesség állandó, a változtatások minden lépésben beépíthetőek (*welcome changes*)
  - a működő szoftver az előrehaladás mérőeszköze, előtérben az egyszerűség, ugyanakkor folyamatos odafigyelés a megfelelő tervezésre, optimalizációra
  - a fejlesztést általában önszervező, kis csapatok végzik, megosztott felelősséggel, folytonos interakcióval, gyors visszajelzésekkel
  - a folyamatos kiadások automatizálhatók, ekkor *continuous deployment*-ről beszélünk

Folytonos integráció és kiadás

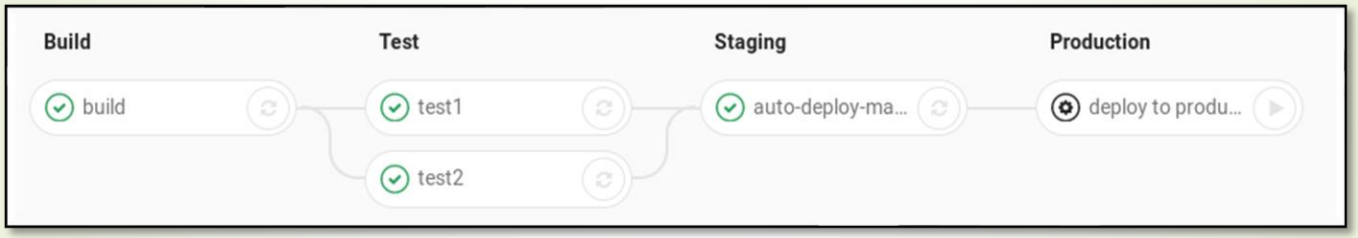
Folyamatos integráció és teljesítés

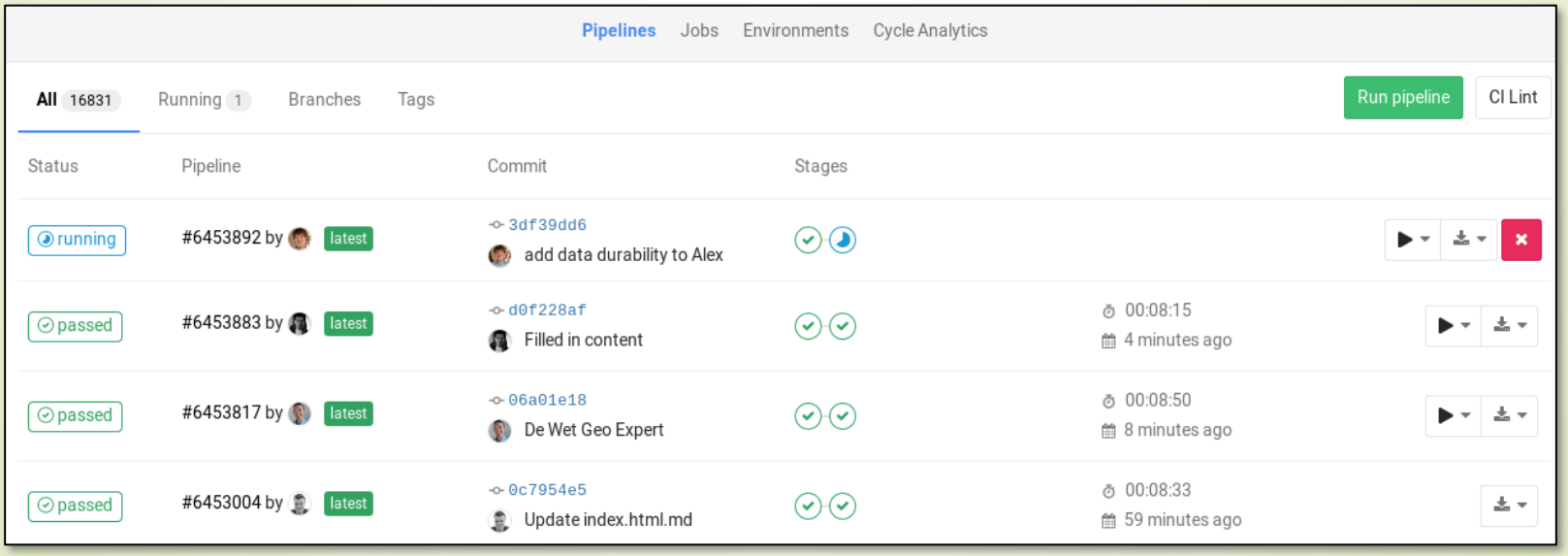


Folytonos integráció és kiadás

Feladatok

- A folyamatos integráció és teljesítés lépéseit egymásra épülő feladatok (*jobs*) láncolataként (*pipelines*) definiálhatjuk

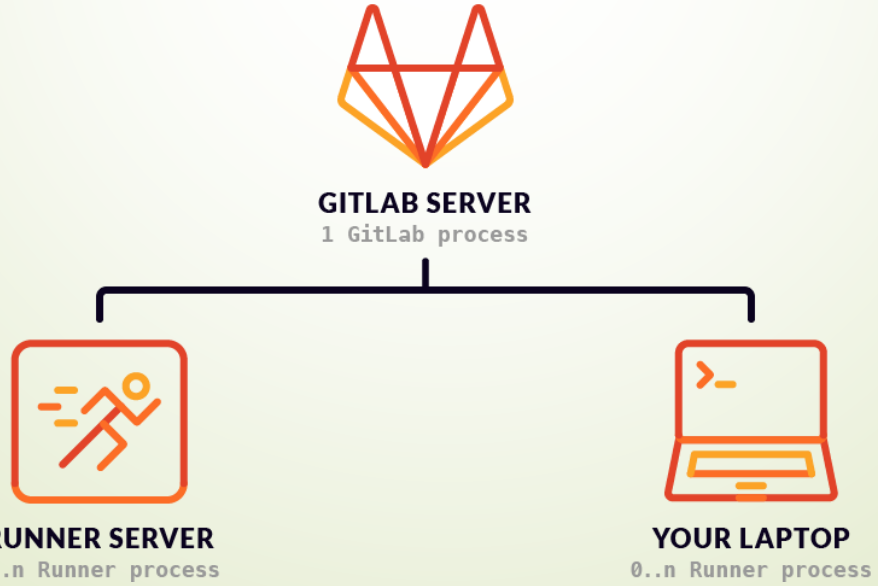




Folytonos integráció és kiadás

GitLab Runners

- A GitLab rendelkezik integrált, saját megoldással a folyamatos integráció és teljesítés támogatására
  - a feladatokat (*jobs*) a GitLab szervertől független ún. *GitLab Runner* példányok hajtják végre
    - Shell, SSH, VirtualBox, Docker, Kubernetes, stb.
  - a *runnerek* egyedileg konfigurálhatóak, lehetnek megosztottak vagy projekthez rendeltek



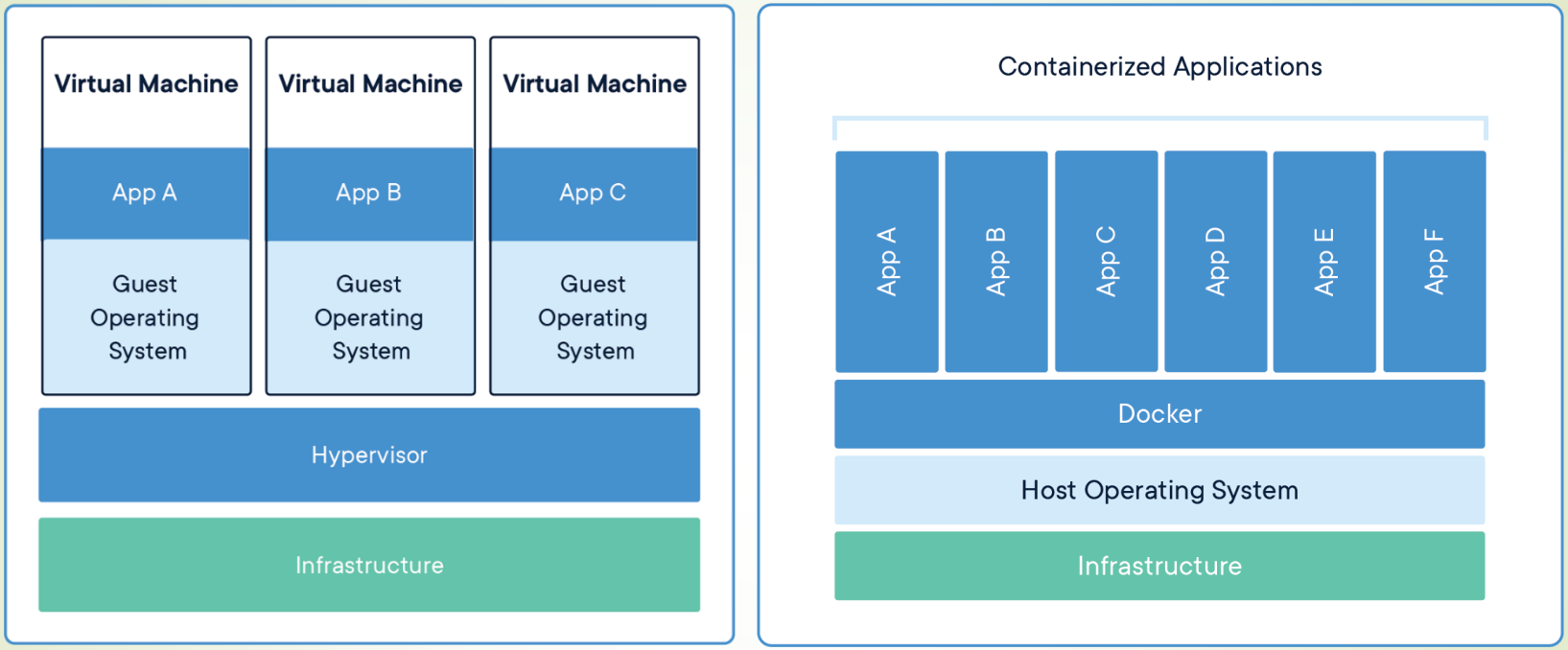
Folytonos integráció és kiadás

Docker

- A folyamatos integrációt egy izolált, reprodukálható környezetben érdemes végezni
- A Docker napjainkban a legelterjedtebb *container framework*
  - a *container* hasonlít a virtuális gépekhez (*VM)* olyan tekintetben, hogy egy teljesen elkülönített, virtualizált környezett biztosít, amelynek a gazdaszámítógép szolgáltat erőforrásokat
  - a fő különbség a *containerek* és a virtuális gépek között, hogy minden *container* osztozik a gazda kerneljén a többi *containerrel*, a virtualizált hardver és az OS nem része, csupán az alkalmazásunkhoz kötődő könyvtárak, binárisok és a felhasználói terület
  - a *containerek* ezáltal nagyságrendekkel kisebb *overheaddel* bírnak a VM-ekhez képest, így könnyebb súlyú megoldást nyújtanak a virtualizációra

Folytonos integráció és kiadás

Virtuális gépek és *containerek*



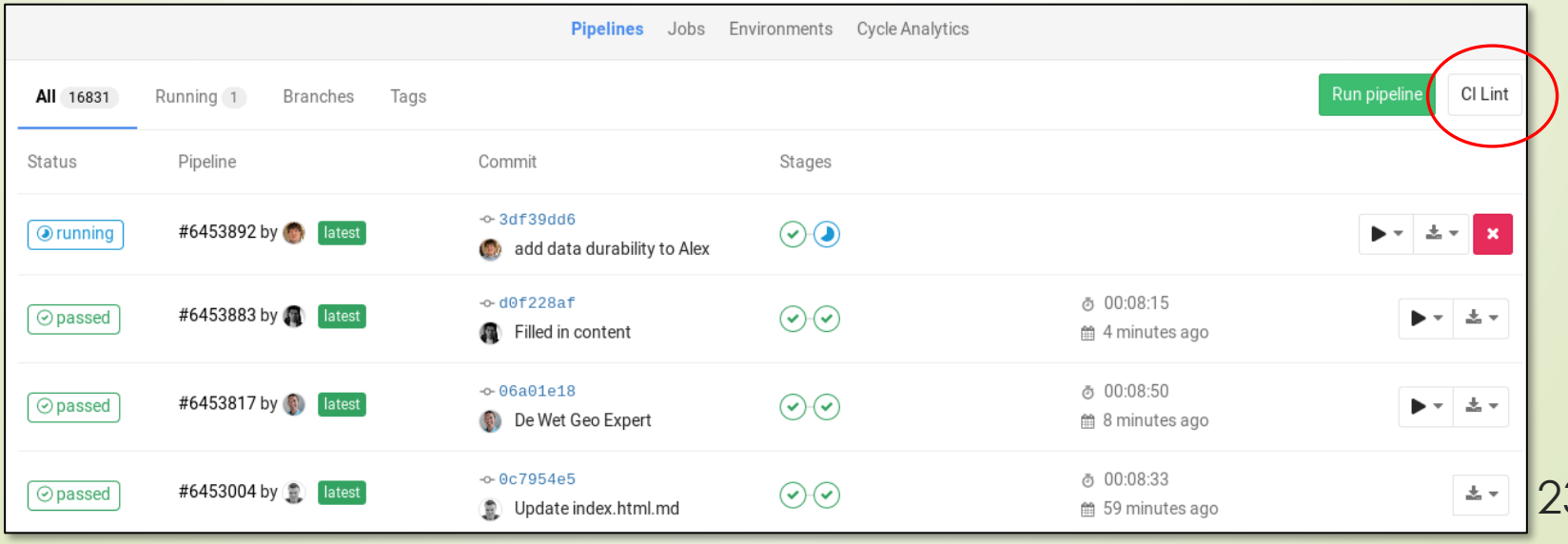
Forrás: docker.com

- a *container framework*ök lehetőséget adnak hordozható alkalmazások létrehozására és menedzselésére
- az alkalmazások modularizálhatóak és skálázhatóak, a komponensek külön *container*ben futhatnak

Folytonos integráció és kiadás

GitLab CI/CD

- A folyamatos integráció konfigurációját a `.gitlab-ci.yml` fájl tartalmazza, YAML formátumban
  - a YAML (*YAML Ain't Markup Language*) egy emberi szemmel könnye(bbe)n olvasható strukturált leíró nyelv
  - https://yaml.org/spec/1.2.2/
- A GitLab webes felületén elérhető egy CI Lint funkció a formátum validálására beküldés előtt



Folytonos integráció és kiadás

YAML szintaxis

- Példa YAML kód:

```
name: Gipsz Jakab # kulcs-érték párok
age: 42
details: # beágyazott kollekció
  givenname: Jakab
  familyname: Gipsz
  birthyear: 1978
languages: # értékek listája (tömb)
  - Hungarian
  - English
  - German
# több soros szöveg
intro_multi: |
  multiple line
  introduction
intro_single: >
  single line
  introduction
```

Folytonos integráció és kiadás

GitLab CI/CD: jobs

- A `.gitlab-ci.yml` fájlban feladatokat (*jobs*) definiálhatunk, amelyekben megadhatjuk milyen utasításokat kell végrehajtaniuk (`script`).
- Pl.:

```
build_program:
  script:
    - apt-get update -qq
    - apt-get install -yqq build-essential
    - make
```

Folytonos integráció és kiadás

GitLab CI/CD: multiple jobs

- Több feladat is definiálható, továbbá megadható egy globális `before_script` elem is, amelyet minden *job* előtt végre kell hajtani. (Felüldefiniálható az egyes feladatokban.)

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  script:
    - mkdir build && cd build
    - cmake ..
    - make install

test_program:
  script:
    - mkdir build && cd build
    - cmake ..
    - make test
```

Folytonos integráció és kiadás

GitLab CI/CD: stages

- A folyamatos integráció feladatait egymást követő szakaszokra (*stages*) oszthatjuk
  - alapértelmezetten 3 *stage* van: *build*, *test*, *deploy*
  - ez tetszőlegesen felüldefiniálhatjuk

```
stages:
  - lint
  - build
```

- Egy *stage* feladatai egymástól függetlenül párhuzamosítva végrehajthatóak (több *runner* bevonásával)
  - a *stagek* egymásra épülnek, amennyiben egy *stage* valamely feladata hibával zárul, a rá épülő *stagek* nem kerülnek végrehajtásra

Folytonos integráció és kiadás

GitLab CI/CD: stages

- A folyamat *stage*kre osztása:

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make install

test_program:
  stage: test
  script:
    - mkdir build && cd build
    - cmake ..
    - make test
```

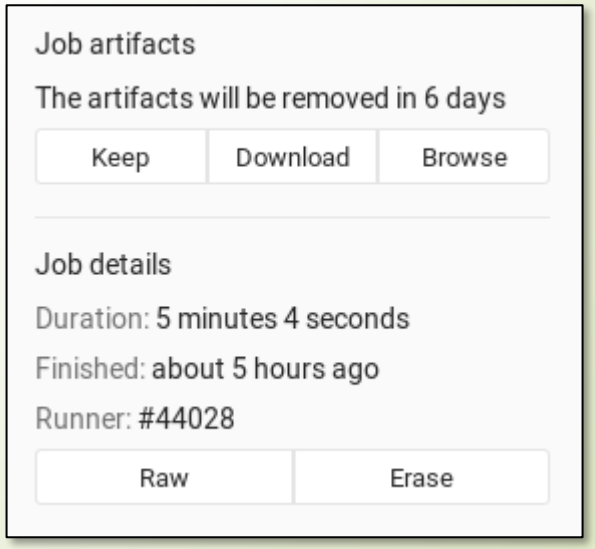
Folytonos integráció és kiadás

GitLab CI/CD: artifacts

- A CI feladatok részeként előállított bináris vagy egyéb állományokat megőrizhetjük (*artifact*)

```
pdf:
  script:
    - pdftex paper.tex
  artifacts:
    paths:
      - paper.pdf
    expire_in: 1 month
```

- Az *artifactok* a GitLab webes felületéről könnyen letölthetőek



Folytonos integráció és kiadás

GitLab CI/CD: artifacts

- *Artifactok* definiálása:

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake .. -DCMAKE_INSTALL_PREFIX=../install
    - make install
  artifacts:
    paths:
      - install/
    expire_in: 1 week
```

Folytonos integráció és kiadás

GitLab CI/CD: dependencies

- Az egymástól függő programok (modulok) ellenőrzése könnyen redundáns végrehajtáshoz vezethet:

```
before_script: ...

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make

test_program:
  stage: test
  script:
    - mkdir build && cd build
    - cmake ..
    - make
    - make test
```

Folytonos integráció és kiadás

GitLab CI/CD: dependencies

- *Artifactok* átadása *jobok* között:

```
before_script: ...

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make
  artifacts:
    paths:
      - build/

test_program: # függ a build kimenetétől
  stage: test
  script:
    - cd build
    - make test
  dependencies:
    - build_program
```

Folytonos integráció és kiadás

Docker images

- *Docker container* alapú GitLab Runner esetén megadhatjuk melyik *Docker image*-ből kívánunk kiindulni:

```
image: ubuntu:22.04
```

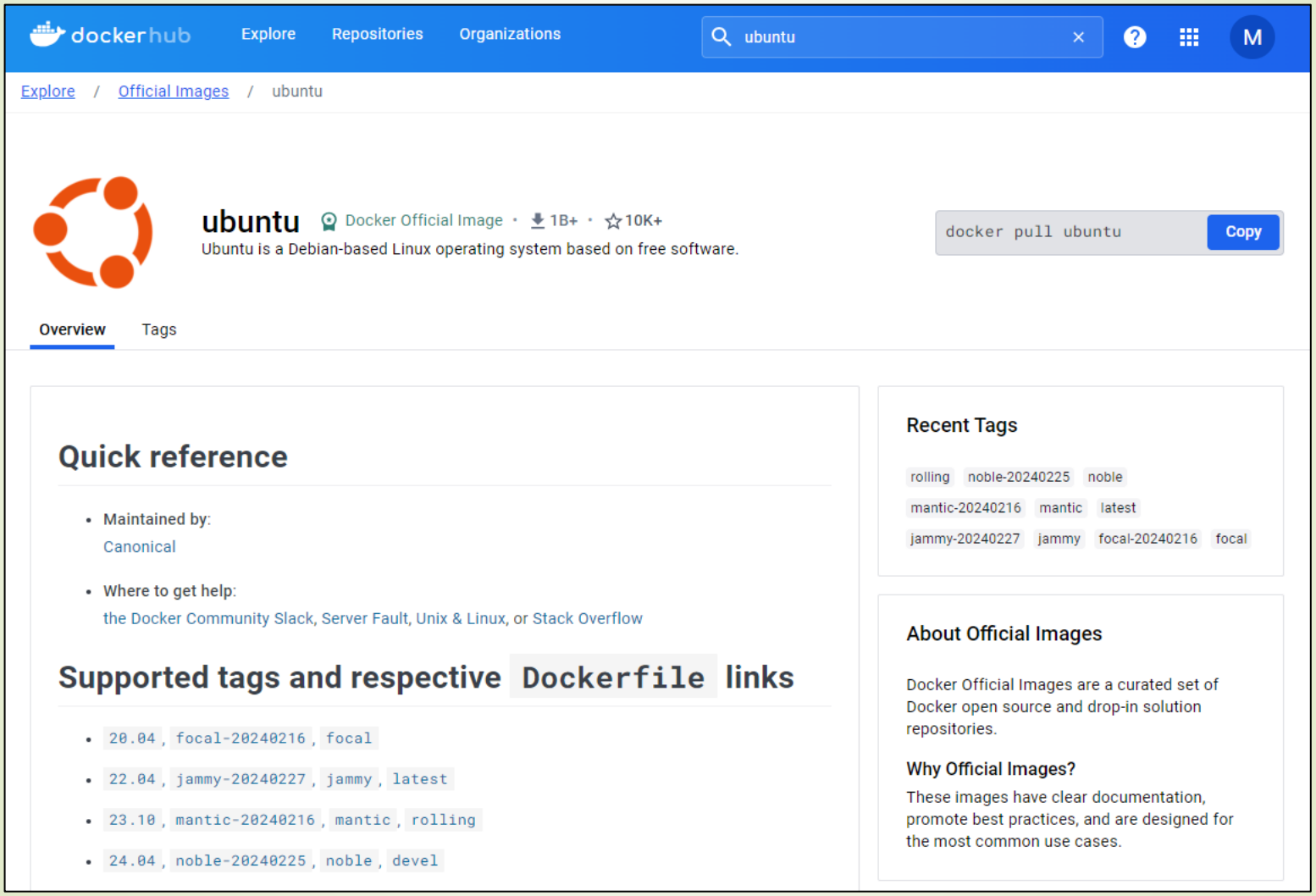
image név

tag név

- *Docker image*-t egy *Docker registryből* kérhetünk:
  - Alapértelmezetten a publikus Docker Hub-ot használjuk, ahová saját image is feltölthető: https://hub.docker.com/
  - Használható privát *Docker registry* is (pl. vállalati környezet)
    `image: mycompany.com:5000/custom:latest`
- Ha nem adjuk meg, akkor a runner konfigurációja adja meg a használandó image-t
  - a szofttech.inf.elte.hu runnerjei az `ubuntu:22.04` imagere vannak konfigurálva
  - A Windows runnerekhez az alapértelmezett image: `mcr.microsoft.com/windows/servercore:1809`

Folytonos integráció és kiadás

Docker Hub



Folytonos integráció és kiadás

GitLab CI/CD

- További lehetőségek (teljesség igénye nélkül):
  - `only, except:` CI jobok végrehajtásának feltételhez kötése (például csak a *master* branch-en futtatni)
  - `when:` CI jobok végrehajtásának feltételhez kötése (manuális vs. automatikus végrehajtás)

```
deploy_program:
  stage: deploy
  script:
    - ...
  only:
    - master
  when: manual
```

Folytonos integráció és kiadás

GitLab CI/CD

- `rules:` feltételes végrehajtás haladóbb konfigurációja (`only` és `except` szabályokhoz képest)
  - `only, except` kulcsszavak már elavultak (*deprecated*), újabb fejlesztéseket már nem kapnak
  - pl. végrehajtás kizárólag a *master* fejlesztési ágon:

```
rules:
  - if: $CI_COMMIT_BRANCH == "master"
```

  - pl. végrehajtás csak a *Dockerfile* változása esetén:

```
rules:
  - changes:
    - Dockerfile
```

  - ezen kondíciók tetszőlegesen kombinálhatóak is

Folytonos integráció és kiadás

GitLab CI/CD

- `variables:` változók definiálása.
  A futtató környezetre számos változó már előre definiált: https://docs.gitlab.com/ee/ci/variables/
- `services:` szolgáltatások (pl. adatbázis motor) külön Docker containerben futtatása (*docker-compose*)

```
services:
    - mysql:latest
variables:
    MYSQL_DATABASE: my_db
    MYSQL_USER: my_user
    MYSQL_PASSWORD: very_secret_password
```

  - a hosztnév `mysql` lesz (`alias` opcióval megadható más)

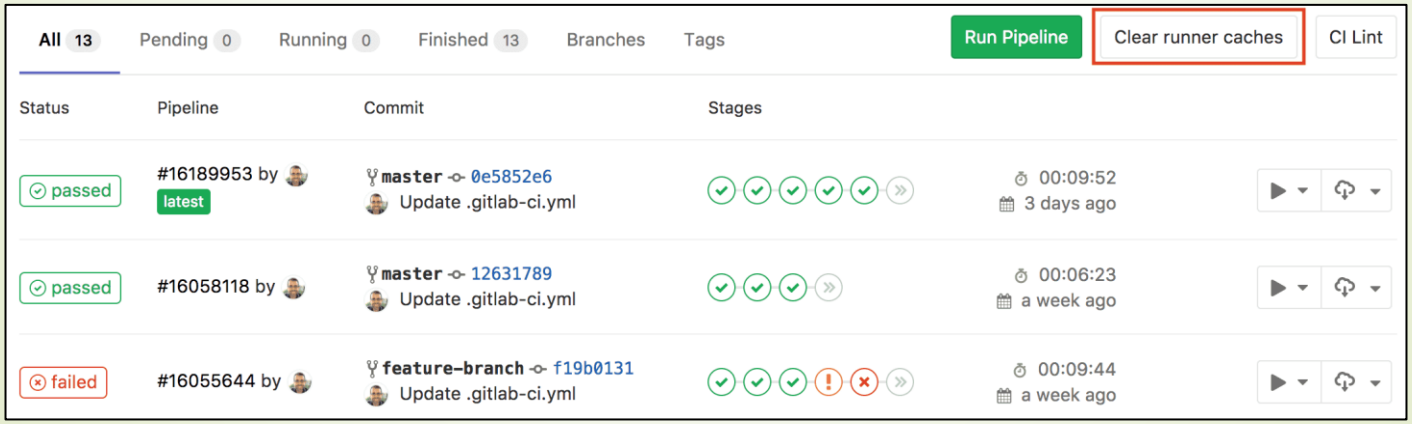
Folytonos integráció és kiadás

GitLab CI/CD

- `cache:` fájlok, könyvtárak (tipikusan függőségek) megőrzése CI jobok és pipelineok között
  - pl.:

```
cache:
  paths:
    - vendor/
```

  - a cache kiüríthető manuálisan:



Folytonos integráció és kiadás

GitLab CI/CD

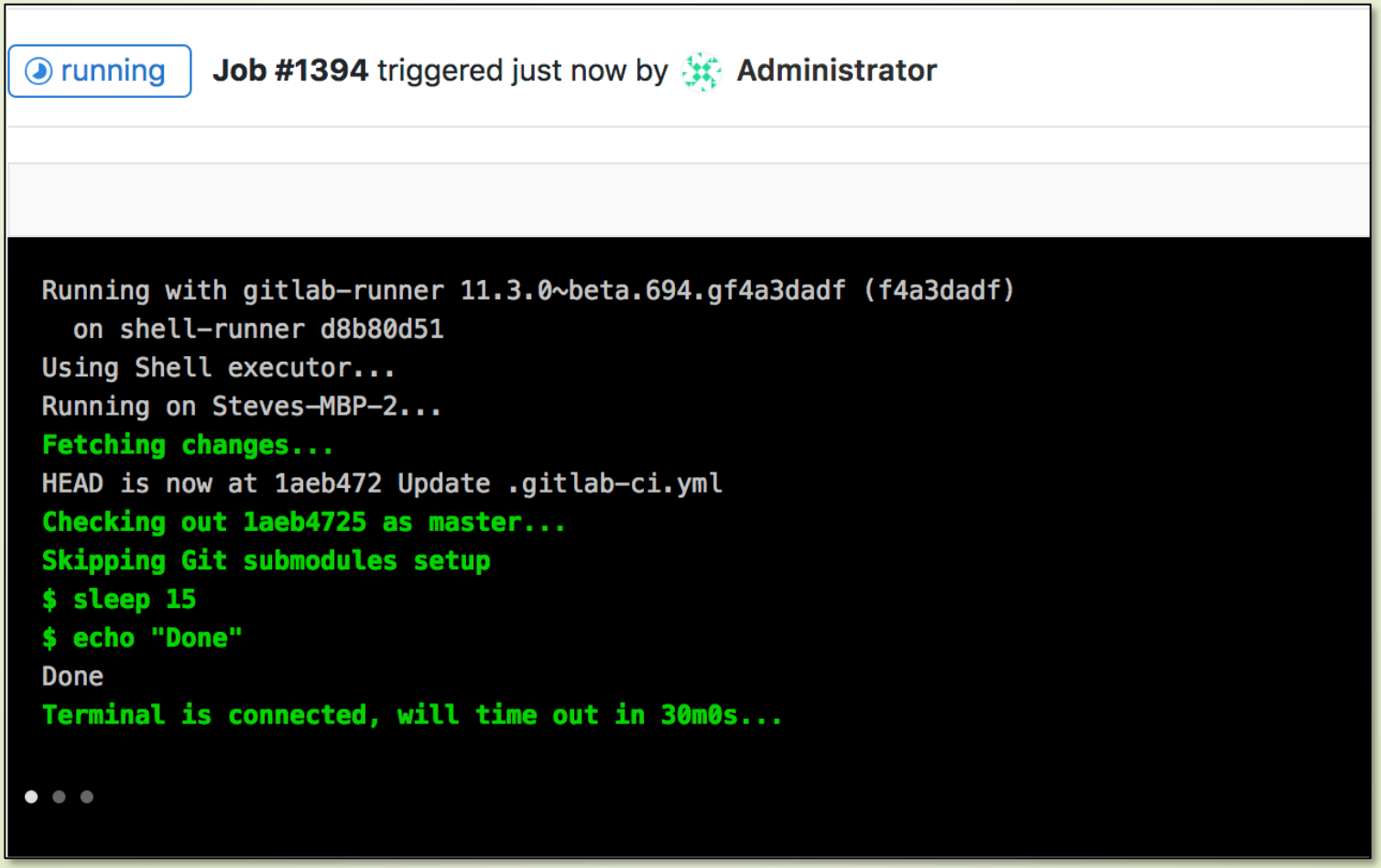
- Egy kulcs (`key`) is megadható, amellyel pl. CI *job*-onként vagy *branch*-enként külön cache használható.
- Pl. egy .NET alkalmazás esetén megadható a letöltendő NuGet csomagok *cachelése*:

```
cache:
  # Külön cache jobonként és branchenként
  key: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
  paths:
    - .nuget/

build_program:
  script:
    # A projekt gyökerébe, a .nuget
    # könyvtárba töltjük le függőségeket.
    - dotnet restore --packages .nuget
    - dotnet build --no-restore
```

Folytonos integráció és kiadás

GitLab CI/CD: terminals

- A folyamatos integráció feladatainak végrehajtását egy online terminál ablakon keresztül követhetjük a GitLab webes felületén



Folytonos integráció és kiadás

GitLab CI/CD: példa projektek

- Amőba C++/Qt implementációval: https://szofttech.inf.elte.hu/mate/tictactoe-qt

```
image: ubuntu:20.04

stages:
  - build
  - test

before_script:
  - apt-get update -yqq
  - apt-get install -yqq build-essential cmake
  - apt-get install -yqq qt5-default

# Build
build_game:
  stage: build
  script:
    - cd TicTacToeGame
    - qmake
    - make

# Test
test_model:
  stage: test
  script:
    - cd TicTacToeTest
    - qmake
    - make
    - make check
```

Folytonos integráció és kiadás

GitLab CI/CD: példa projektek

- Amőba C#/.NET implementációval: https://szofttech.inf.elte.hu/mate/tictactoe-dotnet

```yaml
image: mcr.microsoft.com/dotnet/sdk:6.0

stages:
  - build
  - test

before_script:
  - dotnet --version

# Build
build_model:
  stage: build
  script:
    - dotnet build TicTacToeGame.Model.Basic
    - dotnet build TicTacToeGame.Persistence.Text
    - dotnet build TicTacToeGame.Persistence.Binary

build_view:
  stage: build
  image: mcr.microsoft.com/dotnet/sdk:6.0-windowsservercore-ltsc2019
  tags: [windows]
  script:
    - dotnet build TicTacToeGame.sln

# Test
test_model:
  stage: test
  script:
    - dotnet test TicTacToeGame.Test

```

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

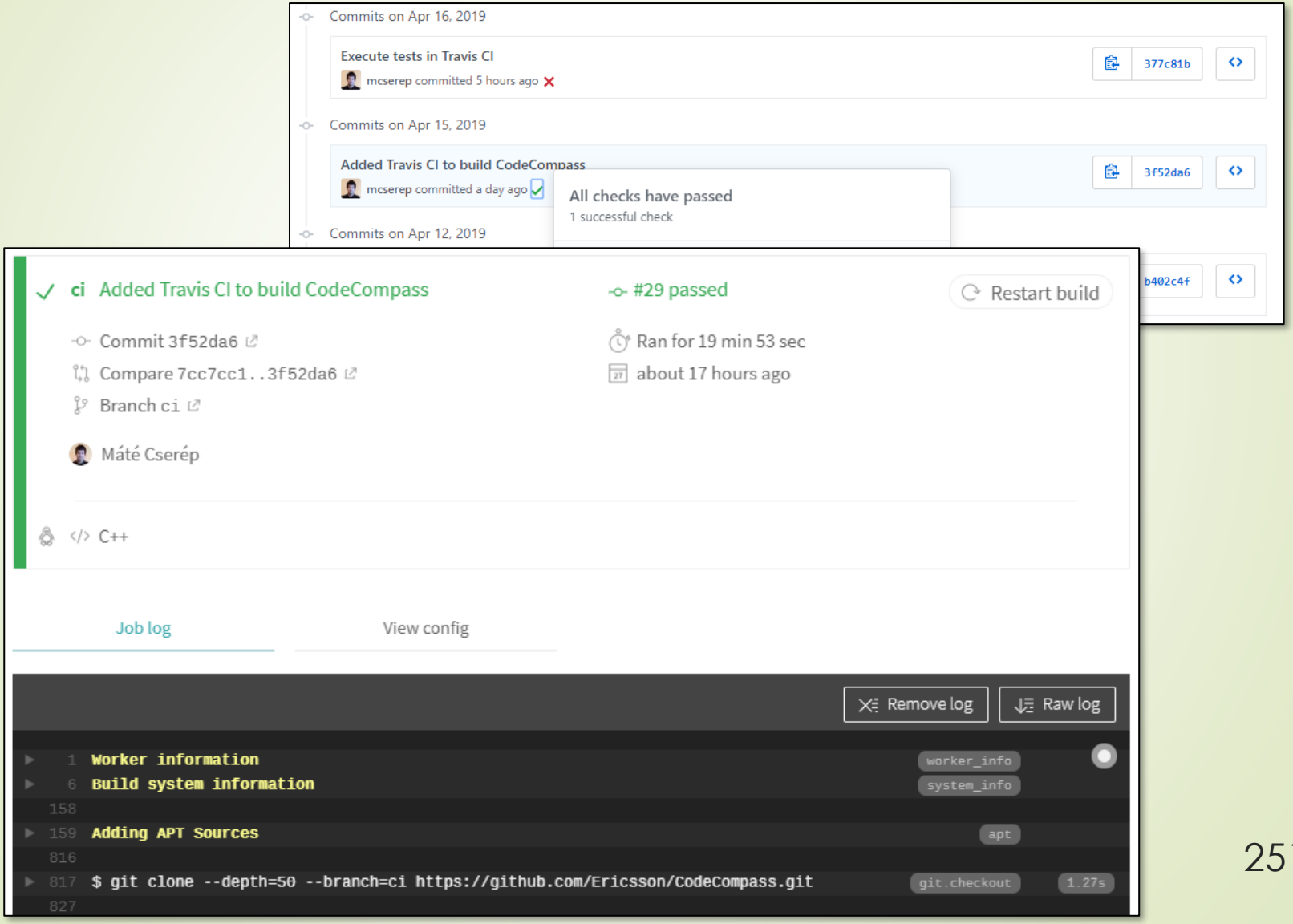
Travis CI

- Folyamatos integrációs szolgáltatás GitHub projektekhez
  - Korábban népszerű, nyílt forráskódú projektekhez ingyenesen használható, de 2021 ót már nem.
  - https://travis-ci.com/
- Támogatja a Linux, a Windows és a macOS operációs rendszereket, több előkészített környezettel (*image*)
  - A környezetet a használt programozási nyelvhez állíthatjuk be, az elterjedtebb fordító eszközökkel és könyvtárakkal
- A CI konfigurációt a `.travis.yml` fájlban adhatjuk meg

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

Travis CI – GitHub integration



Folytonos integráció és kiadás

Travis CI

- Például:

```
os: linux
dist: xenial
language: cpp

stages:
  - compile
  - test
  - deploy

# ...

jobs:
  include:
    - stage: compile
        - cd $TRAVIS_BUILD_DIR
        - mkdir build && cd build
        - cmake ..
        - make

# ...
```

Folytonos integráció és kiadás

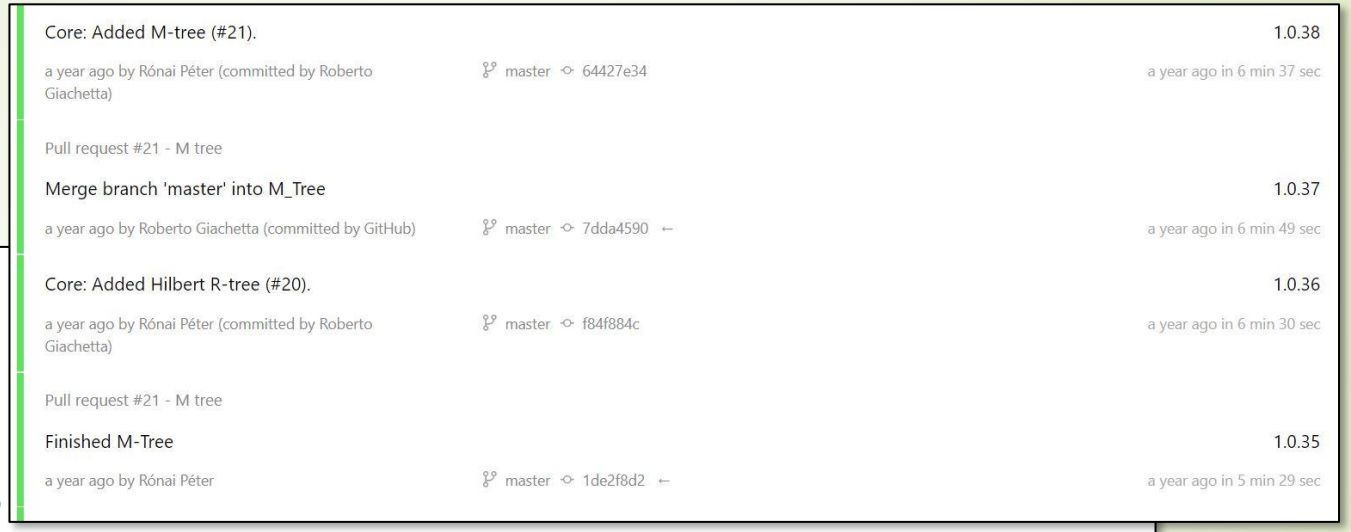
AppVeyor CI

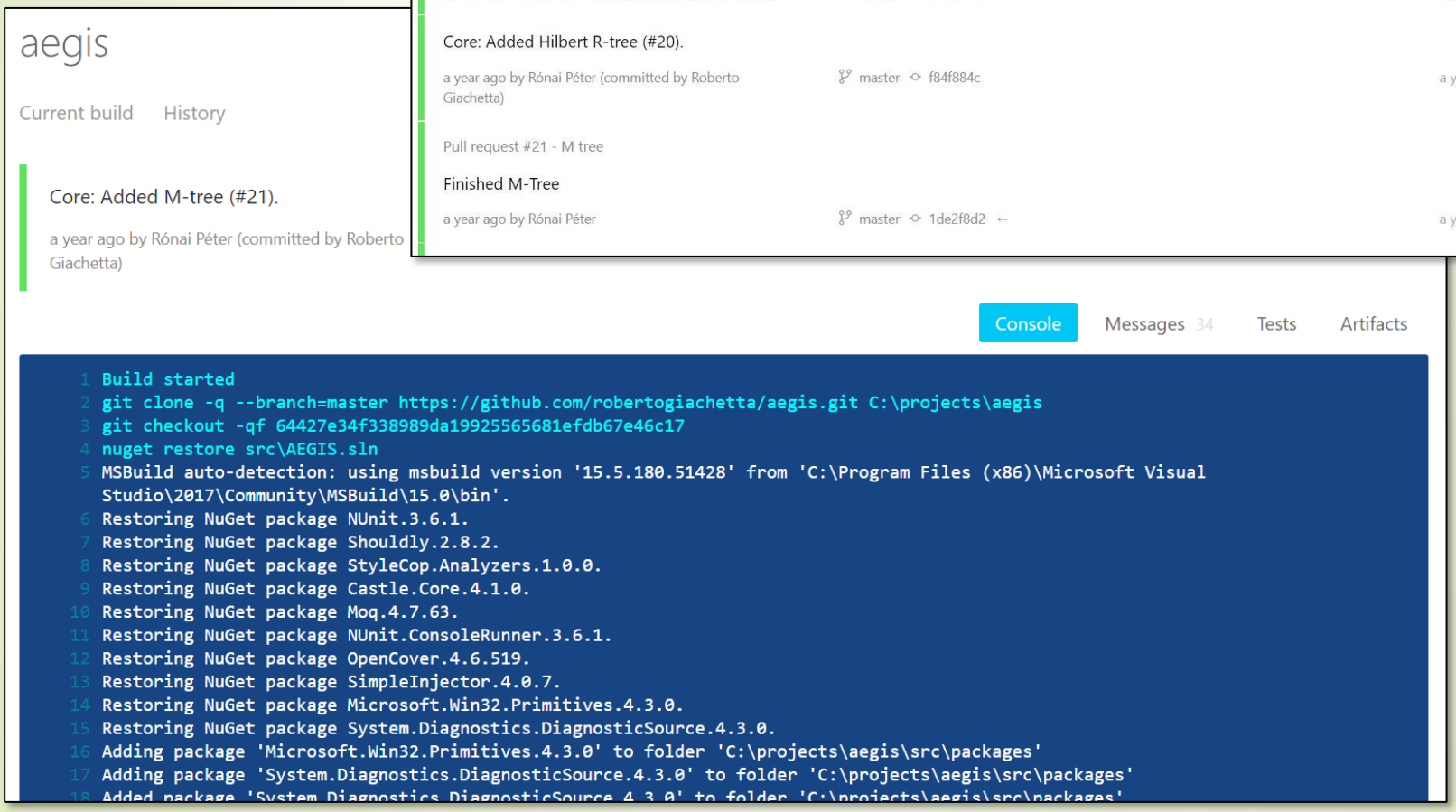
- Folyamatos integrációs szolgáltatás
  - Integrálható a GitHub, GitLab, BitBucket, Visual Studio Team Services platformokkal
  - Nyílt forráskódú projektekhez ingyenesen használható
  - https://www.appveyor.com/
- Támogatja a Linux, a Windows, a macOS operációs rendszereket, több előkészített környezettel (*image*)
  - Kiemelt .NET és Visual Studio támogatás
- A CI konfigurációt a `appveyor.yml` fájlban adhatjuk meg

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

AppVeyor CI – webes interfész





Folytonos integráció és kiadás

AppVeyor CI

- Például:

```
version: 1.0.{build} # Version format
image: Visual Studio 2017 # Build worker image
platform: Any CPU # Build platform
configuration: Debug # Build Configuration

# Execute script before build
before_build:
  - dotnet restore src\MyProject.sln

# Execute build script
build_script:
  - dotnet build src\MyProject.sln

# Execute test script
test_script:
  - dotnet test src\MyProject.sln
```

Folytonos integráció és kiadás

Jenkins

- Nyílt forráskódú folyamatos integrációs szolgáltatás
  - Integrálható a GitHub, GitLab, és egyéb projektvezető szolgáltatásokkal
  - Nem nyújt hoszting szolgáltatást, de más vállalkozások kínálnak (pl. CloudBees)
  - https://jenkins.io/
- A CI konfigurációt a `Jenkinsfile` adja meg, például:

```
pipeline {
  agent { docker { image 'ubuntu:18.04' } }
  stages {
    stage('build') {
      steps {
        sh 'apt-get install build-essential'
        sh 'make'
      }
    }
  }
}
```

Folytonos integráció és kiadás

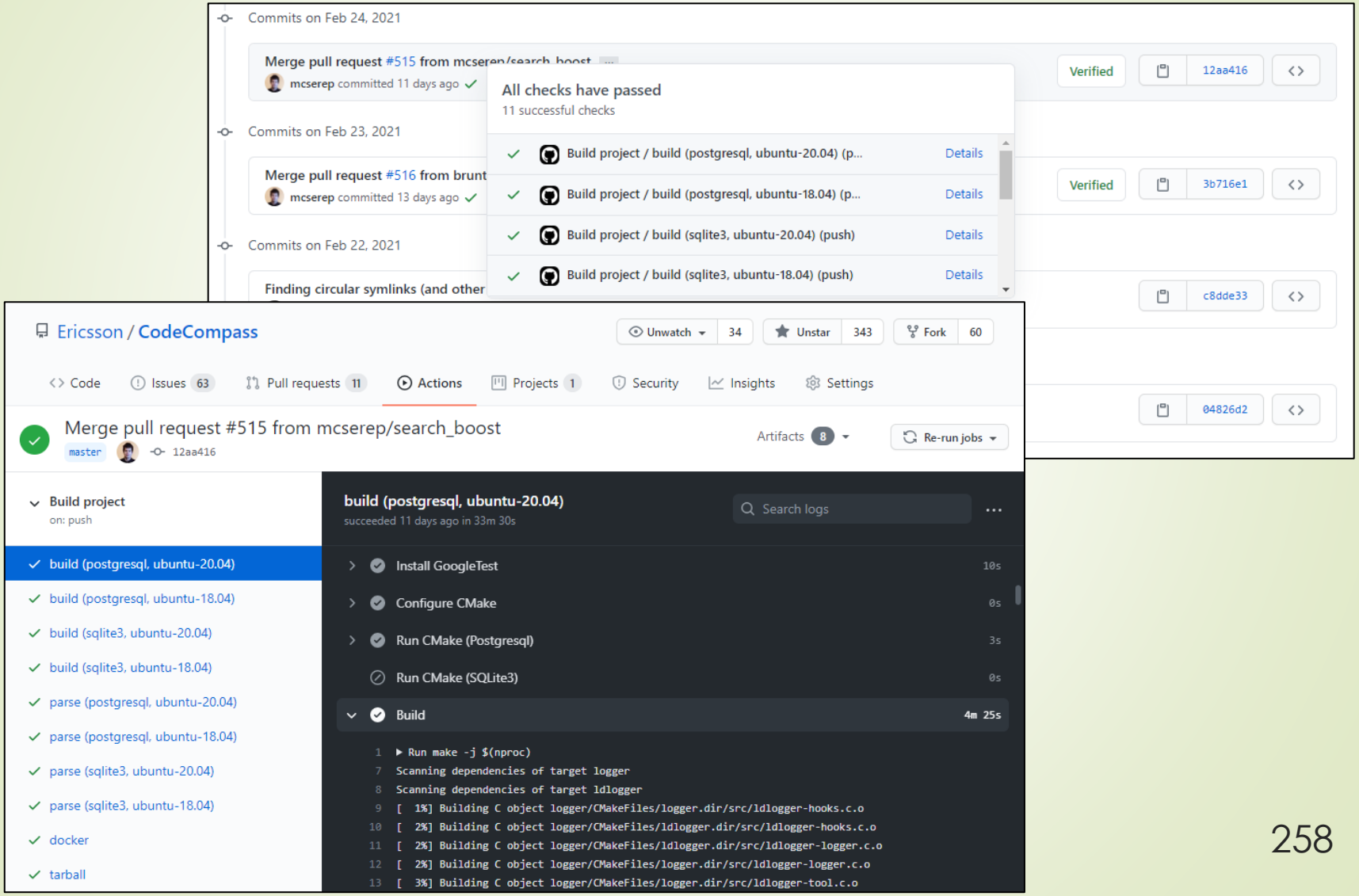
GitHub Actions

- A GitHub viszonylag új, saját CI/CD szolgáltatása
  - Ingyenes fiókkal korlátozott mértékig (jelenleg 2000 futtatási perc / hó) ingyenesen használható, ez előfizetéssel bővíthető
  - https://docs.github.com/en/actions
- Linux (Ubuntu), Windows és macOS operációs rendszer alapú virtuális gépek futtatását támogatja, de használható Docker
  - A virtuális gépek a Microsoft felhőjében (*Azure*) futnak
  - Telepíthető saját *self-hosted* GitHub Actions Runner is
- A CI konfigurációt a `.github/workflows/` könyvtárban, YAML fájlokban adhatjuk meg
  - Shell utasítások mellett magunk vagy mások által elkészített *action*-öket is felhasználhatunk, mint komplex építő elemek

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

GitHub Actions



Folytonos integráció és kiadás

GitHub Actions

- Például:

```
name: Build project
on: [push, pull_request]
jobs:
  build:
    runs-on: [windows-latest] # vagy ubuntu-latest
    steps:
    - uses:  actions/checkout@v4
    - name:  Setup .NET SDK
      uses:  actions/setup-dotnet@v4
      with:
        dotnet-version: '6.0.x'
    - name:  Restore NuGet packages
      run:   dotnet restore AEGIS.sln
    - name:  Build the solution
      run:   dotnet build AEGIS.sln -c Release
    - name:  Run unit tests
      shell: powershell
      run:   dotnet test AEGIS.sln
```

Szoftvertechnológia

Folytonos integráció és kiadás

Continuous integration & delivery

C szakirány

Dr. Cserép Máté

ELTE Informatikai Kar

2025.

Folytonos integráció és kiadás

Folyamatos integráció

- A *folytonos integráció* (*continuous integration*, *CI*) egy olyan gyakorlati módszer, amely lehetővé teszi a programkódok ellenőrzésének és tesztelésének felgyorsítását
  - célja a lehetséges hibák, integrációs problémák azonnali, automatizált kiszűrése, visszajelzés a fejlesztőnek
  - a programkódok verziókezelő rendszer segítségével egy központi tárhelyre kerülnek, naponta többször
  - a tárhely tartalma minden módosítást követően automatikusan fordításra kerül (*build automation*), a fordítással pedig a lekódolt tesztek is végrehajtódnak
  - az így ellenőrzött kódot további tesztelés követheti

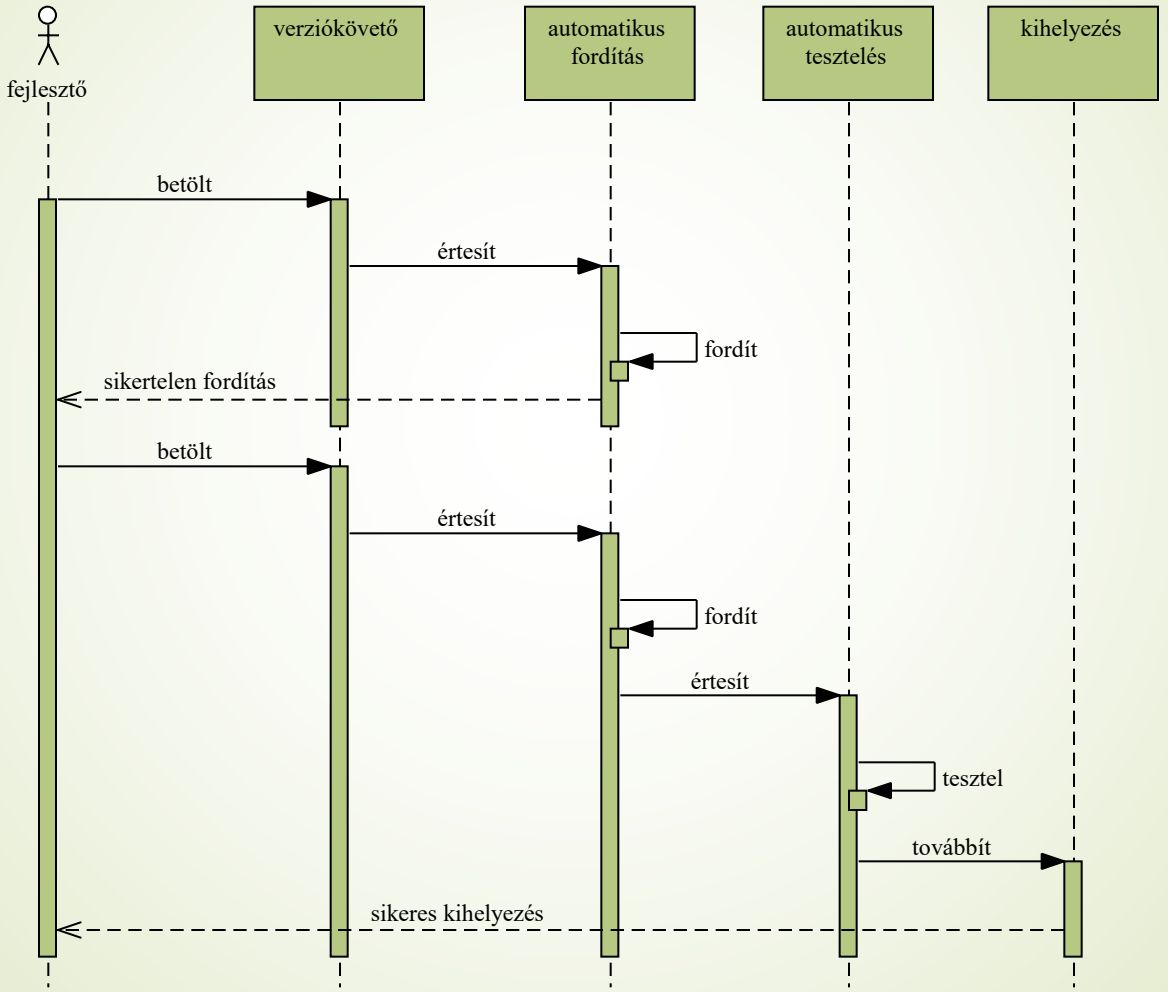
Folytonos integráció és kiadás

Folyamatos teljesítés

- Az agilis szoftverfejlesztés (*agile software development*) célja a gyors alkalmazásfejlesztés megvalósítása, inkrementális alapon
  - a szoftver folyamatos fejlesztés és kiadás alatt áll (*continuous delivery*), a sebesség állandó, a változtatások minden lépésben beépíthetőek (*welcome changes*)
  - a működő szoftver az előrehaladás mérőeszköze, előtérben az egyszerűség, ugyanakkor folyamatos odafigyelés a megfelelő tervezésre, optimalizációra
  - a fejlesztést általában önszervező, kis csapatok végzik, megosztott felelősséggel, folytonos interakcióval, gyors visszajelzésekkel
  - a folyamatos kiadások automatizálhatók, ekkor *continuous deployment*-ről beszélünk

Folytonos integráció és kiadás

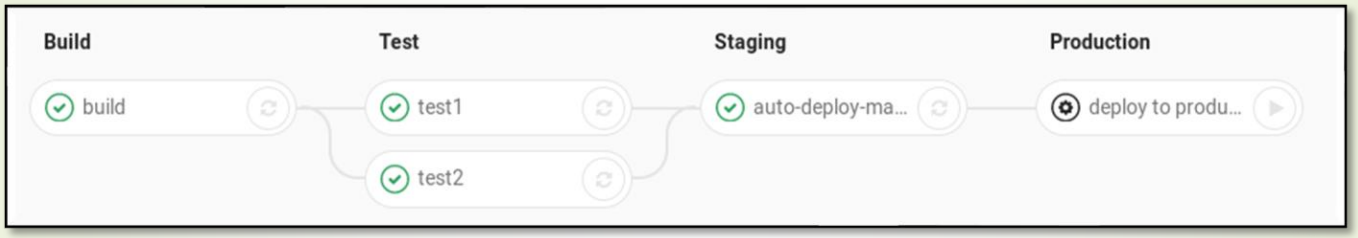
Folyamatos integráció és teljesítés

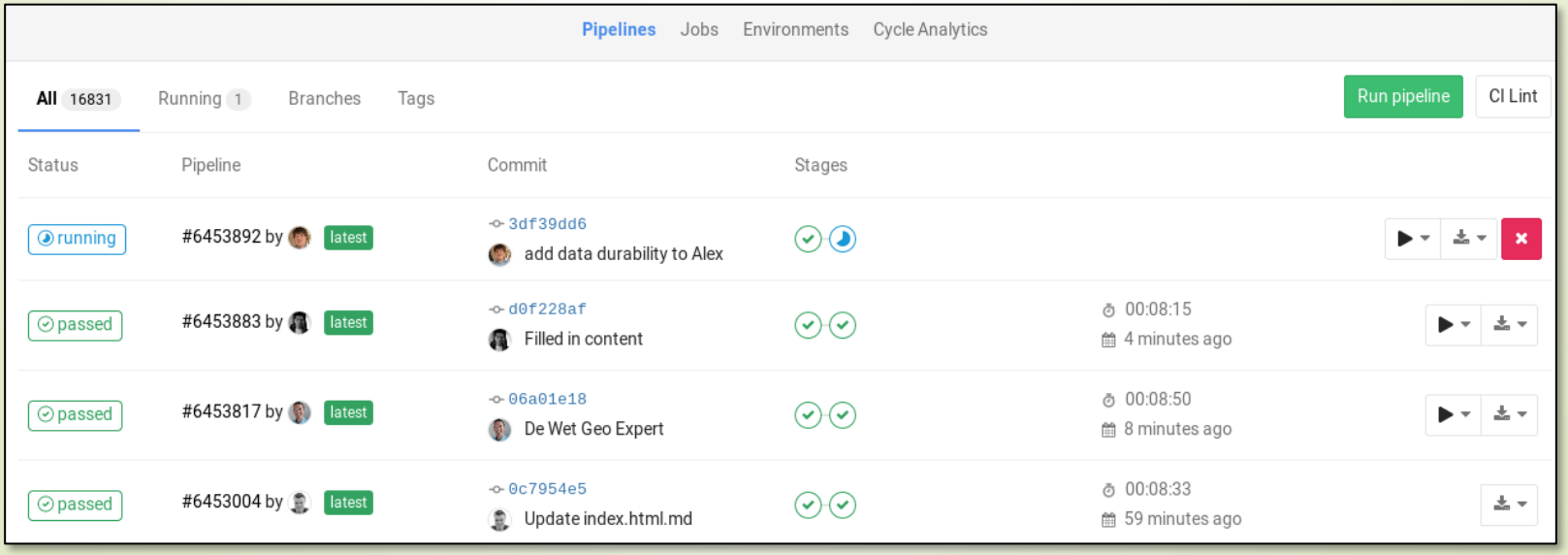


Folytonos integráció és kiadás

Feladatok

- A folyamatos integráció és teljesítés lépéseit egymásra épülő feladatok (*jobs*) láncolataként (*pipelines*) definiálhatjuk

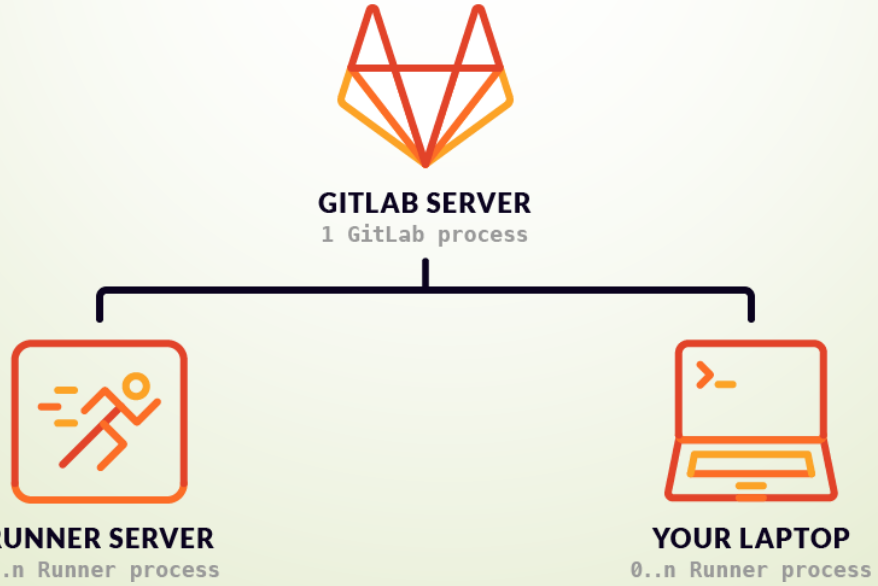




Folytonos integráció és kiadás

GitLab Runners

- A GitLab rendelkezik integrált, saját megoldással a folyamatos integráció és teljesítés támogatására
  - a feladatokat (*jobs*) a GitLab szervertől független ún. *GitLab Runner* példányok hajtják végre
    - Shell, SSH, VirtualBox, Docker, Kubernetes, stb.
  - a *runnerek* egyedileg konfigurálhatóak, lehetnek megosztottak vagy projekthez rendeltek



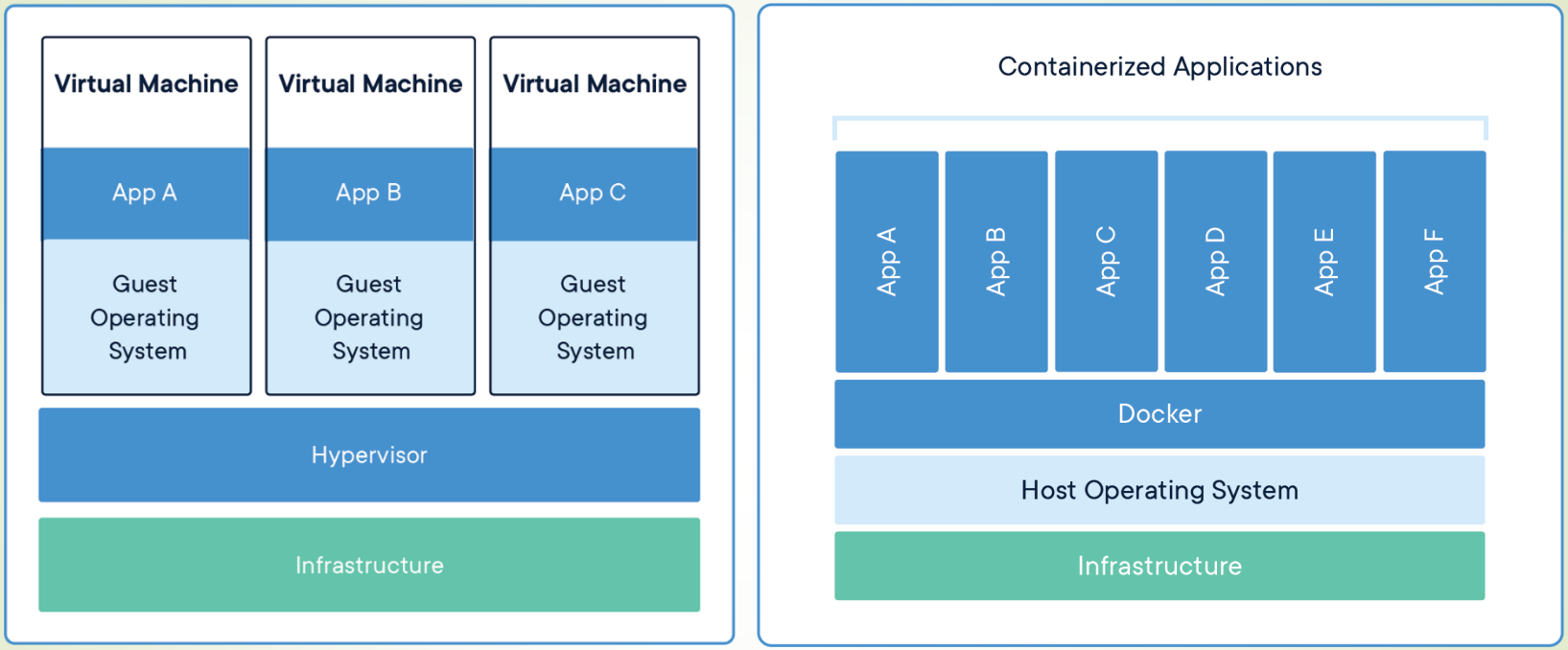
Folytonos integráció és kiadás

Docker

- A folyamatos integrációt egy izolált, reprodukálható környezetben érdemes végezni
- A Docker napjainkban a legelterjedtebb *container framework*
  - a *container* hasonlít a virtuális gépekhez (*VM)* olyan tekintetben, hogy egy teljesen elkülönített, virtualizált környezett biztosít, amelynek a gazdaszámítógép szolgáltat erőforrásokat
  - a fő különbség a *containerek* és a virtuális gépek között, hogy minden *container* osztozik a gazda kerneljén a többi *containerrel*, a virtualizált hardver és az OS nem része, csupán az alkalmazásunkhoz kötődő könyvtárak, binárisok és a felhasználói terület
  - a *containerek* ezáltal nagyságrendekkel kisebb *overheaddel* bírnak a VM-ekhez képest, így könnyebb súlyú megoldást nyújtanak a virtualizációra

Folytonos integráció és kiadás

Virtuális gépek és *containerek*



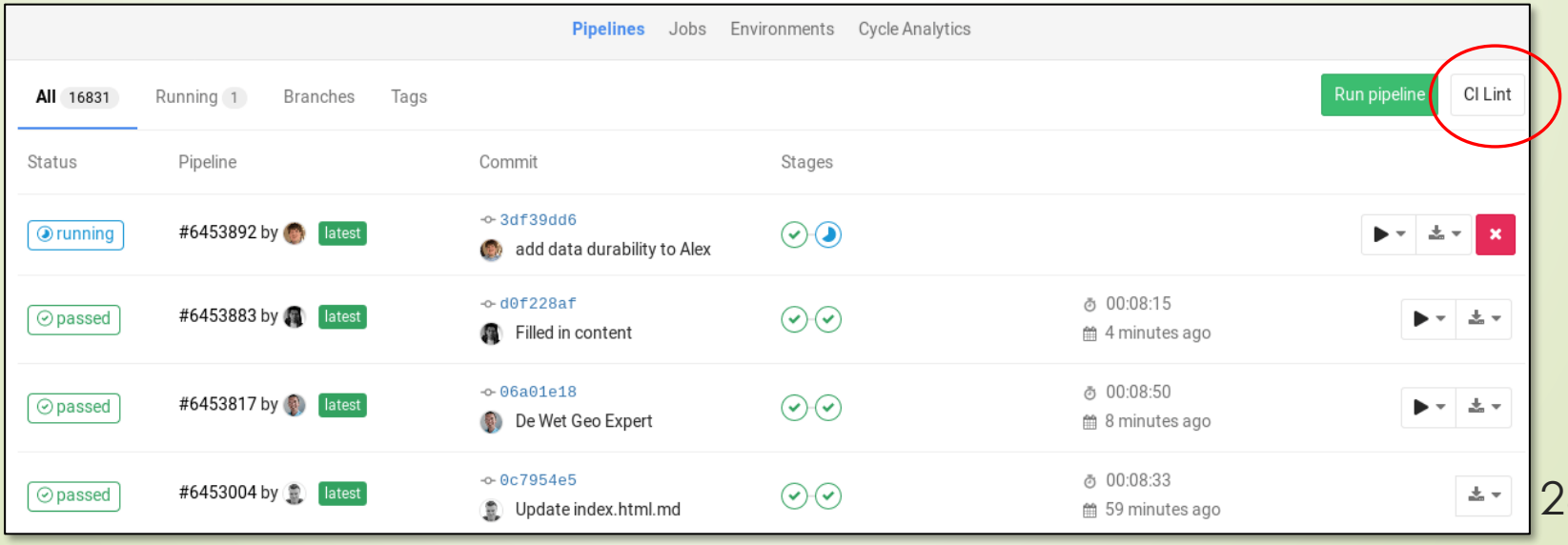
Forrás: docker.com

- a *container framework*ök lehetőséget adnak hordozható alkalmazások létrehozására és menedzselésére
- az alkalmazások modularizálhatóak és skálázhatóak, a komponensek külön *container*ben futhatnak

Folytonos integráció és kiadás

GitLab CI/CD

- A folyamatos integráció konfigurációját a `.gitlab-ci.yml` fájl tartalmazza, YAML formátumban
  - a YAML (*YAML Ain't Markup Language*) egy emberi szemmel könnye(bbe)n olvasható strukturált leíró nyelv
  - https://yaml.org/spec/1.2/spec.html
- A GitLab webes felületén elérhető egy CI Lint funkció a formátum validálására beküldés előtt



Folytonos integráció és kiadás

YAML szintaxis

- Példa YAML kód:

```
name: Gipsz Jakab # kulcs-érték párok
age: 42
details: # beágyazott kollekció
  givenname: Jakab
  familyname: Gipsz
  birthyear: 1978
languages: # értékek listája (tömb)
  - Hungarian
  - English
  - German
# több soros szöveg
intro_multi: |
  multiple line
  introduction
intro_single: >
  single line
  introduction
```

Folytonos integráció és kiadás

GitLab CI/CD: jobs

- A `.gitlab-ci.yml` fájlban feladatokat (*jobs*) definiálhatunk, amelyekben megadhatjuk milyen utasításokat kell végrehajtaniuk (`script`).
- Pl.:

```
build_program:
  script:
    - apt-get update -qq
    - apt-get install -yqq openjdk-11-jdk ant
    - ant compile
    - ant jar
```

Folytonos integráció és kiadás

GitLab CI/CD: multiple jobs

- Több feladat is definiálható, továbbá megadható egy globális `before_script` elem is, amelyet minden *job* előtt végre kell hajtani. (Felüldefiniálható az egyes feladatokban.)

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant junit

build_program:
  script:
    - ant compile
    - ant jar

test_program:
  script:
    - ant compile-test
    - ant test
```

Folytonos integráció és kiadás

GitLab CI/CD: stages

- A folyamatos integráció feladatait egymást követő szakaszokra (*stages*) oszthatjuk
  - alapértelmezetten 3 *stage* van: *build*, *test*, *deploy*
  - ez tetszőlegesen felüldefiniálhatjuk

```
stages:
  - lint
  - build
```

- Egy *stage* feladatai egymástól függetlenül párhuzamosítva végrehajthatóak (több *runner* bevonásával)
  - a *stagek* egymásra épülnek, amennyiben egy *stage* valamely feladata hibával zárul, a rá épülő *stagek* nem kerülnek végrehajtásra

Folytonos integráció és kiadás

GitLab CI/CD: stages

- A folyamat *stage*kre osztása:

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant junit

build_program:
  stage: build
  script:
    - ant compile
    - ant jar

test_program:
  stage: test
  script:
    - ant compile-test
    - ant test
```

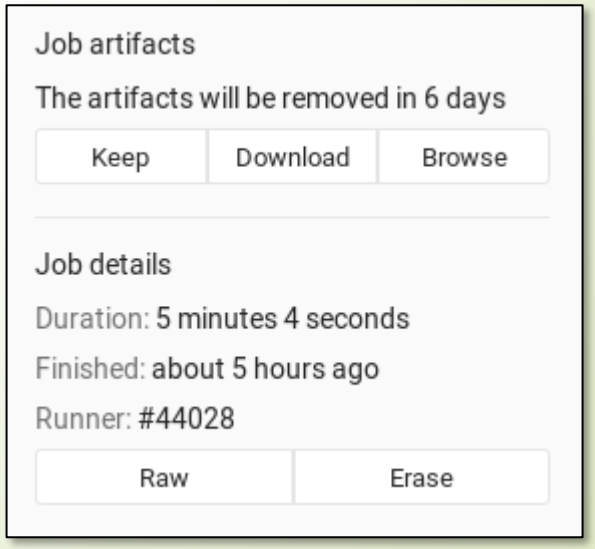
Folytonos integráció és kiadás

GitLab CI/CD: artifacts

- A CI feladatok részeként előállított bináris vagy egyéb állományokat megőrizhetjük (*artifact*)

```
pdf:
  script:
    - pdftex paper.tex
  artifacts:
    paths:
      - paper.pdf
    expire_in: 1 month
```

- Az *artifactok* a GitLab webes felületéről könnyen letölthetőek



Folytonos integráció és kiadás

GitLab CI/CD: artifacts

- *Artifactok* definiálása:

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant

build_program:
  stage: build
  script:
    - ant compile
    - ant jar
  artifacts:
    paths:
      - dist/program.jar
    expire_in: 1 week
```

Folytonos integráció és kiadás

GitLab CI/CD: dependencies

- Az egymástól függő programok (modulok) ellenőrzése könnyen redundáns végrehajtáshoz vezethet:

```
before_script: ...

build_program_A:
  stage: build
  script:
    - cd module_A
    - ant jar

build_program_B: # függ program_A.jar állománytól
  stage: build
  script:
    - cd module_A
    - ant jar
    - cd ..
    - cd module_B
    - ant jar
```

Folytonos integráció és kiadás

GitLab CI/CD: dependencies

- *Artifactok* átadása *jobok* között:

```
before_script: ...

build_program_A:
  stage: build
  script:
    - cd module_A
    - ant jar
  artifacts:
    paths:
      - module_A/dist/program_A.jar

build_program_B: # függ program_A.jar állománytól
  stage: build
  script:
    - cd module_B
    - ant jar
  dependencies:
    - build_program_A
```

Folytonos integráció és kiadás

Docker images

- *Docker container* alapú GitLab Runner esetén megadhatjuk melyik *docker image*-ből kívánunk kiindulni:

```
image: ubuntu:22.04
```

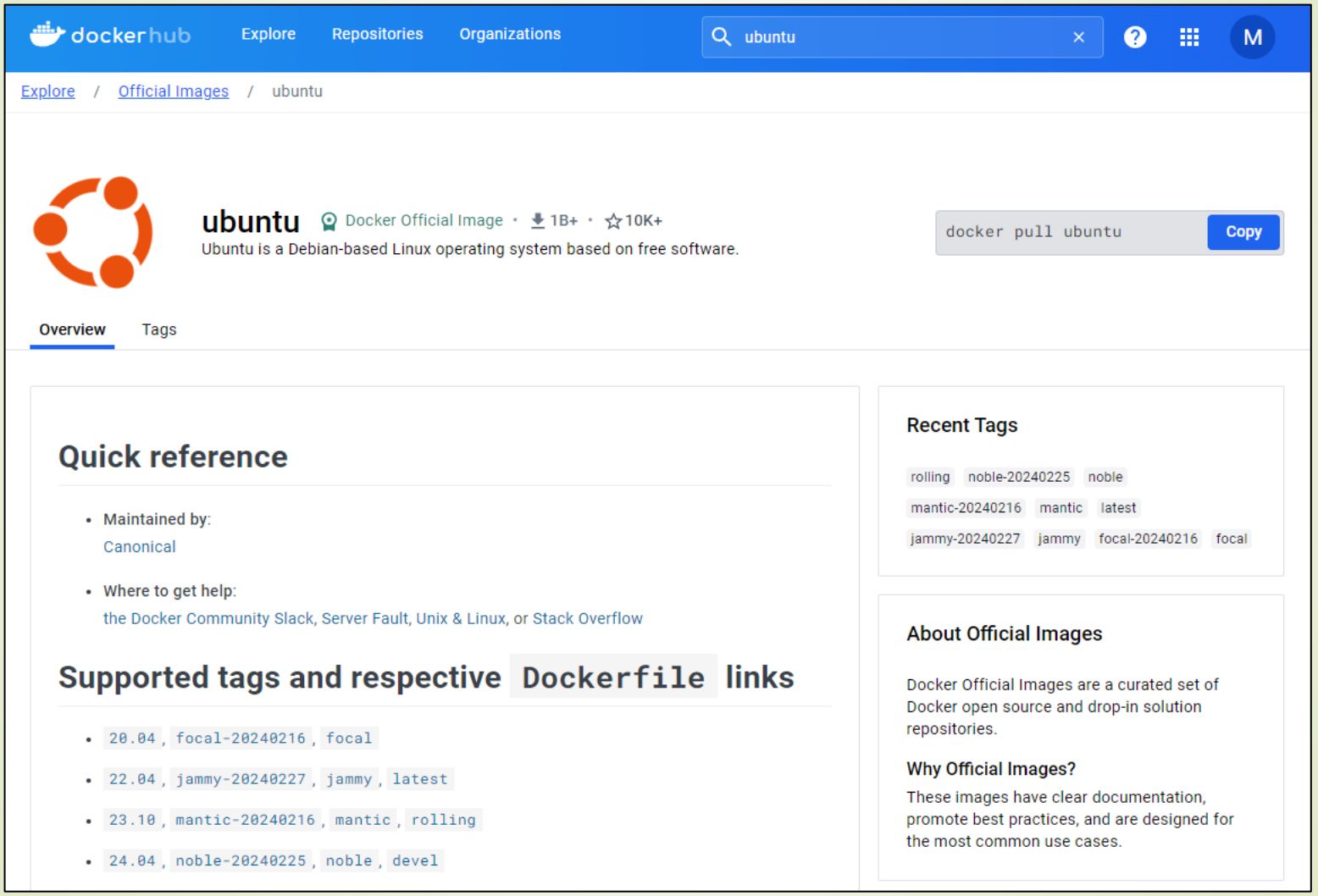
image név

tag név

- *Docker image*-t egy *Docker registryből* kérhetünk:
  - Alapértelmezetten a publikus Docker Hub-ot használjuk, ahová saját image is feltölthető: https://hub.docker.com/
  - Használható privát *Docker registry* is (pl. vállalati környezet)
    `image: mycompany.com:5000/custom:latest`
- Ha nem adjuk meg, akkor a runner konfigurációja adja meg a használandó image-t
  - a szofttech.inf.elte.hu runnerjei az `ubuntu:22.04` imagere vannak konfigurálva
  - A Windows runnerekhez az alapértelmezett image: `mcr.microsoft.com/windows/servercore:1809`

Folytonos integráció és kiadás

Docker Hub



Folytonos integráció és kiadás

GitLab CI/CD

- További lehetőségek (teljesség igénye nélkül):
  - `only, except:` CI jobok végrehajtásának feltételhez kötése (például csak a *master* branch-en futtatni)
  - `when:` CI jobok végrehajtásának feltételhez kötése (manuális vs. automatikus végrehajtás)

```
image: maven:latest
build_program:
  stage: build
  script:
    - mvn compile
test_program:
  stage: test
  script:
    - mvn test
deploy_program:
  stage: deploy
  script:
    - mvn deploy
  only:
    - master
  when: manual
```

Folytonos integráció és kiadás

GitLab CI/CD

- `rules:` feltételes végrehajtás haladóbb konfigurációja (`only` és `except` szabályokhoz képest)
  - `only, except` kulcsszavak már elavultak (*deprecated*), újabb fejlesztéseket már nem kapnak
  - pl. végrehajtás kizárólag a *master* fejlesztési ágon:

```
rules:
  - if: $CI_COMMIT_BRANCH == "master"
```

  - pl. végrehajtás csak a *Dockerfile* változása esetén:

```
rules:
  - changes:
    - Dockerfile
```

  - ezen kondíciók tetszőlegesen kombinálhatóak is

Folytonos integráció és kiadás

GitLab CI/CD

- `variables:` változók definiálása.
  A futtató környezetre számos változó már előre definiált: https://docs.gitlab.com/ee/ci/variables/
- `services:` szolgáltatások (pl. adatbázis motor) külön Docker containerben futtatása (*docker-compose*)

```
services:
    - mysql:latest
variables:
    MYSQL_DATABASE: my_db
    MYSQL_USER: my_user
    MYSQL_PASSWORD: very_secret_password
```

  - a hosztnév `mysql` lesz (`alias` opcióval megadható más)

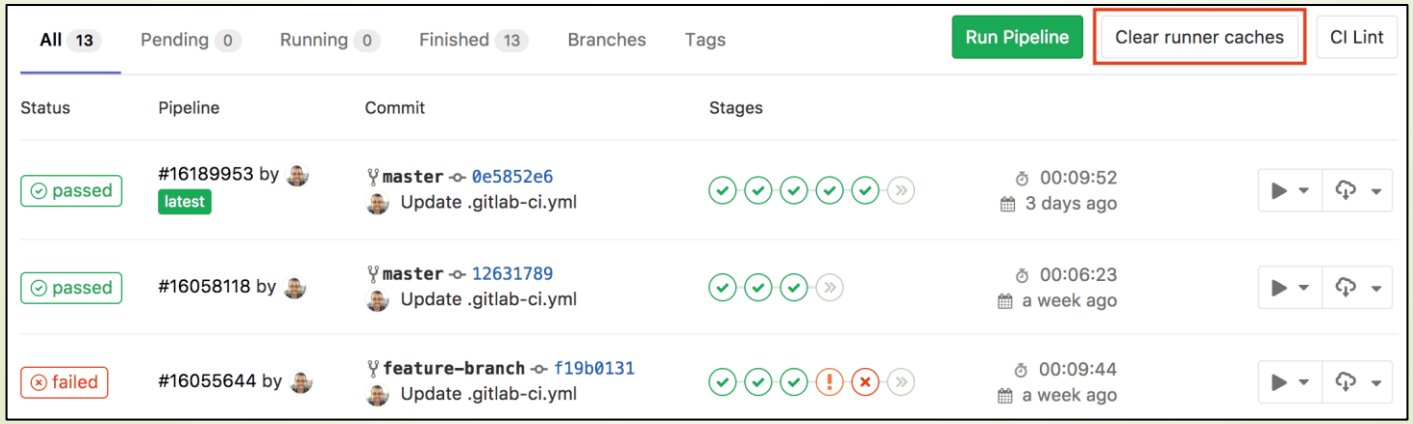
Folytonos integráció és kiadás

GitLab CI/CD

- `cache:` fájlok, könyvtárak (tipikusan függőségek) megőrzése CI jobok és pipelineok között
  - pl.:

```
cache:
  paths:
    - vendor/
```

  - a cache kiüríthető manuálisan:



Folytonos integráció és kiadás

GitLab CI/CD

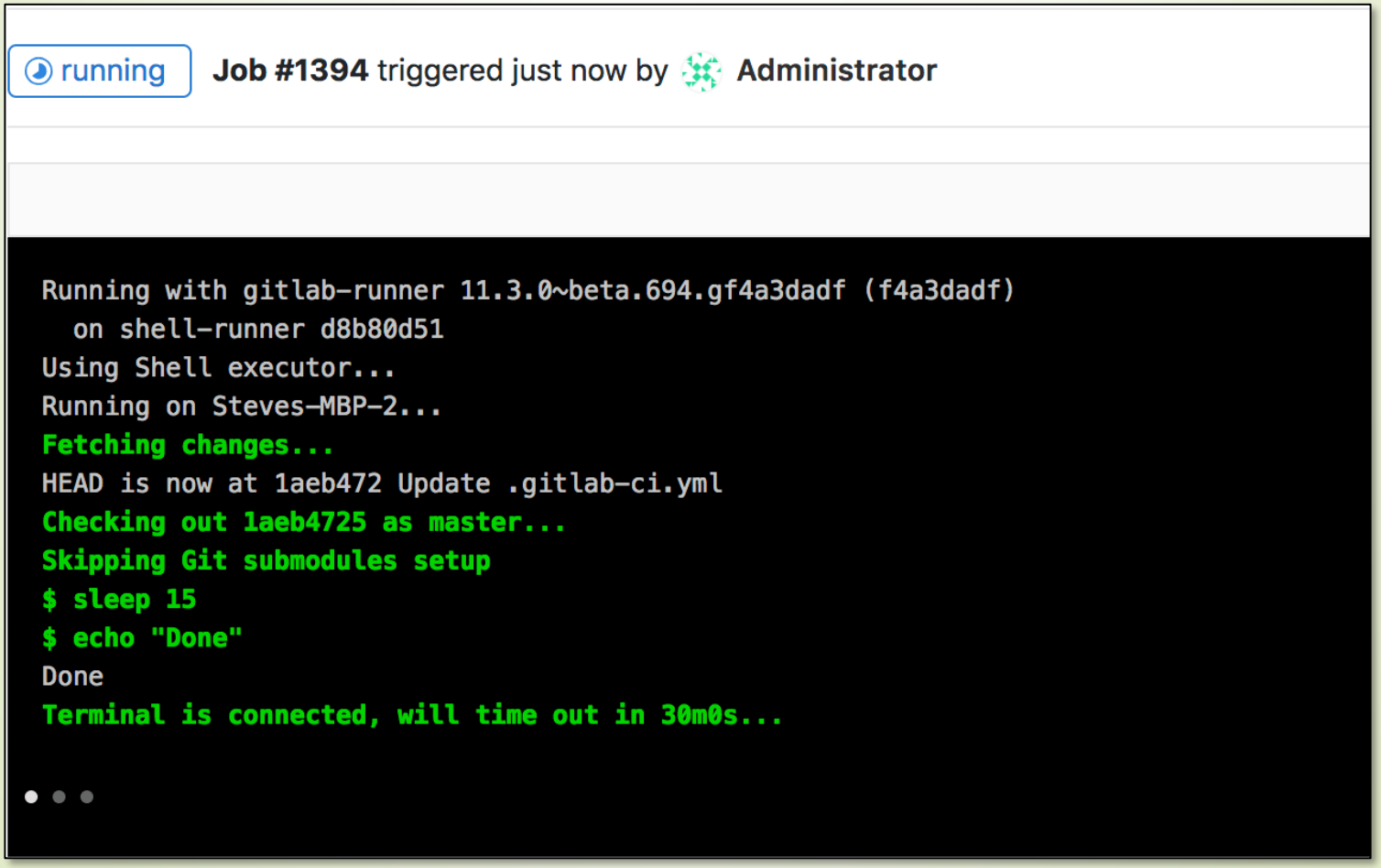
- Cacheléskor egy kulcs (`key`) is megadható, amellyel pl. CI *job*-onként vagy *branch*-enként külön cache használható.
- Pl. egy Maven alkalmazás esetén megadható a letöltendő csomagok *cacheléseː*

```
variables:
  MAVEN_OPTS: "-Dmaven.repo.local=
               $CI_PROJECT_DIR/.m2/repository"

cache:
  # Külön cache jobonként és branchenként
  key: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
  paths:
    - .m2/repository

build_program:
  script:
    - mvn compile
```

Folytonos integráció és kiadás

GitLab CI/CD: terminals

- A folyamatos integráció feladatainak végrehajtását egy online terminál ablakon keresztül követhetjük a GitLab webes felületén



Folytonos integráció és kiadás

GitLab CI/CD: példa projekt

- Hálózati Pacman játék Java implementációval: https://szofttech.inf.elte.hu/mate/pacman-java/

```yaml
image: openjdk:11

stages:
  - build
  - test

before_script:
  - apt-get update -yqq
  - apt-get install -yqq ant junit4

# Build
build_game:
  stage: build
  script:
    - ant compile
    - ant jar

# Test
test_model:
  stage: test
  script:
    - >
      ant test
      -Dlibs.junit_4.classpath=/usr/share/java/junit4.jar
      -Dlibs.hamcrest.classpath=/usr/share/java/hamcrest-core.jar

```

- Ant build rendszerrel: `ant` fejlesztési ág

Folytonos integráció és kiadás

GitLab CI/CD: példa projekt

- Maven build rendszerrel: `maven` fejlesztési ág

```yaml
image: maven:latest

variables:
  MAVEN_OPTS: "-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository"

cache:
  key: maven-build
  paths:
    - .m2/repository

build:
  stage: build
  script:
    - mvn compile
test:
  stage: test
  script:
    - mvn test
```

- A cache egy kulccsal azonosítható, így pl. szabályozható, hogy mely fejlesztési ágak között kerüljön megosztásra.

Folytonos integráció és kiadás

GitLab CI/CD: példa projekt

- Gradle build rendszerrel: `gradle` fejlesztési ág

```yaml
image: gradle:latest

variables:
  GRADLE_USER_HOME: "$CI_PROJECT_DIR/.gradle"

# Cache and keep downloaded dependencies between CI pipelines
cache:
  key: gradle-build
  paths:
    - .gradle

build:
  stage: build
  script:
    - gradle assemble

test:
  stage: test
  script:
    # build also executes test
    - gradle build

```

Folytonos integráció és kiadás

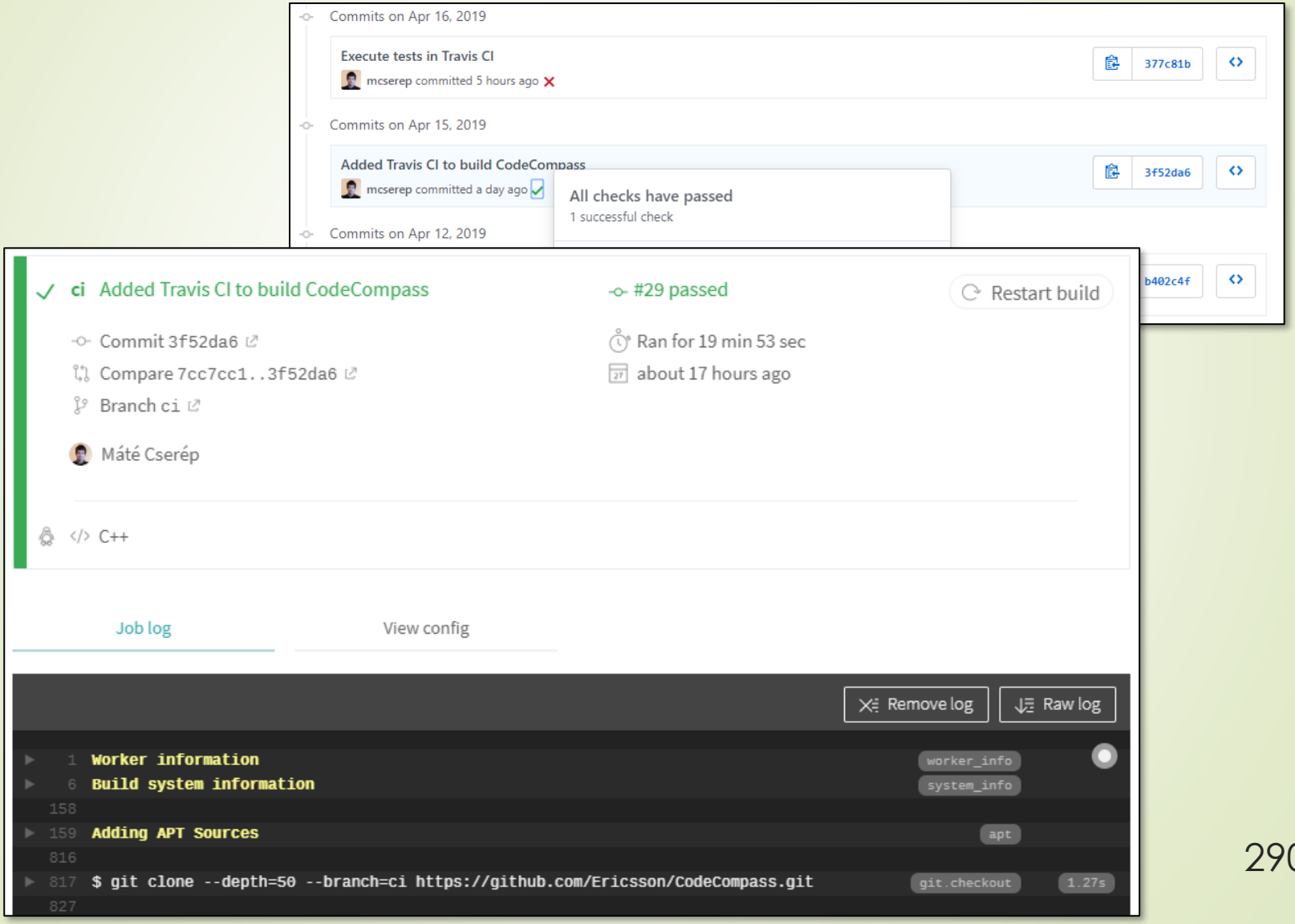
Travis CI

- Folyamatos integrációs szolgáltatás GitHub projektekhez
  - Korábban népszerű, nyílt forráskódú projektekhez ingyenesen használható, de 2021 ót már nem.
  - https://travis-ci.com/
- Támogatja a Linux, a Windows és a macOS operációs rendszereket, több előkészített környezettel (*image*)
  - A környezetet a használt programozási nyelvhez állíthatjuk be, az elterjedtebb fordító eszközökkel és könyvtárakkal
- A CI konfigurációt a `.travis.yml` fájlban adhatjuk meg

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

Travis CI – GitHub integration



Folytonos integráció és kiadás

Travis CI

- Például:

```
os: linux
dist: xenial
language: java

stages:
  - compile
  - test
  - deploy

# ...

jobs:
  include:
    - stage: compile
        - mvn compile
    - stage: test
        - mvn test

# ...
```

Folytonos integráció és kiadás

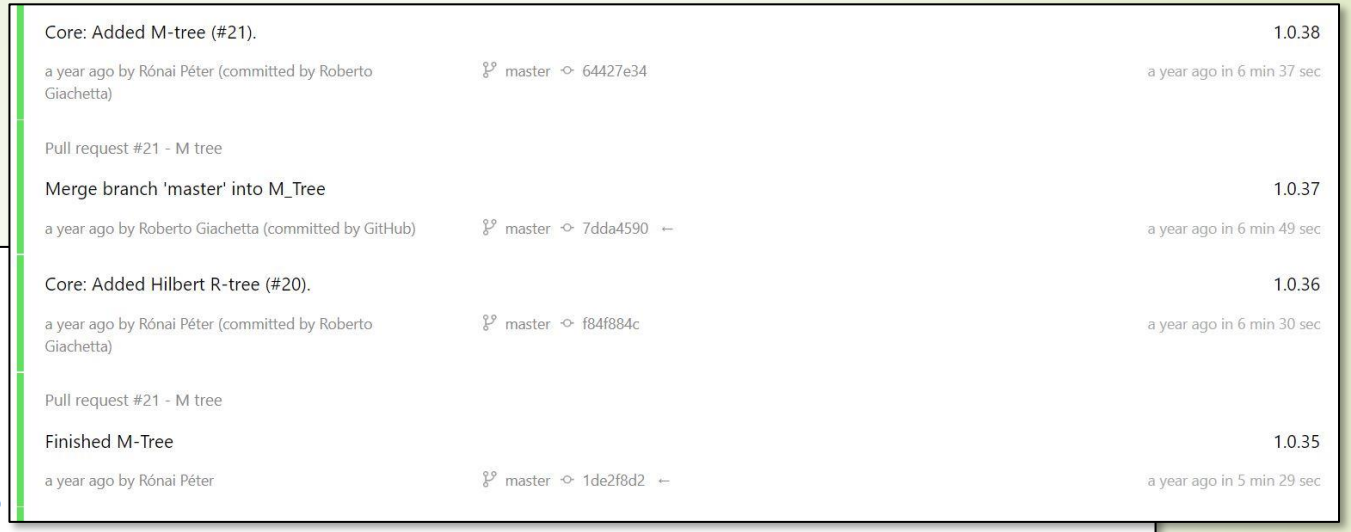
AppVeyor CI

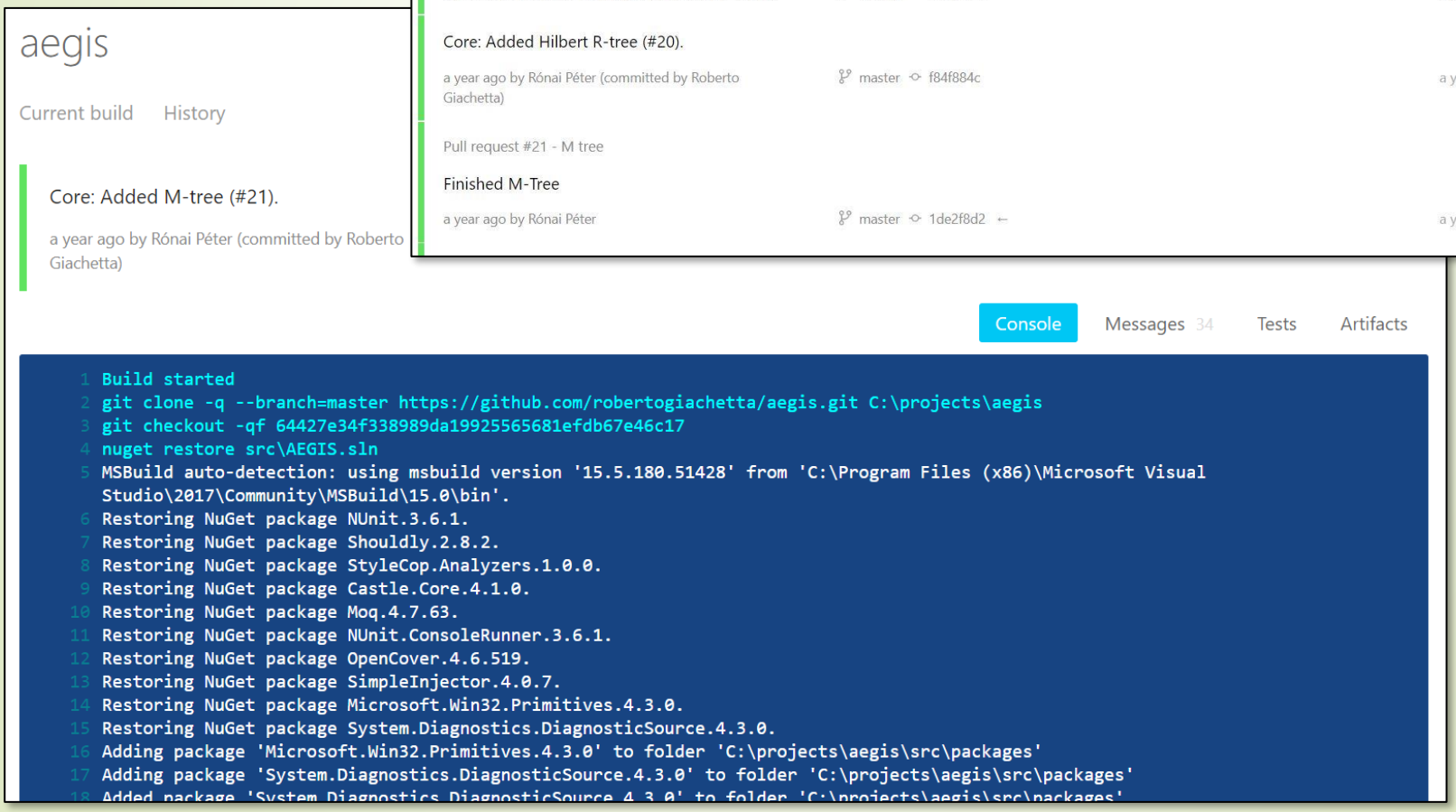
- Folyamatos integrációs szolgáltatás
  - Integrálható a GitHub, GitLab, BitBucket, Visual Studio Team Services platformokkal
  - Nyílt forráskódú projektekhez ingyenesen használható
  - https://www.appveyor.com/
- Támogatja a Linux, a Windows és a macOS operációs rendszereket, több előkészített környezettel (*image*)
  - Kiemelt .NET és Visual Studio támogatás
- A CI konfigurációt a `appveyor.yml` fájlban adhatjuk meg

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

AppVeyor CI – webes interfész





Folytonos integráció és kiadás

AppVeyor CI

- Például:

```
version: 1.0.{build} # Version format
image: Visual Studio 2017 # Build worker image
platform: Any CPU # Build platform
configuration: Debug # Build Configuration

# Execute script before build
before_build:
  - dotnet restore src\MyProject.sln

# Execute build script
build_script:
  - dotnet build src\MyProject.sln

# Execute test script
test_script:
  - dotnet test src\MyProject.sln
```

Folytonos integráció és kiadás

Jenkins

- Nyílt forráskódú folyamatos integrációs szolgáltatás
  - Integrálható a GitHub, GitLab, és egyéb projektvezető szolgáltatásokkal
  - Nem nyújt hoszting szolgáltatást, de más vállalkozások kínálnak (pl. CloudBees)
  - https://jenkins.io/
- A CI konfigurációt a `Jenkinsfile` adja meg, például:

```
pipeline {
    agent { docker { image 'maven:3.3.3' } }
    stages {
        stage('build') {
            steps {
                sh 'mvn compile'
            }
        }
    }
}
```

Folytonos integráció és kiadás

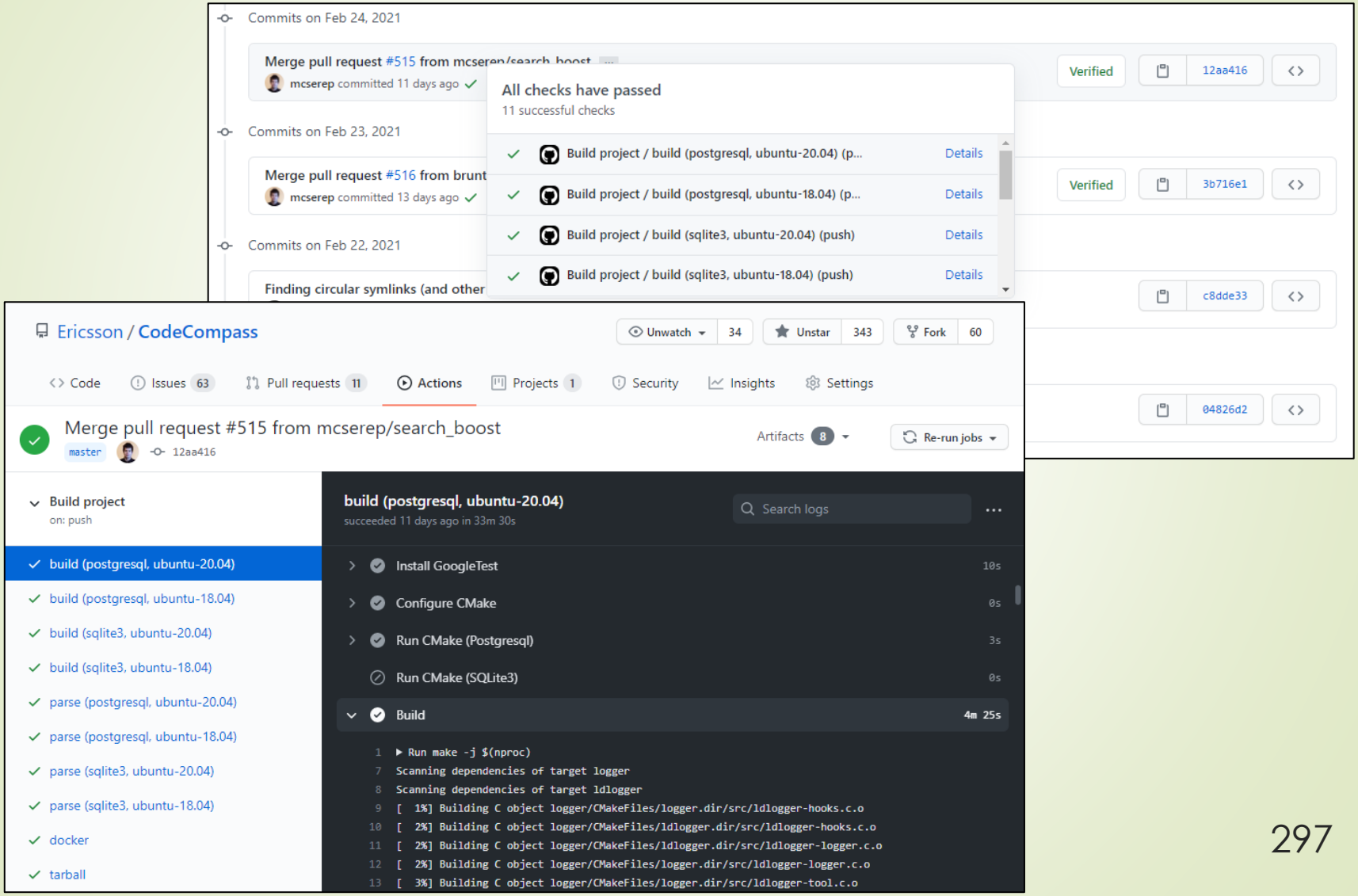
GitHub Actions

- A GitHub viszonylag új, saját CI/CD szolgáltatása
  - Ingyenes fiókkal korlátozott mértékig (jelenleg 2000 futtatási perc / hó) ingyenesen használható, ez előfizetéssel bővíthető
  - https://docs.github.com/en/actions
- Linux (Ubuntu), Windows és macOS operációs rendszer alapú virtuális gépek futtatását támogatja, de használható Docker
  - A virtuális gépek a Microsoft felhőjében (*Azure*) futnak
  - Telepíthető saját *self-hosted* GitHub Actions Runner is
- A CI konfigurációt a `.github/workflows/` könyvtárban, YAML fájlokban adhatjuk meg
  - Shell utasítások mellett magunk vagy mások által elkészített *action*-öket is felhasználhatunk, mint komplex építő elemek

Folytonos integráció és kiadás

ELTE | IK
INFORMATIKAI KAR

GitHub Actions



Folytonos integráció és kiadás

GitHub Actions

- Például:

```
name: Build project
on: [push, pull_request]
jobs:
  build:
    runs-on: [ubuntu-latest]
    steps:
    - uses: actions/checkout@v4

    - name: Set up JDK 1.21
      uses: actions/setup-java@v4
      with:
        distribution: zulu # Azul Zulu OpenJDK
        java-version: 21

    - name: Build the project
      run:  mvn compile

    - name: Run the unit tests
      run:  mvn test
```

Software Technology

Project Management Tools

Dr. Máté Cserép
ELTE, Faculty of Informatics
2025.

Project Management Tools

Software tools

- The work of the development team must be supported by appropriate software tools, i.e.:
  - a *project management system* (*project tracking system*) that supports documentation and task tracking
  - an *advanced design tool* (*case tool*), where the process of development and responsibility can be tracked
  - an *integrated development environment* (*IDE*)
  - a *version control system* (*revision control system*), that allows to track program code changes
  - a *continuous integration* system to ensure early detection of errors

Project Management Tools

Functionality

- The project management tool provides the following options:
  - defining the schedule and risks of development
  - the possibility and generation of simple and continuous documentation of development
  - recording and tracking tasks, activities
  - fixing errors that occurred during testing, and following the repair process
  - integrated version control and source code browsing
  - a web or graphical interface that provides ease of use and access from anywhere
  - *code review* interface for methodological assessment

Project Management Tools

Schedule and timing

- The software gives us the opportunity to prepare the project schedule and keep it in focus
  - we can define *milestones* for which tasks have to be completed
  - developers can see their own tasks separately, and manage their progress
  - we can distribute the resources of the development steps
  - we can define dependencies between program parts
  - we can manage the time course of individual development steps, predict the effects of deviations from the plan on resources and further development times

Project Management Tools

Task and error tracking

- Systems allow designers to set tasks and testers to report errors in the program
  - the tasks can be created and published using so-called *tickets* (or *issues*)
    - they may indicate a new *feature*, *bug*, other development task (*task*), or *documentation*
    - we can specify their description, deadline and the assigned responsible team member
    - they can be commented, closed, reopened
  - the tickets provide logging of the development and testing process

Project Management Tools

ELTE | IK
INFORMATIKAI KAR

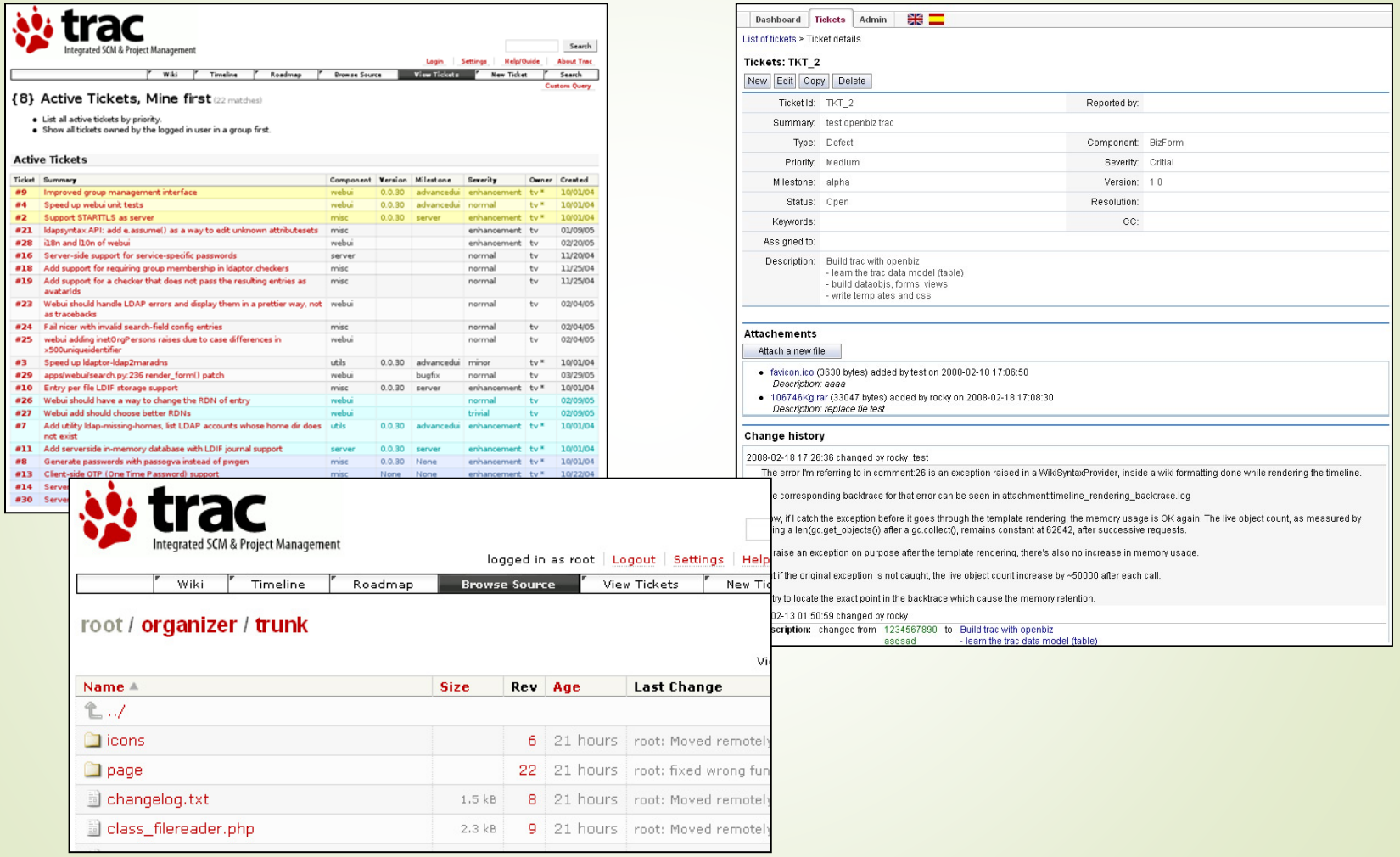
Examples

- The UI of these tools is implemented by suitable web technology, while the data storage is implemented by a database engine
  - most tools are open source and their project management is done with the same tool
- Some popular project management tools:
  - *Trac*: Python based, MySQL / SQLite / PostgreSQL database background
  - *Redmine*: Ruby on Rails based, MySQL / SQLite / PostgreSQL database backgrounds
  - *Azure DevOps Services*: Microsoft's solution, Git and TFVC version control support, ASP.NET and MSSQL backend (known as *Microsoft Team Foundation Server* until 2019)
  - *YouTrack*: JetBrains' system, Java based, Xodus Database background (NoSQL)

Project Management Tools

ELTE | IK
INFORMATIKAI KAR

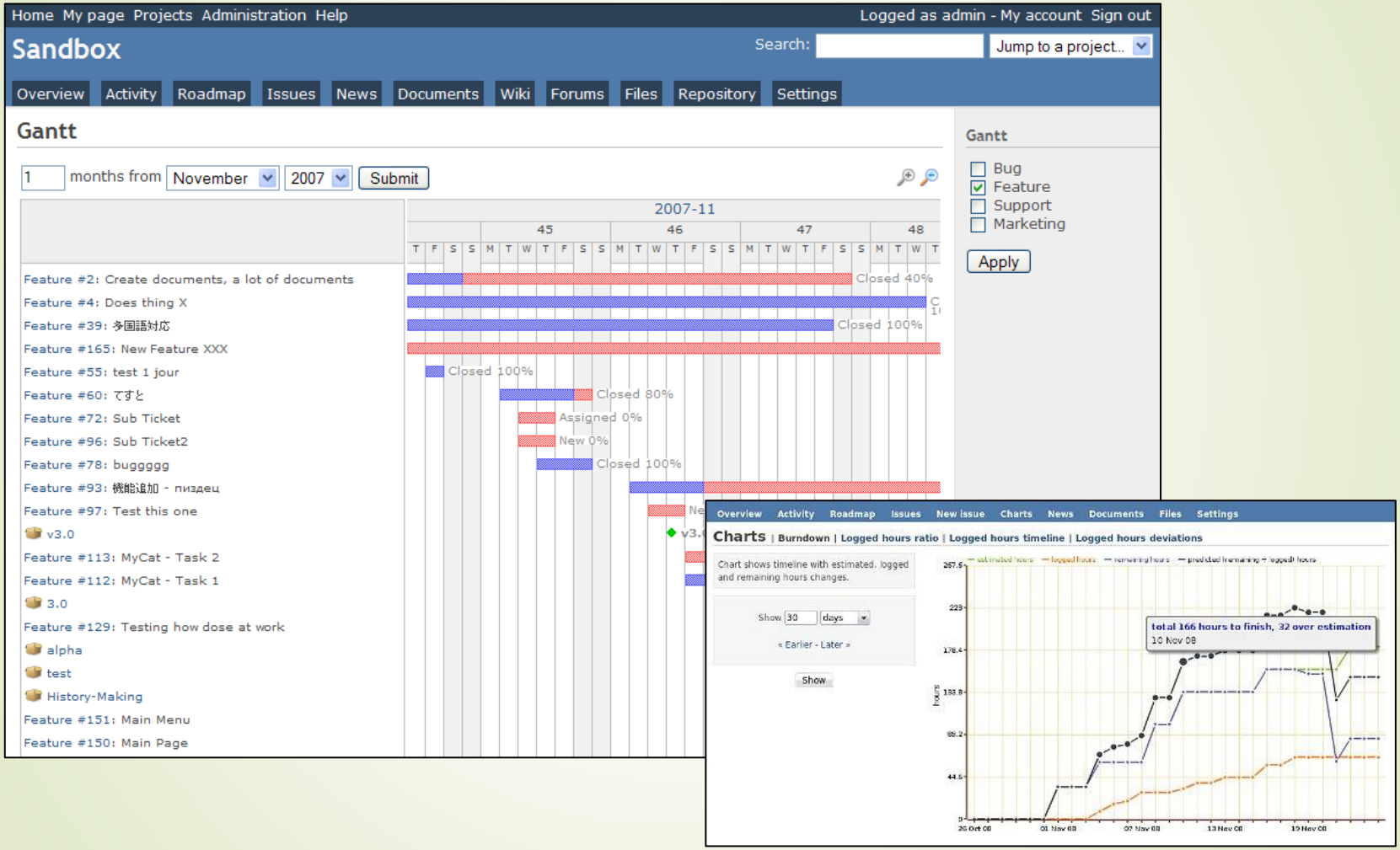
The Trac project manager



Project Management Tools

ELTE | IK
INFORMATIKAI KAR

The Redmine project manager



Project Management Tools

Project hosting services

- *Project hosting services* usually provide most project development tools and functionalities as online services
  - project management, code storage, code viewing, version control, core review, documentation (wiki), mailing list
  - can be integrated with other project management and development tools
  - guarantees the availability and integrity of the codebase
  - usually it is free of charge for open source software
  - E.g.: *GitHub*, *GitLab, SourceForge, Bitbucket, Azure DevOps services*

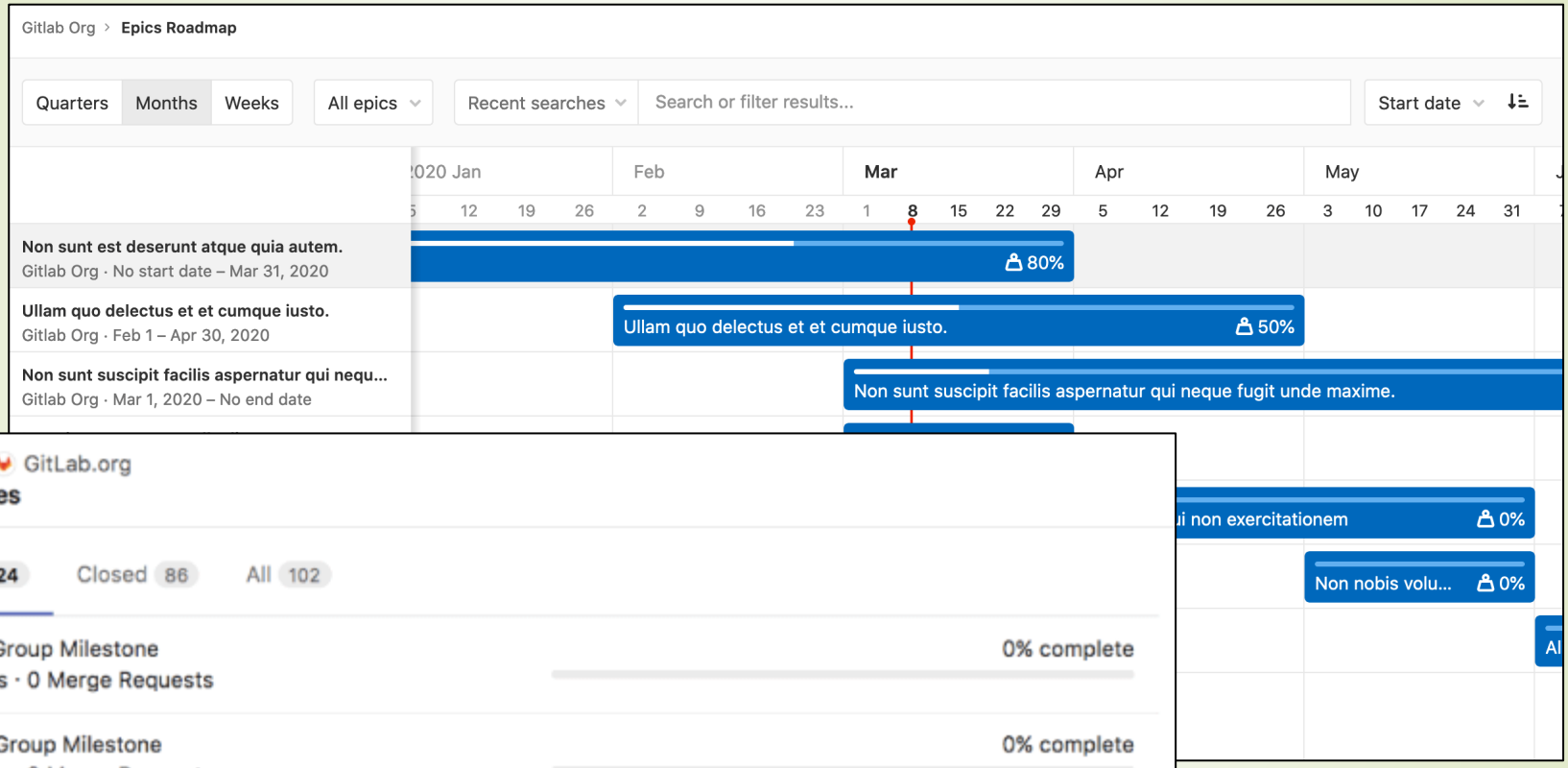
Project Management Tools

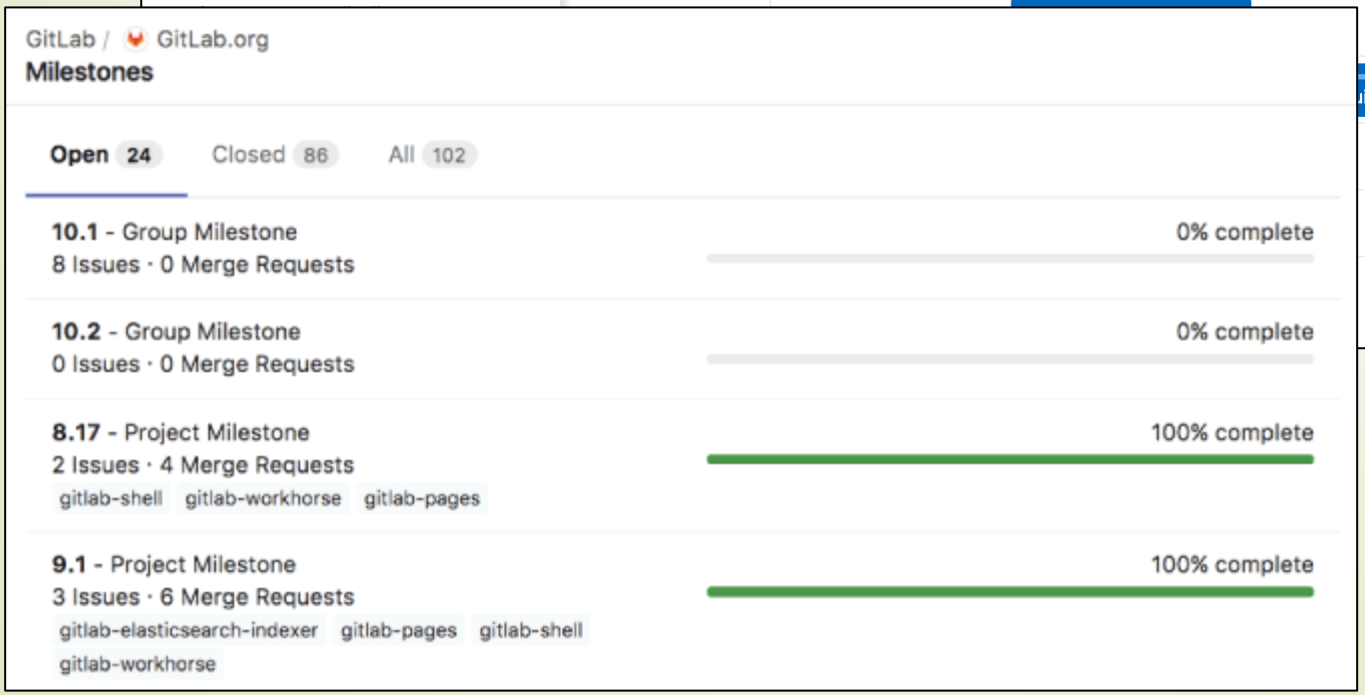
The GitLab Project manager services

- GitLab is a web-based project manager that integrates many software tools
  - Can be used via gitlab.com (SaaS)
    - similar to GitHub
    - free, unlimited (private) project (subscription available)
  - Gitlab Community Edition
    - *self-hosted*
    - suitable for small and medium sized teams
    - free, open source
  - GitLab Enterprise Edition
    - with extra functionality that is typically required for larger developer team (100+ people)

Project Management Tools

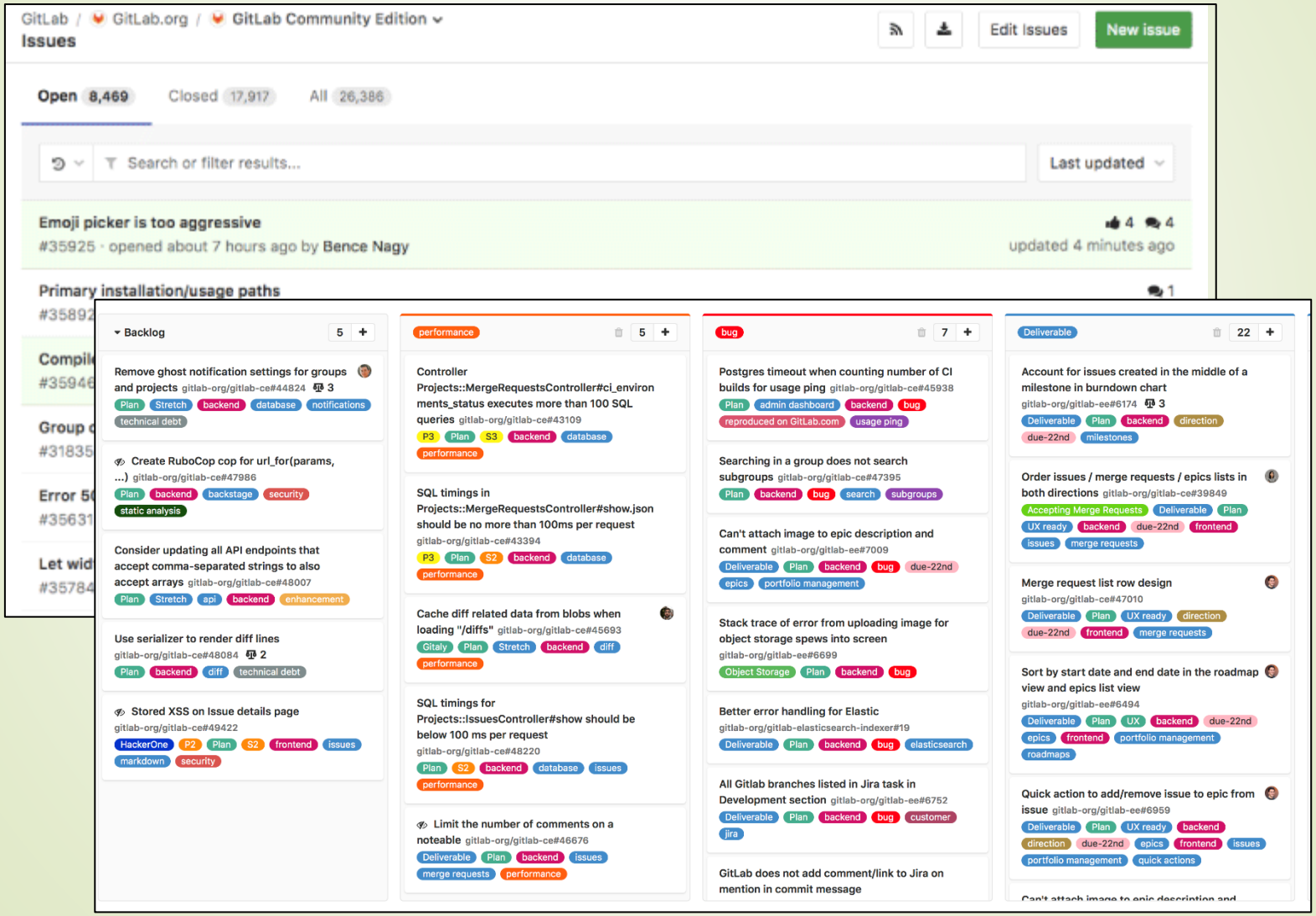
GitLab – Project management





Project Management Tools

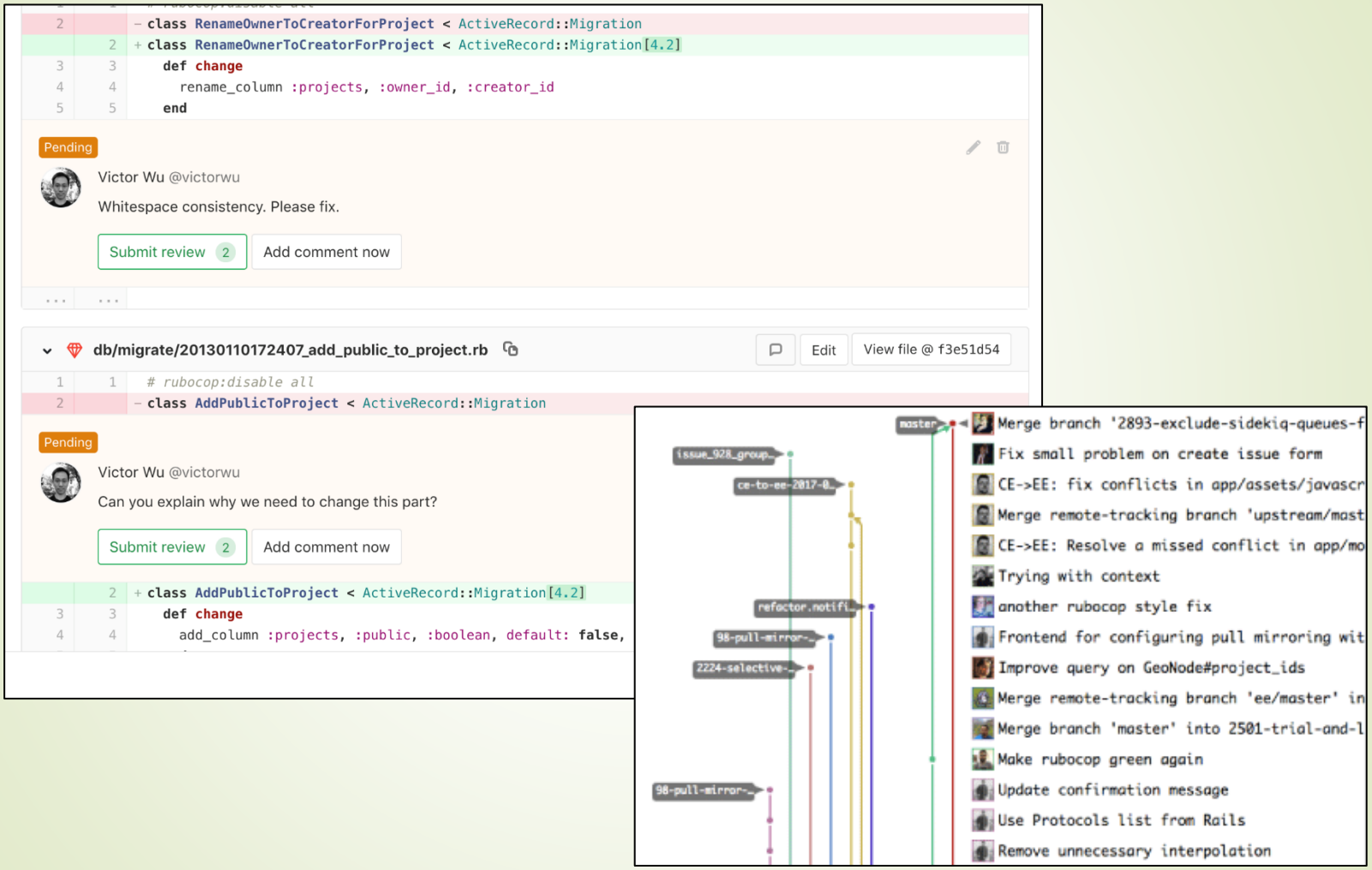
GitLab – Issue tracking



Project Management Tools

ELTE | IK
INFORMATIKAI KAR

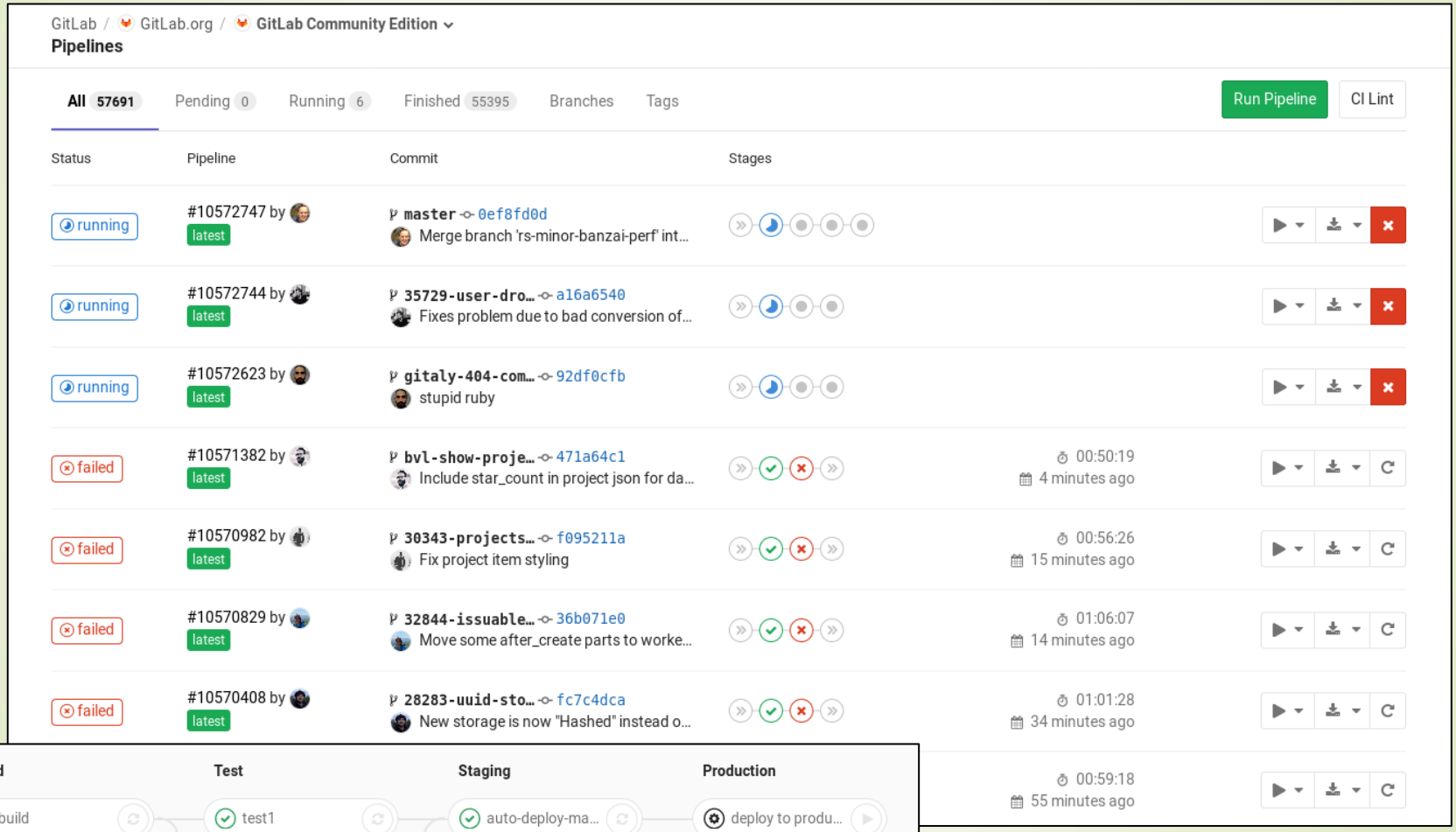
GitLab – Source code management



Project Management Tools

ELTE | IK
INFORMATIKAI KAR

GitLab – Continuous integration



Project Management Tools

ELTE | IK
INFORMATIKAI KAR

Faculty GitLab server

- Throughout the semester the group projects must be administered on the faculty GitLab server instance for the Software Technology course.
  - URL: https://szofttech.inf.elte.hu/
  - Authentication: *INF account* (same you log into the laboratory computers)
- Both the specification, the implementation, the documentation and the testing can be done with GitLab.
  - Wiki pages with markdown syntax
  - Issue tracking
  - Source code management
  - Continuous integration, automated testing

Software Technology

Version control systems

Specializations A & B

Dr. Máté Cserép

ELTE, Faculty of Informatics

2025.

Version control systems

Historical background

- The exponential growth in size and complexity resulted in a *software crisis* in the 1970s and lead to an increment in the:
  - size of program source code,
  - time required to complete software projects,
  - required human resource (developers).
- As the software industry advanced into a new era, more and more applications were developed
  - the life-cycle of software projects usually did not end with the release of the first public version,
  - maintenance and further development phases followed.

- **Software projects increased both in size, complexity, development time and participating programmers.**

Version control systems

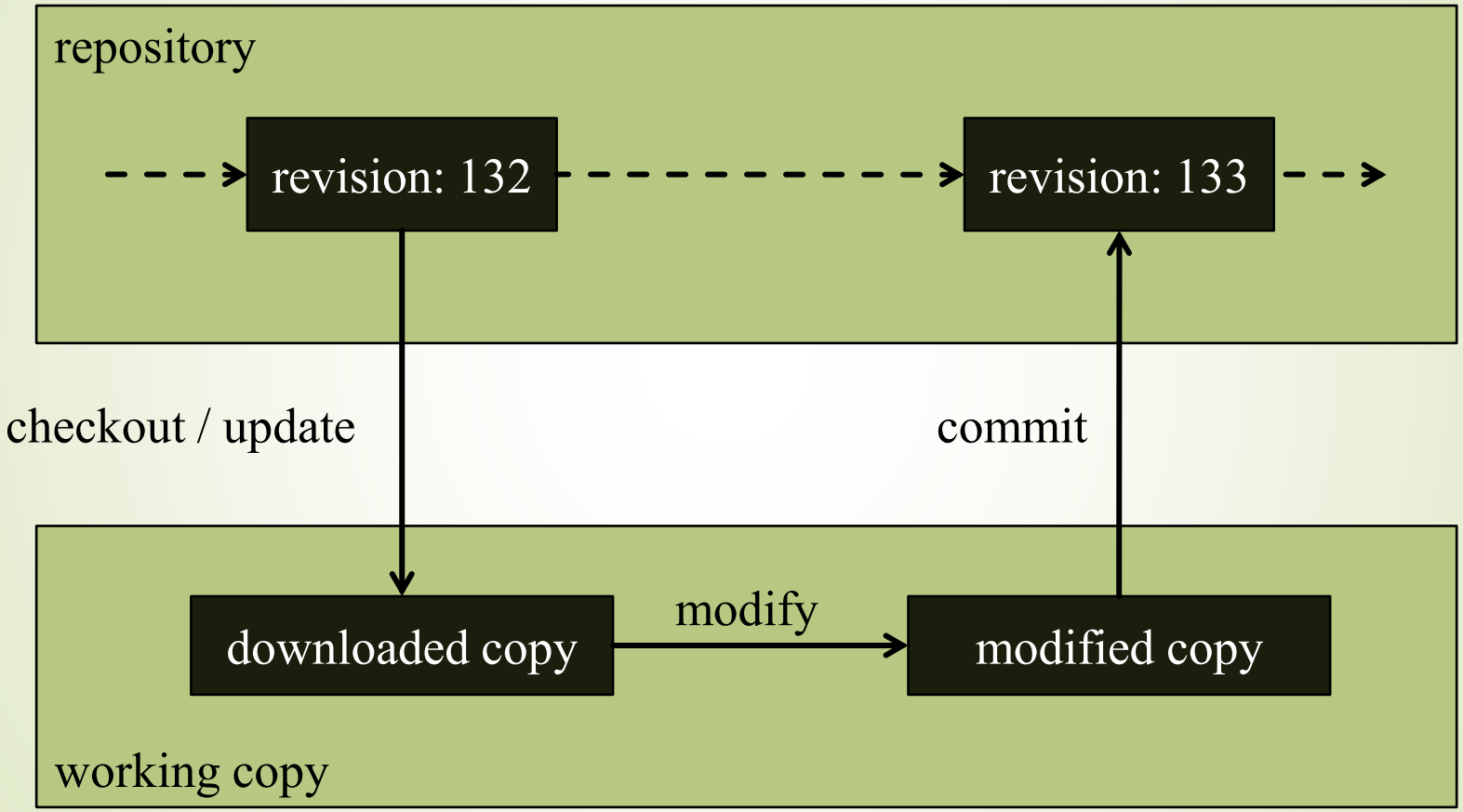
Functionality

- As implementation is carried out in multiple phases, often by multiple developers, it is necessary for the subsequent and parallel states of the program to be easily trackable. This task is accomplished through the *revision control systems*
  - e.g. *CVS*, *Apache Subversion* (*SVN*), *Mercurial*, *Git*
  - the source code is stored in a shared, central *repository*
  - from which the programmers may create their local *working copy* for development
  - the modifications are uploaded back to the central repository (*commit*)
  - after the initial creation of the working copies (*checkout*), they must be continuously *updated*

Version control systems

ELTE | IK
INFORMATIKAI KAR

Functionality



Version control systems

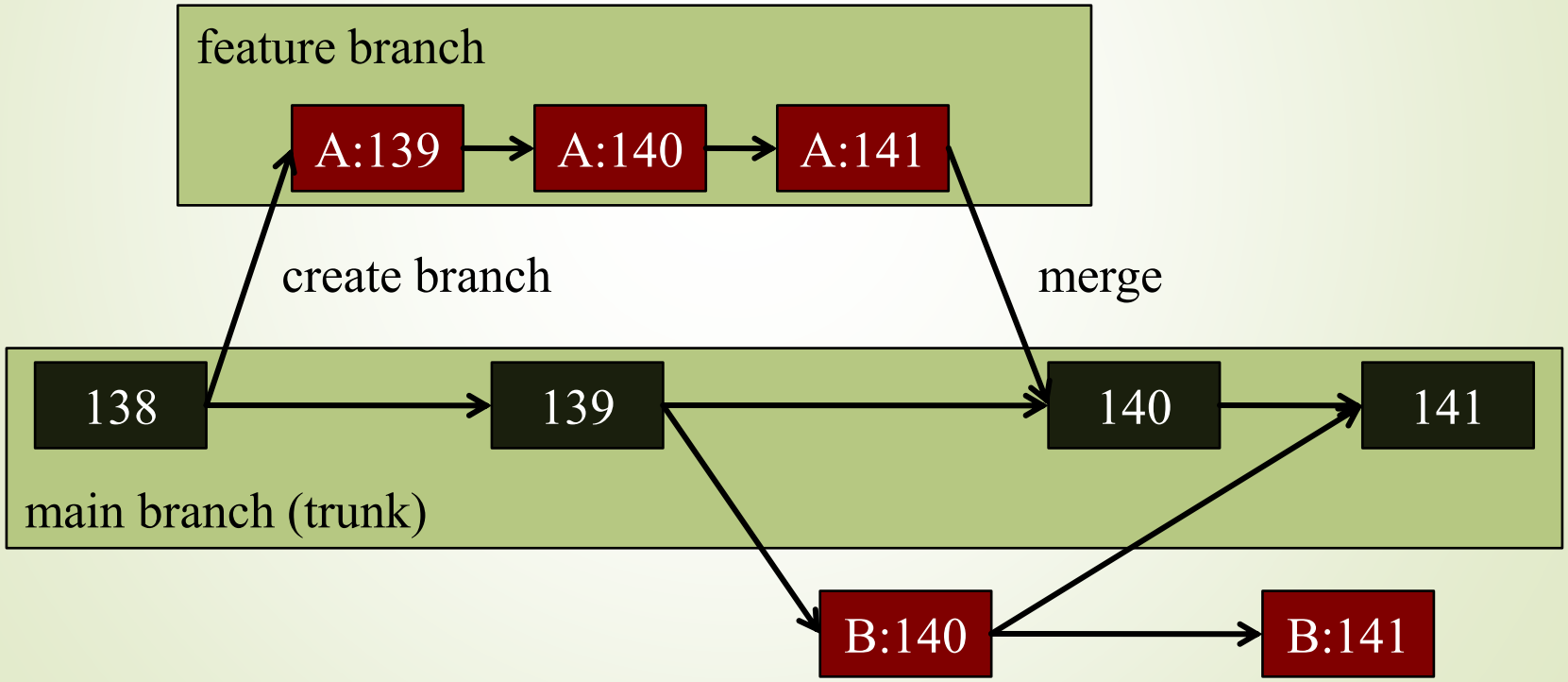
Functionality

- Revision control systems enables us to:
  - store all previous *revisions* and to make a local working copy from them
  - access the main line of development (usually named *baseline*, *master* or *trunk*), the most recent version (*head*), and to upload new revisions with documentation
  - recording the changes between specific revisions
  - undoing changes, *restoring* previous revisions
  - checking possible source code conflicts resulting from modifications and *resolving* them

Version control systems

Functionality

- branching new lines of development (*branch*), which run alongside the main line, and can be merged with each other or back to the main line of development



Version control systems

Functionality

- merging often requires additional manual interaction and corrections
- the changed code parts can usually be merged automatically through code analysis, which can be *two-way*, using only the 2 states of the modified file, or *three-way*, analyzing the state of their common ancestor
- *exclusively lock* code parts to avoid conflicts
- creating a *snapshot* from a specific revision and marking it with a *tag* (e.g. for easy or public access)
- treating commits as atomic operations (e.g. to handle network failures during upload)

Version control systems

Local version control system (1st generation)

- Tracking the changes of source code, managing the combination of features added to different releases
  - local repository (though can be accessed by multiple developers when deployed on e.g. a *mainframe*)
  - file based operations (1 revision contains changes for 1 file)
  - concurrency control through exclusive locks

- The theoretical foundations defined in the 1970s:
  - Source Code Control System (SCCS) – 1972
  - Revision Control System (RCS) - 1982

Version control systems

Centralized version control system (2nd generation)

- Parallel development by multiple programmers
  - centralized model with a client-server architecture
  - fileset based operations (1 revision contains changes for a set of files)
  - concurrency control usually by *merging before committing*
- Became widespread in the 1990s:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
  - SourceSafe, Perforce, Team Foundation Server, etc.
- *Drawback: the central role of the server (e.g. failure), furthermore version control operations requires network connection, hindering offline work*

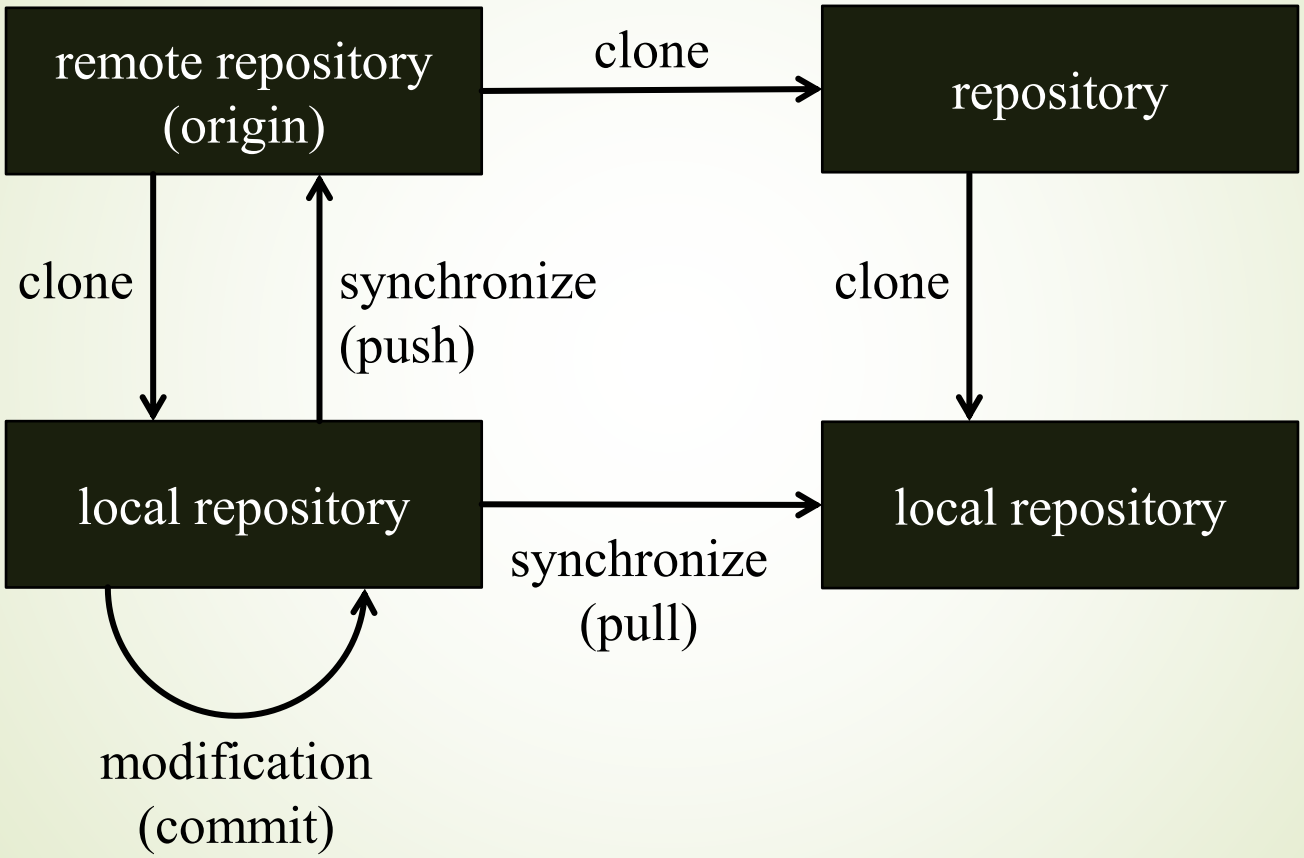
Version control systems

Distributed version control system (3rd generation)

- Network communication is decoupled from the classical version control operations and become separate actions which can be initiated by the user
  - decentralized, distributed network model
  - each client has a copy of the complete codebase, including its full version history
  - version control operations are executed on the local repository of the client
  - synchronization is done based on a *peer-to-peer* approach, but well-known servers can be established to boost efficiency
  - concurrency control usually by *committing before merging*
- Introduced in the early 2000s:
  - Monotone, Darcs, Git, Mercurial, Bazaar, etc.

Version control systems

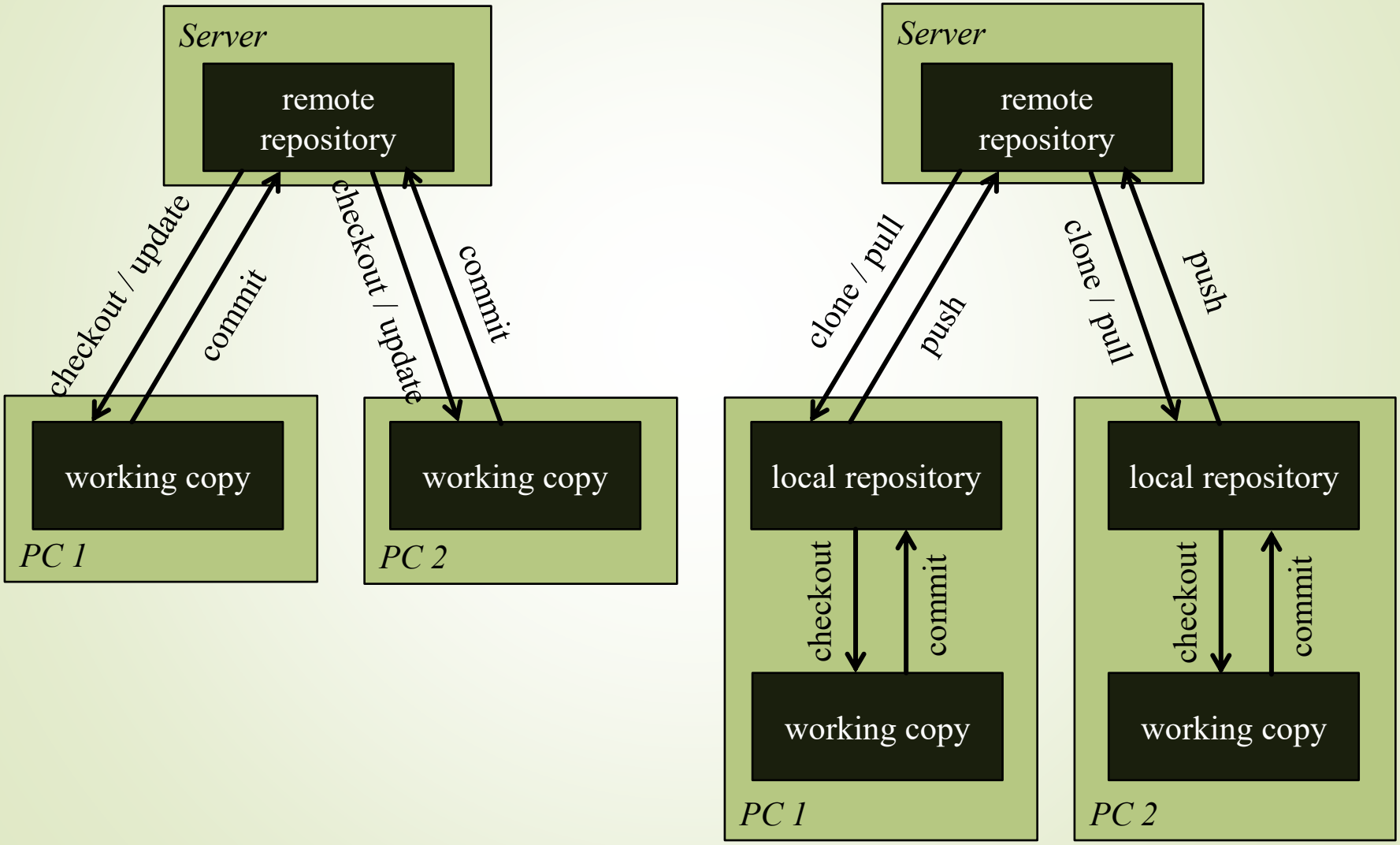
Distributed version control system (3rd generation)



Version control systems

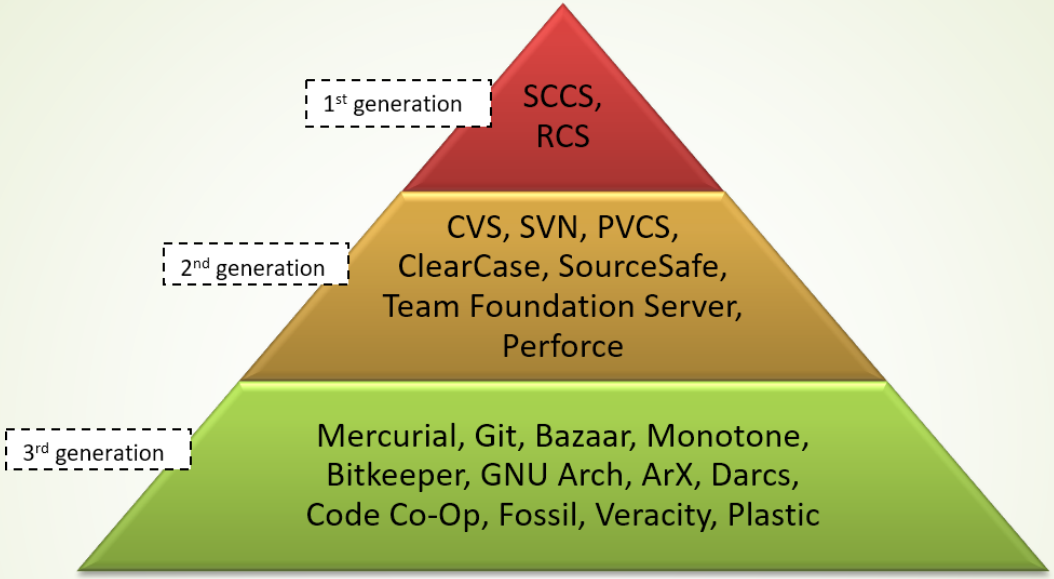
ELTE | IK
INFORMATIKAI KAR

Centralized and distributed version control systems



Version control systems

Generation model



Generation	Network model	Operation atomicity	Concurrence handling
First	Local	Single files (non-atomic commits)	Exclusive locks
Second	Centralized	Fileset (atomic commits)	Merge before commit
Third	Distributed	Fileset (atomic commits)	Commit before merge

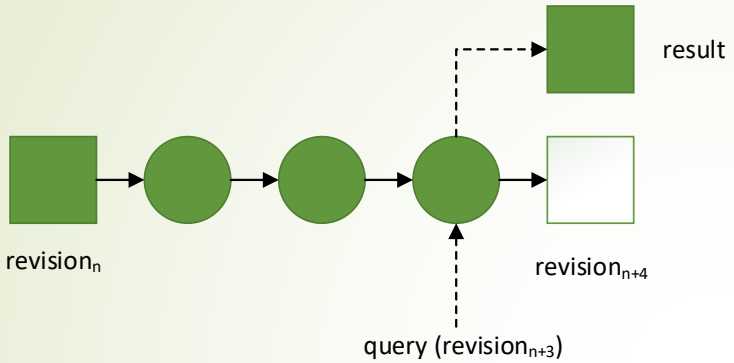
Version control systems

Changeset representation

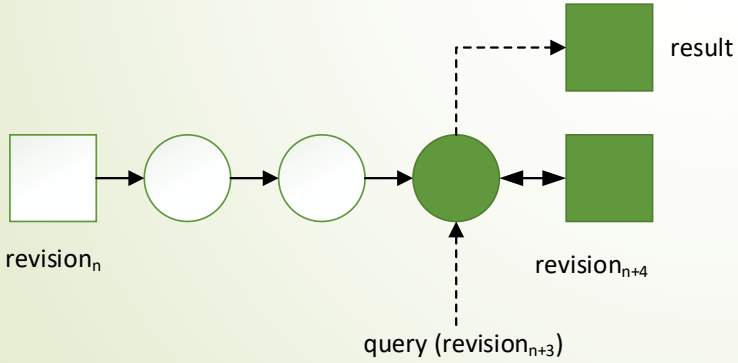
- Storing the full state of all revisions in the repository would waste significant storage space and raise processing costs
- Hence previous (1st and 2nd generation) revision control systems only store the difference (*changeset*, *delta*) between any two subsequent revisions
  - some systems (e.g. Mercurial) may create a *snapshot* from the full content periodically
- Initially (SCCS) *forward deltas* were utilized to create the versions from the previous ones
- It was early realized (RCS), that using *reverse deltas* and storing the full state of the newest revision can provide better performance, since recent revisions of a branch are checked out more frequently compared to older ones
  - Mixed solution: using reverse deltas on the main line and forward deltas on the side branches

Version control systems

Changeset representation



Forward deltas

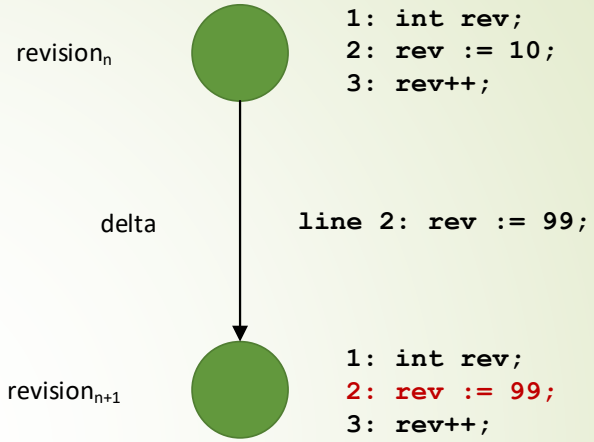


Reverse deltas

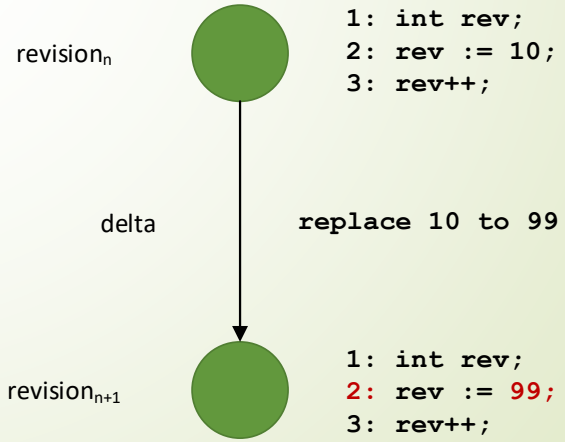
Version control systems

Changeset representation

- Determining changes in textual documents, like source code files is usually done on a *state-based* approach
  - in most cases comparing content line-by-line
    - e.g. GNU diff
  - for structured content the unit of comparison might be something different (e.g. XML, JSON, UML)
- For binary data (e.g. images) *operation-based* approaches can also be used.



State-based

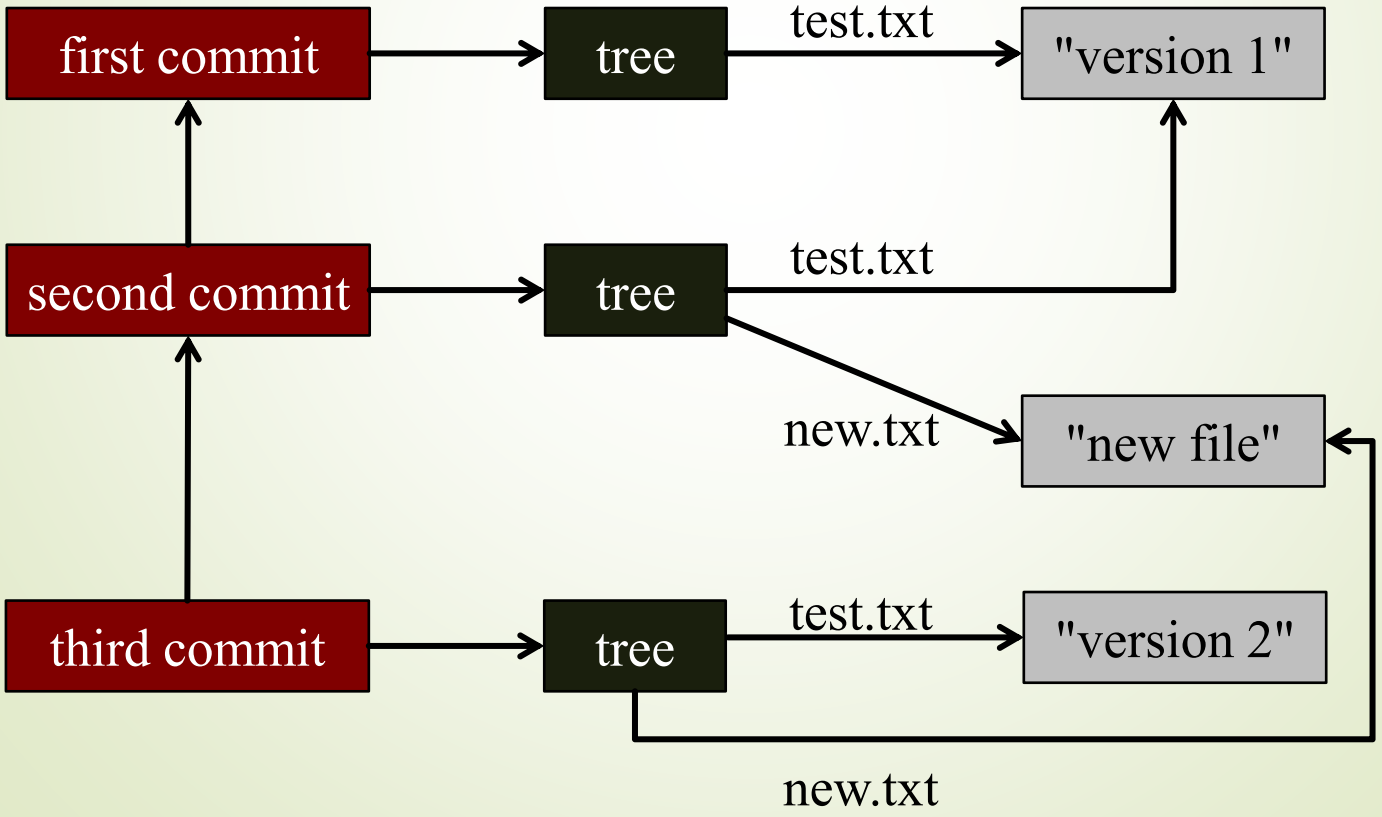


Operation-based

Version control systems

Changeset representation

- By the time of 3rd generation version control systems, storage capacity significantly increased, while their cost was reduced
  - Newer version control tools (like Git) store the entire content of modified files, not only the changed parts (*delta*)



Version control systems

Git

- During the seminar classes, student project will be version controlled with the Git version control system, which is tightly integrated into the faculty GitLab instance.
  - Faculty GitLab server: https://szofttech.inf.elte.hu/
- The *remote repository* available on the GitLab server can be browsed and viewed on the web user interface of GitLab. Basic edit operations are also possible (these will also be *commits*).
- Version control of the source code is usually performed on a *local repository* with a client application. The local repository is regularly synchronized with the remote. We can use:
  - the command-line commands; or
  - a graphical desktop client application.

Version control systems

Git: installation

- The Git version control system can be easily installed:
  - Windows, Mac installer: https://git-scm.com/downloads
  - Debian/Ubuntu: `apt-get install git`
  - Other UNIX systems: https://git-scm.com/download/linux
- After installation we can immediately start to work with the command-line instructions.
  - Therefore, on a Windows system, it is recommended to add the Git binaries to the PATH environment variable.
  - Graphical client applications might be installed separately.
- For each Git commit the name and email address of the author (the *committer* to be more precise) must be assigned. The values shall be configured beforehand globally.

```
git config --global user.name "Sample Student"
git config --global user.email student@inf.elte.hu
```

Git: creating a repository

- A new empty repository can be initialized:

```
git init
```

- Or an existing remote repository can be cloned:

```
git clone https://mysite.com/best-project.git
```

- We usually need to clone remote repositories.
  - The remote repository, which was cloned will get the alias name *origin* in our local repository (by default), This can be used later, e.g. on synchronization.

Git: tracking files

- We can start tracking the changes of files with the `git add` command, adding them to the *staging area*:

```
git add main.cpp
```

  - Not only single files, but patterns (e.g. `*.cpp`) or complete directories can be added.

- The current state of our working copy can be checked with the `git status` command anytime.

```
git status
> On branch master
> Your branch is up to date with 'origin/master'.
> Changes to be committed:
>   (use "git reset HEAD <file>..." to unstage)
>         new file:   main.cpp
```

Git: creating a new revision

- Once we have finished added the needed changes to the *staging area*, we can commit it as a new revision to the local repository with the `git commit` command:

```
git commit -m "Added main program."
> [master d26c7a9] Added main program.
>  1 file changed, 1 insertion(+)
>  create mode 100644 main.cpp
```

Version control systems

Git: tracking further changes

- We can add new files to the *staging area*:

```
git add rectangle.h

git commit -m "Added Rectangle class."
```

- A new revision can contain new files and the modification of already version controlled files:

```
git add circle.h

git add main.cpp

git commit -m "Added Circle class.
               Modified the main program."
```

Git: synchronization with a remote repository

- Mew revisions created in our local repository have to be synchronized with the originally cloned remote repository from time to time, so our modifications will be accessible to our teammates. This can be done with the `git push` command.

```
git push origin master
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 247 bytes | 123.00 KiB/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To /path/to/workspace/folder
   d45172c..80a39a2  master -> master
```

  - We define which remote repository we would like to synchronize with and which branch. We can use the *origin* alias name for the cloned remote repository. The branch is the *master* in the above example.
  - When using *tracking branches* the name of the remote and the branch can be omitted, simplifying the command to `git push`.
- New branches and revision committed by other developers can be downloaded with the `git pull` command into out local repository.

Git: creating branches

- New branches can be created with the `git branch` command.

```
git branch new-branch
```

  - This new branch will diverge from the currently checked out revision in our working copy.
- We can switch between branches with the `git checkout` command.

```
git checkout new-branch
```

  - The default branch is usually named *master*.
- A new branch can be created and checked out in a single step:

```
git checkout -b new-branch
```

Git: merging branches

- After the completion of the purpose of a branch (e.g. developing a new feature), the changes should be merged back to the main development branch (master).
  - We can merge branches with the `git merge` command.
  - First we need to check out the master branch:

```
git checkout master
```

  - Then we can merge the modifications of the side branch:

```
git merge new-branch
```

- In case the same files and the same parts of them were modified on both branches, merging can usually cannot be done automatically. This is called a *merge conflict*.

Git: resolving merge conflicts

- In such case the conflict must be resolved manually, by editing the affected files.
- Conflicting code segments are marked with a special syntax by Git inside the source files themselves:

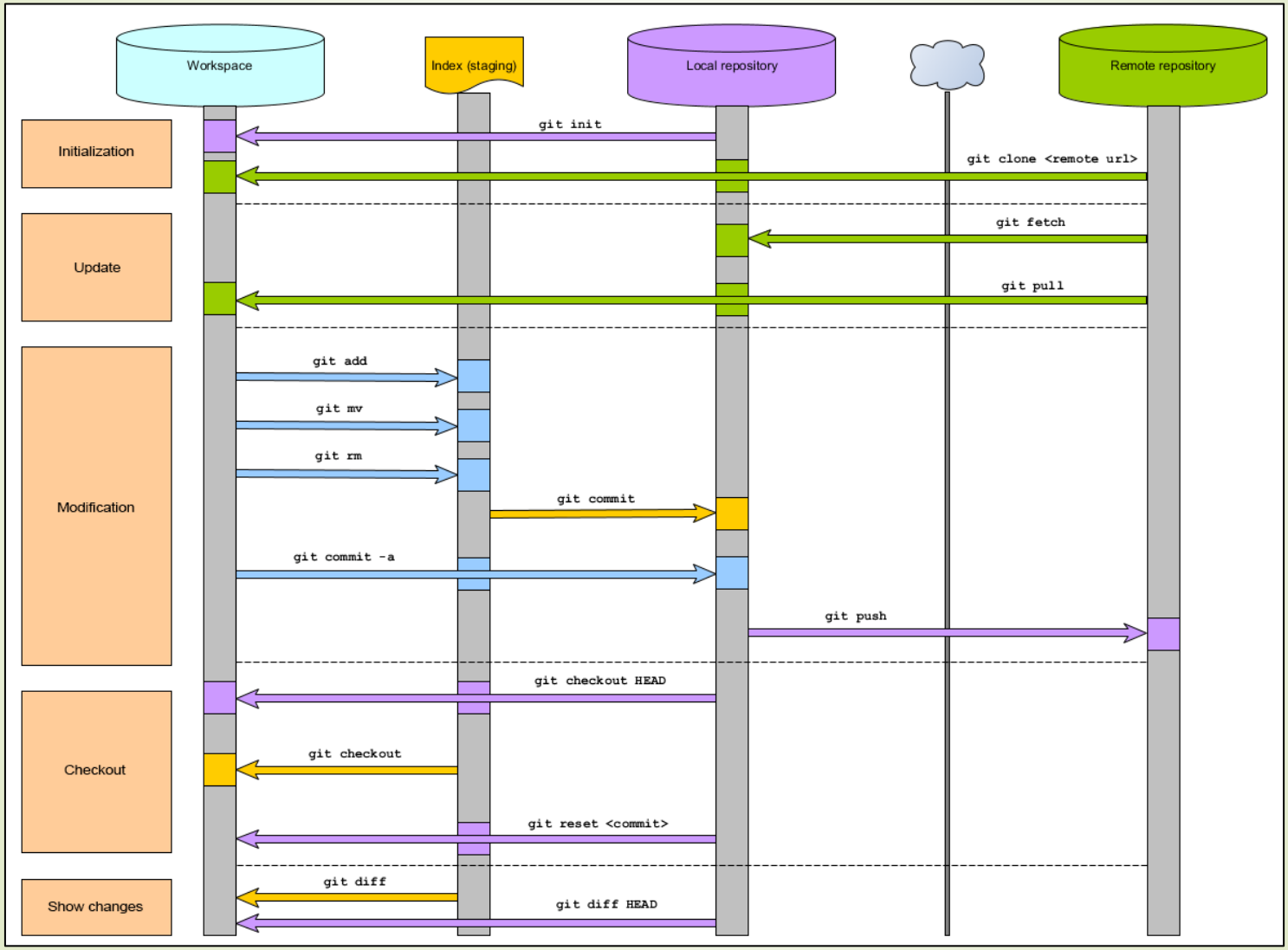
```
<<<<<<< HEAD
The content on the current branch, which is the master
in this example
=======
The content on the branch to be merged, which is the
new-branch in this example
>>>>>>> new-branch
```

  - It is the task of the developer to decide which version to use from the two; or define a new, third version as a resolution.
  - The resolved files must be added to the *staging area* (`git add`), then the modifications have to be committed (`git commit`).

Version control systems

ELTE | IK
INFORMATIKAI KAR

Git: overview of basic command line instructions

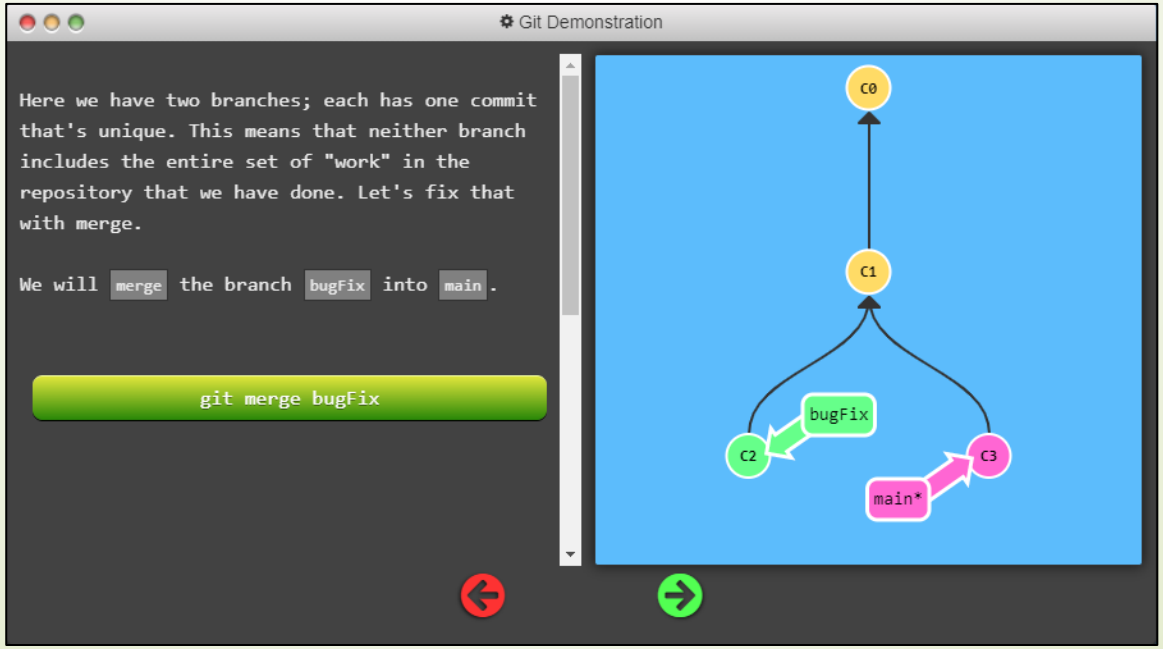


Source: Krisztián Nagy (ELTE FI)

Version control systems

Online tools to learn using Git

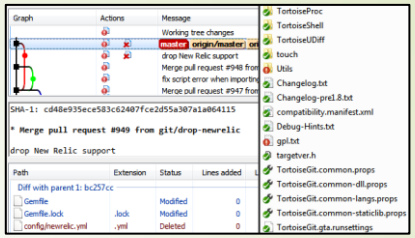
- Multiple online resources support understanding Git through visualizing how the Git history changes for the executed commands. The following tools are recommended:
  - https://git-school.github.io/visualizing-git/
  - https://learngitbranching.js.org/

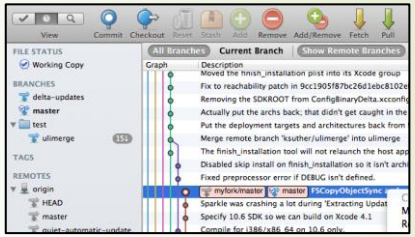


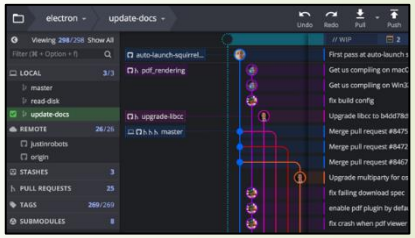
Version control systems

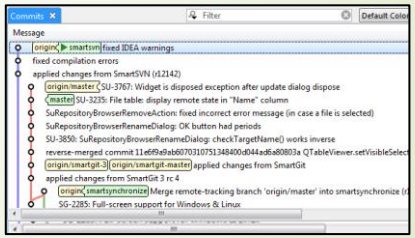
Git GUI clients

- TortoiseGit
  - Windows
  - *available in computer labs*
- SourceTree
  - Windows, Mac
- GitKraken
  - Linux, Windows, Mac
- SmartGit
  - Linux, Windows, Mac
- Further:
  - https://git-scm.com/downloads/guis





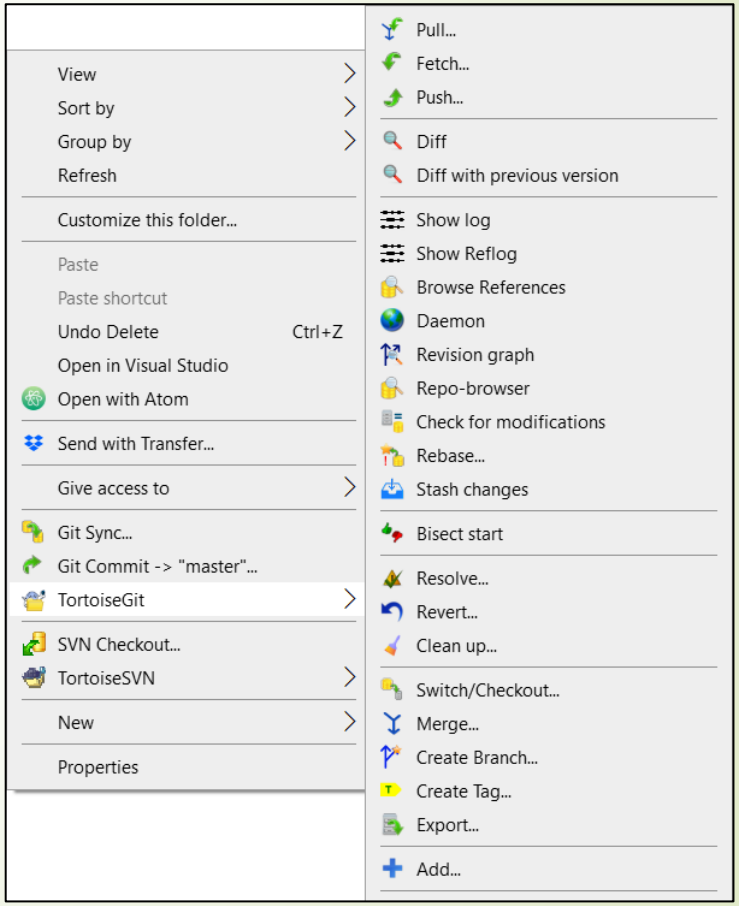




Version control systems

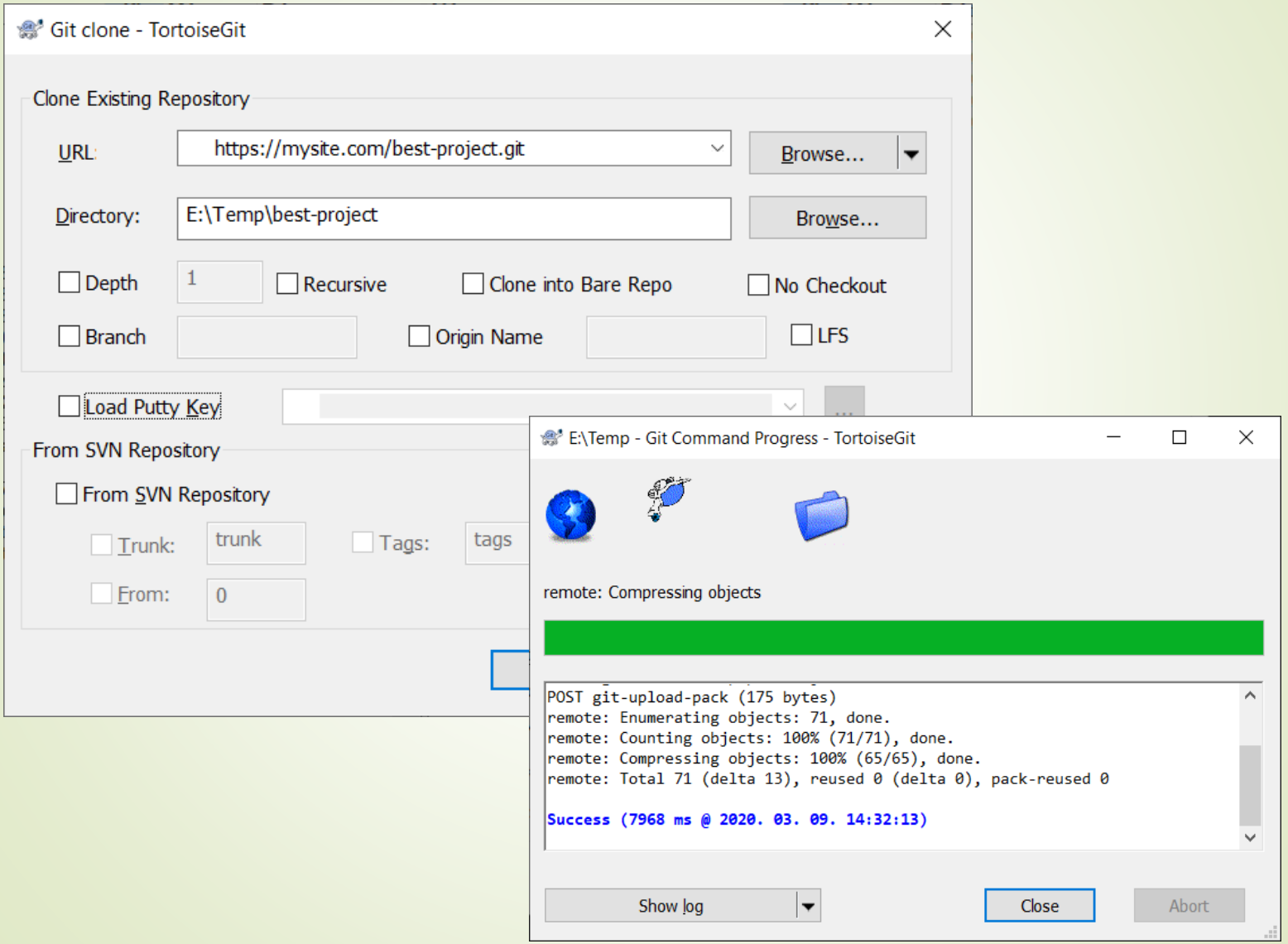
Using TortoiseGit

- TortoiseGit is a free graphical desktop client for the Windows operating system
  - Homepage, download: https://tortoisegit.org/
  - Does not include Git, which must be installed separately.
- When using TortoiseGit, the usual Git commands can be invoked from the (right mouse click) context menu of the *File Explorer.*



Version control systems

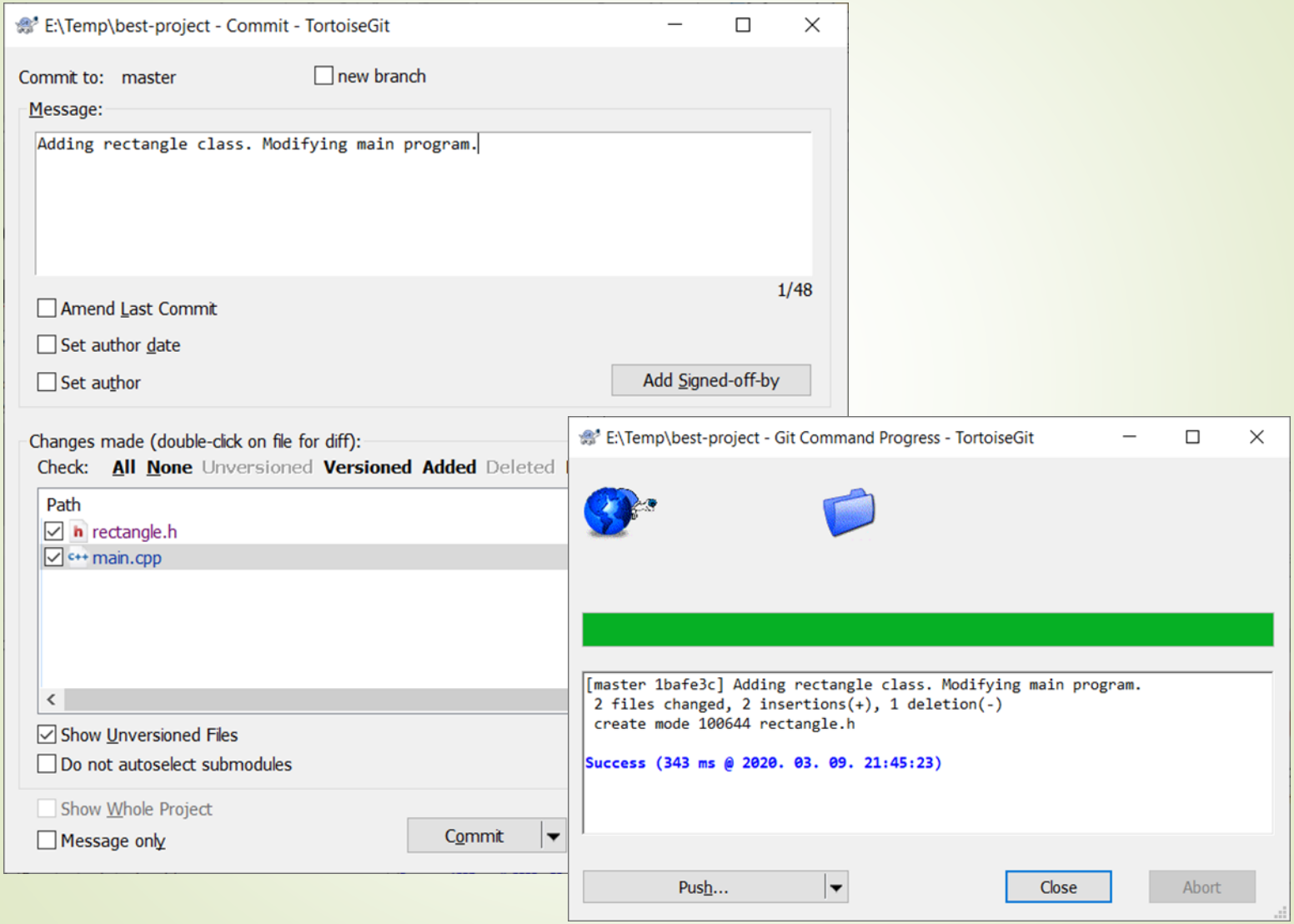
TortoiseGit: clone a repository



Version control systems

ELTE | IK
INFORMATIKAI KAR

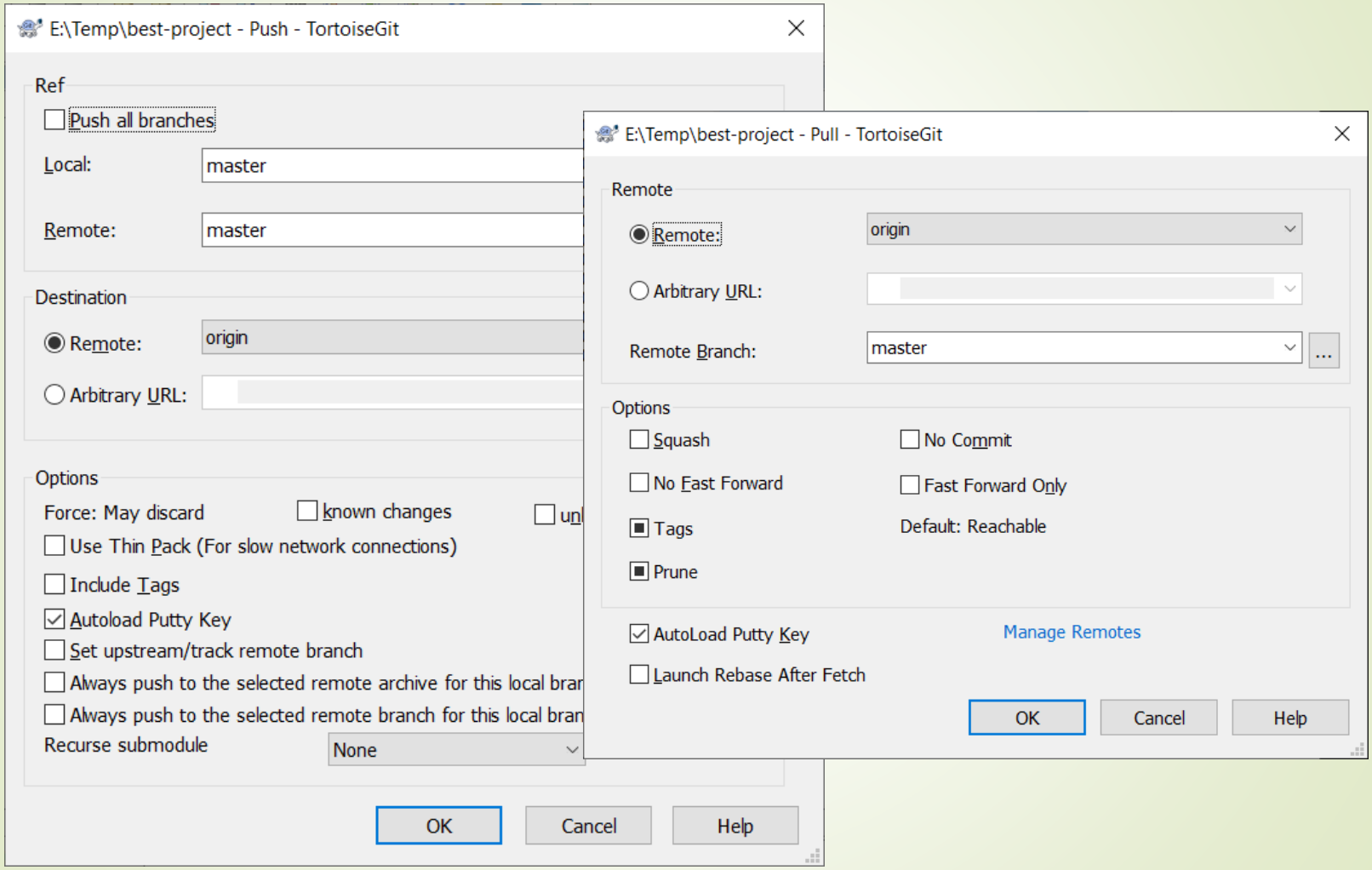
TortoiseGit: commit a new revision



Version control systems

ELTE | IK
INFORMATIKAI KAR

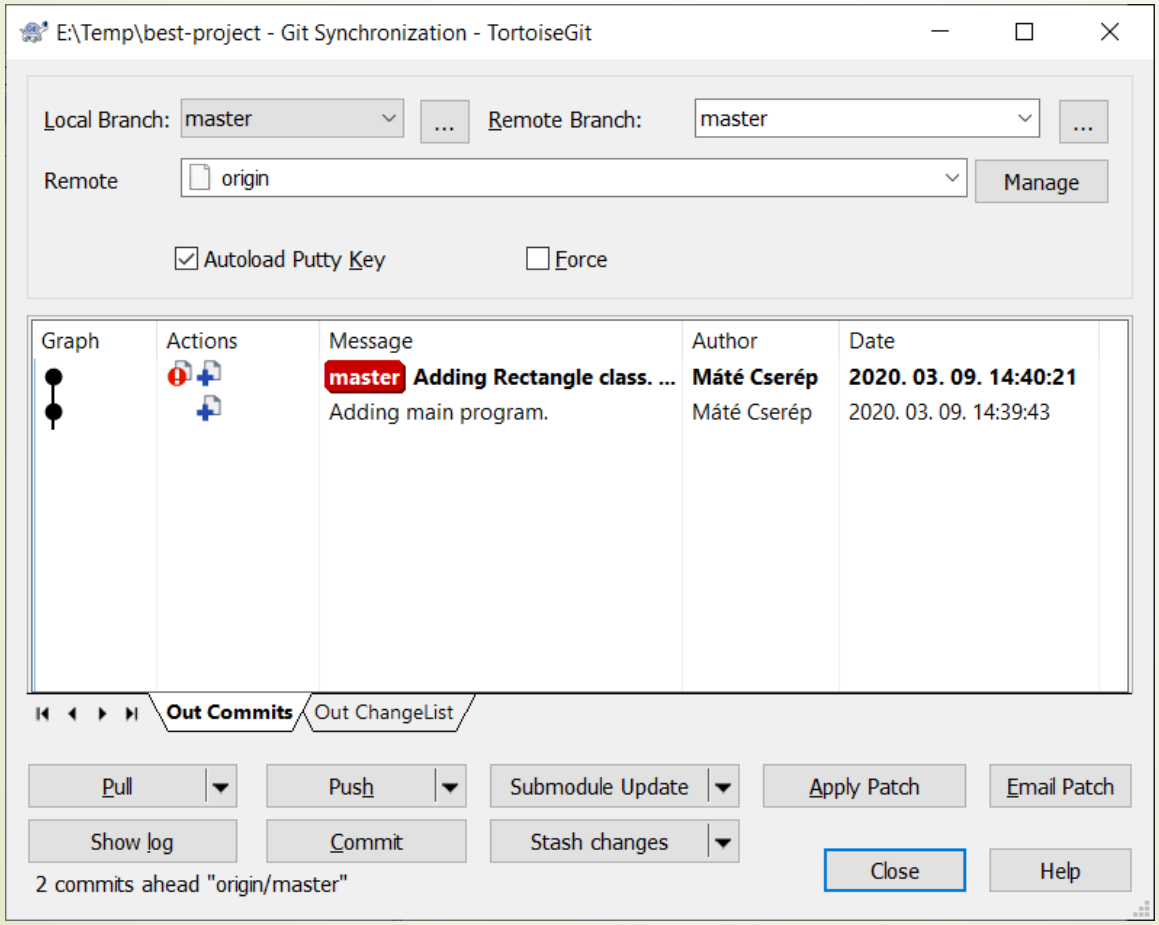
TortoiseGit: synchronize with remote (*push* & *pull*)



Version control systems

TortoiseGit: synchronize with remote (*sync*)

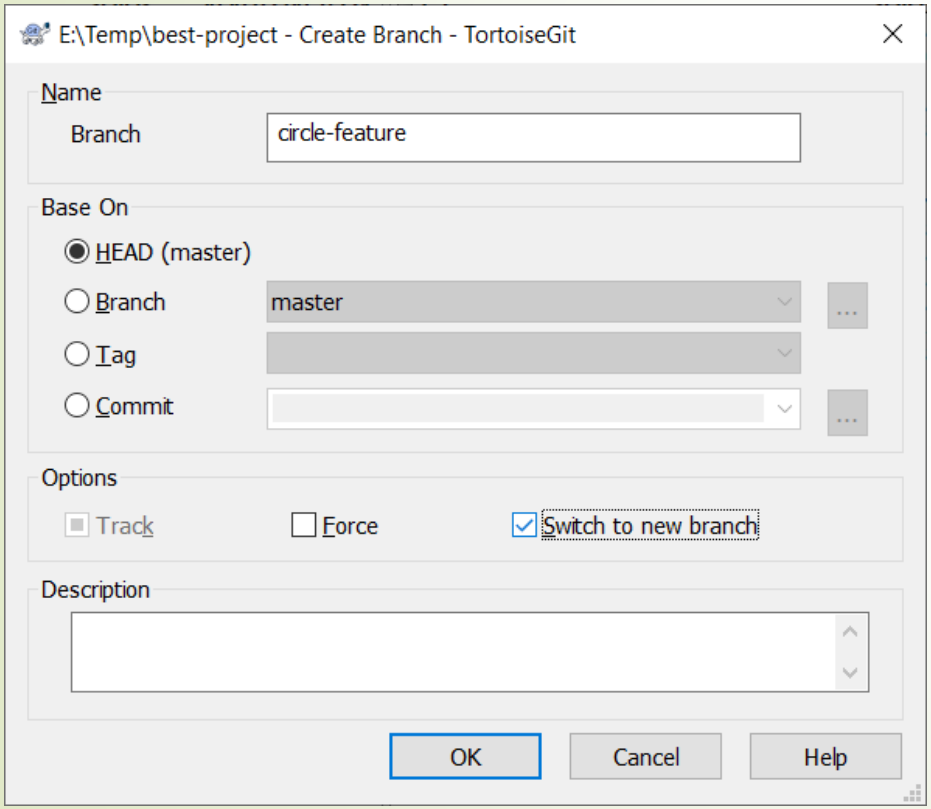
- The *Sync* option provides an overview interface where both the *pull* and the *push* functionalities can be accessed.

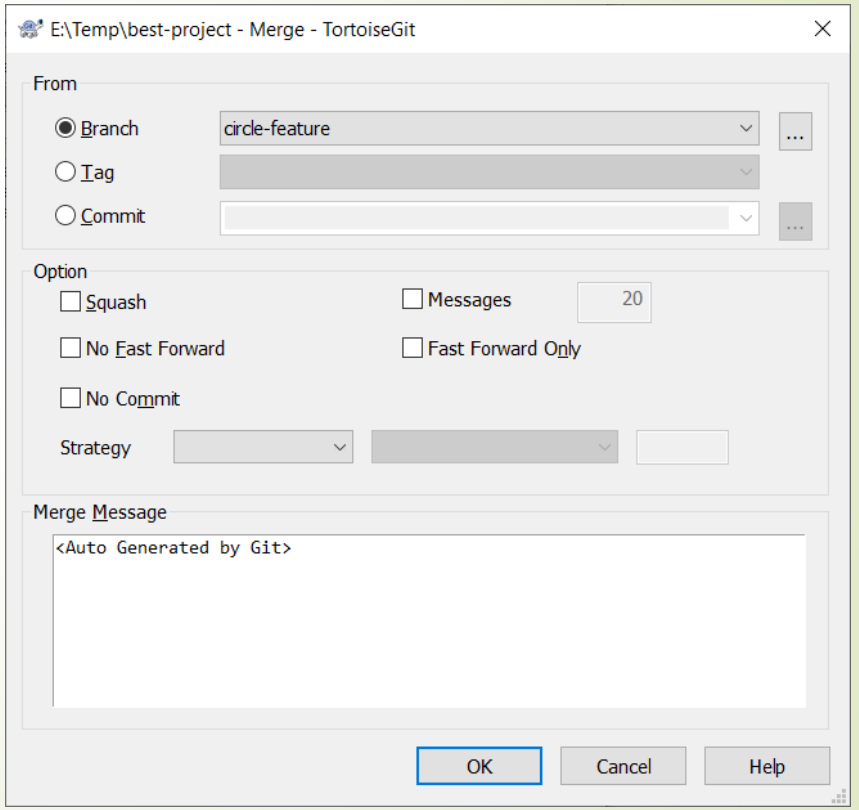


Version control systems

TortoiseGit: create and merge branches

- Beginner Git users often find creating and merging branches a complex task with the command-line tools. The graphical interface of TortoiseGit can ease this job.

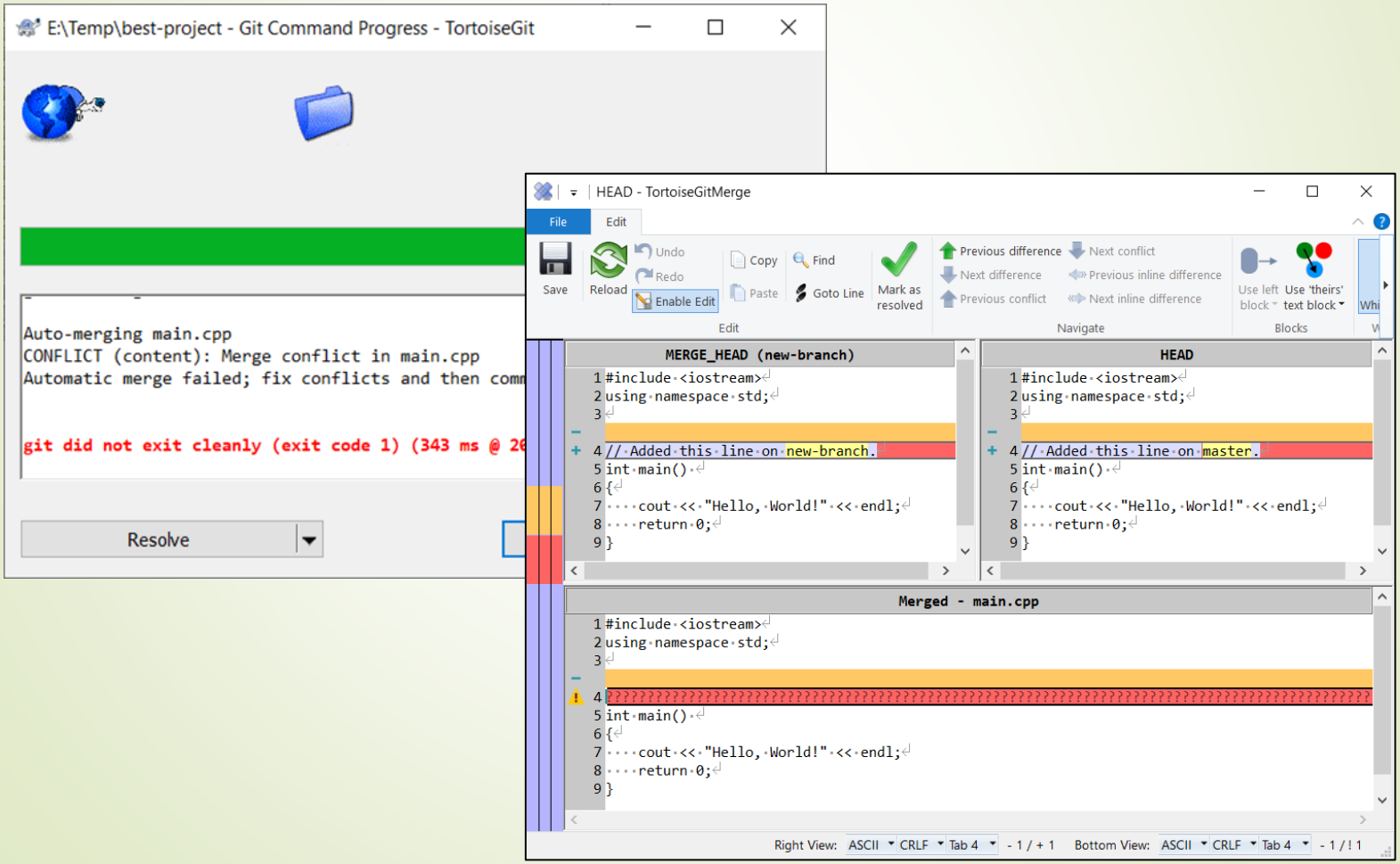




Version control systems

TortoiseGit: resolving merge conflicts

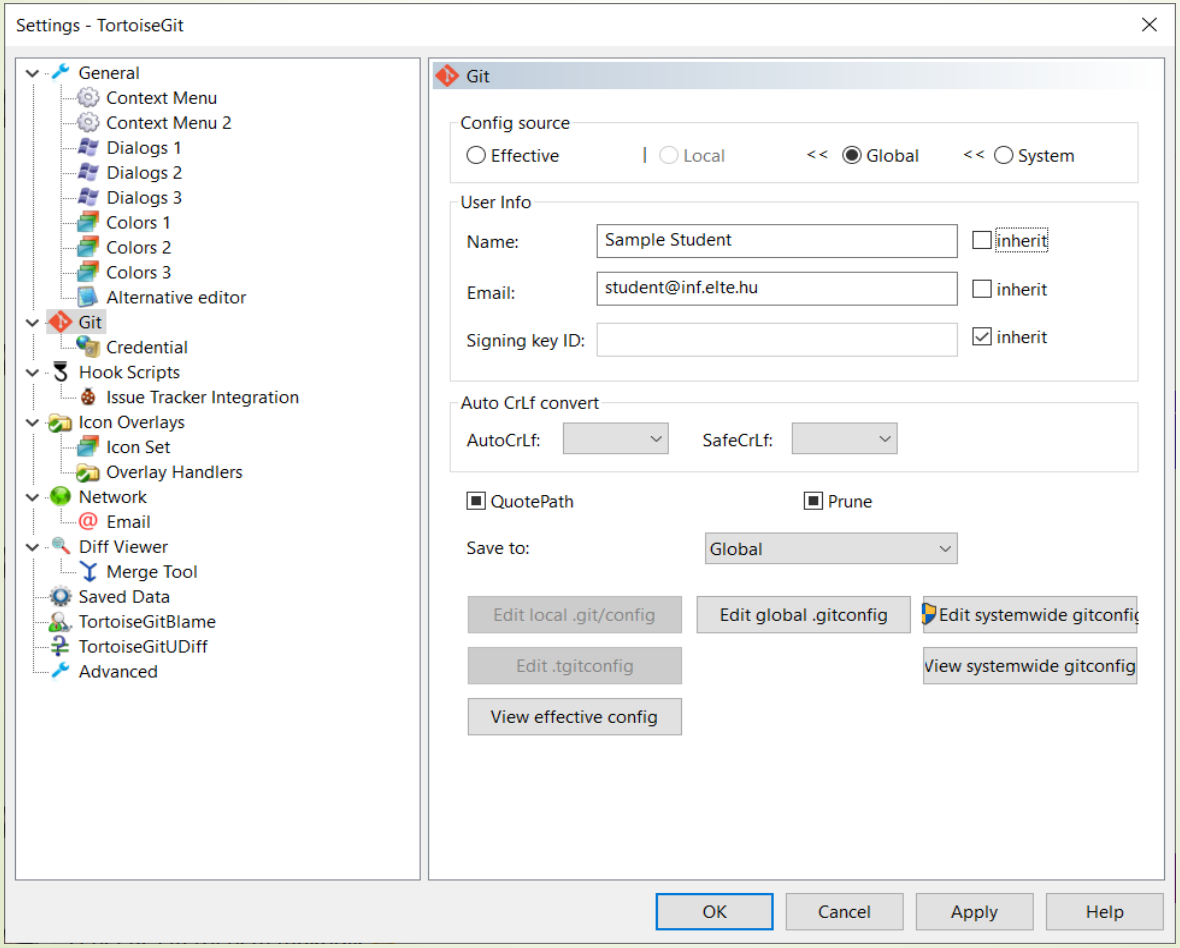
- This is especially true when resolving conflicts. TortoiseGit provides a side-by-side comparison view of the 2 file states to be merged.



Version control systems

TortoiseGit: settings

- In the *TortoiseGit -> Settings* option in the menu, we can modify the settings, e.g. the commiter's name and email address.

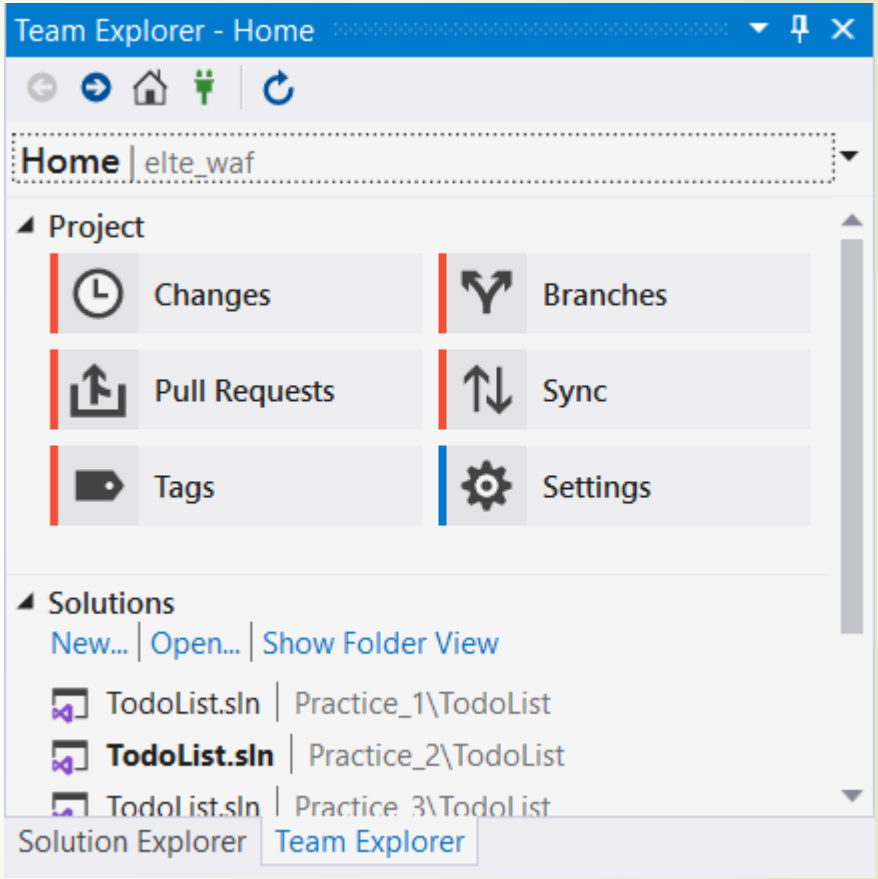


Version control systems

ELTE | IK
INFORMATIKAI KAR

Support in integrated development environments

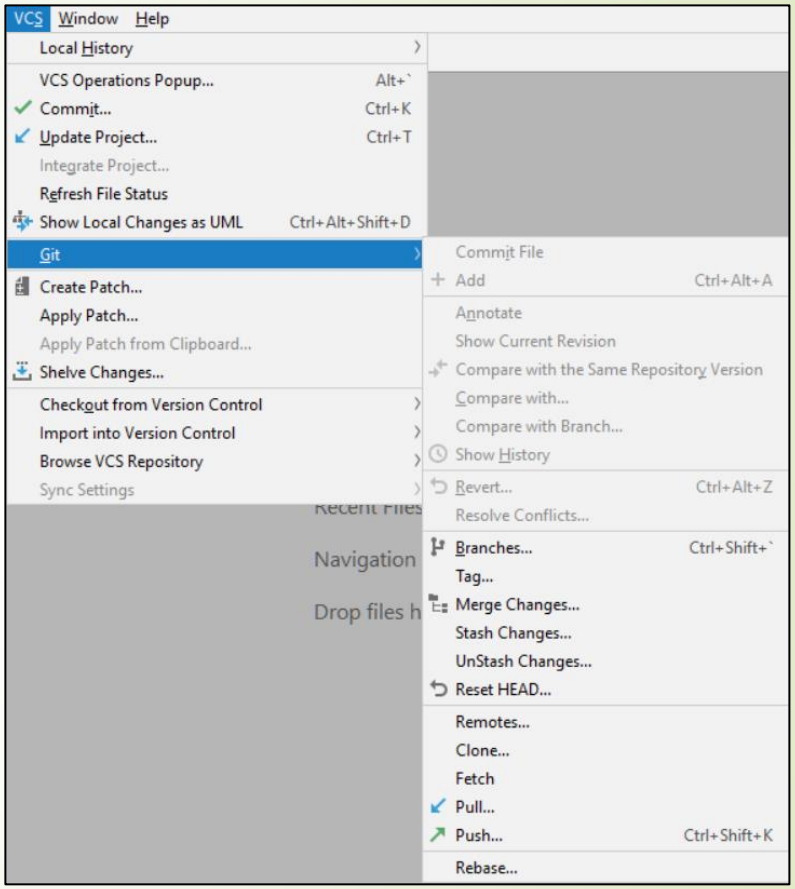
- Most modern integrated development environments (IDE) offers native solution for version control.
  - They are called *integrated* for a reason.
  - Therefore switching between multiple applications during development can be avoided.
- In *Visual Studio* the version control functionalities are located under the *Team Explorer* menu.

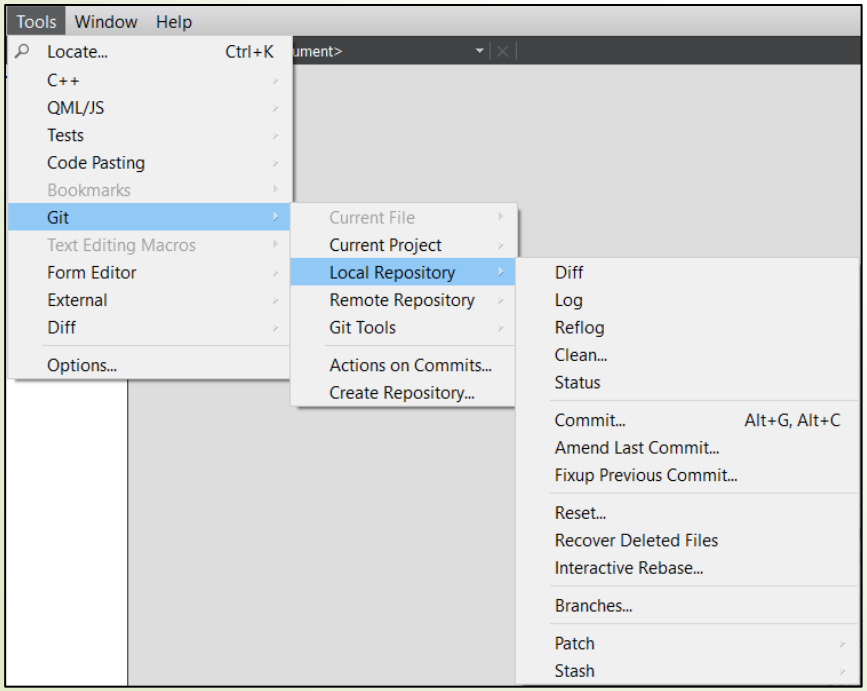


Version control systems

Support in integrated development environments

- In *IntelliJ CLion* these functionalities are locate under the VCS (*version control system*) menu.
- For *QtCreator* it is available in the *Tools -> Git* menu





Version control systems

Which files shall be version controlled?

- Git is efficient is version controlling text files, including code files. Therefore the purpose of the version control system is to track the changes of the source code of the software under development.
- For a general software project, it is NOT recommended to place under version control:
  - intermediate and final binary files of the compilation process, as they are reproducible from the source code and will cause merge conflicts regularly.
  - personal configuration of the development environment (e.g. the *.vs/* folder for Visual Studio or the *nbproject/private/* folder for NetBeans), as the differ among developers.
  - large binary files (e.g. videos, large images), as Git is not efficient version controlling them. Although the size of Git repositories scale well, the size of an easily manageable *repository* does not exceed 1-2 GB.

Excluding files from version control

- Only files explicitly added to version control (`git add`) are tracked, other files are excluded.
- To avoid tracking files accidentally, we can mark files and directories which must always be ignored from version control.
  - The rules can be defined in special `.gitignore` files.
  - These files shall be version controlled, so the configuration is shared among team members.
- Inside a `.gitignore` file, each line contains a pattern to match files and directories. The matched files are excluded from version control.
  - This configuration is applied transitively to subdirectories, so a single `.gitignore` file in the project root directory can often be sufficient.

Version control systems

Git: .gitignore pattern

Pattern	Matches	Description
program.exe	/program.exe /bin/program.exe	Matches all **program.exe** files.
/program.exe	/program.exe **but not:** /bin/program.exe	Matches the **program.exe** file at the given directory level.
*.exe	/program.exe /bin/main.exe	Matches all files with the **exe** extension.
bin/	/bin/ /project/bin/ **but not:** /logs/bin (file)	Matches all **bin** folders (but not the files with the name **bin**!)
/bin/*.exe	/bin/program.exe /bin/main.exe **but not:** /bin/program.dll /project/bin/program.exe	Matches all files with the **exe** extension in the **bin** folder at the given directory level.

Version control systems

Git: .gitignore pattern

Pattern	Matches	Description
*.log !important.log	/application.log /logs/application.log **but not:** /logs/important.log	Matches all files with the **log** extension, except when the file's name is **important.log**.
logs/**/*.log	/logs/application.log /logs/runtime/main/01.log /project/logs/deploy.log **but not:** /runtime.log	Matches all files with the **log** extension, which are inside a folder named **logs** (arbitrary folder depth allowed).

- For most programming languages an IDEs, a general purpose, sample `.gitignore` file is available on *GitHub*. It is a good idea to copy such a sample file or even combine multiple of them.
  - URL: https://github.com/github/gitignore

Version control systems

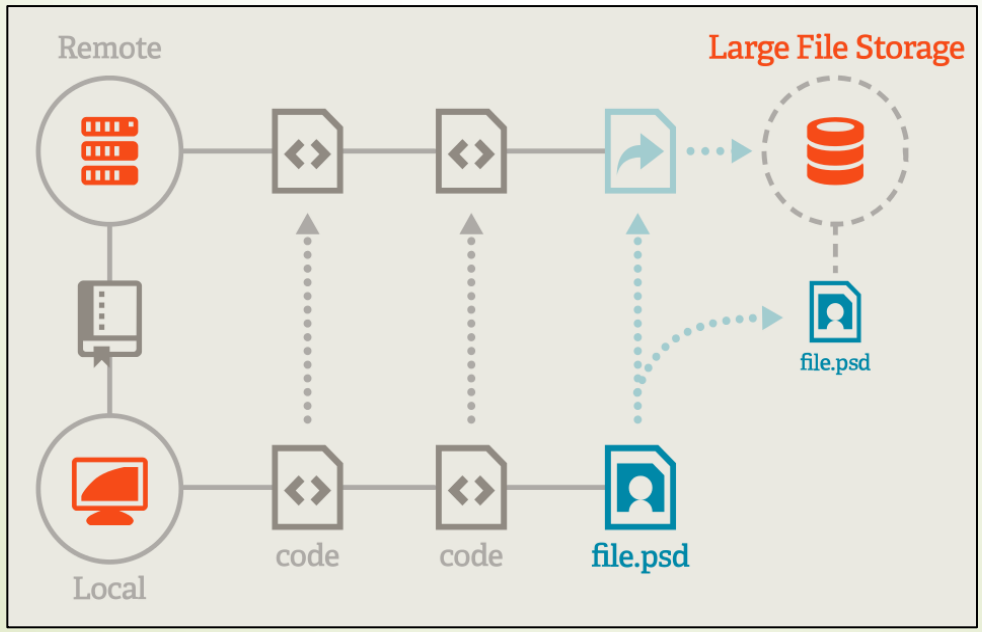
Version control of large binary resources

- Version control of large video, image and audio resource files requires extra caution.
  - Git is not efficient in tracking changes of large binary files.
  - It significantly increases the network traffic required to download a local repository (*git clone*).
  - Developers do not necessarily need the *assets* created by designers during development and testing time.
- Therefore even in the case the large binary resource files in a project change rarely, usually it does not worth to track them in a version control system like Git.

Version control systems

Git Large File Storage

- When required, the version control of large binary files can be done with the *Git Large File Storage* (Git LFS) system.
  - It replaces large binary files with a reference and the content of the files themselves are not stored in the Git repository, but on a different (remote) server.
  - Thus the size of our Git repository remains small.



Source: git-lfs.github.com

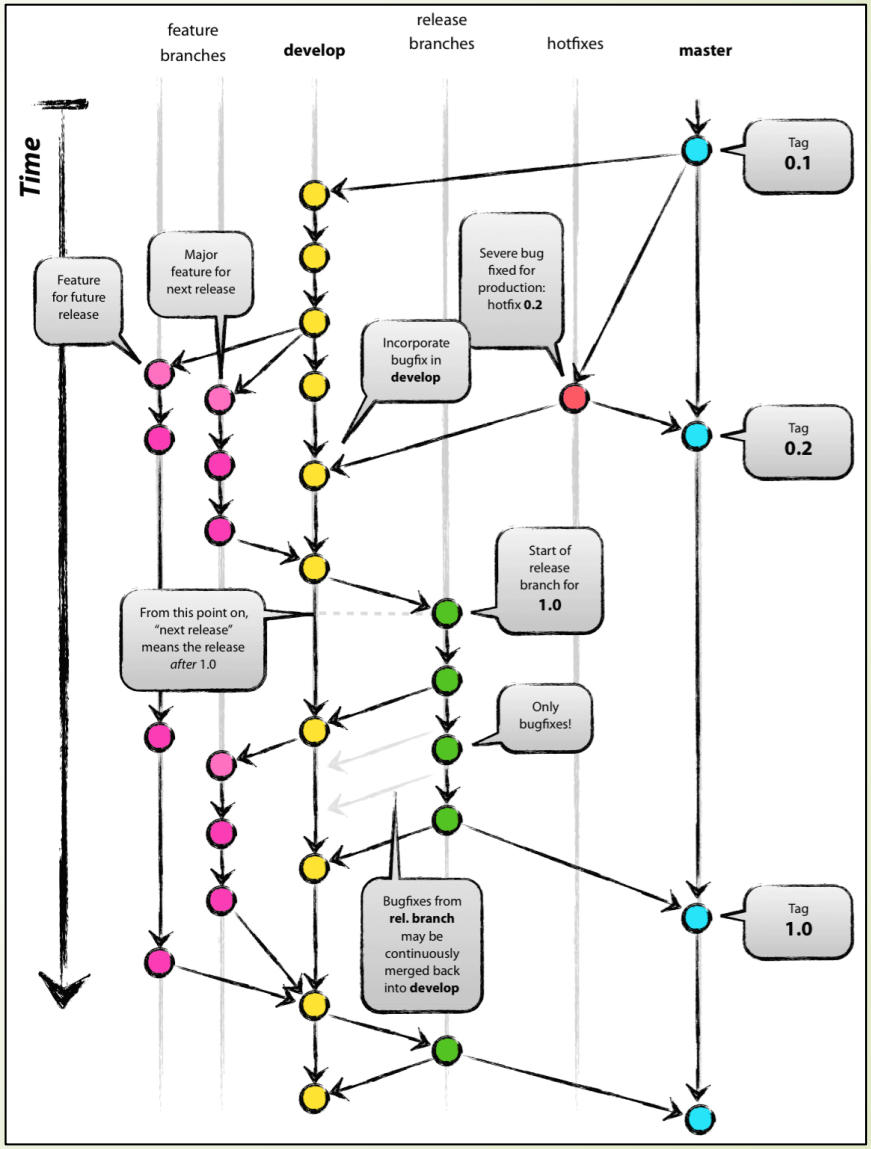
Git Large File Storage

- The GitLab instance on szofttech.inf.elte.hu supports Git LFS.
- To utilize it, Git LFS must be installed on the client computers (the computers you use for development).
  - Download: https://git-lfs.github.com/
  - Usage: https://docs.gitlab.com/ee/administration/lfs/manage_large_binaries_with_git_lfs.html

Version control systems

Gitflow feature branching model

- Preferred for large software with multiple supported releases
- Main branches:
  - master
  - develop
- Supporting branches:
  - feature branches
  - release branches
  - hotfix branches



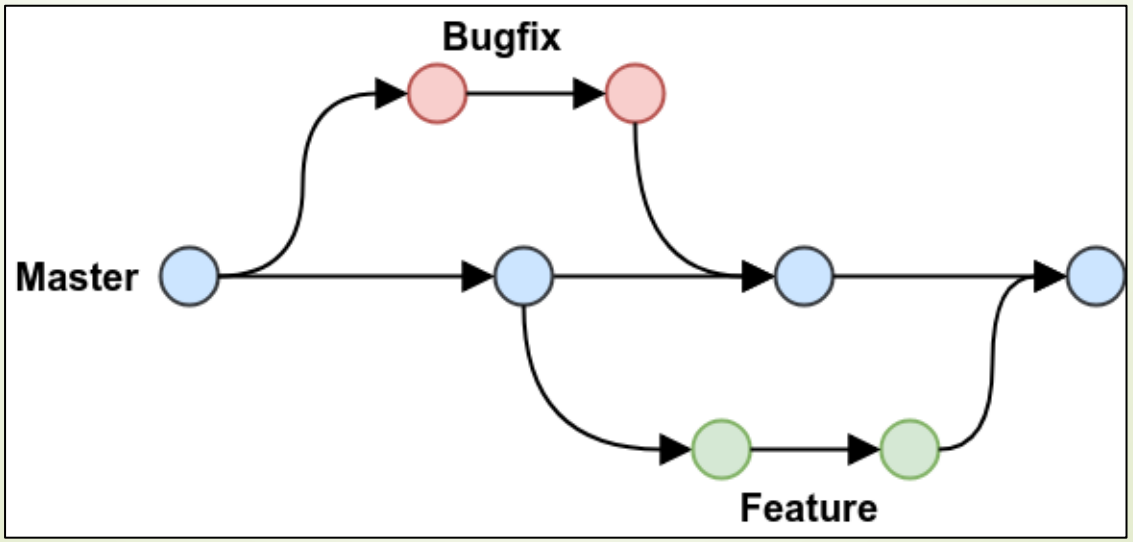
Source: nvie.com

Version control systems

GitHub feature branching model

- Preferred for software with a single supported release and a fast cycle to roll out new releases.
  - What is on the main branch, can be released!

- Main branch: *master*
- Supporting branches: *feature* and *bugfix* branches

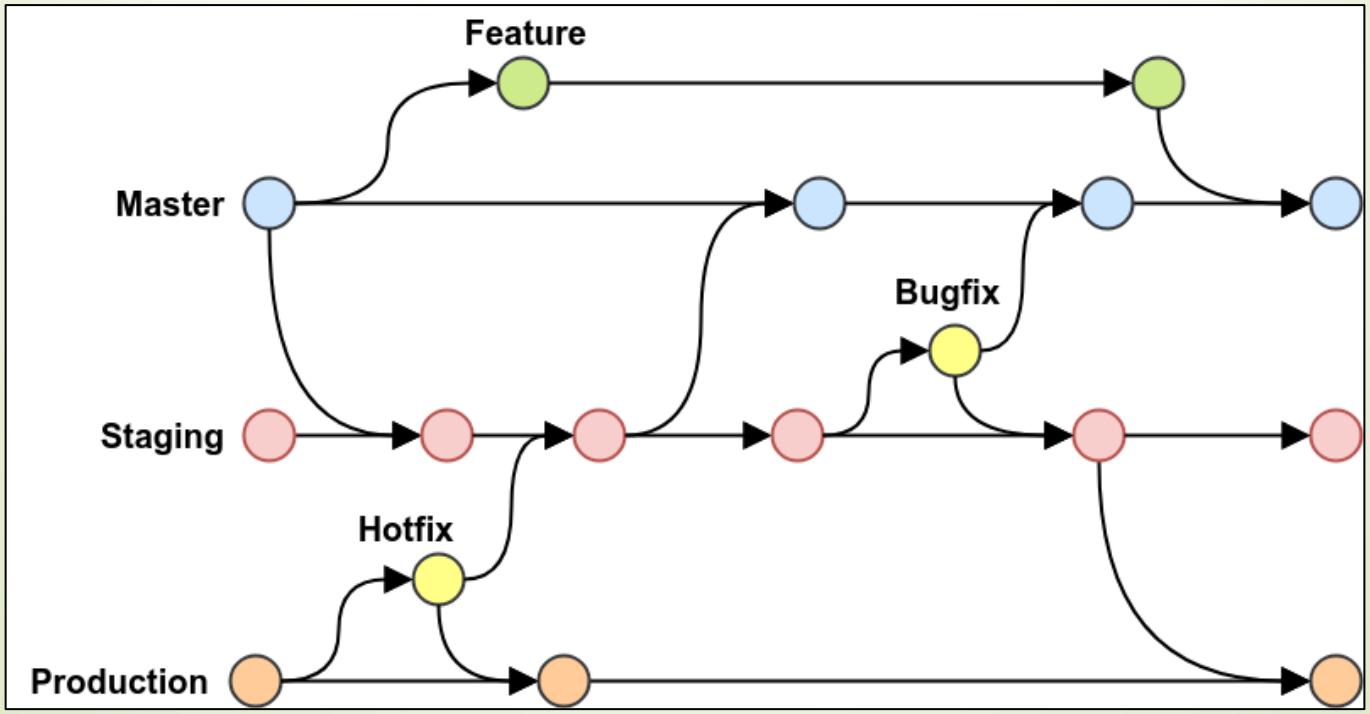


Source: blog.programster.org

Version control systems

GitLab Flow feature branching model

- It may not always be possible to release a new version of the software on the main development branch.
  - Main development branch: *master*
  - Production release branch: *production*
  - Staging (testing) release branch: *staging* (optional)



Source: blog.programster.org

Software Technology

Version control systems

Specialization C

Dr. Máté Cserép

ELTE, Faculty of Informatics

2025.

Version control systems

Historical background

- The exponential growth in size and complexity resulted in a *software crisis* in the 1970s and lead to an increment in the:
  - size of program source code,
  - time required to complete software projects,
  - required human resource (developers).
- As the software industry advanced into a new era, more and more applications were developed
  - the life-cycle of software projects usually did not end with the release of the first public version,
  - maintenance and further development phases followed.

- **Software projects increased both in size, complexity, development time and participating programmers.**

Version control systems

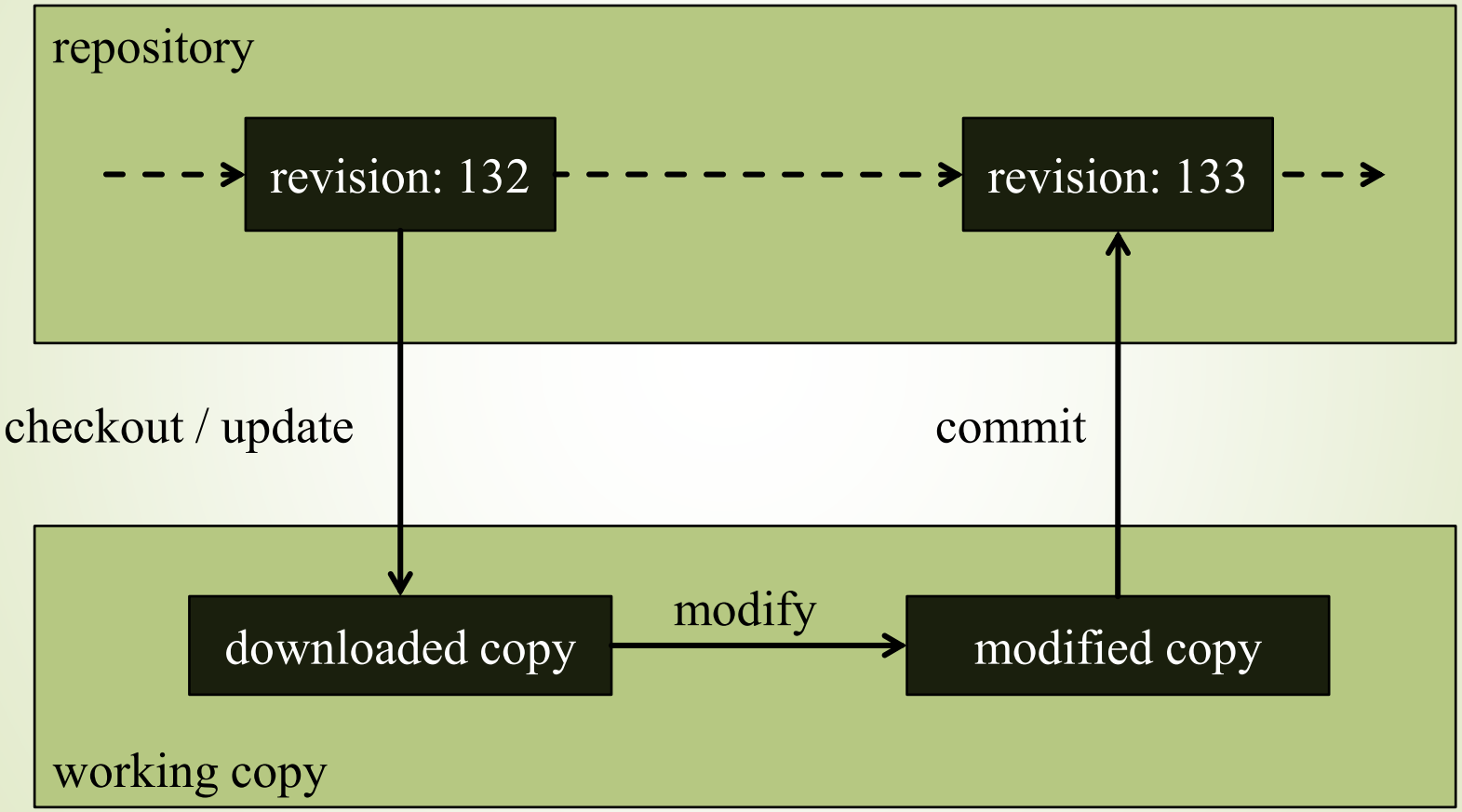
Functionality

- As implementation is carried out in multiple phases, often by multiple developers, it is necessary for the subsequent and parallel states of the program to be easily trackable. This task is accomplished through the *revision control systems*
  - e.g. *CVS*, *Apache Subversion* (*SVN*), *Mercurial*, *Git*
  - the source code is stored in a shared, central *repository*
  - from which the programmers may create their local *working copy* for development
  - the modifications are uploaded back to the central repository (*commit*)
  - after the initial creation of the working copies (*checkout*), they must be continuously *updated*

Version control systems

ELTE | IK
INFORMATIKAI KAR

Functionality



Version control systems

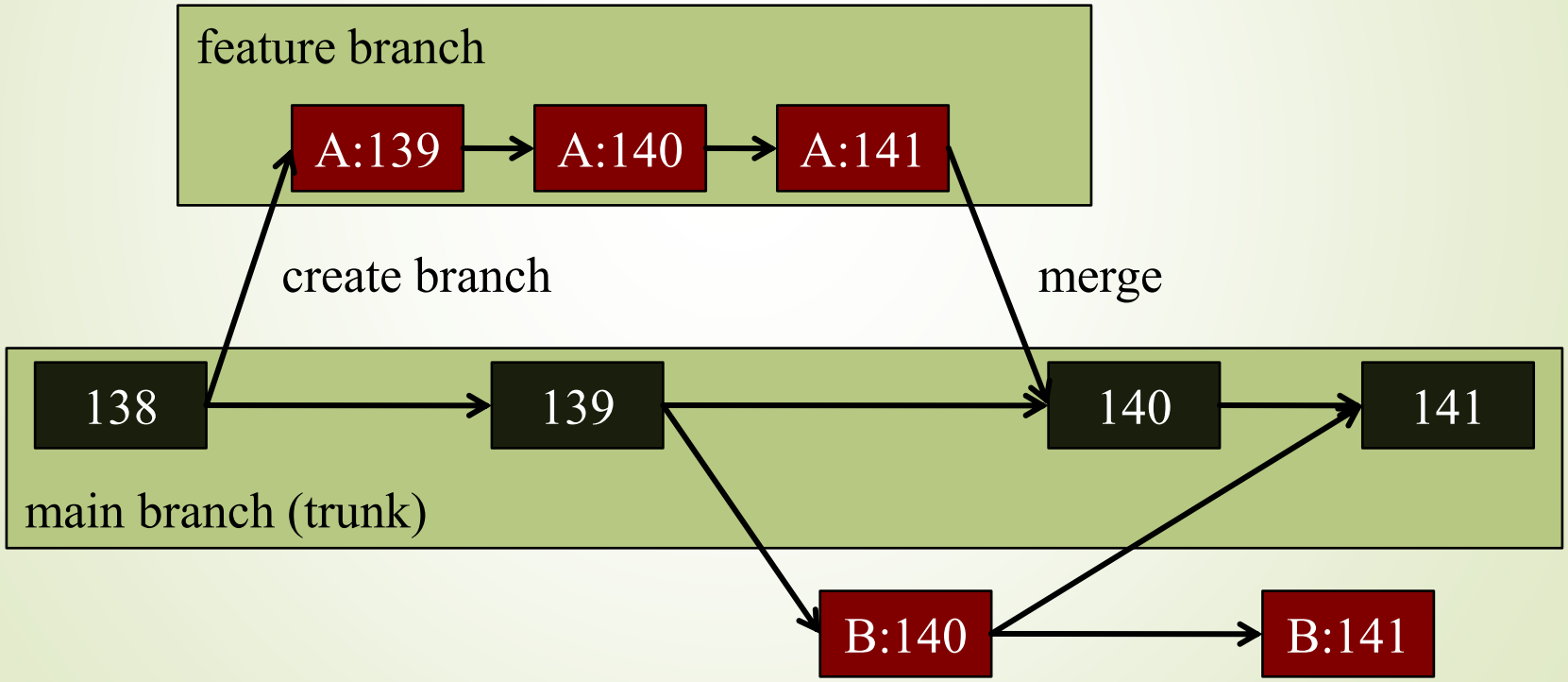
Functionality

- Revision control systems enables us to:
  - store all previous *revisions* and to make a local working copy from them
  - access the main line of development (usually named *baseline*, *master* or *trunk*), the most recent version (*head*), and to upload new revisions with documentation
  - recording the changes between specific revisions
  - undoing changes, *restoring* previous revisions
  - checking possible source code conflicts resulting from modifications and *resolving* them

Version control systems

Functionality

- branching new lines of development (*branch*), which run alongside the main line, and can be merged with each other or back to the main line of development



Version control systems

Functionality

- merging often requires additional manual interaction and corrections
- the changed code parts can usually be merged automatically through code analysis, which can be *two-way*, using only the 2 states of the modified file, or *three-way*, analyzing the state of their common ancestor
- *exclusively lock* code parts to avoid conflicts
- creating a *snapshot* from a specific revision and marking it with a *tag* (e.g. for easy or public access)
- treating commits as atomic operations (e.g. to handle network failures during upload)

Version control systems

Local version control system (1st generation)

- Tracking the changes of source code, managing the combination of features added to different releases
  - local repository (though can be accessed by multiple developers when deployed on e.g. a *mainframe*)
  - file based operations (1 revision contains changes for 1 file)
  - concurrency control through exclusive locks

- The theoretical foundations defined in the 1970s:
  - Source Code Control System (SCCS) – 1972
  - Revision Control System (RCS) - 1982

Version control systems

Centralized version control system (2nd generation)

- Parallel development by multiple programmers
  - centralized model with a client-server architecture
  - fileset based operations (1 revision contains changes for a set of files)
  - concurrency control usually by *merging before committing*
- Became widespread in the 1990s:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
  - SourceSafe, Perforce, Team Foundation Server, etc.
- *Drawback: the central role of the server (e.g. failure), furthermore version control operations requires network connection, hindering offline work*

Version control systems

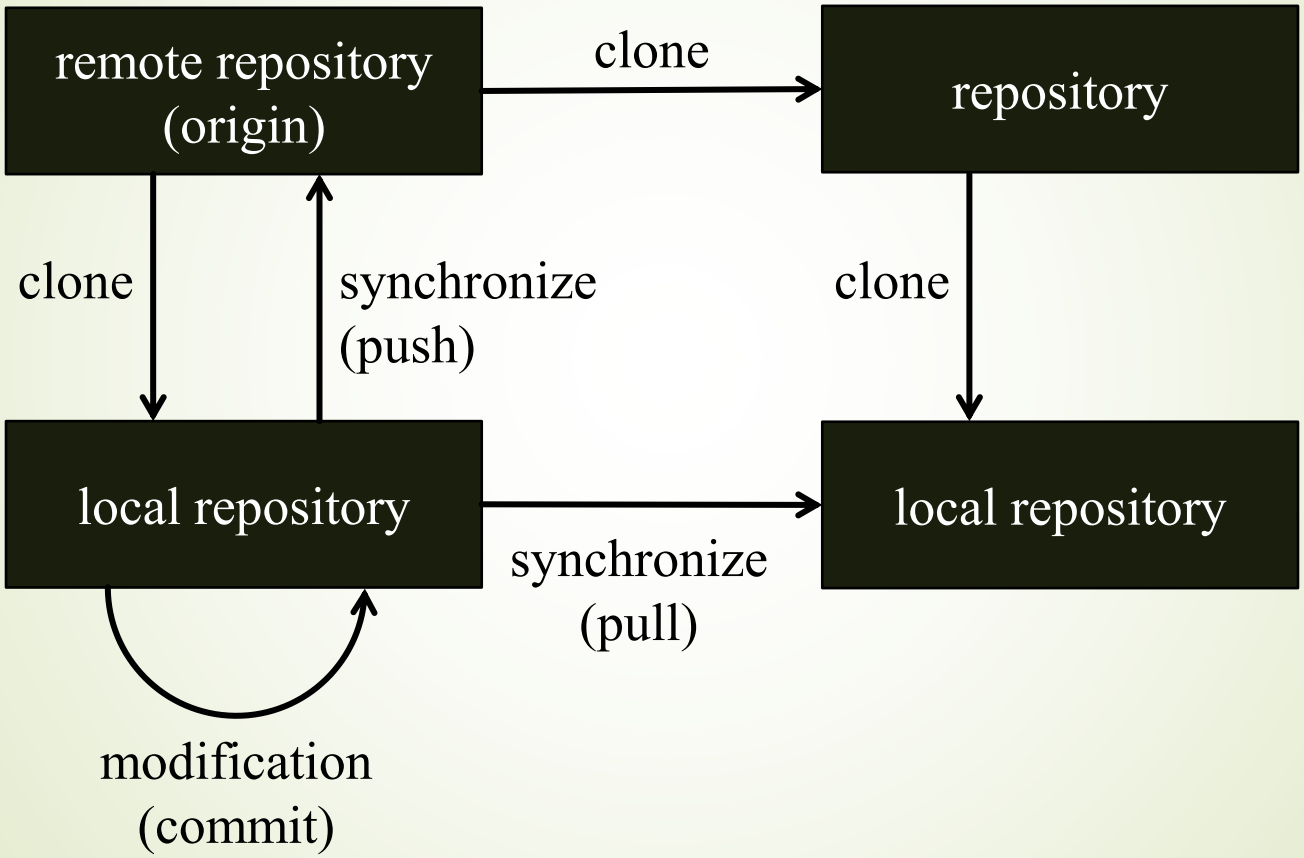
Distributed version control system (3rd generation)

- Network communication is decoupled from the classical version control operations and become separate actions which can be initiated by the user
  - decentralized, distributed network model
  - each client has a copy of the complete codebase, including its full version history
  - version control operations are executed on the local repository of the client
  - synchronization is done based on a *peer-to-peer* approach, but well-known servers can be established to boost efficiency
  - concurrency control usually by *committing before merging*
- Introduced in the early 2000s:
  - Monotone, Darcs, Git, Mercurial, Bazaar, etc.

Version control systems

ELTE | IK
INFORMATIKAI KAR

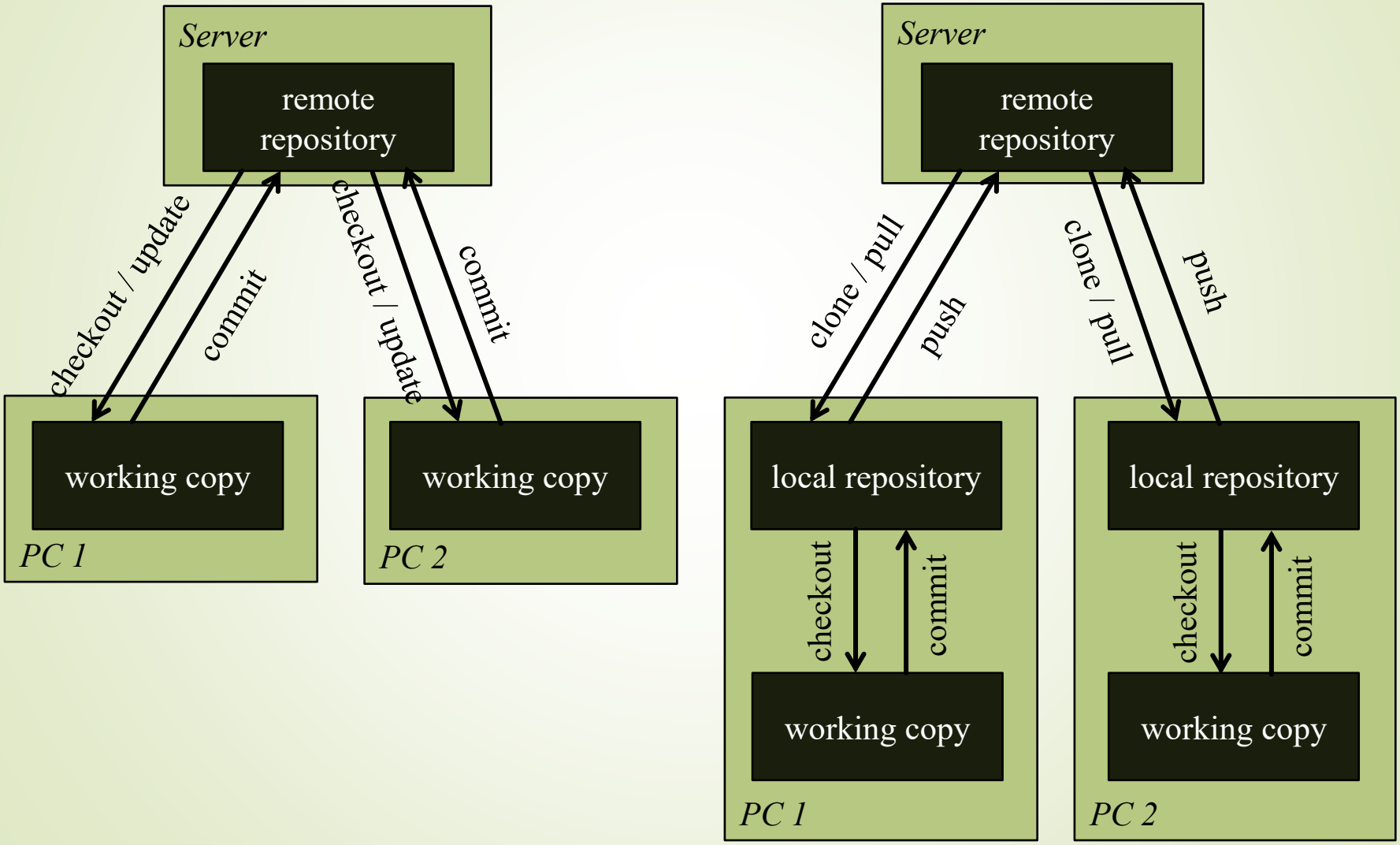
Distributed version control system (3rd generation)



Version control systems

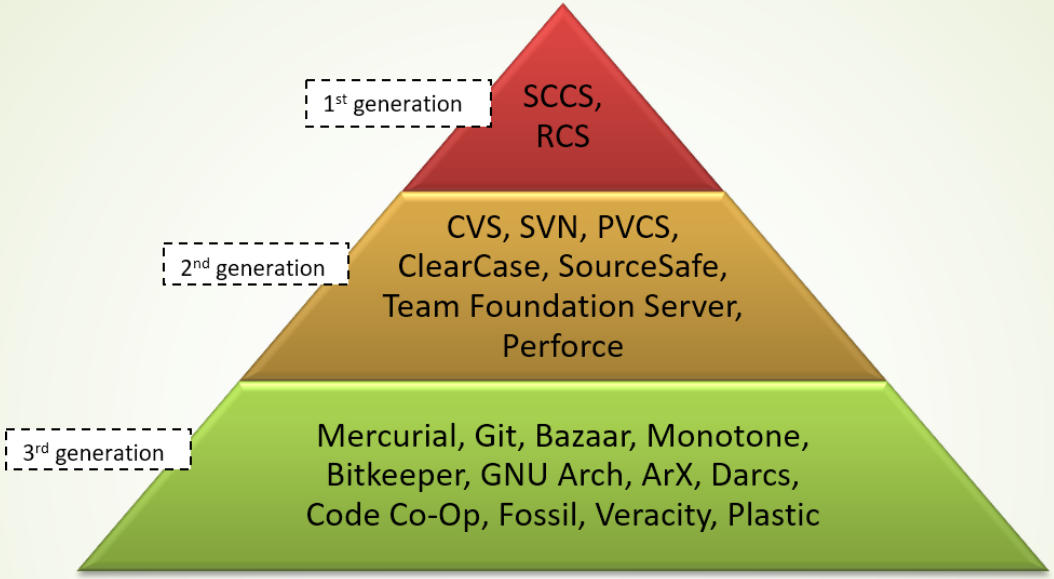
ELTE | IK
INFORMATIKAI KAR

Centralized and distributed version control systems



Version control systems

Generation model



Generation	Network model	Operation atomicity	Concurrence handling
First	Local	Single files (non-atomic commits)	Exclusive locks
Second	Centralized	Fileset (atomic commits)	Merge before commit
Third	Distributed	Fileset (atomic commits)	Commit before merge

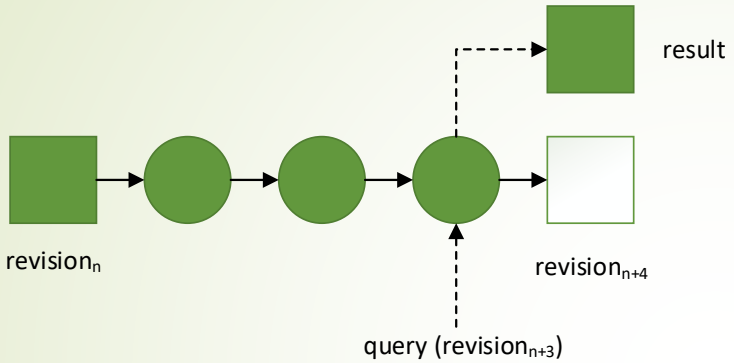
Version control systems

Changeset representation

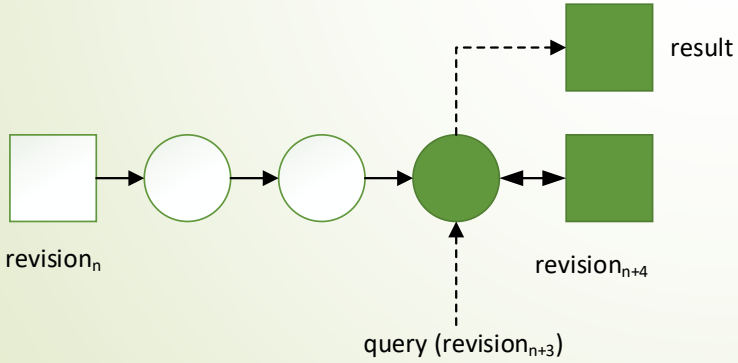
- Storing the full state of all revisions in the repository would waste significant storage space and raise processing costs
- Hence previous (1st and 2nd generation) revision control systems only store the difference (*changeset*, *delta*) between any two subsequent revisions
  - some systems (e.g. Mercurial) may create a *snapshot* from the full content periodically
- Initially (SCCS) *forward deltas* were utilized to create the versions from the previous ones
- It was early realized (RCS), that using *reverse deltas* and storing the full state of the newest revision can provide better performance, since recent revisions of a branch are checked out more frequently compared to older ones
  - Mixed solution: using reverse deltas on the main line and forward deltas on the side branches

Version control systems

Changeset representation



Forward deltas

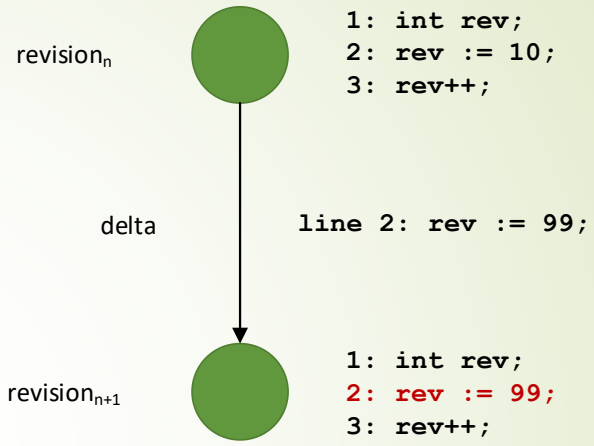


Reverse deltas

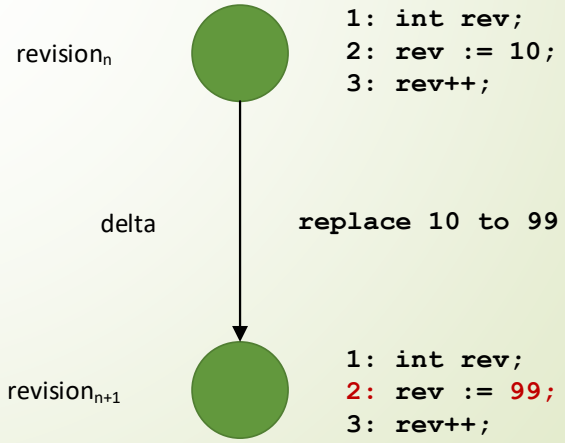
Version control systems

Changeset representation

- Determining changes in textual documents, like source code files is usually done on a *state-based* approach
  - in most cases comparing content line-by-line
    - e.g. GNU diff
  - for structured content the unit of comparison might be something different (e.g. XML, JSON, UML)
- For binary data (e.g. images) *operation-based* approaches can also be used.



State-based

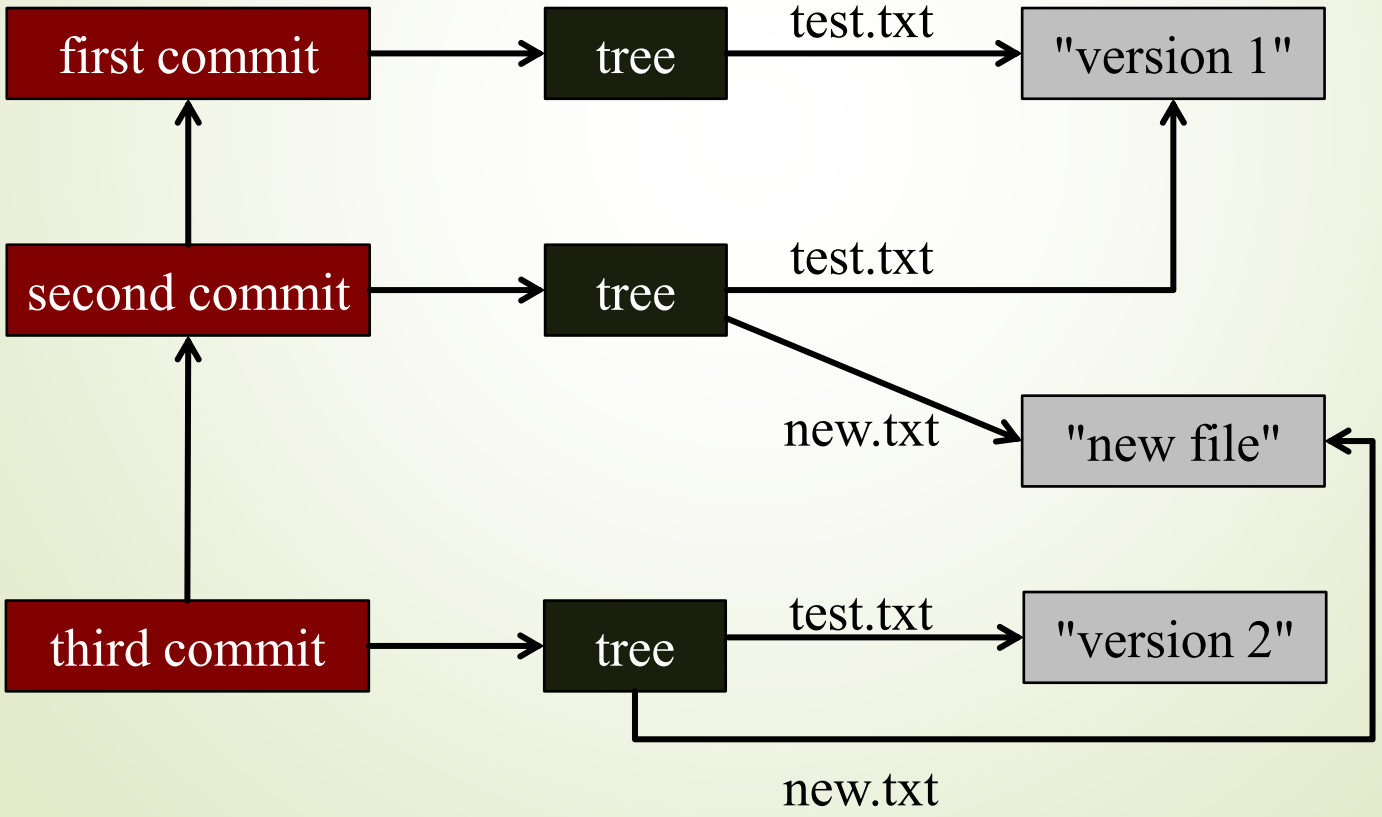


Operation-based

Version control systems

Changeset representation

- By the time of 3rd generation version control systems, storage capacity significantly increased, while their cost was reduced
  - Newer version control tools (like Git) store the entire content of modified files, not only the changed parts (*delta*)



Version control systems

Git

- During the seminar classes, student project will be version controlled with the Git version control system, which is tightly integrated into the faculty GitLab instance.
  - Faculty GitLab server: https://szofttech.inf.elte.hu/
- The *remote repository* available on the GitLab server can be browsed and viewed on the web user interface of GitLab. Basic edit operations are also possible (these will also be *commits*).
- Version control of the source code is usually performed on a *local repository* with a client application. The local repository is regularly synchronized with the remote. We can use:
  - the command-line commands; or
  - a graphical desktop client application.

Git: installation

- The Git version control system can be easily installed:
  - Windows, Mac installer: https://git-scm.com/downloads
  - Debian/Ubuntu: `apt-get install git`
  - Other UNIX systems: https://git-scm.com/download/linux
- After installation we can immediately start to work with the command-line instructions.
  - Therefore, on a Windows system, it is recommended to add the Git binaries to the PATH environment variable.
  - Graphical client applications might be installed separately.
- For each Git commit the name and email address of the author (the *committer* to be more precise) must be assigned. The values shall be configured beforehand globally.

```
git config --global user.name "Sample Student"
git config --global user.email student@inf.elte.hu
```

Git: creating a repository

- A new empty repository can be initialized:

```
git init
```

- Or an existing remote repository can be cloned:

```
git clone https://mysite.com/best-project.git
```

- We usually need to clone remote repositories.
  - The remote repository, which was cloned will get the alias name *origin* in our local repository (by default), This can be used later, e.g. on synchronization.

Version control systems

Git: tracking files

- We can start tracking the changes of files with the `git add` command, adding them to the *staging area*:

```
git add Main.java
```

  - Not only single files, but patterns (e.g. `*.java`) or complete directories can be added.

- The current state of our working copy can be checked with the `git status` command anytime.

```
git status
> On branch master
> Your branch is up to date with 'origin/master'.
> Changes to be committed:
>   (use "git reset HEAD <file>..." to unstage)
>         new file:   Main.java
```

Git: creating a new revision

- Once we have finished added the needed changes to the *staging area*, we can commit it as a new revision to the local repository with the `git commit` command:

```
git commit -m "Added main program."
> [master d26c7a9] Added main program.
>  1 file changed, 1 insertion(+)
>  create mode 100644 Main.java
```

Version control systems

Git: tracking further changes

- We can add new files to the *staging area*:

```
git add Rectangle.java

git commit -m "Added Rectangle class."
```

- A new revision can contain new files and the modification of already version controlled files:

```
git add Circle.java

git add Main.java

git commit -m "Added Circle class.
               Modified the main program."
```

Git: synchronization with a remote repository

- Mew revisions created in our local repository have to be synchronized with the originally cloned remote repository from time to time, so our modifications will be accessible to our teammates. This can be done with the `git push` command.

```
git push origin master
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 247 bytes | 123.00 KiB/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To /path/to/workspace/folder
   d45172c..80a39a2  master -> master
```

  - We define which remote repository we would like to synchronize with and which branch. We can use the *origin* alias name for the cloned remote repository. The branch is the *master* in the above example.
  - When using *tracking branches* the name of the remote and the branch can be omitted, simplifying the command to `git push`.
- New branches and revision committed by other developers can be downloaded with the `git pull` command into out local repository.

Git: creating branches

- New branches can be created with the `git branch` command.

```
git branch new-branch
```

  - This new branch will diverge from the currently checked out revision in our working copy.
- We can switch between branches with the `git checkout` command.

```
git checkout new-branch
```

  - The default branch is usually named *master*.
- A new branch can be created and checked out in a single step:

```
git checkout -b new-branch
```

Git: merging branches

- After the completion of the purpose of a branch (e.g. developing a new feature), the changes should be merged back to the main development branch (master).
  - We can merge branches with the `git merge` command.
  - First we need to check out the master branch:

```
git checkout master
```

  - Then we can merge the modifications of the side branch:

```
git merge new-branch
```

- In case the same files and the same parts of them were modified on both branches, merging can usually cannot be done automatically. This is called a *merge conflict*.

Git: resolving merge conflicts

- In such case the conflict must be resolved manually, by editing the affected files.
- Conflicting code segments are marked with a special syntax by Git inside the source files themselves:

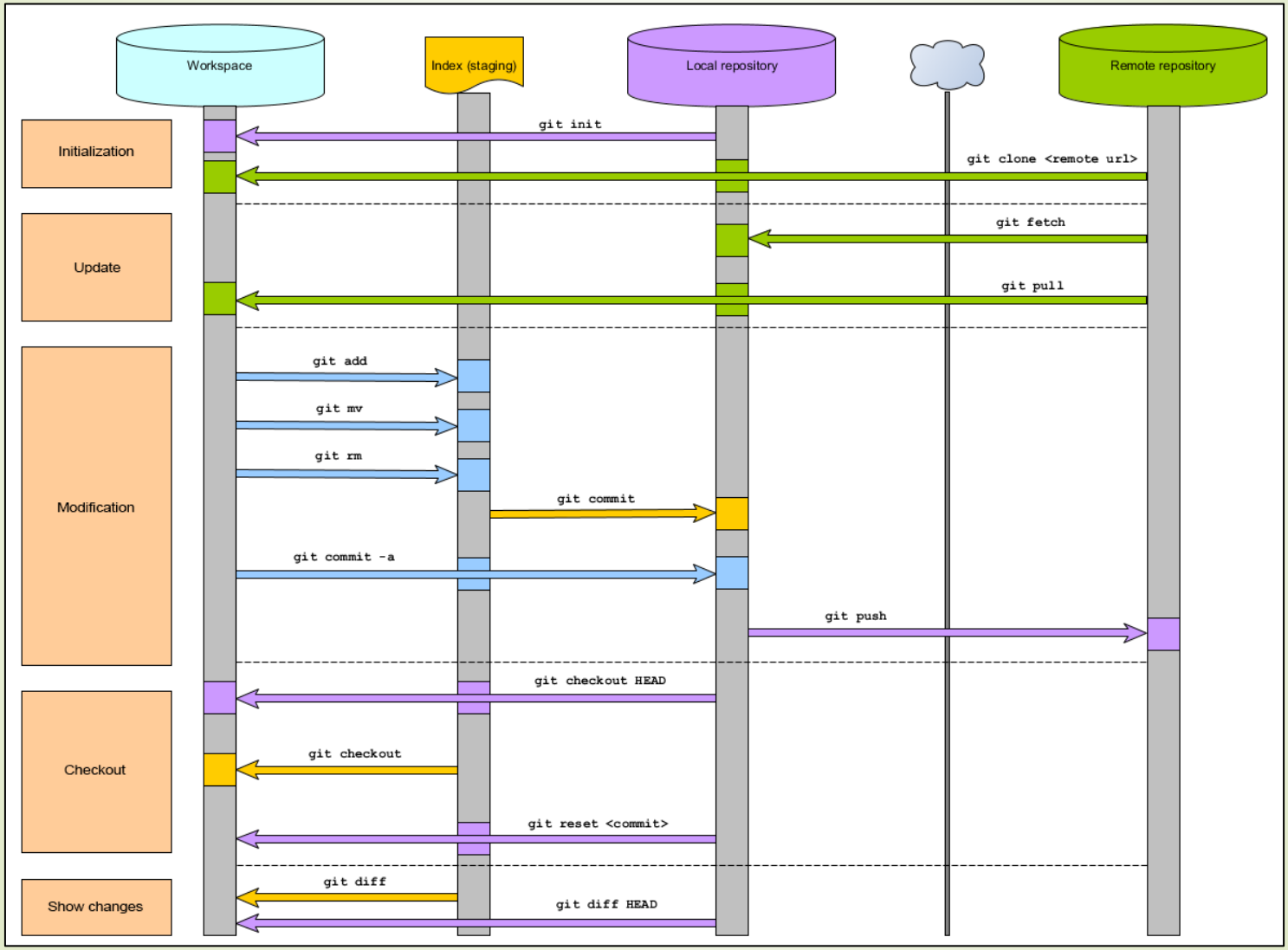
```
<<<<<<< HEAD
The content on the current branch, which is the master
in this example
=======
The content on the branch to be merged, which is the
new-branch in this example
>>>>>>> new-branch
```

  - It is the task of the developer to decide which version to use from the two; or define a new, third version as a resolution.
  - The resolved files must be added to the *staging area* (`git add`), then the modifications have to be committed (`git commit`).

Version control systems

ELTE | IK
INFORMATIKAI KAR

Git: overview of basic command line instructions

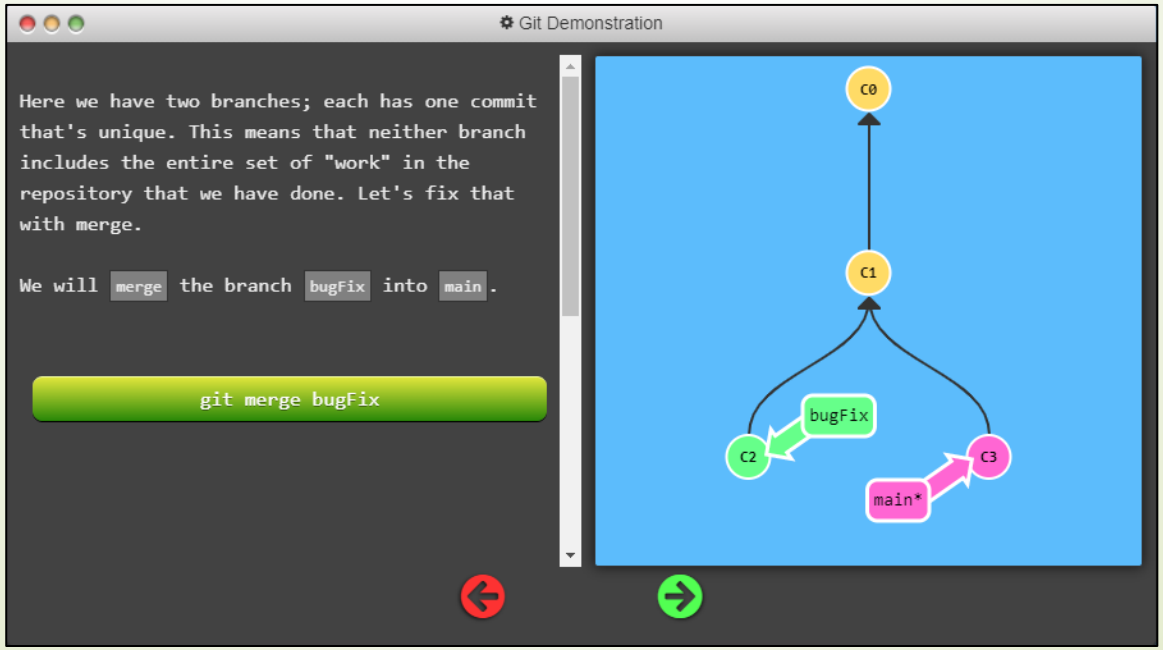


Source: Krisztián Nagy (ELTE FI)

Version control systems

Online tools to learn using Git

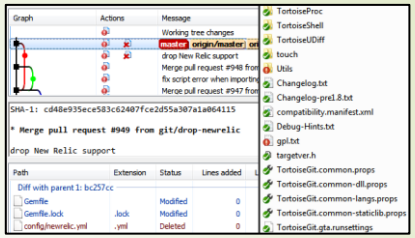
- Multiple online resources support understanding Git through visualizing how the Git history changes for the executed commands. The following tools are recommended:
  - https://git-school.github.io/visualizing-git/
  - https://learngitbranching.js.org/

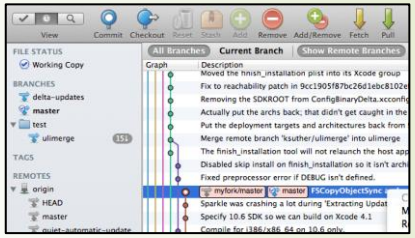


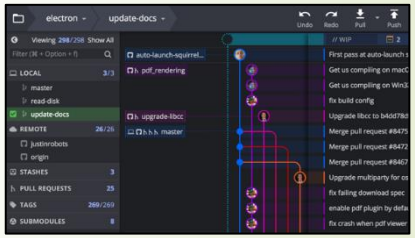
Version control systems

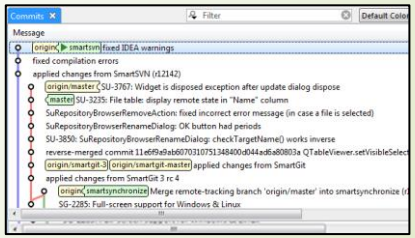
Git GUI clients

- TortoiseGit
  - Windows
  - *available in computer labs*
- SourceTree
  - Windows, Mac
- GitKraken
  - Linux, Windows, Mac
- SmartGit
  - Linux, Windows, Mac
- Further:
  - https://git-scm.com/downloads/guis





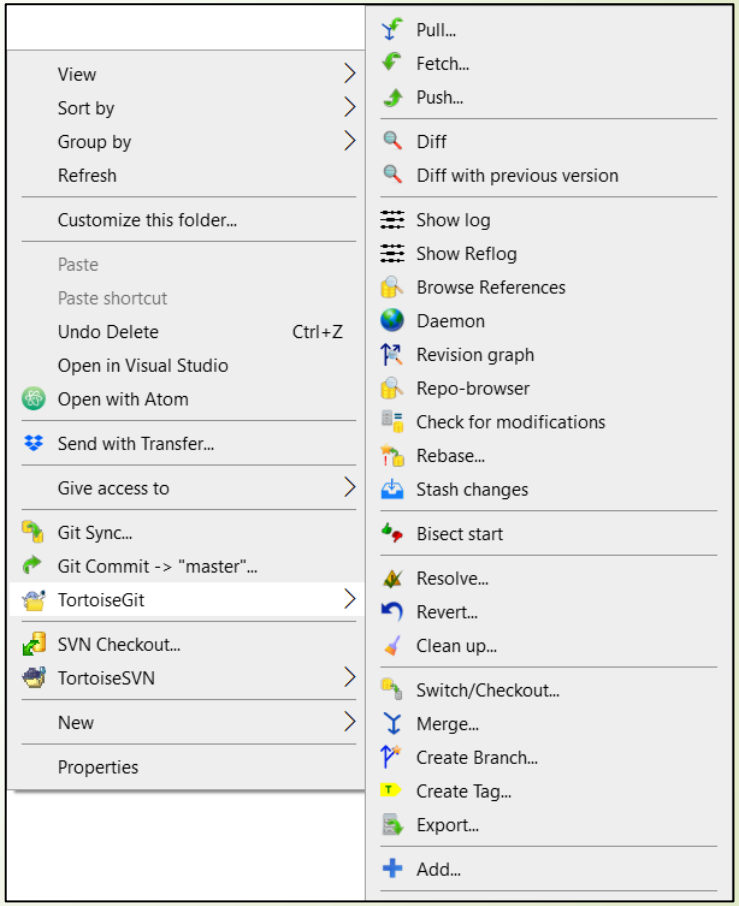




Version control systems

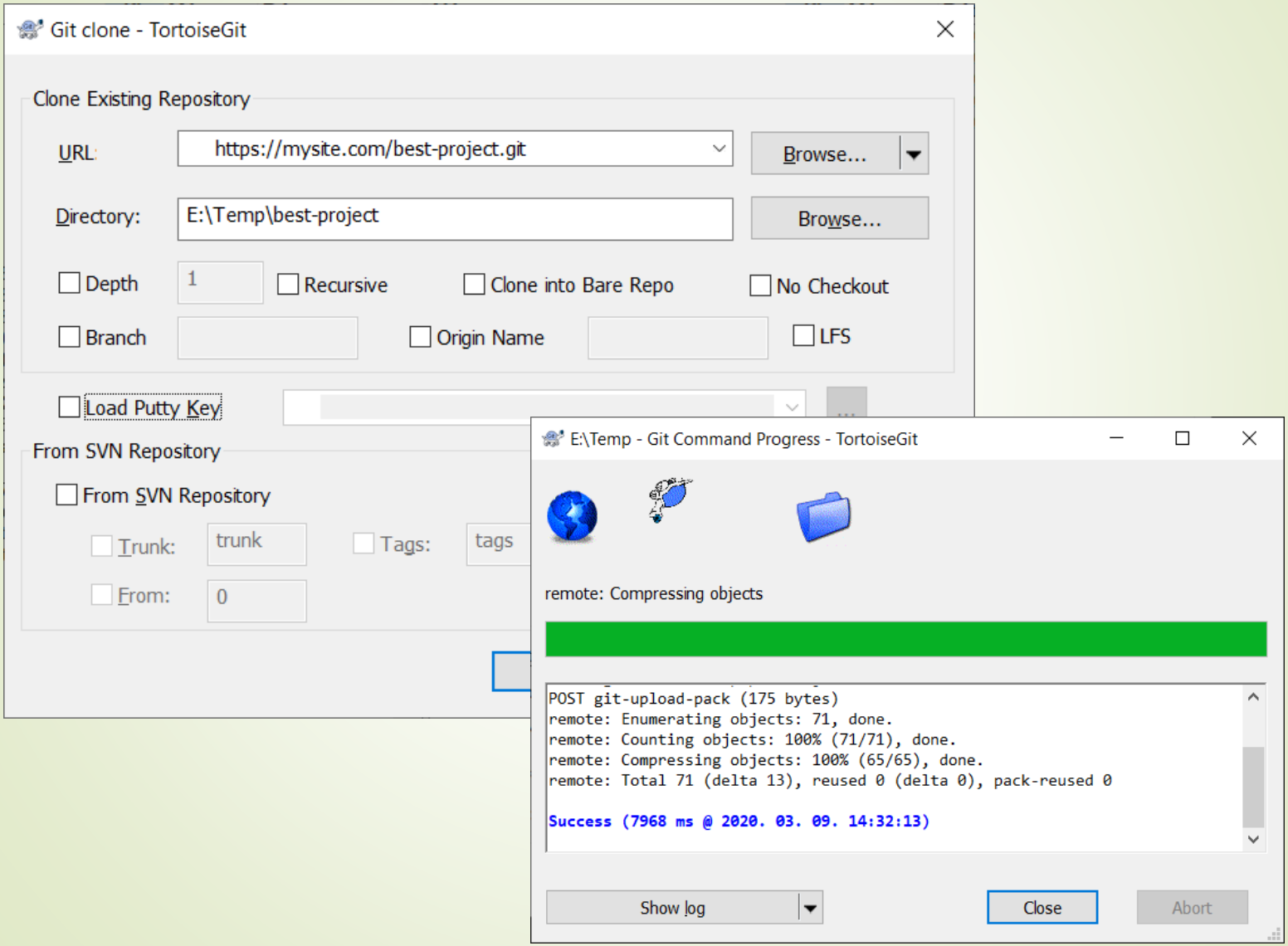
Using TortoiseGit

- TortoiseGit is a free graphical desktop client for the Windows operating system
  - Homepage, download: https://tortoisegit.org/
  - Does not include Git, which must be installed separately.
- When using TortoiseGit, the usual Git commands can be invoked from the (right mouse click) context menu of the *File Explorer.*



Version control systems

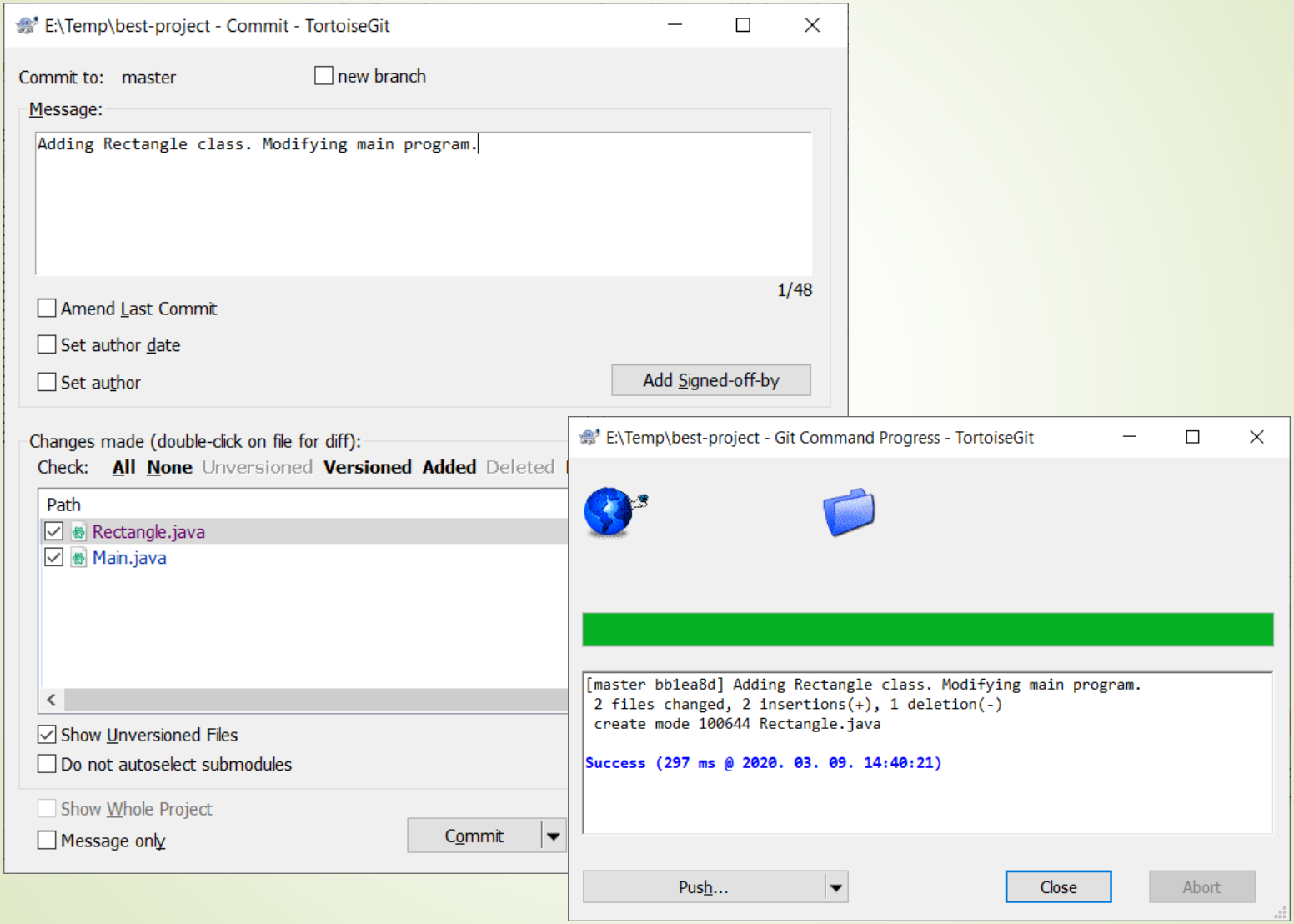
TortoiseGit: clone a repository



Version control systems

ELTE | IK
INFORMATIKAI KAR

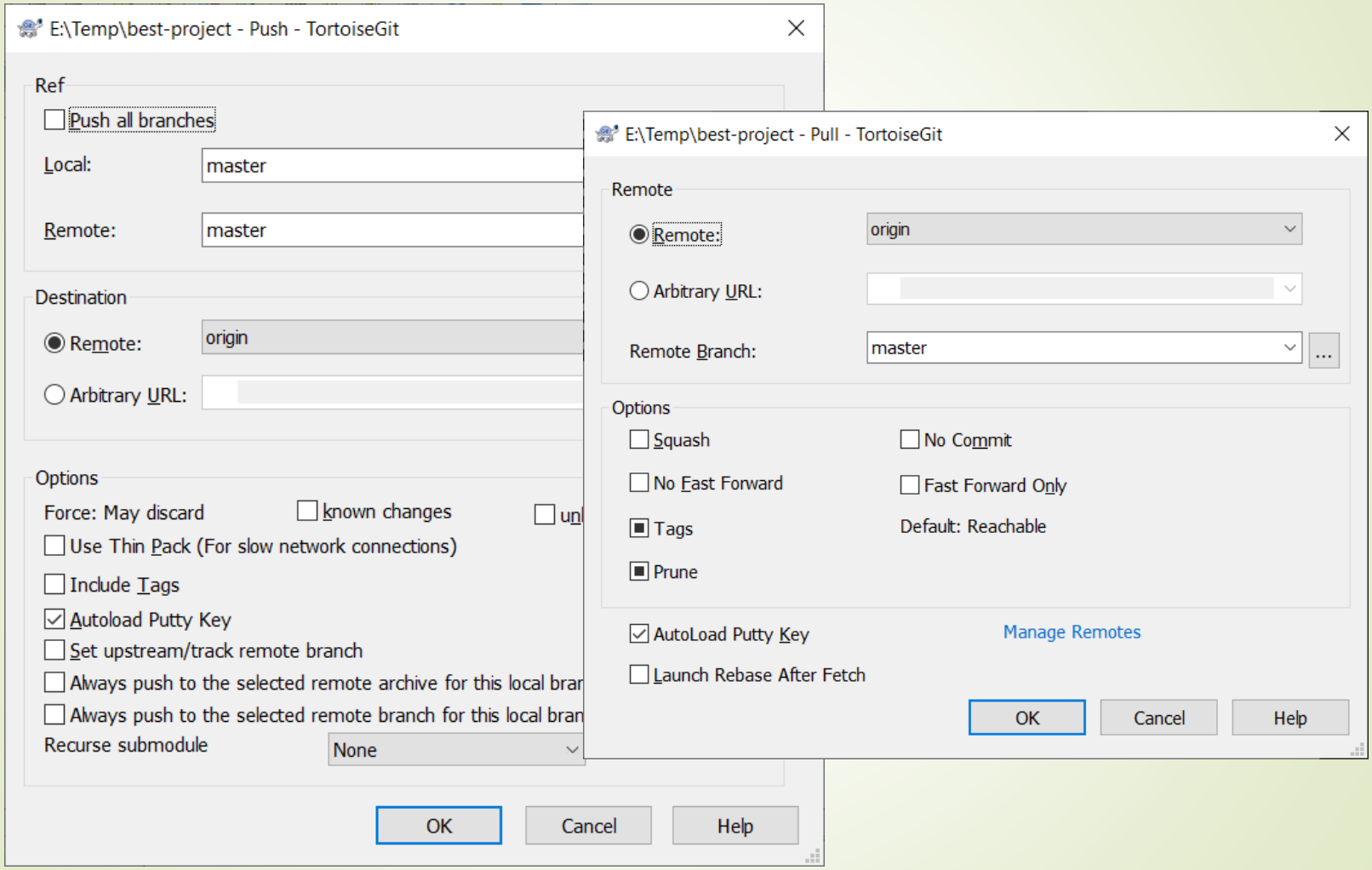
TortoiseGit: commit a new revision



Version control systems

ELTE | IK
INFORMATIKAI KAR

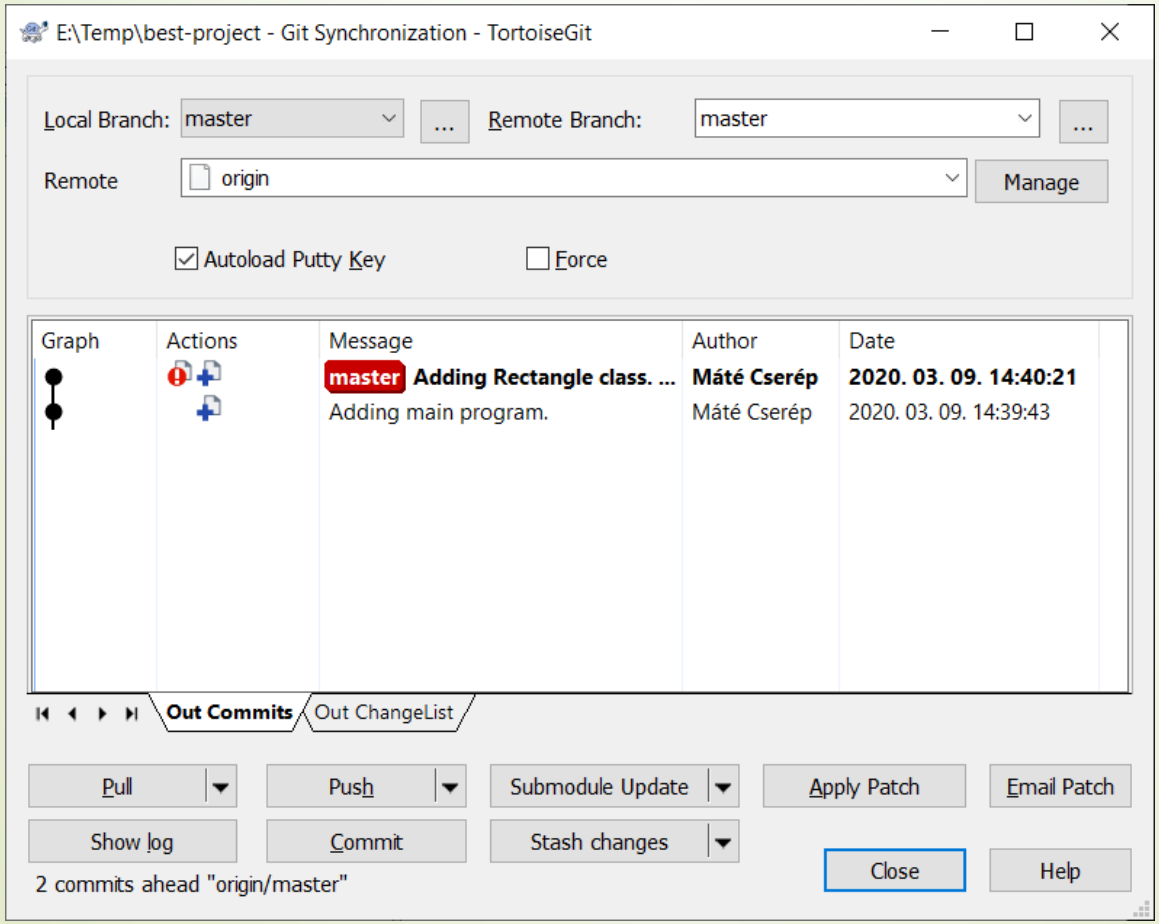
TortoiseGit: synchronize with remote (*push* & *pull*)



Version control systems

TortoiseGit: synchronize with remote (*sync*)

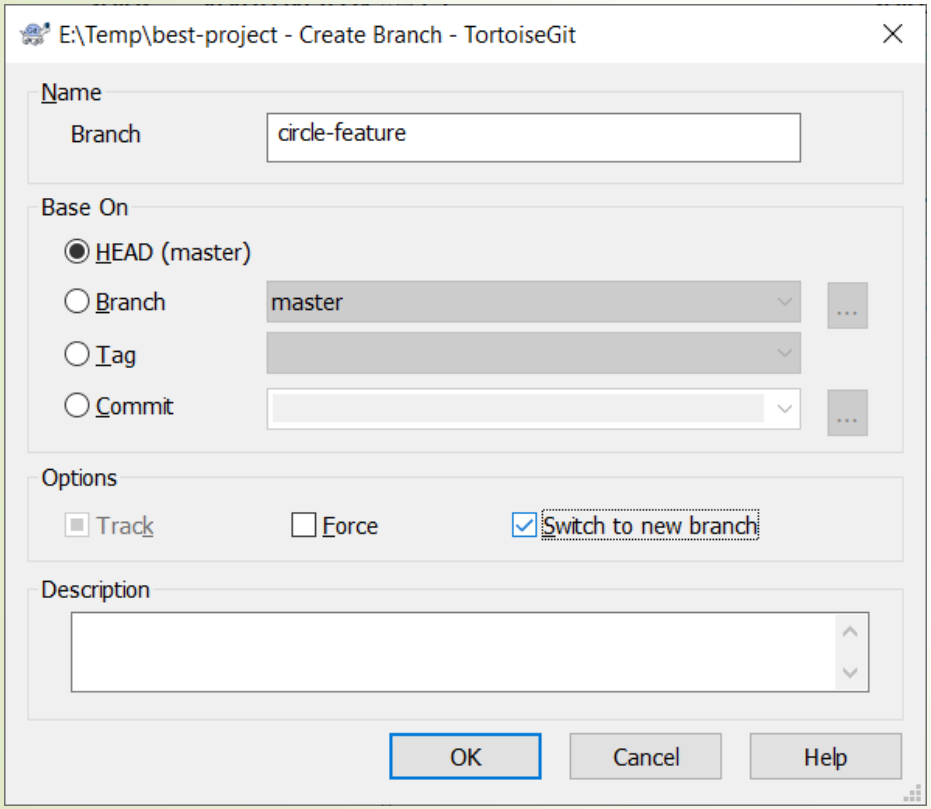
- The *Sync* option provides an overview interface where both the *pull* and the *push* functionalities can be accessed.

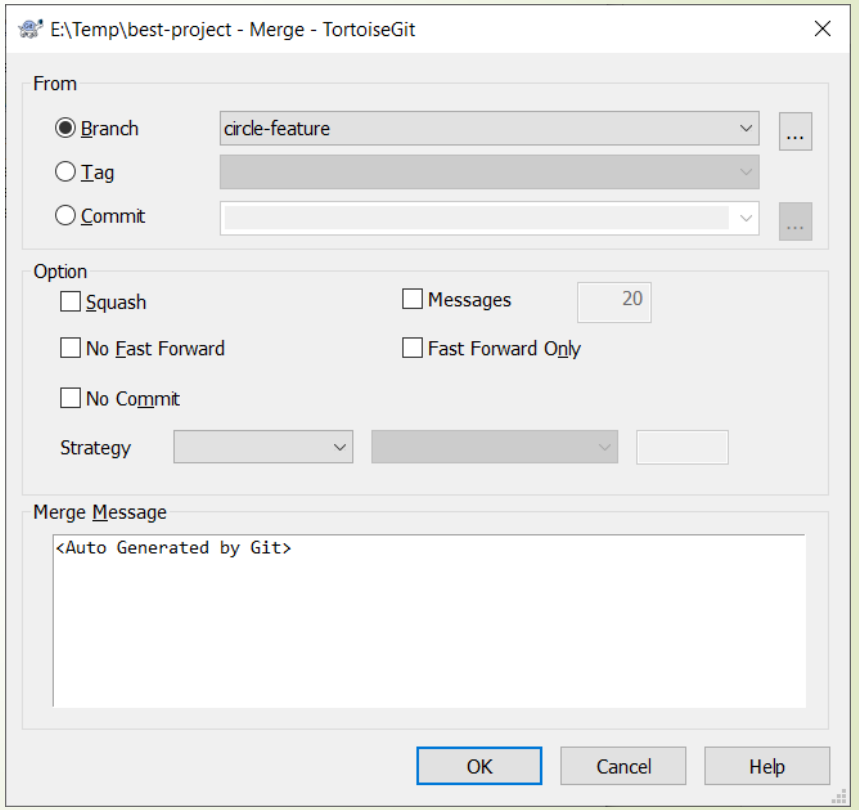


Version control systems

TortoiseGit: create and merge branches

- Beginner Git users often find creating and merging branches a complex task with the command-line tools. The graphical interface of TortoiseGit can ease this job.

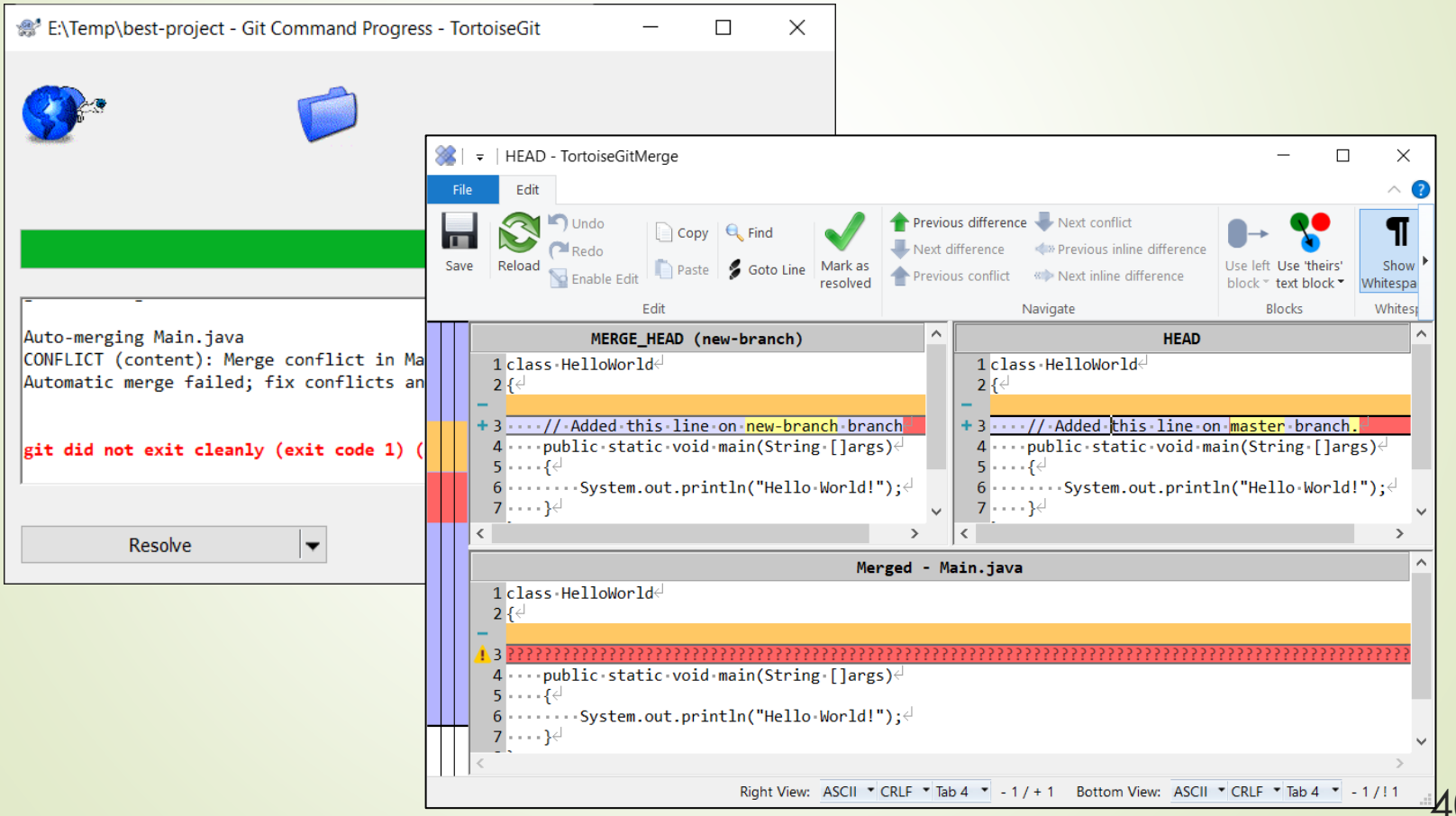




Version control systems

TortoiseGit: resolving merge conflicts

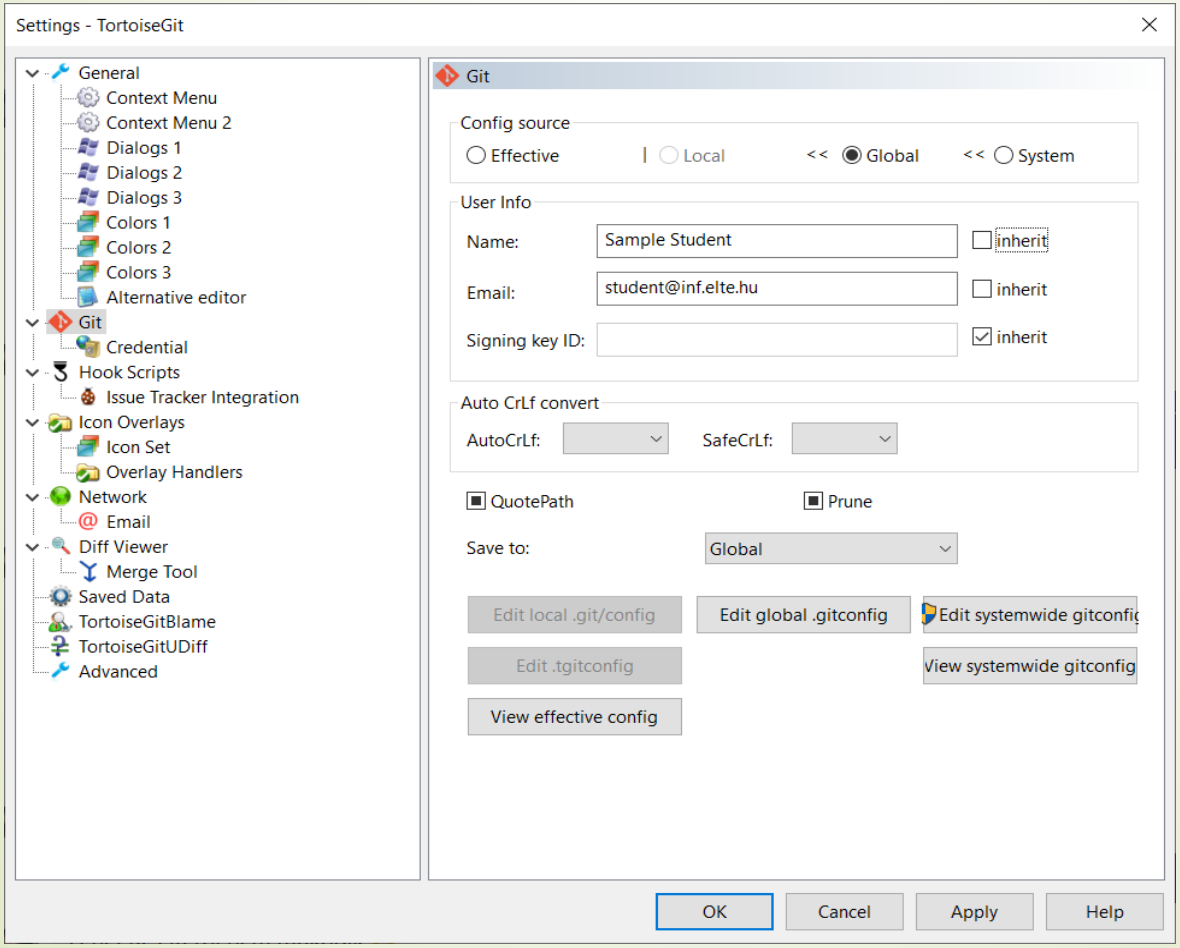
- This is especially true when resolving conflicts. TortoiseGit provides a side-by-side comparison view of the 2 file states to be merged.



Version control systems

TortoiseGit: settings

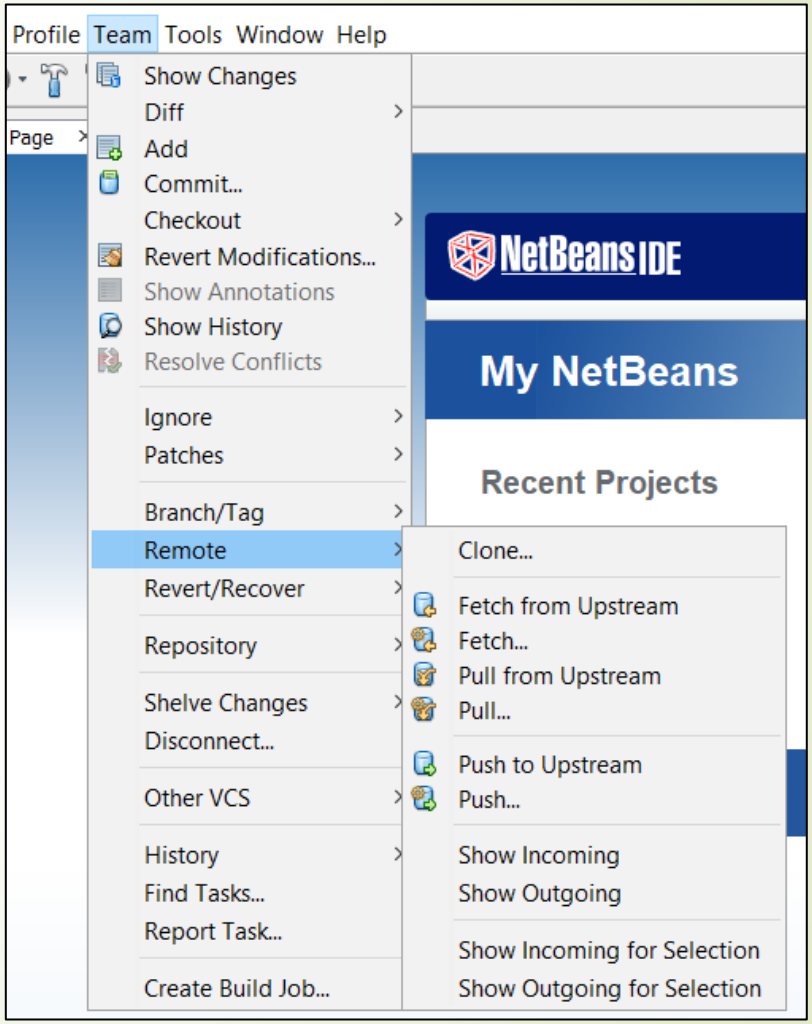
- In the *TortoiseGit -> Settings* option in the menu, we can modify the settings, e.g. the commiter's name and email address.



Version control systems

Support in integrated development environments

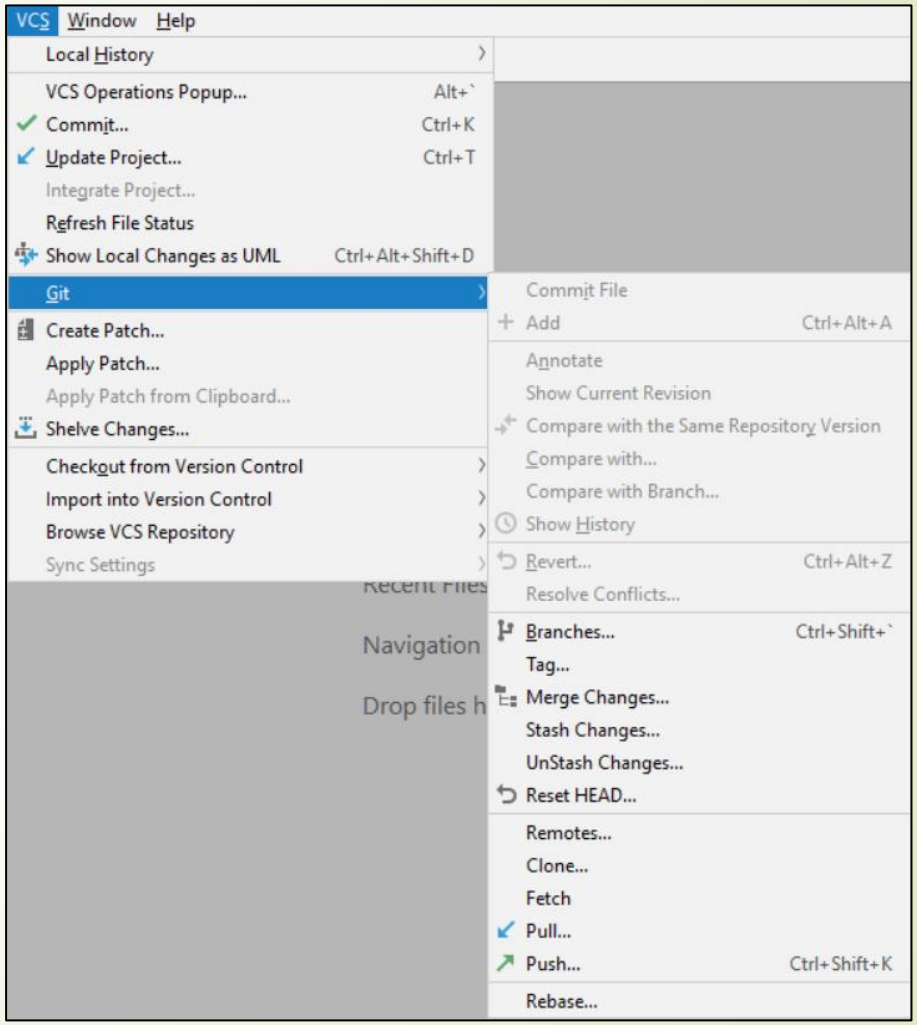
- Most modern integrated development environments (IDE) offers native solution for version control.
  - They are called *integrated* for a reason.
  - Therefore switching between multiple applications during development can be avoided.
- In *NetBeans* the version control functionalities are located under the *Team* menu.



Version control systems

Support in integrated development environments

- In *IntelliJ IDEA* these functionalities are locate under the VCS (*version control system*) menu.



Version control systems

Which files shall be version controlled?

- Git is efficient is version controlling text files, including code files. Therefore the purpose of the version control system is to track the changes of the source code of the software under development.
- For a general software project, it is NOT recommended to place under version control:
  - intermediate and final binary files of the compilation process, as they are reproducible from the source code and will cause merge conflicts regularly.
  - personal configuration of the development environment (e.g. the *.vs/* folder for Visual Studio or the *nbproject/private/* folder for NetBeans), as the differ among developers.
  - large binary files (e.g. videos, large images), as Git is not efficient version controlling them. Although the size of Git repositories scale well, the size of an easily manageable *repository* does not exceed 1-2 GB.

Excluding files from version control

- Only files explicitly added to version control (`git add`) are tracked, other files are excluded.
- To avoid tracking files accidentally, we can mark files and directories which must always be ignored from version control.
  - The rules can be defined in special `.gitignore` files.
  - These files shall be version controlled, so the configuration is shared among team members.
- Inside a `.gitignore` file, each line contains a pattern to match files and directories. The matched files are excluded from version control.
  - This configuration is applied transitively to subdirectories, so a single `.gitignore` file in the project root directory can often be sufficient.

Version control systems

Git: .gitignore pattern

Pattern	Matches	Description
program.jar	/program.jar /bin/program. jar	Matches all **program.jar** files.
/program.jar	/program.jar **but not:** /bin/program.jar	Matches the **program.jar** file at the given directory level.
*.jar	/program.jar /bin/main.jar	Matches all files with the **jar** extension.
bin/	/bin/ /project/bin/ **but not:** /logs/bin (file)	Matches all **bin** folders (but not the files with the name **bin**!)
/bin/*.jar	/bin/program.jar /bin/main.jar **but not:** /bin/program.class /project/bin/program.jar	Matches all files with the **jar** extension in the **bin** folder at the given directory level.

Git: .gitignore pattern

Pattern	Matches	Description
*.log !important.log	/application.log /logs/application.log **but not:** /logs/important.log	Matches all files with the **log** extension, except when the file's name is **important.log**.
logs/**/*.log	/logs/application.log /logs/runtime/main/01.log /project/logs/deploy.log **but not:** /runtime.log	Matches all files with the **log** extension, which are inside a folder named **logs** (arbitrary folder depth allowed).

- For most programming languages an IDEs, a general purpose, sample `.gitignore` file is available on *GitHub*. It is a good idea to copy such a sample file or even combine multiple of them.
  - URL: https://github.com/github/gitignore

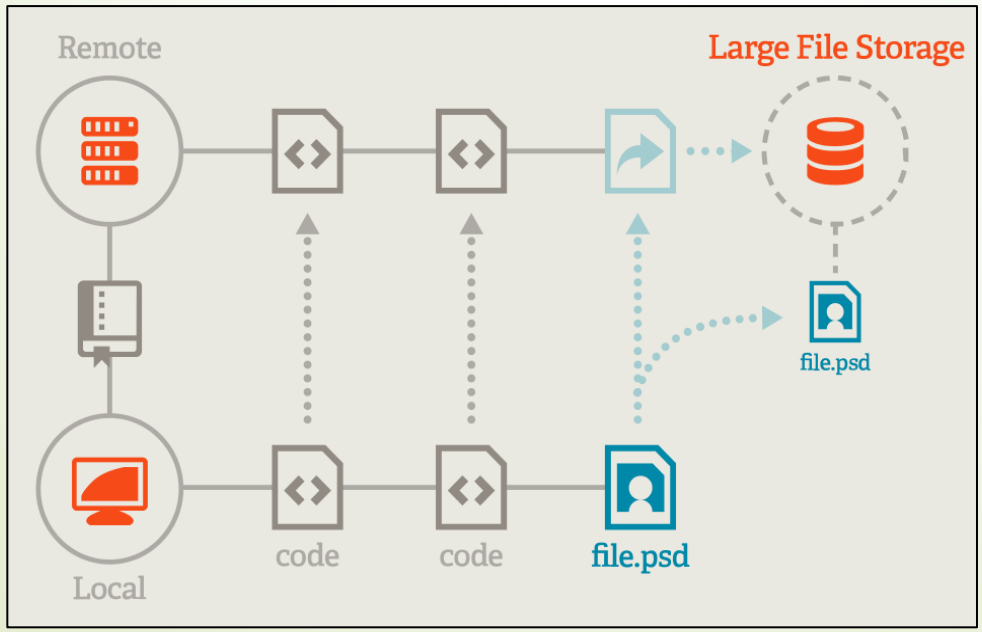
Version control of large binary resources

- Version control of large video, image and audio resource files requires extra caution.
  - Git is not efficient in tracking changes of large binary files.
  - It significantly increases the network traffic required to download a local repository (*git clone*).
  - Developers do not necessarily need the *assets* created by designers during development and testing time.
- Therefore even in the case the large binary resource files in a project change rarely, usually it does not worth to track them in a version control system like Git.

Version control systems

Git Large File Storage

- When required, the version control of large binary files can be done with the *Git Large File Storage* (Git LFS) system.
  - It replaces large binary files with a reference and the content of the files themselves are not stored in the Git repository, but on a different (remote) server.
  - Thus the size of our Git repository remains small.



Source: git-lfs.github.com

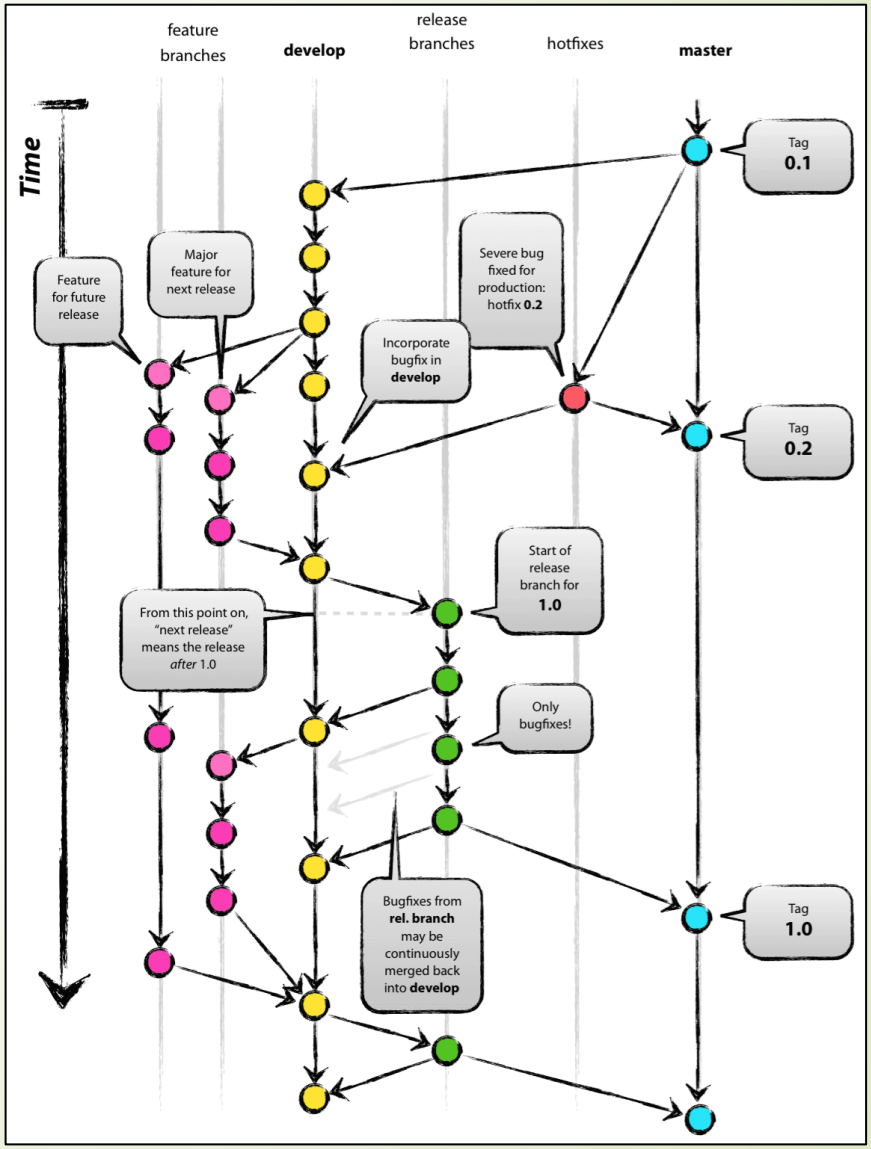
Git Large File Storage

- The GitLab instance on szofttech.inf.elte.hu supports Git LFS.
- To utilize it, Git LFS must be installed on the client computers (the computers you use for development).
  - Download: https://git-lfs.github.com/
  - Usage: https://docs.gitlab.com/ee/administration/lfs/manage_large_binaries_with_git_lfs.html

Version control systems

Gitflow feature branching model

- Preferred for large software with multiple supported releases
- Main branches:
  - master
  - develop
- Supporting branches:
  - feature branches
  - release branches
  - hotfix branches



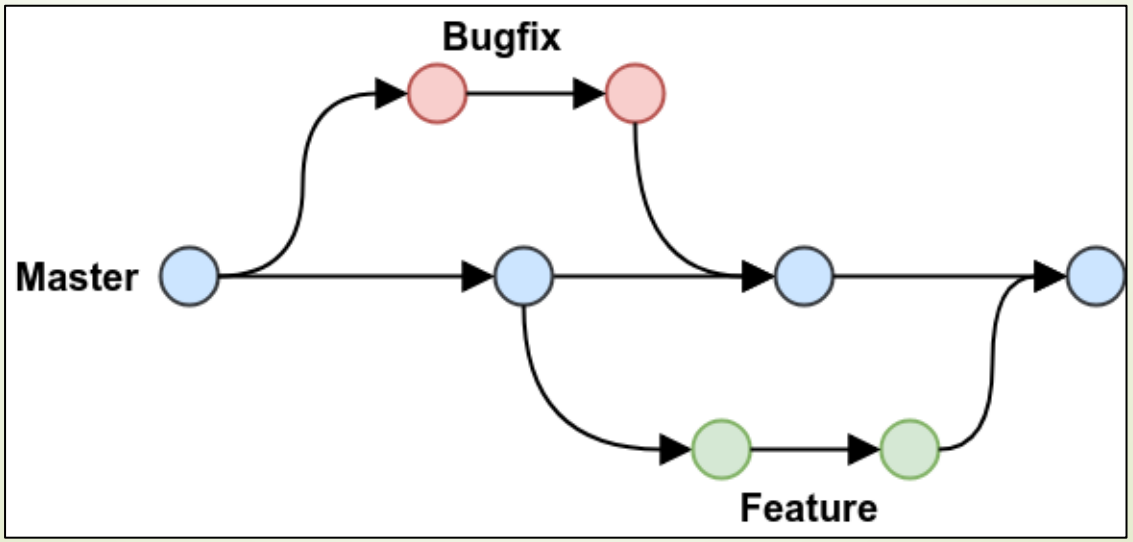
Source: nvie.com

Version control systems

GitHub feature branching model

- Preferred for software with a single supported release and a fast cycle to roll out new releases.
  - What is on the main branch, can be released!

- Main branch: *master*
- Supporting branches: *feature* and *bugfix* branches

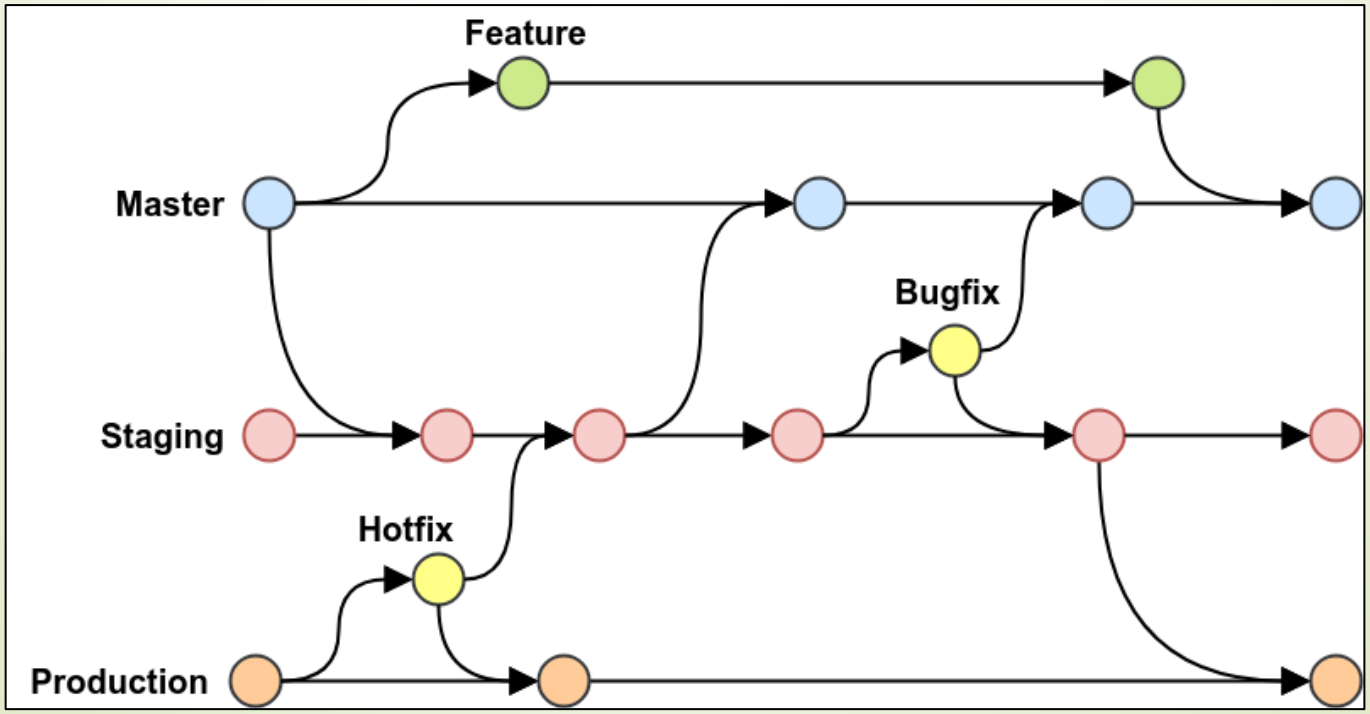


Source: blog.programster.org

Version control systems

GitLab Flow feature branching model

- It may not always be possible to release a new version of the software on the main development branch.
  - Main development branch: *master*
  - Production release branch: *production*
  - Staging (testing) release branch: *staging* (optional)



Source: blog.programster.org

Software technology

Build systems

Specializations A & B

Dr. Máté Cserép

ELTE, Faculty of Informatics

2025.

Build systems

„The problem of handling the compilation of a project has been encountered well before you were born. That's why there's a single method to do it today, being a consensus since several decades.

Ha! Ha! Just kidding!"

Julien Jorge

Build systems

Compiling C++ programs

- How can we compile a C++ application using console tools (e.g., with the GNU g++ compiler)?

```
g++ -c -o foo.o foo.cpp \
    -O2 -std=c++17 -pedantic -I./include/


... additional translation units ...


g++ -c -o main.o main.cpp \
    -O2 -std=c++17 -pedantic -I./include/
g++ -o program.exe  foo.o ...  main.o
```

Build systems

Compiling C++ programs

- As we can see, even for a relatively simple program consisting of a few source files, the solution is not straightforward.

- **Problems:**
  - Manually compiling programs can quickly become difficult to manage even for smaller projects.
  - For larger programs, it becomes difficult to track which translation units need to be recompiled.
    - Rebuilding the entire application can take a significant amount of time for real-world enterprise applications.

Build systems

Categorization of build tools

- *Standalone*
  - Automates the building of projects based on specified instructions and/or rules.
  - Examples: Make, NMake, Scons, Jom, BJam, Ninja
- Integrated into development environments (*integrated*)
  - Build systems that can be used within IDEs
  - Examples: Visual Studio, Xcode, Eclipse
- *Generators*
  - Tools that generate input files for other *standalone* build systems automatically.
  - Examples: CMake, Autotools, GYP

Build systems – Make

Installation

- GNU Make
  - https://www.gnu.org/software/make/
- Installation:
  - On Windows, it is recommended to install MinGW (*Minimalist GNU for Windows*) and add its *bin* directory to the *PATH* environment variable.
  - On UNIX systems, it is available in the *package repository*.
    - Debian/Ubuntu: `apt-get install make`
  - It is often installed together with development environments (e.g., JetBrains CLion, Qt Creator)

Build systems – Make

Characteristics

- Compilation is defined through *targets* and *rules*.
- Rules can contain *dependencies* and *commands*.
- The set of rules is stored in a *Makefile*.
- To run: **`make`**
  - If the *Makefile* has a different name:
    `make -f MyMakefile`
- Example *Makefile*:

```
program: foo.cpp main.cpp
	g++ -o program foo.cpp main.cpp -std=c++17
```

dependency

target

command

rule

Build systems – Make

Rules

- *Makefile* rules are structured as follows:
  - *Target*: the file to be generated by the rule, often (but not necessarily) a binary file.
  - *Dependencies*: if the target does not exist or any of the dependencies are updated, the target must be regenerated.
    - Dependency updates can be determined based on file timestamps.
  - *Commands*: the actions required to generate the target.

Build systems – Make

Variables

- In addition to rules, we can define variables in the *Makefile*:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic

program: foo.cpp main.cpp
	$(CXX) -o program foo.cpp main.cpp $(CFLAGS)
```

Build systems – Make

Handling multiple translation units

- By defining additional rules, multiple translation units can be handled:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic

foo.o: foo.cpp foo.h
  $(CXX) -c -o foo.o foo.cpp $(CFLAGS)

main.o: main.cpp
  $(CXX) -c -o main.o main.cpp $(CFLAGS)

program: foo.o main.o
  $(CXX) -o program foo.o main.o $(CFLAGS)
```

Build systems – Make

Automatic variables

- Automatic variables allows to generalize the rules based on patterns.
- Important automatic variables:
  - $@: the name of the target (file)
  - $<: the name of the first dependency
  - $^: the names of all dependencies
  - $?: the names of dependencies newer than the target
  - $(@D): the directory part of the target's name
  - $(@F): the file part of the target's name

Build systems – Make

Automatic variables

- *Pattern rule* for compiling object files:

```
CXX=g++
CFLAGS=-std=c++17 -pedantic
DEPS=foo.h

%.o: %.cpp $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)

program: foo.o main.o
  $(CXX) -o program foo.o main.o $(CFLAGS)
```

Build systems – Make

Automatic variables

- Extracting object files (and avoiding redundancy):

```
CXX=g++
CFLAGS=-std=c++17 -pedantic
DEPS=foo.h
OBJ=foo.o main.o

%.o: %.c $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)

program: $(OBJ)
  $(CXX) -o $@ $^ $(CFLAGS)
```

Build systems – Make

ELTE | IK
INFORMATIKAI KAR

Handling project directory structure

- Managing source and build directories (e.g. *include*, *obj*):

```
IDIR=../include
ODIR=obj
CXX=g++
CFLAGS=-std=c++17 -pedantic -I$(IDIR)
_DEPS=foo.h
DEPS=$(patsubst %, $(IDIR)/%, $(_DEPS))
_OBJ=foo.o main.o
OBJ = $(patsubst %, $(ODIR)/%, $(_OBJ))

$(ODIR)/%.o: %.cpp $(DEPS)
  $(CXX) -c -o $@ $< $(CFLAGS)
$(ODIR)/program: $(OBJ)
  $(CXX) -o $@ $^ $(CFLAGS)
```

../include/foo.h

obj/foo.o obj/main.o

Build systems – Make

Phony targets

- In a Makefile, targets normally refer to files produced as a result of the rule. Let's look at an example where this is not the case:

```
clean:
	rm -f $(ODIR)/*.o $(ODIR)/program
```

- If a file named *clean* exists, the *make clean* command will not execute the commands (since there are no dependencies).
- Such targets are marked as *phony* targets. A typical example:

```
.PHONY: clean
clean:
	rm -f $(ODIR)/*.o $(ODIR)/program

.PHONY: all
all: $(ODIR)/program
```

Build systems – Make

Modules

- GNU Make supports modularized projects. In this case, each subdirectory contains its own *Makefile*, and the parent directory's *Makefile* triggers their execution:

```
.PHONY: all

all: program

  $(MAKE) -C module_a
  $(MAKE) -C module_b
  $(CXX) -o program $(CFLAGS) \
    module_a/obj/modul_a.o \
    module_b/obj/modul_b.o
```

Build systems – Make

ELTE | IK
INFORMATIKAI KAR

Example

- Sample project: Console thesis number generator application
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp

```
# Programs & tools (Linux)
#CXX     = g++
#RM      = rm -f
#MKDIR   = mkdir -p
#CP      = cp -f

# Programs & tools (Windows)
CXX     = g++
RM      = rmdir /s /q
MKDIR   = mkdir
CP      = copy /y

# Flags & options
CFLAGS  = -O2 -std=c++17 -pedantic
INCLUDE = -I./include/

# Output
OBJDIR  = build
MAIN    = thesisgenarator.exe
TARGET  = $(OBJDIR)/$(MAIN)

_OBJ = GeneratorModel.o main.o
OBJ = $(patsubst %, $(OBJDIR)/%, $(_OBJ))

# Make targets
.PHONY: all
all: $(TARGET)

$(OBJDIR):
  $(MKDIR) $(OBJDIR)

$(OBJDIR)/GeneratorModel.o: GeneratorModel.cpp GeneratorModel.h
  $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)

$(OBJDIR)/main.o: main.cpp GeneratorModel.h
  $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)

$(TARGET): $(OBJDIR) $(OBJ)
  $(CXX) -o $@ $(OBJ) $(CFLAGS)

.PHONY: clean
clean:
  $(RM) $(OBJDIR)
```

Build systems – CMake

Installation

- CMake
  - https://cmake.org/
- Platform-independent and compiler-independent generator for lower-level build tools.
  - Examples: GNU Make, MSVC, Ninja.
- Installation:
  - Binaries and installers for Windows, Linux, and macOS are available on the official website: https://cmake.org/download/
  - On UNIX-based systems, it can typically be installed from the *package repository*.
    - Debian/Ubuntu: `apt-get install cmake`

Build systems – CMake

Usage

- The configuration is specified in the *CMakeLists.txt* file.
- Run with: **`cmake <parameters>`**
- Example CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)
add_executable(HelloWorld main.cpp foo.cpp)
```

Minimum required CMake version

Project name

Adding a new executable binary target:
Create the *HelloWorld* binary:
from the *main.cpp* and *foo.cpp* source files

Build systems – CMake

Variables

- Variables can be defined and referenced using `${name}` syntax:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)

set(SRCS \
  main.cpp \
  foo.cpp \
  foo.h)

add_executable(HelloWorld ${SRCS})
```

Build systems – CMake

Configuring C++ compiler options

- You can customize the C++ compiler options:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)
add_executable(HelloWorld main.cpp foo.cpp)

target_compile_features(
  HelloWorld PRIVATE cxx_std_17)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)
```

GNU GCC specific flags

Visibility rules will be discussed later

Build systems – CMake

ELTE | IK
INFORMATIKAI KAR

Building program libraries

- Creating static or dynamic libraries:

```
cmake_minimum_required (VERSION 3.16)
project(MyStack)

add_library(my_stack_static STATIC mystack.cpp)
add_library(my_stack_dynamic SHARED mystack.cpp)
```

  - Static libraries have `.lib` extension on Windows and `.a` extension on UNIX systems.
  - Dynamic libraries have `.dll` extension on Windows and `.so` extension on UNIX systems.

Build systems – CMake

Configuring include path

- Extend the *include* path with the *include* directory:

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)


target_compile_features(
  HelloWorld PRIVATE cxx_std_17)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)


include_directories("include")


add_executable(HelloWorld main.cpp foo.cpp)
```

Build systems – CMake

ELTE | IK
INFORMATIKAI KAR

Linking build targets

```
cmake_minimum_required (VERSION 3.16)
project(HelloWorld)

add_executable(HelloWorld
  main.cpp foo.cpp mystack.h)
add_library(MyStack STATIC mystack.cpp mystack.h)
target_compile_options(
  MyStack PRIVATE -O2 -pedantic)

target_link_libraries(HelloWorld MyStack)
```

- The visibility control rules mentioned earlier are important here.

Linking build targets

- The following visibility options are available:
  - **PRIVATE**: the flags apply only to the given build target. In the example, the `-O2 -pedantic` flags would not apply to the `HelloWorld` target.
  - **PUBLIC**: the flags also apply to additional targets processing the given build target.
  - **INTERFACE**: the flags do not apply to the given build target, but they do apply to additional targets processing the given build target.

Build systems – CMake

Packages

- CMake can manage many commonly used external dependencies as packages, allowing them to be built and linked into our program.

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

find_package(SomePackage)
include_directories(...)
link_directories(...)
link_libraries(...)
```

- It can be enough to link to the target:

```
target_link_directories(mytarget ...)
target_link_libraries(mytarget ...)
```

Build systems – CMake

ELTE | IK
INFORMATIKAI KAR

Packages

- Example for using the OpenCV library:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

find_package(OpenCV REQUIRED)
link_libraries(${OpenCV_LIBS})
```

Build systems – CMake

ELTE | IK
INFORMATIKAI KAR

Packages

- Example for using the Boost library:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

set(Boost_USE_STATIC_LIBS ON)
set(Boost_USE_STATIC ON)
find_package(Boost REQUIRED
  COMPONENTS filesystem log program_options)
include_directories(${Boost_INCLUDE_DIRS})
link_libraries(${Boost_LIBRARIES})
```

Build systems – CMake

Packages

- Example for using the Qt library:

```
cmake_minimum_required (VERSION 3.16)
project(QtHelloWorld)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTOUIC ON)

find_package(Qt5Widgets CONFIG REQUIRED)

set(SRCS mainwindow.ui mainwindow.cpp main.cpp)
add_executable(helloworld WIN32 ${SRCS})
target_link_libraries(helloworld Qt5::Widgets)
```

Build systems – CMake

Modules

- CMake supports modularized projects. In this case, each subdirectory contains its own *CMakeLists.txt* file, and the parent directory references the subdirectories to manage them:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

... Configuration  ...

add_subdirectory(module_a)
add_subdirectory(module_b)
```

Build systems – CMake

Generators

- Typically, we want to build the project into a separate *build directory*:

```
mkdir build && cd build
```

`cmake ../` or `cmake ../src`

- We can choose between several generators:
  - Using GNU Makefile:
    `cmake -G "Unix Makefiles" ../`
  - Microsoft Visual Studio:
    `cmake -G "Visual Studio 17 2022" -A x64 ../`
  - Xcode:
    `cmake -G Xcode ../`
- Then we compile the program:
  - Makefile: `make`
  - MSVC: `msbuild MyProgram.sln`

Build systems – CMake

Deployment

- We can specify an install directory where the final binaries should be copied:

```
cmake -DCMAKE_INSTALL_PREFIX=../install ../src
```

- Example CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)
project(MyProgram)

set(SRCS main.cpp foo.cpp foo.h)
add_executable(HelloWorld ${SRCS})
install(TARGETS HelloWorld
  DESTINATION ${CMAKE_INSTALL_PREFIX})
```

- Makefiles: **`make install`**
- MSVC: **`msbuild INSTALL.vcxproj`**

Build systems – CMake

ELTE | IK
INFORMATIKAI KAR

Example

- Sample project: console thesis number generator application
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp
- CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.16)
project(ThesisGenerator)
include_directories(./)
add_executable(ThesisGenerator
    GeneratorModel.cpp
    GeneratorModel.h
    main.cpp)
target_compile_features(ThesisGenerator PUBLIC cxx_std_17)
install(TARGETS ThesisGenerator
    DESTINATION ${CMAKE_INSTALL_PREFIX})
```

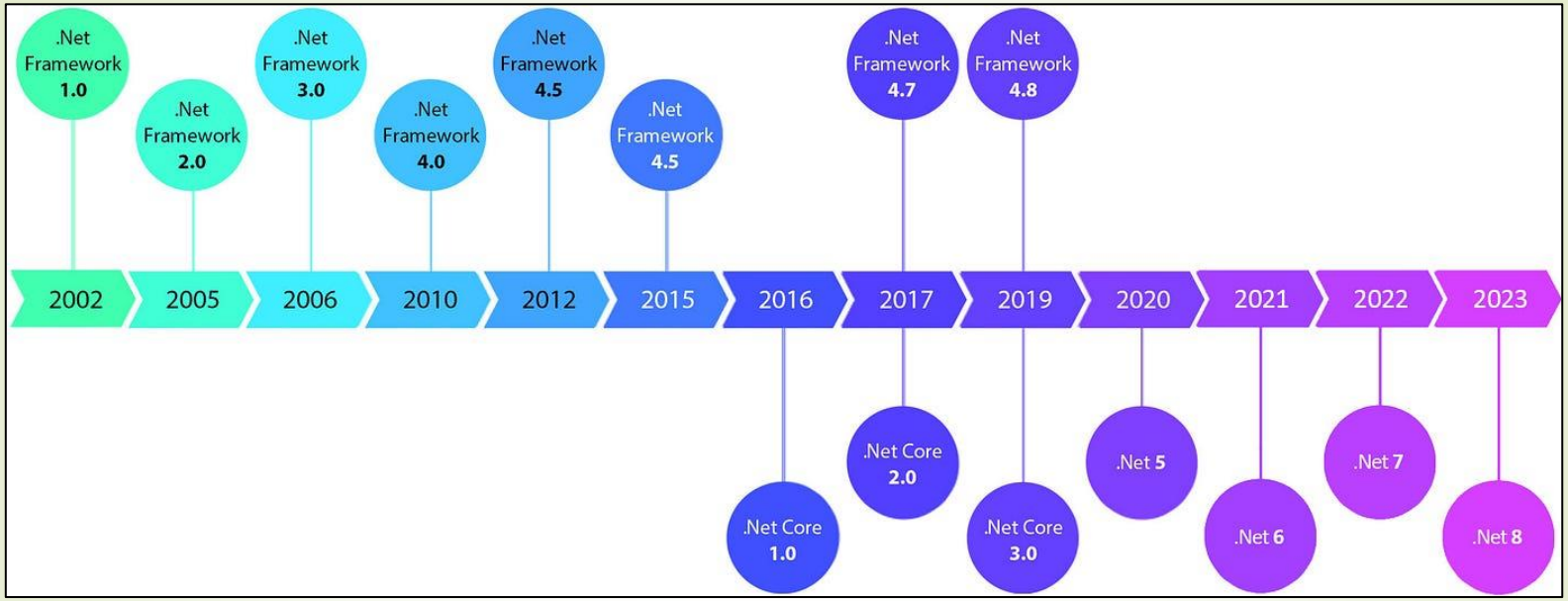
Build systems – Qt

Usage

- Qt's own build system is based on *Makefiles*.
- Creating and building a Qt project consists of 3 steps:
  - `qmake -project`: create a new Qt project file (.pro file).This project file can be edited manually or by an IDE and keeps track of project files and configurations.
  - `qmake`: generate a *Makefile* based on the project file.
  - `make`: compile the project using GNU Make.
- This process can be automated with a single click using an IDE like *Qt Creator*.
- Sample project: Qt-based thesis number generator application
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-qt
- Qt officially supports CMake as well (e.g. through Qt Creator).

Build systems – .NET

.NET Framework



- *Problems with the .NET Framework:*
  - Windows-centered approach
  - Monolithic, poorly modularized structure
  - Closed source (for contribution)

Build systems – .NET

.NET Core

- *The .NET Core addresses these problems:*
  - Cross-platform (Windows, Linux, macOS) and open-source (https://github.com/dotnet/core)
    - Can be used for GUI applications as well (since .NET Core 3), using *desktop packs,* although applications can become platform-dependent this way
  - Modular structure, only components required by the application need to be installed in runtime
    - Useful for e.g. creating microservices, containerization (e.g., Docker), in embedded systems
  - High performance and scalability (significantly better performance with .NET Core and ASP.NET Core)

Build systems – .NET

ELTE | IK
INFORMATIKAI KAR

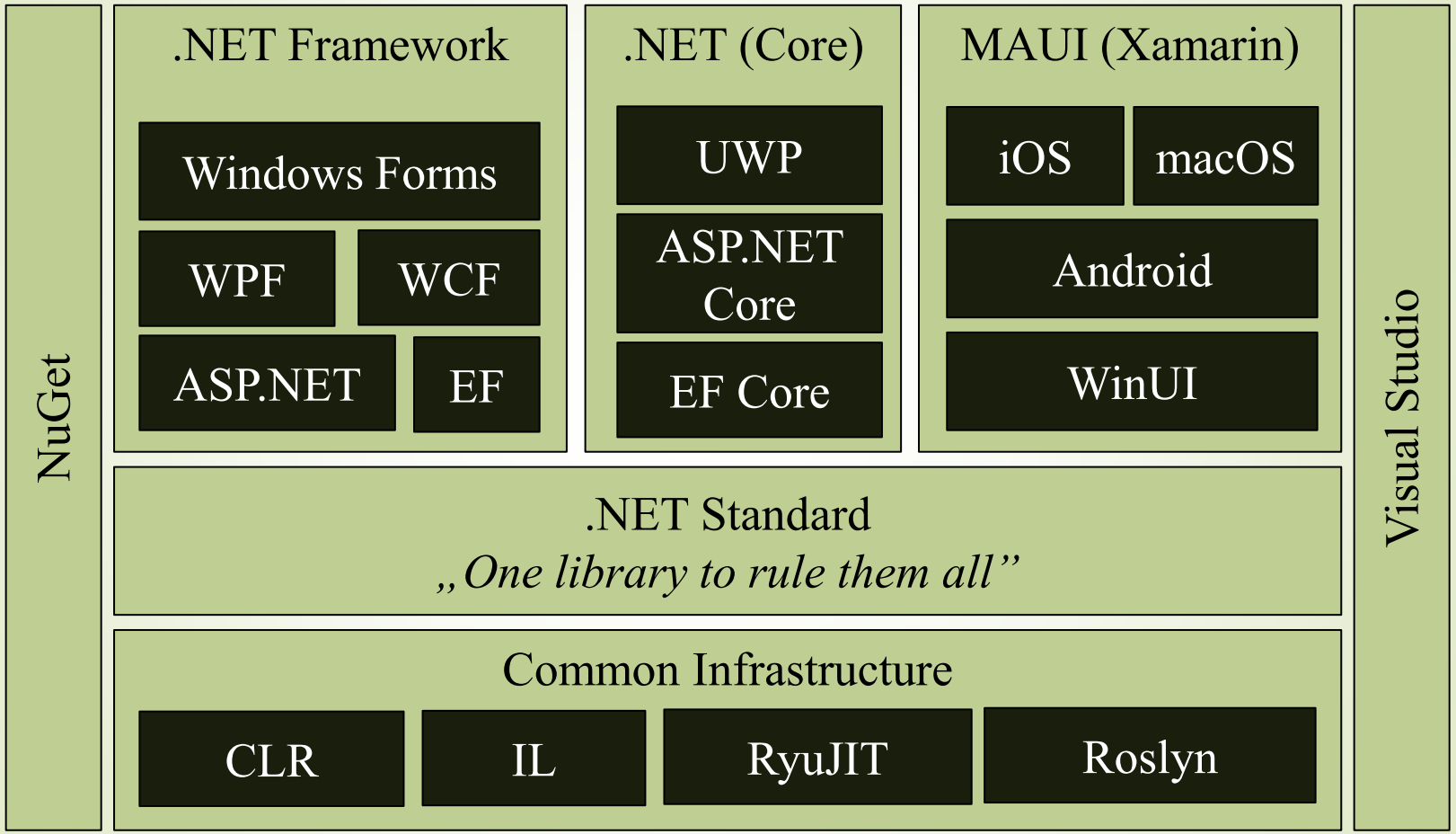
.NET Standard

- *Arising problems:*
  - Integrating applications written in different .NET frameworks
  - Developing general-purpose libraries
- *.NET Standard:*
  - A common, shared API across different framework Base Class Libraries (BCLs)
  - Replaces the older PCL (Portable Class Library) projects

Build systems – .NET

ELTE | IK
INFORMATIKAI KAR

.NET Standard



Build systems – .NET

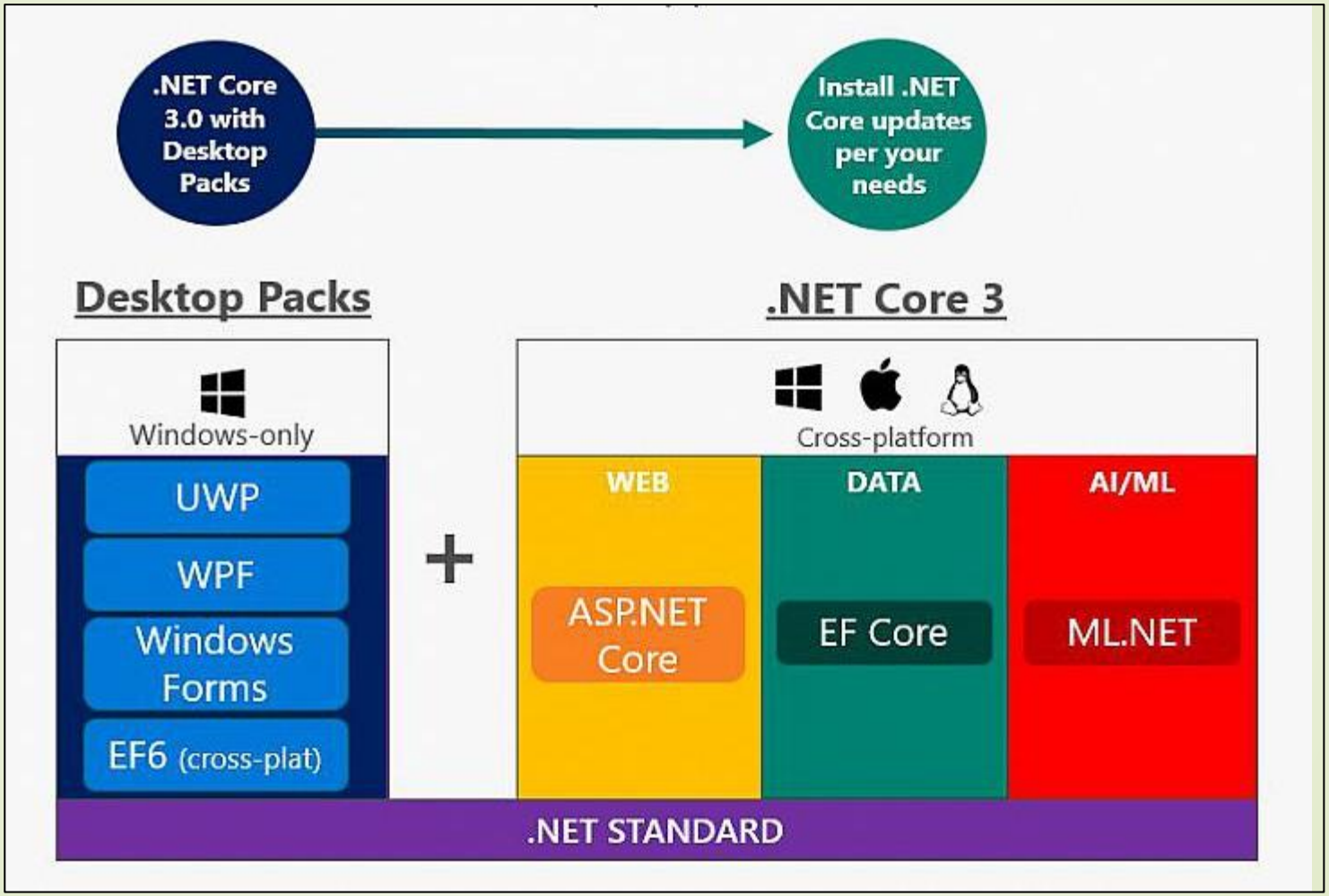
.NET Standard

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	*N/A*
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	*N/A*
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2021.2

Source: https://docs.microsoft.com/en-us/dotnet/standard/net-standard

Build systems – .NET

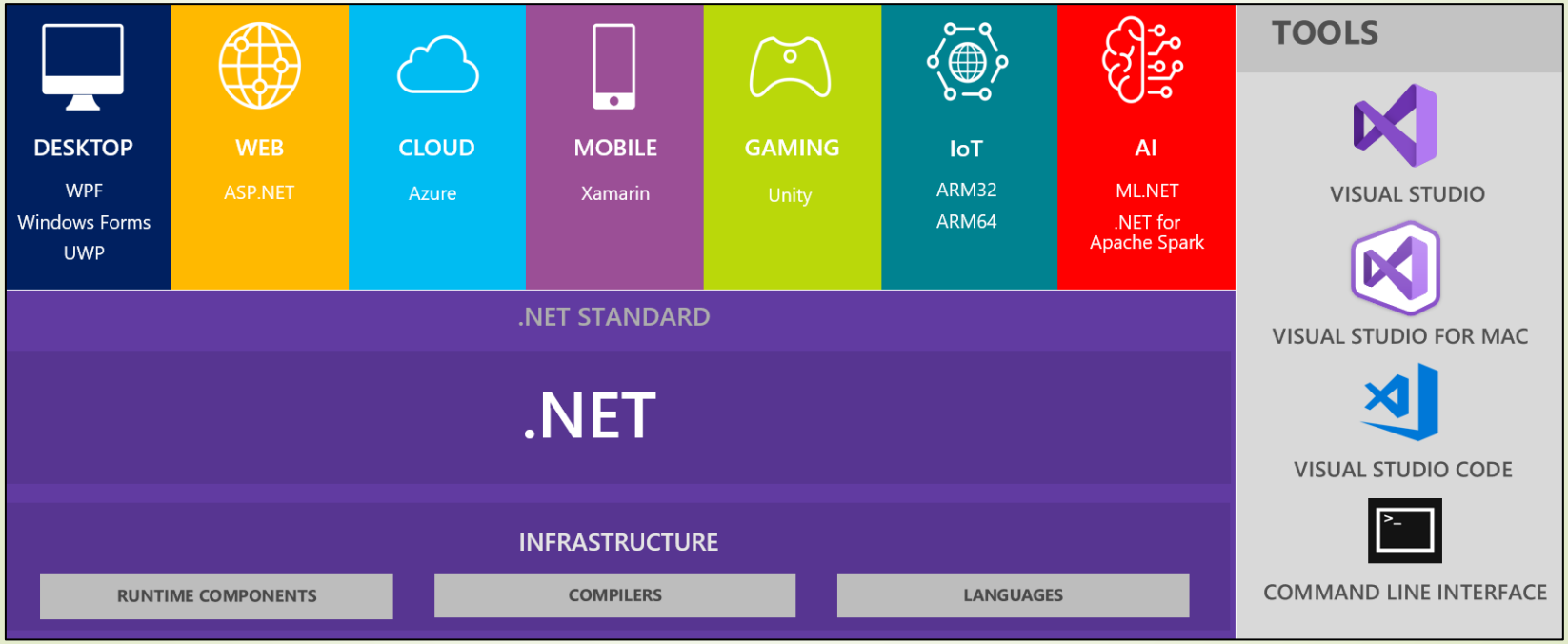
GUI applications with .NET Core 3+



Build systems – .NET

The future of .NET

- .NET Framework 4.8 is the final version.
- After .NET Core 3.1, the next version was named .NET 5 to avoid versioning confusion.
  - Currently, .NET 8 is the latest LTS (*Long Term Support*) version, and .NET 9 is available with a shorter support period.



Build systems – .NET

Building under .NET Framework

- Microsoft's build tool for Visual Studio projects.
  - It compiles based on *.sln* solution and *.csproj/.vcxproj* project files.
  - The project files are in XML format and can contain compilation configurations and other settings in addition to the project files.
- For example:

```
msbuild SolutionFile.sln /t:Build /p:Configuration=Release
```

  - for the solution specified in the *SolutionFile.sln*
  - executing the *Build* target
  - using the *Release* configuration
- MSBuild's implementation under the Mono project is *Xbuild.*

Build systems – .NET

Building under .NET Core

- For the cross-platform .NET Core, the SDK can be accessed using the *dotnet* CLI command.
  - Uses *MSBuild* internally, which has become cross-platform meanwhile.
- Projects are still split into solutions (*.sln* file) and projects *(.csproj/.vcxproj* files), which are in XML format.
- Installation on Debian/Ubuntu: `apt-get install dotnet`
- Example of using the .NET Core build system:
  - Restore NuGet packages: `dotnet nuget restore`
  - Build: `dotnet build path/to/SolutionFile.sln`
    For the current directory: `dotnet build`
  - Build using the *Release* configuration:
    `dotnet build --configuration Release`
  - Run: `dotnet run path/to/SolutionFile.sln`
    For an already built binary: `dotnet path/to/Binary.exe`

Build systems – .NET

Building under .NET Core

- The output of a .NET application build may not include all necessary files for running the application.
  - Deployment (publishing) refers to copying all compiled binaries, configuration files, and dependencies to the installation location:
    - This can be a simple output folder or, in the case of a web application, directly onto a web server.
  - Can be performed via Visual Studio or through CLI:
    `dotnet publish -c Release -o out_dir`
  - After this, the application can run even without the full SDK, only needing the .NET Runtime.
    - By using the `-s` option, *self-contained* deployment is possible, which includes the necessary components of the .NET Runtime.

Build systems – .NET

ELTE | IK
INFORMATIKAI KAR

Recommended structure for seminar projects

- Model project: UI-independent, implemented as a .NET Standard library. Can be referenced by .NET Core or other framework projects.
- View project: specific (WinForms, WPF or Avalonia UI) project using the .NET 8 LTS version. Can be implemented using *Desktop pack* but will still be platform specific (Windows).
- Test project: .NET 8 project, allowing Continuous Integration (CI) for the model on Linux as well. *MSTest*, *NUnit*, or *xUnit* frameworks can be used.

- Sample project: console thesis number generator application
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-net

Software technology

Build systems

Specialization C

Dr. Máté Cserép

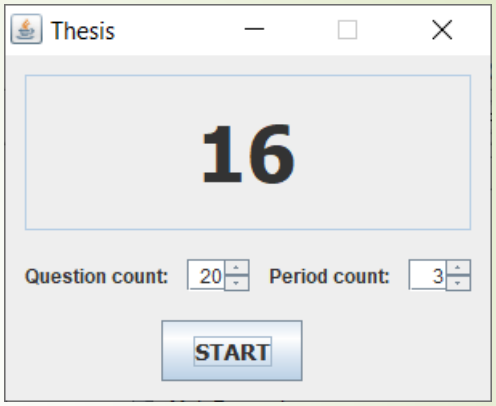
ELTE, Faculty of Informatics

2025.

Build systems

Sample application

- Let's have a sample Java application.
- The *ThesisGenerator* application can generate thesis serial number for a verbal examination.
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-java
- Model-View architecture:
  - **`thesisGenerator.model`** package: UI independent business logic
  - **`thesisGenerator.view`** package: Swing based UI
  - **`thesisGenerator`** package: Main program



Build systems

Compiling Java programs

```
mkdir dist
javac -d dist
  src\thesisGenerator\*.java
  src\thesisGenerator\model\*.java
  src\thesisGenerator\view\*.java


cd dist
jar -cfe thesis-generator.jar
  thesisGenerator.ThesisGenerator
  thesisGenerator\*.class
  thesisGenerator\model\*.class
  thesisGenerator\view\*.class


java -jar thesis-generator.jar
```

Build systems

Compiling Java programs

- That was not simple for such a basic program with a few source files all together.

- **Problem statement:**
  - Compiling programs manually with console commands can easily get difficult to manage even for smaller applications with a couple source files.
  - Working with larger programs is becomes untrackable which translation units require recompilation.
    - Recompiling the complete application can take a long time for an enterprise application.

Build systems

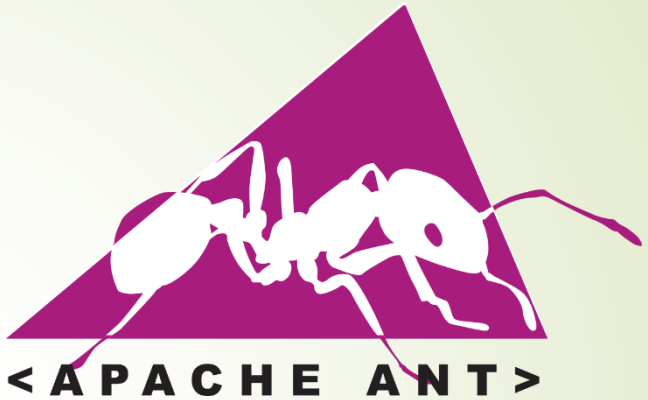
Requirements towards build systems

- Compiling the code
  - Manage dependencies of compilation targets
- Packaging the binaries
  - Support multiple release options
- Perform automatized tests
- Deploying the binaries to the test server
- Copying the code from one location to another

- Management of package repositories

Build systems - Ant

Features



- Imperative approach
- Typically used for Java projects
- XML-based build file
  - Named *build.xml* by default
- Official website and tutorial:
  - https://ant.apache.org/
  - https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- Installation:
  - Windows installer can be downloaded from official website.
  - UNIX systems: from package repository.
    - Debian/Ubuntu: `apt-get install ant`
  - Together with an IDE, e.g. NetBeans installs Ant.

Build systems - Ant

Build file (build.xml)

- The *build.xml* file contains a *project* root element, which sets:
  - Name of the project
  - Default target (discussed later)
  - The base directory of the project, typically the current folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="projname"
         default="deftarget"
         basedir=".">
   ...
</project>
```

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Define a target

- Inside the project element multiple type of elements can be defined. The most important are *targets*:

```
<project ...>
  <target name="compile">
    ...
  </target>
</project>
```

- Targets can be executed on the command line:
  `ant` **`compile`**

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Directory creation

- Create a target which creates the *classes* directory for the *.class* files to be compiled:

```
<project ...>
  <target name="prepare">
    <mkdir dir="classes"/>
  </target>
</project>
```

- Command line:
  `ant` **`prepare`**

Build systems - Ant

Target dependencies

- Define a target to compile all Java source files in the *src* directory. Place the output inside the *classes* folder.
  - Make the *compile* target depend on the *prepare* target!

```
<project ...>
  <target name="compile" depends="prepare">
    <javac srcdir="src" destdir="classes"/>
  </target>
</project>
```

  - Command line:
    `ant` **`compile`**

Build systems - Ant

Cleanup target

- Add a cleanup target, which removes all compilation binaries.

```
<project ...>
  <target name="clean">
    <delete>
      <fileset dir="classes" includes="*"/>
    </delete>
    <delete dir="classes"/>
  </target>
</project>
```

- Deleting the files are not required, removing the classes folder removes its content recursively.
- Command line:
  `ant` **`clean`**

Cleanup target

- The `failonerror` attribute configures the target whether to fail the complete process if that target fails.

```
<project ...>
  <target name="clean" failonerror="false">
    <delete dir="classes"/>
  </target>
</project>
```

- Command line:
  `ant` **`clean`**

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Properties

- Inside a project we can also defines properties.
  - Properties are key-value pairs.
  - Evaluated at runtime with the `${name}` syntax.

```
<project ...>
  <property name="jarname"
            value="filename.jar" />
  ...
</project>
```

  - There are also built-in properties, e.g. the `${basedir}` is the base project directory.
    - https://ant.apache.org/manual/properties.html

Build systems - Ant

Packaging

```
<project ...>
  <target name="jar" depends="compile">
    <jar destfile="${jarname}">
      <fileset dir="classes">
        <include name="*.class"/>
      </fileset>
      <manifest>
        <attribute name="Main-Class" value="Main"/>
      </manifest>
    </jar>
  </target>
</project>
```

- **Note:** it is important to set the entry point in the manifest!

Build systems - Ant

Target: complex example

```
<target name="compile" depends="prepare,init">
  <javac destdir="build/classes" debug="on">
    <src path="src1/java"/>
    <src path="src2/java"/>
    <include name="**/*.java"/>
    <exclude name="com/comp/xyz/applet/*.java"/>
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </javac>
</target>
```

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Deploying (file operations)

- Deploy by copying the final binaries to a target destination:

```
<target name="install" depends="jar">
  <mkdir dir="build/war/WEB-INF/lib"/>
  <copy todir="build/war/WEB-INF/lib">
    <fileset dir="lib">
      <include name="*.jar"/>
      <exclude name="servlet-api.jar"/>
      <exclude name="catalina-ant.jar"/>
      <exclude name="el-api.jar"/>
    </fileset>
  </copy>
</target>
```

  - Command line:
    `ant` **`install`**

Build systems - Ant

JVM launch

- A target for executing the compiled and packaged JAR file can also be defined:

```
<target name="run" depends="jar">
  <java jar="${jarname}" fork="true" />
</target>
```

  - The `fork` attribute causes the task to run in a different process, and a different Java virtual machine (JVM).
  - Command line:
    `ant` **`run`**

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

JVM launch

- More complex example:

```
<target name="run">
  <java classname="com.comp.foo.TestClient"
        jvmargs="-Xdebug server=y,suspend=n">
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </java>
</target>
```

  - Command line:
    `ant` **`run`**

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Generating API documentation

```
<target name="doc">
  <tstamp>
    <format property="timestamp" pattern="d.M.yyyy"
            locale="en"/>
  </tstamp>
  <mkdir dir="doc"/>
  <javadoc sourcepath="src" destdir="doc"
           windowtitle="Project documentation">
    <header>Very Important Project</header>
    <footer>Javadocs compiled ${timestamp}></footer>
  </javadoc>
</target>
```

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Executing unit tests

```
<target name="test" depends="compile">
  <mkdir dir="report"/>
  <junit printsummary="yes" haltonfailure="no">
    <classpath location="build/classes" />
    <classpath location="build/test/classes" />
    <classpath location="lib/junit-4.12.jar" />
    <classpath location="lib/hamcrest-core-1.3.jar" />

    <formatter type="plain" />
    <test name="com.comp.foo.ServerTest"
          haltonfailure="no" todir="report" />
  </junit>
</target>
```

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Executing unit tests

```
<target name="test" depends="compile">
  <mkdir dir="report"/>
  <junit printsummary="yes" haltonfailure="no">
    <classpath location="build/classes" />
    <classpath location="build/test/classes" />
    <classpath location="lib/junit-4.12.jar" />
    <classpath location="lib/hamcrest-core-1.3.jar" />

    <formatter type="plain" />
    <batchtest fork="yes" todir="report">
      <fileset dir="test">
          <include name="**/*Test.java" />
      </fileset>
    </batchtest>
  </junit>
</target>
```

Build systems - Ant

ELTE | IK
INFORMATIKAI KAR

Complete build.xml file for the ThesisGenerator app

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="ThesisGenerator" default="jar" basedir=".">
    <target name="clean">
        <delete dir="build"/>
    </target>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/ThesisGenerator.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="thesisGenerator.ThesisGenerator"/>
            </manifest>
        </jar>
    </target>
    <target name="run" depends="jar">
        <java jar="build/jar/ThesisGenerator.jar" fork="true"/>
    </target>
    <target name="doc" depends="compile">
      <tstamp>
        <format property="timestamp" pattern="d.M.yyyy" locale="en"/>
      </tstamp>
      <mkdir dir="doc"/>
      <javadoc sourcepath="src" destdir="doc" windowtitle="Thesis Generator">
        <header>Thesis Generator</header>
        <footer>Javadocs compiled ${timestamp}></footer>
        <fileset dir="src/" includes="**/*.java" />
      </javadoc>
    </target>
    <target name="test" depends="compile">
      <mkdir dir="report"/>
      <junit printsummary="yes" haltonfailure="no">
        <classpath location="lib/junit-4.12.jar" />
        <classpath location="lib/hamcrest-core-1.3.jar" />
        <classpath location="build/classes" />
        <classpath location="build/test/classes" />
        <formatter type="xml" />
        <formatter type="plain" />
        <batchtest fork="yes" todir="report">
          <fileset dir="test">
              <include name="**/*Test.java" />
          </fileset>
        </batchtest>
      </junit>
    </target>
</project>
```

Build systems - Ant

NetBeans

- By default Netbeans uses Ant as a build system.
  - *build.xml* is located in the project root
  - references *nbproject/build-impl.xml*, which is generated by Netbeans and shouldn't be modified
  - Targets as hooks can be definied in *build.xml*, which will be called by Netbeans's build process automatically:
    - *-pre-init*, *-post-init*, *-pre-compile*, *-post-compile*, *-pre-jar*, *-post-jar*, *-post-clean*, etc.

Build systems - Maven

Features

- Software project management tool
- Building project, running tests, managing dependencies, documentation
- Packages: automatic download of dependencies
- Declarative specification of the build process
- Fix, predefined directory structure, conventions
- Typically Java, but it can handle other language via plugins
- XML-based build file
  - Named *pom.xml* by default
- Official website and a recommended tutorial:
  - http://maven.apache.org/
  - https://www.baeldung.com/maven

Build systems - Maven

Installation

- Standalone installation
  - Binaries are available on official website for download
  - Simply extract it to a preferred location
  - Set the `MAVEN_HOME` env. variable to point to this location
  - Also ensure that the `JAVA_HOME` env. variable points to your JDK installation folder
  - Add the `MAVEN_HOME\bin` folder to your `PATH`.
- UNIX package repository installation
  - Usually available
  - Debian/Ubuntu: `apt-get install maven`
- Typically ships bundled with IDEs (e.g. NetBeans, IntelliJ)

Build systems - Maven

Project

- Project Object Model (POM)
- Project uniquely identified by project's group, artifact Id, version, the 3 abbreviated as GAV together
- Project can be divided into multiple modules that can be handled independently

Build systems - Maven

Project Object Model (pom.xml)

- The Project Object Model (pom.xml) is a specification of the project's all important information:
  - identifiers: groupId, artifactId, version
  - how the project is built
  - result of the build
  - test cases for the project
  - dependencies of the project

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Project Object Model (pom.xml)

- The root element of the pom.xml file is also a `project` element.
- The following elements must be defined inside the project:
  - `modelVersion`: version of the POM specification
  - `groupId`: unique base name of the company or group that created the project. Group ID should follow Java's package name rules. This means it starts with a reversed domain name, e.g. *hu.elte.inf*
  - `artifactId`: unique name of the project
  - `version`: version of the project
  - `packaging`: applied packaging method (default is jar, other options: pom, maven-plugin, ejb, war, ear, rar, par)

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Project Object Model (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.software</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
</project>
```

Build systems - Maven

Directory structure

- A Maven project has a directory structure based on defined conventions.
  - The default directory layout can be overridden using project descriptors, but this is uncommon and discouraged.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Directory structure override

```
<project>
   ...
   <build>
     <directory>target</directory>
     <outputDirectory>classes</outputDirectory>
     <finalName>${project.artifactId}-${project.version}</finalName>
     <testOutputDirectory>test-classes</testOutputDirectory>
     <sourceDirectory>src/main/java</sourceDirectory>
     <scriptSourceDirectory>src/main/scripts
     </scriptSourceDirectory>
     <testSourceDirectory>/src/test/java</testSourceDirectory>
     ...
   </build>
</project>
```

Build systems - Maven

Lifecycle phases

- Maven build system follows a specified *lifecycle*, consisted of phases. The most important phases of the default lifecycle:
  - **validate:** validate the project is correct and all necessary information is available
  - **compile:** compile the source code of the project
  - **test:** test the compiled source code using a suitable unit testing framework. These tests should not require the code to be packaged or deployed
  - **package:** take the compiled code and package it in its distributable format, such as a JAR.
  - **integration-test:** process and deploy the package if necessary into a testing environment where additional integration tests can be run

Build systems - Maven

Lifecycle phases

- **verify:** run any checks to verify the package is valid and meets quality criteria
- **install:** install the package into the local repository, for use as a dependency in other projects locally
- **deploy:** done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

- Further details: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
- Command line: `mvn install`
  - Performs the phases until *install* in the default lifecycle, but not the *deploy* phase.

Lifecycle phases

- Beside the default lifecycle, there are other lifecycles, e.g. the clean lifecycle, which can be used to purge previously built binaries from a project.
  - This lifecycle has 3 phases: pre-clean, clean, post-clean.
- Command line: `mvn clean install`
  - Performs the *clean* and then the *install* phases and all phases before them.
  - Ultimately this will remove and rebuild all binaries.

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Goals

- Compilation phases consist of one or multiple *goals*
- The goal is a task that is related to the project's compilation or management
  - The order of these goals depends on the phase's binding
  - Many phases contain only one goal
  - E.g. `compile` phase consists of the `compiler:compile` goal
- Not only phases, but goals can also be passed to the Maven command, peforming only that goal without the previous phases and their goals.
  - E.g. `mvn compiler:compile`
- Custom phases and goals can be defined (in the pom.xml)

Build systems - Maven

Repositories

- A repository holds build artifacts and dependencies.
  - The default local repository in the developer's home folder: *~/.m2/repository*
- If an artifact is available in the local repository, Maven uses it.
- Otherwise, it is downloaded from a central repository and stored in the local repository.
  - Network traffic and build time can be significantly increased for the first build of a project.
  - The default central repository is the Maven Central: https://repo.maven.apache.org
  - Maven can be configured to which repositories and mirrors to use in the *~/.m2/settings.xml* file.
    - Companies often have internal central repositories.

Build systems - Maven

Result of build process

- *target* directory is created during compilation which stores the new files that were generated at compilation time
  - output, e.g. *my-app-1.0-SNAPSHOT.jar*
  - *classes* directory: class files that were created during compilation but not test classes
  - *test-classes*: classes created from test sources
  - *maven-archiver/pom.properties* file that defines the project's GAV
  - *surefire-reports:* reports of the tests

Build systems - Maven

Project hierarchies

- Maven supports submodule projects:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>parent-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <!-- subprojects -->
  <modules>
    <module>first-child-app</module>
    <module>second-child-app</module>
  </modules>
</project>
```

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Project hierarchies

- Submodule projects also reference their parents:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>parent-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.mycompany.app</groupId>
  <artifactId>first-child-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  ...
</project>
```

Build systems - Maven

Plugins

- Maven's functionality itself is limited to the basic, but it is a pluginable framework.
- Many different plugins are available
  - e.g. C++, LaTeX, ant build, javadoc, etc.
  - The official plugins are listed on their website: https://maven.apache.org/plugins/
- There are also 3rd party plugins and one can write own plugin

Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Plugin example: Javadoc

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

- Generate documentation: `mvn javadoc:javadoc` or `mvn:site`

Build systems - Maven

Dependencies

- The external libraries that a project uses are called dependencies.
- The dependency management feature in Maven ensures automatic download of those libraries from a central repository.

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- GAV uniquely specifies the required artifact
- scope defines how we use the dependency

Build systems - Maven

Dependencies' scope

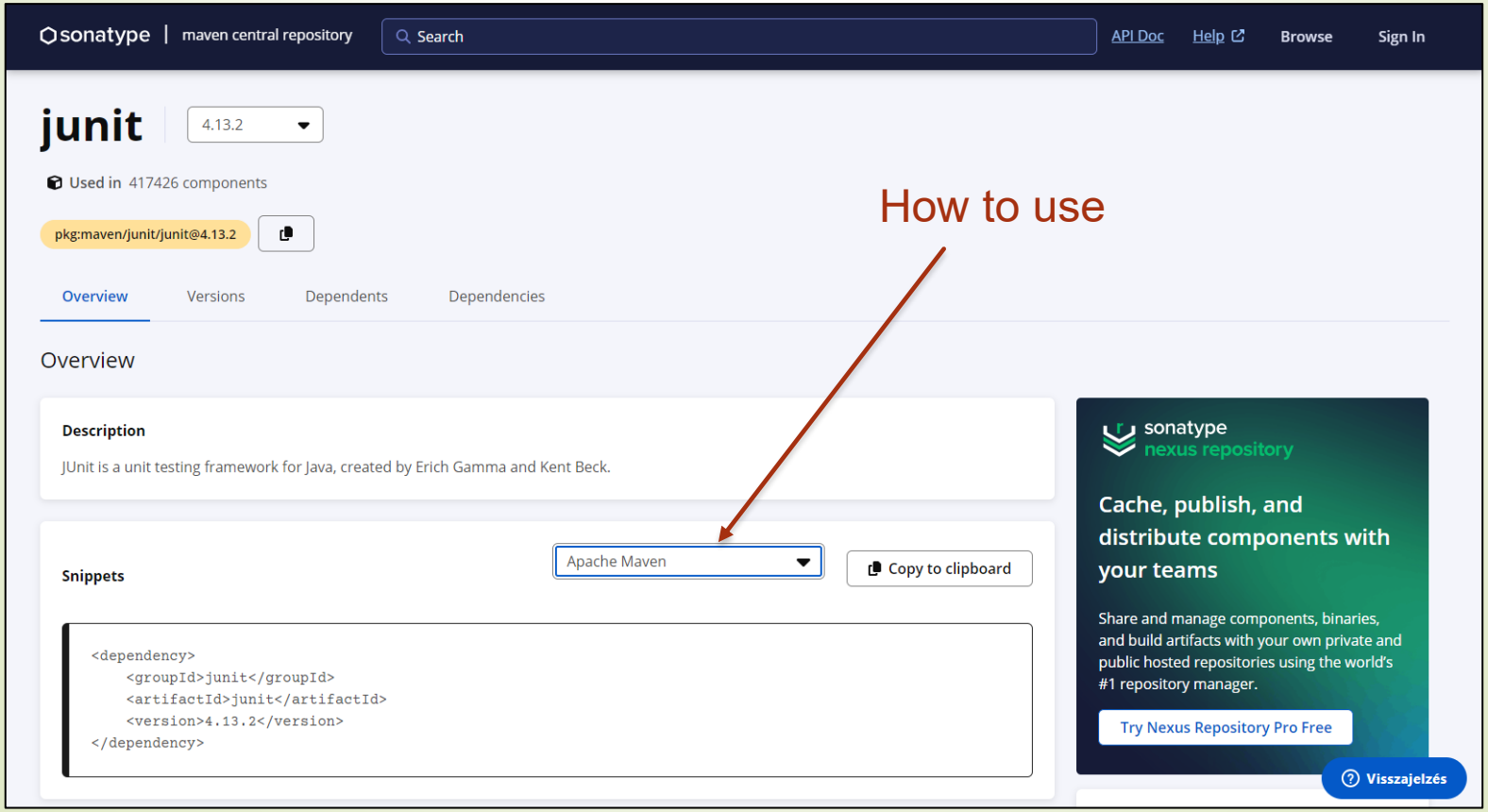
- The most important scopes:
  - **compile:** This is the default if unspecified. Dependencies that required by the compilation
  - **runtime:** Dependency required at runtime, but not required at compilation time.
  - **test:** Dependency is not required in production but it is required for the compilation and execution of testcases.

Build systems - Maven

Search dependencies

- One can browse and search the available libraries in the Maven Central repository:
  - https://search.maven.org/

New default URL and UI:
https://central.sonatype.com/



Build systems - Maven

ELTE | IK
INFORMATIKAI KAR

Complete pom.xml file for the ThesisGenerator app

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>hu.elte.inf</groupId>
    <artifactId>thesis-generator</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>15</maven.compiler.source>
        <maven.compiler.target>15</maven.compiler.target>
        <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
        <maven-project-info-reports-plugin.version>3.0.0</maven-project-info-reports-plugin.version>
        <maven-javadoc-plugin.version>3.2.0</maven-javadoc-plugin.version>
        <junit.version>4.13.2</junit.version>
    </properties>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>${maven-jar-plugin.version}</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <mainClass>thesisGenerator.ThesisGenerator</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Build systems - Maven

Complete pom.xml file for the ThesisGenerator app

```
38      <reporting>
39          <plugins>
40              <plugin>
41                  <groupId>org.apache.maven.plugins</groupId>
42                  <artifactId>maven-project-info-reports-plugin</artifactId>
43                  <version>${maven-project-info-reports-plugin.version}</version>
44              </plugin>
45              <plugin>
46                  <groupId>org.apache.maven.plugins</groupId>
47                  <artifactId>maven-javadoc-plugin</artifactId>
48                  <version>${maven-javadoc-plugin.version}</version>
49                  <configuration>
50                      <doctitle>My API for ${project.name} ${project.version}</doctitle>
51                      <windowtitle>My API for ${project.name} ${project.version}</windowtitle>
52                      <show>public</show>
53                  </configuration>
54              </plugin>
55          </plugins>
56      </reporting>
57      <dependencies>
58          <dependency>
59              <groupId>junit</groupId>
60              <artifactId>junit</artifactId>
61              <version>${junit.version}</version>
62              <scope>test</scope>
63          </dependency>
64      </dependencies>
65  </project>
```

Build systems - Gradle

Features

- Build automation system with increasing popularity [1] [2]
- Aims to merge the best concepts from Ant and Maven
- Supports incremental builds
  - Major performance boost for larger enterprise projects
- Configuration through a Groovy or Kotlin based domain-specific language (DSL) instead of XML
- Official website: https://gradle.org/
- Tutorial: https://docs.gradle.org/current/userguide/getting_started_eng.html

[1] https://www.baeldung.com/java-in-2019

[2] https://www.jetbrains.com/lp/devecosystem-2023/java/

Build systems - Gradle

build.gradle

- In Gradle, the build system is defined in the *build.gradle* file written in Groovy. (For Kotlin use *build.gradle.kts*)
  - The naming is conventional for default usage.
- How to create a *build.gradle* file?
  1. Write from scratch
  2. Use the `gradle init` command to prepare one
  3. Use your preferred IDE to generate one

Build systems - Gradle

Tasks

- In Gradle, builds consist of one or more projects and each project consists of one or more tasks.
  - A task is a single piece of work, e.g. compiling classes, creating archives or publishing them.
  - A simple task can be defined as:

```
task hello {
  doLast {
    println "Hello World"
  }
}
```

Build systems - Gradle

Adding behavior to existing tasks

- Existing tasks can be extended through prefixing or suffixing them with further behavior.

```
task hello {
  doLast {
    println "Hello World"
  }
}

hello.doFirst {
  println "First Line"
}

hello.doLast {
  println "Last Line"
}
```

Build systems - Gradle

Task dependencies

- Dependency relationship can be defined between tasks with the *dependsOn* argument.

```
task init {
  doLast {
    ...
  }
}

task compile(dependsOn: init) {
  doLast {
    ...
  }
}
```

Build systems - Gradle

Plugins

- Existing tasks can be listed with the `gradle tasks` command.
- To simply the build system configuration, Gradle ships with multiple useful plugins, which contains predefined tasks.
  - By utilizing the java plugin, we obtain tasks to build, test and package a general Java-based application.

```
plugins {
  id "java"
}
```

  - Some commands: `gradle assemble`, `gradle build`, `gradle jar`, `gradle test`, `gradle clean`

Build systems - Gradle

Plugins

- By utilizing the application plugin (which also implies the java plugin), we can even create an executable build.

```
plugins {
  id "application"
}
```

- The entry point for the manifest still needs to be defined.

```
application {
  mainClass = "project.MainClass"
}
```

Build systems - Gradle

Dependencies

- Gradle supports a flexible dependency management system.
- Gradle leans heavily on many conventions and facilities established by the Maven, including the option of using Maven Central as a source of library dependencies.

```
repositories {
  mavenCentral()
}
```

- Dependencies are grouped into different configurations, e.g. `implementation`, `testImplementation` or `runtimeOnly`, which can extend each other.

```
dependencies {
  testImplementation "junit:junit:4.13.2"
}
```

Build systems - Gradle

Complete build.gradle file for the ThesisGenerator app

```
plugins {
    // Apply the application plugin. Implies the java plugin.
    id 'application'
}
application {
    // Define the main class for the application.
    mainClass = 'thesisGenerator.ThesisGenerator'
}
repositories {
    // Use the Maven Central Repository for dependencies.
    mavenCentral()
}
dependencies {
    // Use JUnit test framework.
    testImplementation 'junit:junit:4.13.2'
}
// Set source directories. Advised to use the conventional directory layout in general.
sourceSets {
    main.java.srcDirs = ['src']
    test.java.srcDirs = ['test']
}
// Set build directory.
project.buildDir = 'build-gradle'

```

Build systems - Gradle

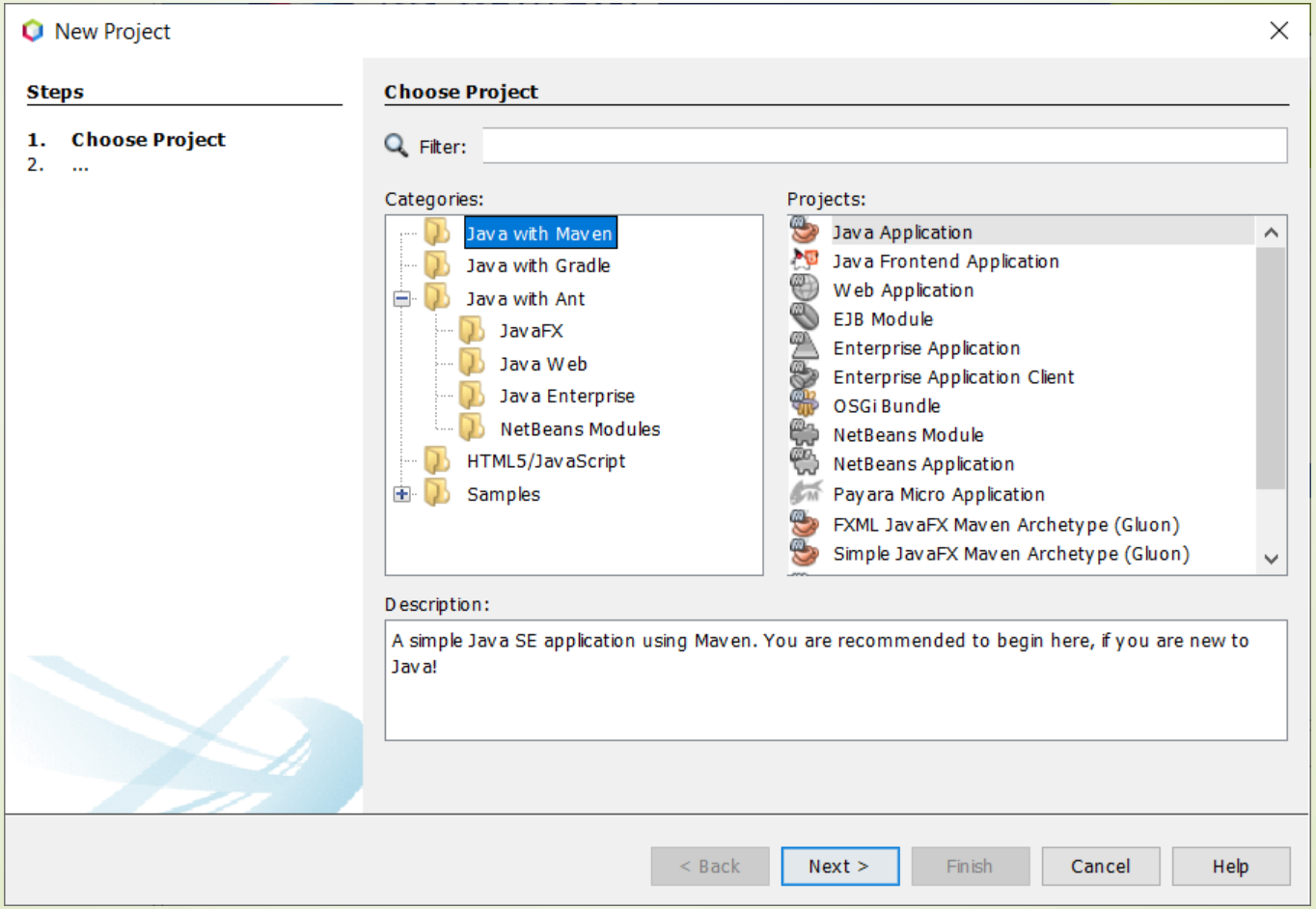
Gradle Wrapper

- Upon creating a project, *Gradle Wrapper* (gradlew) can be added. (Or later with the `gradle wrapper` command.)
  - The Gradle Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary.
- The recommended way to execute any Gradle build is with the help of the Gradle Wrapper. Benefits:
  - No need to manually install Gradle after project creation.
  - Fixed version for a project among developers (and CI).
  - Upgrade to newer Gradle version by simply changing the Wrapper definition.
- Version control: the `gradlew` Bash and Batch script along with the `gradle` folder should be version controlled. The `.gradle` folder containing the downloaded binaries should be not.

Build systems – IDE integration

NetBeans

- NetBeans supports both Ant, Maven and Gradle projects.

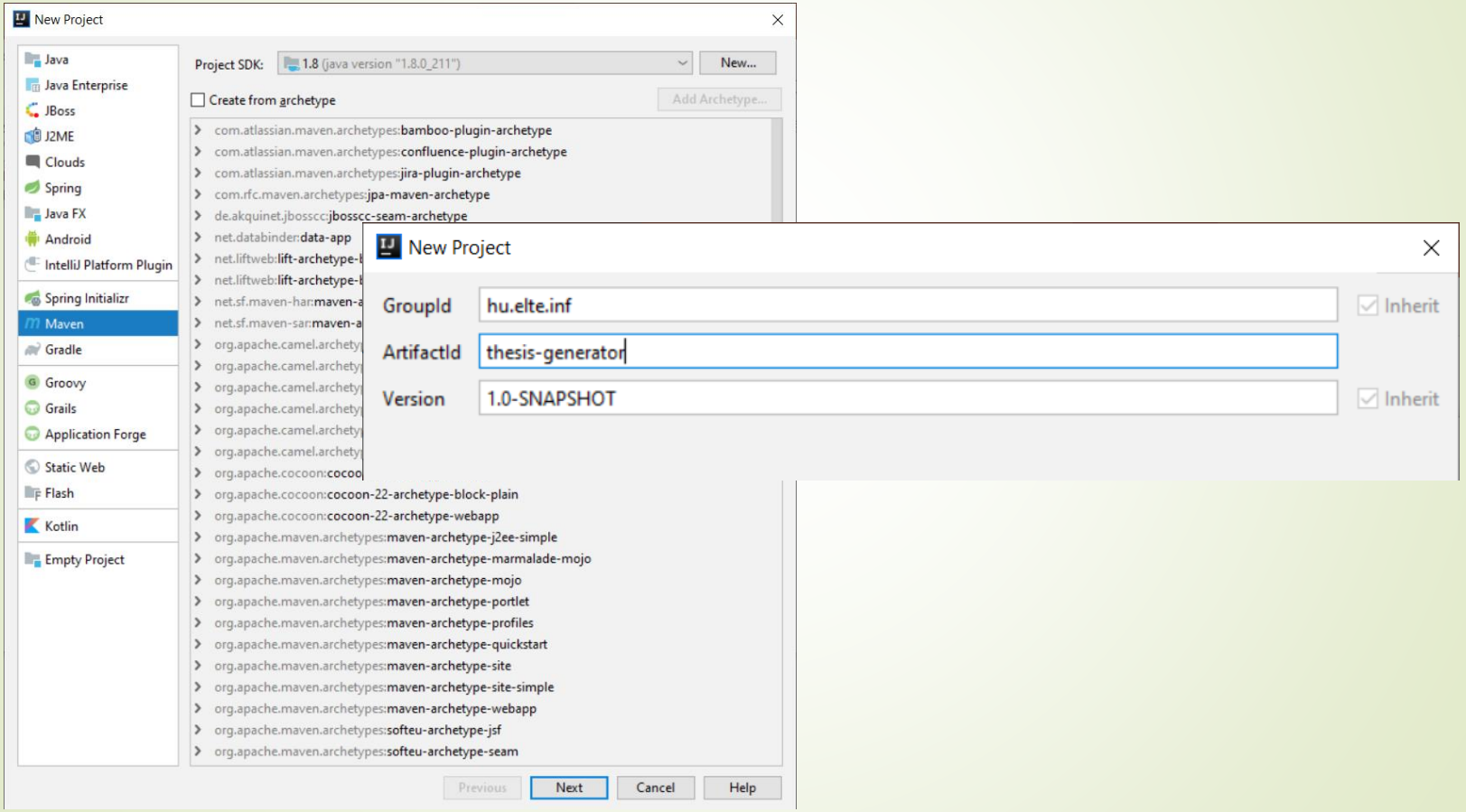


Build systems – IDE integration

ELTE | IK
INFORMATIKAI KAR

IntelliJ

- IntelliJ IDEA has a tight integration with Maven and Gradle.



- *Generate Ant build file is also supported: Build -> Generate Ant Build*

Software Technology

Continuous integration & delivery

Specializations A & B

Dr. Máté Cserép

ELTE Faculty of Informatics

2025.

Continuous integration & delivery

Continuous integration

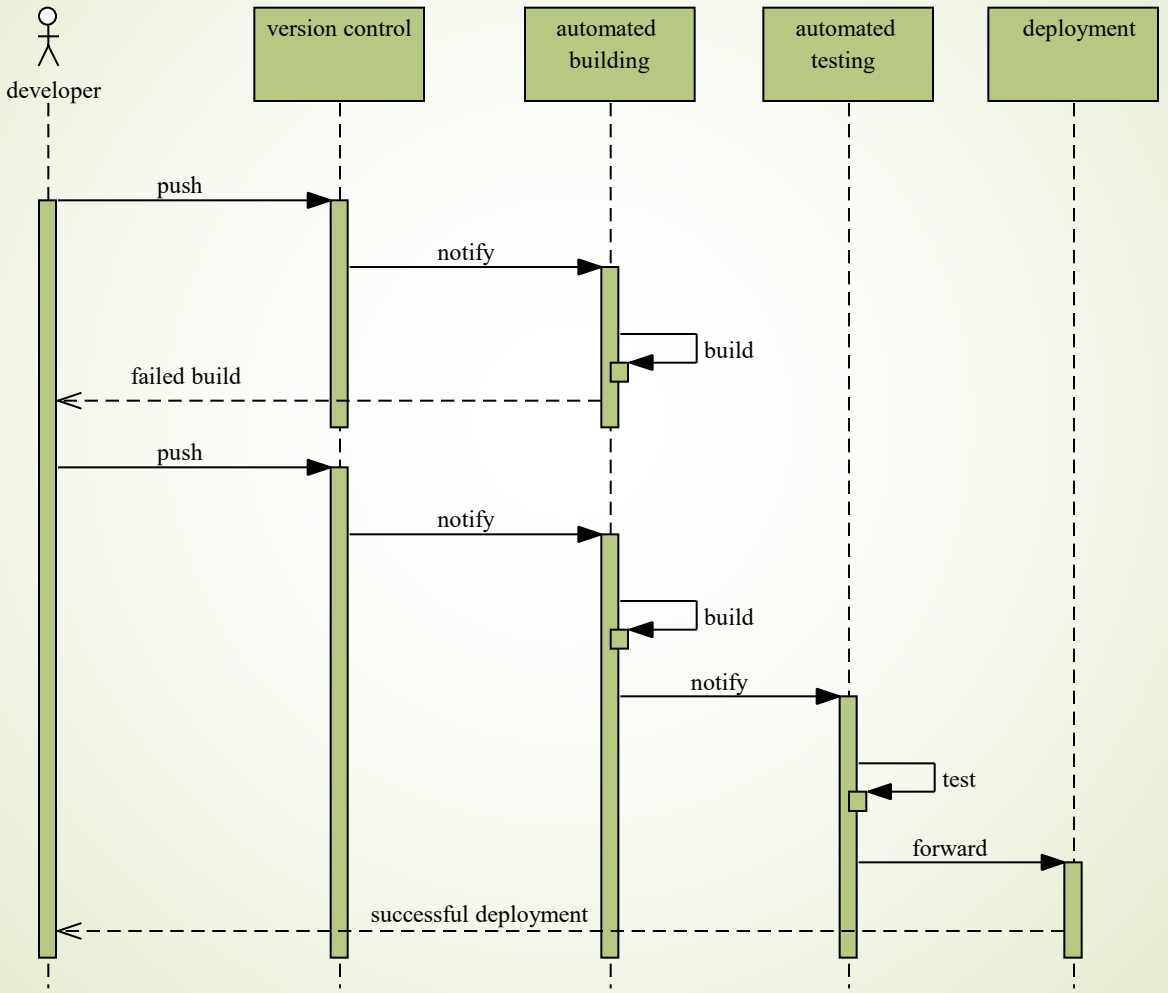
- *Continuous integration* (*CI*) is a practical method that accelerates the verification and testing of program code
  - its goal is to immediately and automatically detect potential errors and integration issues, providing feedback to developers
  - the source code is stored in a central repository using a version control system and is updated multiple times a day
  - after each modification, the repository's content is automatically built (*build automation*), and coded tests are executed along with the build process
  - the verified code can then undergo further testing

Continuous integration & delivery

Continuous delivery

- *Agile software development* aims to achieve rapid application development on an incremental basis
  - the software is continuously developed and released (*continuous delivery*), maintaining a steady pace while allowing changes to be incorporated at every step (*welcome changes*)
  - working software serves as the primary measure of progress, emphasizing simplicity while maintaining a focus on proper design and optimization
  - development is typically carried out by small, self-organizing teams with shared responsibility, continuous interaction, and rapid feedback cycles
  - continuous releases can be automated, which is known as *continuous deployment*

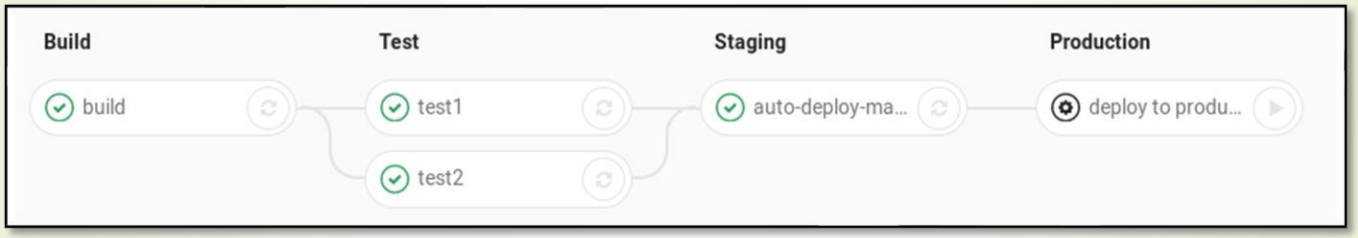
Continuous integration & delivery

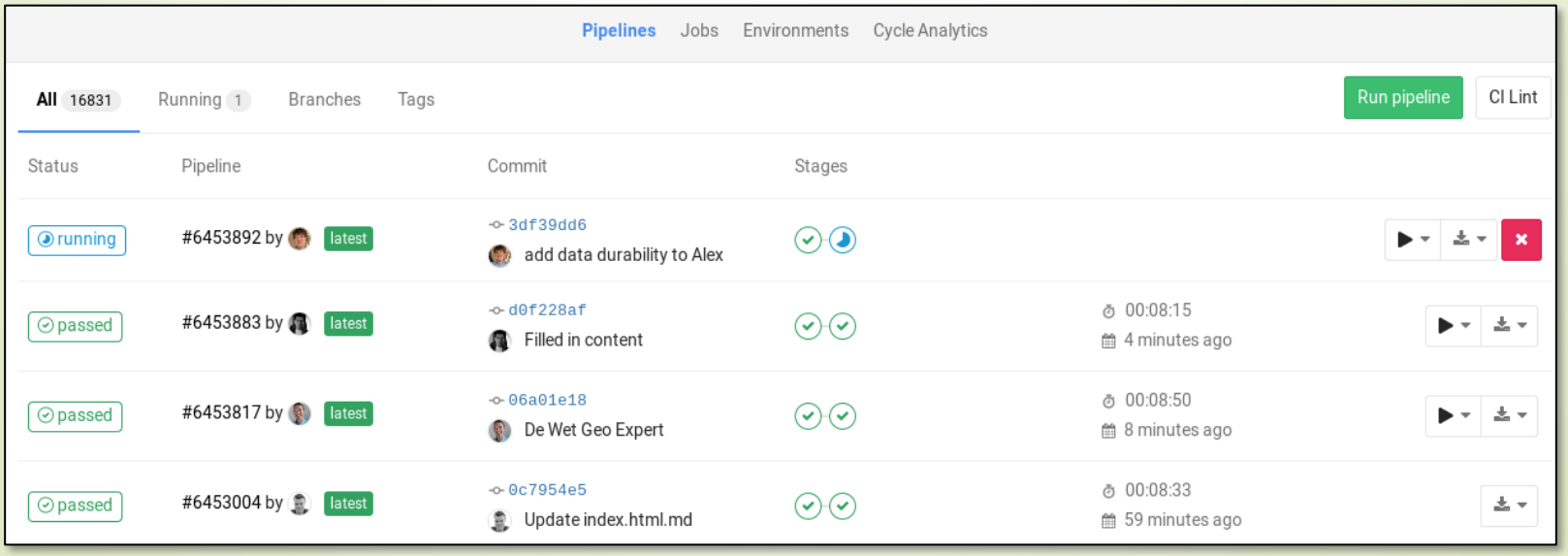


Continuous integration & delivery

Jobs

- The steps of continuous integration and delivery can be defined as *pipelines,* a chain of interdependent *jobs*

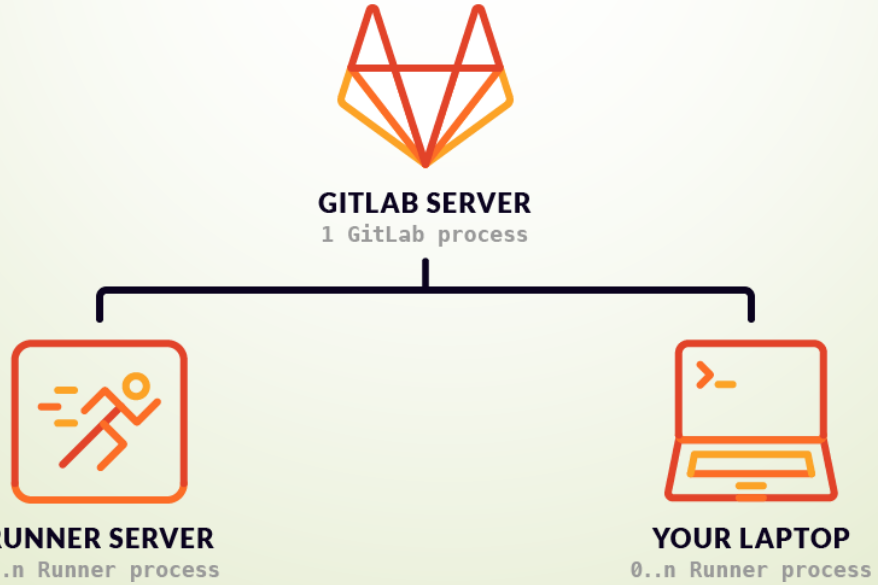




Continuous integration & delivery

GitLab Runners

- GitLab provides an integrated, built-in solution to support continuous integration and delivery
  - *jobs* are executed by so-called *GitLab Runner* instances, independent of the GitLab server
    - Shell, SSH, VirtualBox, Docker, Kubernetes, etc.
  - *runners* can be individually configured and can either be shared or assigned to specific projects



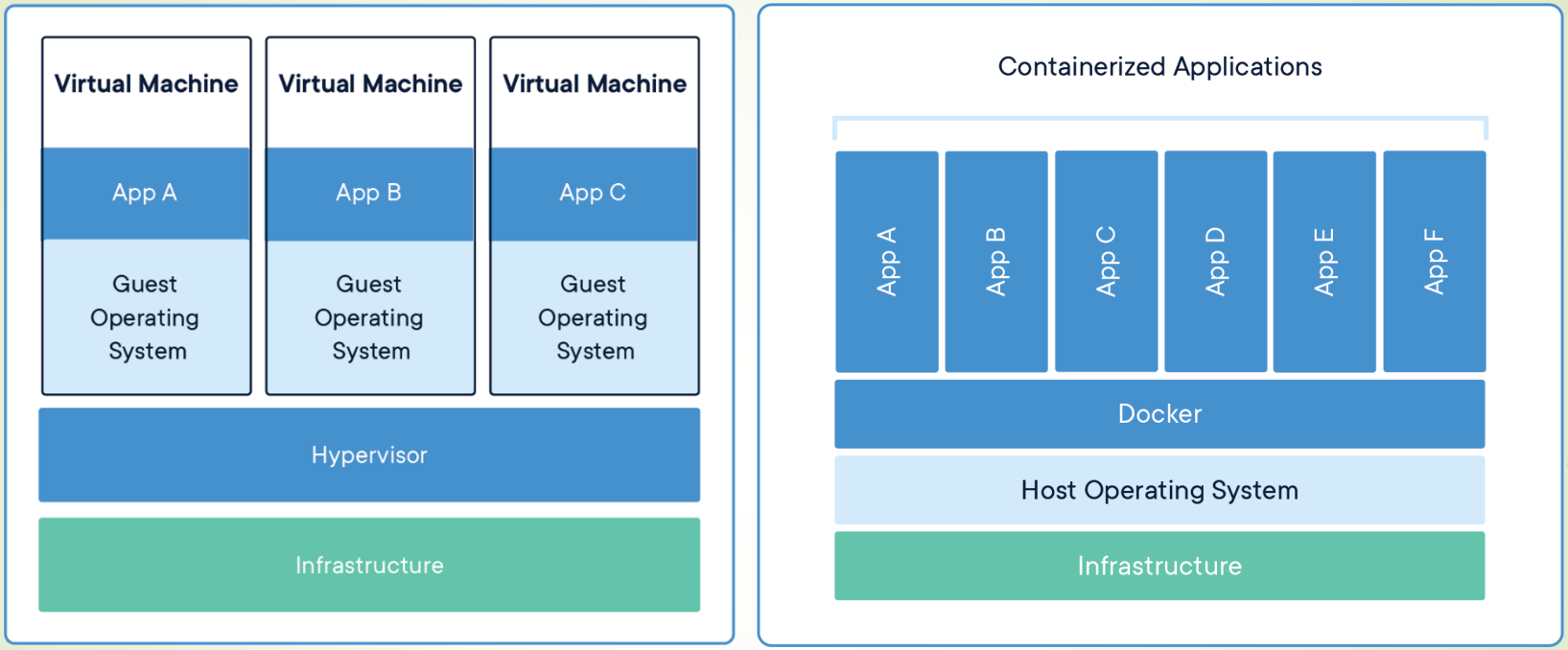
Continuous integration & delivery

Docker

- Continuous integration is best performed in an isolated, reproducible environment
- Docker is currently the most widely used *container framework* system
  - a *container* is similar to a virtual machine (*VM*) in that it provides a fully isolated, virtualized environment where the host machine supplies resources
  - the key difference between *containers* and virtual machines is that all *containers* share the host's kernel, virtualized hardware and an operating system are not included; instead, only the necessary libraries, binaries, and user space are part of the *container*
  - as a result, *containers* have significantly lower *overhead* compared to VMs, making them a more lightweight virtualization solution

Continuous integration & delivery

Virtual machines and *containers*

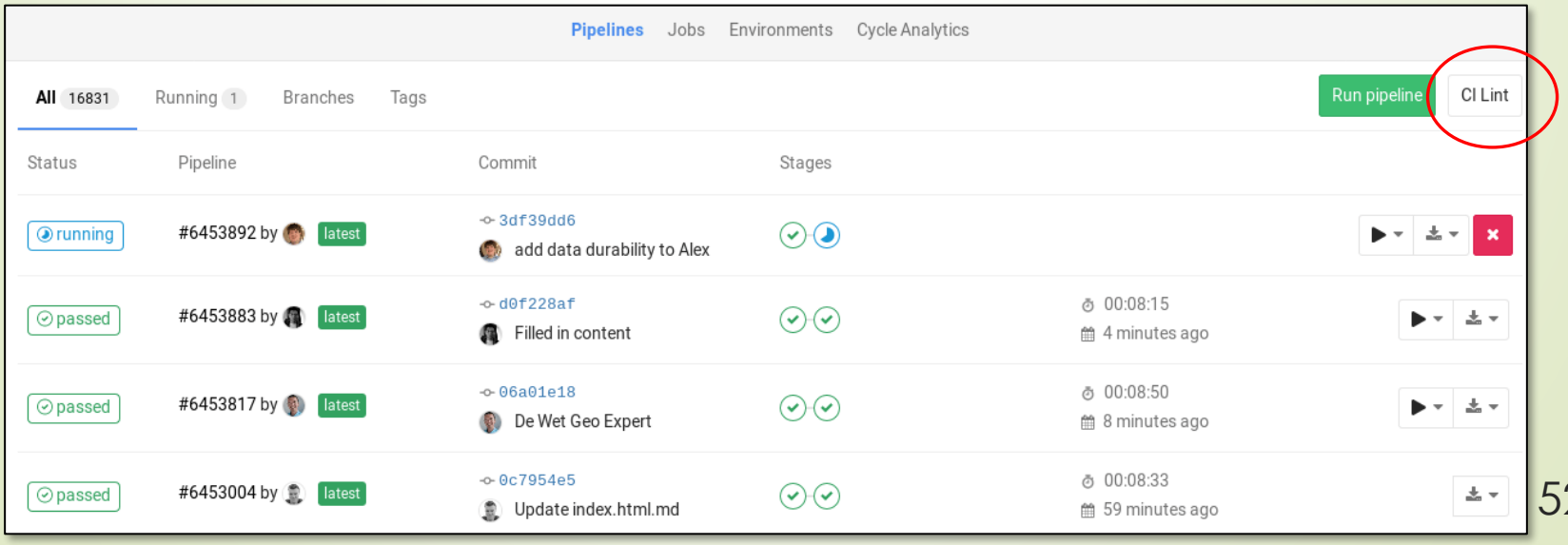


- *container frameworks* enable the creation and management of portable applications
- applications can be modularized and scaled, with components running in separate *containers*

Continuous integration & delivery

GitLab CI/CD

- The configuration for continuous integration is stored in the `.gitlab-ci.yml` file, written in YAML format
  - YAML (*YAML Ain't Markup Language*) is a structured descriptive language that is easily (or more easily) readable by humans
  - https://yaml.org/spec/1.2.2/
- GitLab's web interface includes a CI Lint tool for validating the format before submission



Continuous integration & delivery

YAML syntax

- Example YAML code:

```yaml
name: John Doe # key-value pairs
age: 42
details: # embedded collection
  givenname: John
  familyname: Doe
  birthyear: 1978
languages: # list of values (array)
  - Hungarian
  - English
  - German
# multi-line text
intro_multi: |
  multiple line
  introduction
intro_single: >
  single line
  introduction
```

GitLab CI/CD: jobs

- In the `.gitlab-ci.yml` file, we can define *jobs* specifying which commands they should execute (`script`)
- Example:

```
build_program:
  script:
    - apt-get update -qq
    - apt-get install -yqq build-essential
    - make
```

GitLab CI/CD: multiple jobs

- Multiple *jobs* can be defined, and a global `before_script` can be set to run before every *job*. (This can also be overridden within individual *jobs*.)

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  script:
    - mkdir build && cd build
    - cmake ..
    - make install

test_program:
  script:
    - mkdir build && cd build
    - cmake ..
    - make test
```

GitLab CI/CD: stages

- Continuous integration *jobs* can be divided into sequential *stages*
  - by default, there are 3 *stages*: *build*, *test*, *deploy*
  - this can be redefined as needed

```
stages:
  - lint
  - build
```

- *Jobs* within a *stage* can be run in parallel independently (using multiple *runners*)
  - *stages* are built upon each other: if a *job* in a *stage* fails, the dependent *stages* will not be executed

Continuous integration & delivery

GitLab CI/CD: stages

- Splitting the process into *stages:*

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make install

test_program:
  stage: test
  script:
    - mkdir build && cd build
    - cmake ..
    - make test
```

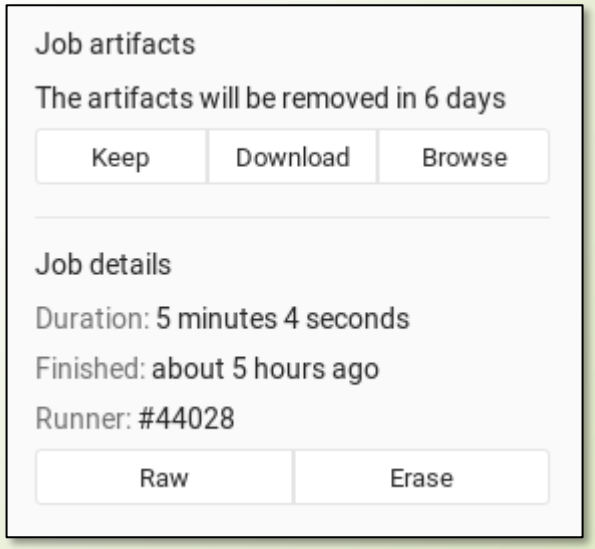
Continuous integration & delivery

GitLab CI/CD: artifacts

- Binaries or other files generated as a part of CI *jobs* can be preserved (*artifact*)

```
pdf:
  script:
    - pdftex paper.tex
  artifacts:
    paths:
      - paper.pdf
    expire_in: 1 month
```

- *Artifacts* can be easily downloaded from the GitLab web interface



Continuous integration & delivery

GitLab CI/CD: artifacts

- Defining *artifacts:*

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq build-essential cmake

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake .. -DCMAKE_INSTALL_PREFIX=../install
    - make install
  artifacts:
    paths:
      - install/
    expire_in: 1 week
```

Continuous integration & delivery

GitLab CI/CD: dependencies

- The verification of interdependent programs (modules) can easily lead to redundant execution:

```
before_script: ...

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make

test_program:
  stage: test
  script:
    - mkdir build && cd build
    - cmake ..
    - make
    - make test
```

GitLab CI/CD: dependencies

- Passing *artifacts* between *jobs:*

```
before_script: ...

build_program:
  stage: build
  script:
    - mkdir build && cd build
    - cmake ..
    - make
  artifacts:
    paths:
      - build/

test_program: # depends on the build output
  stage: test
  script:
    - cd build
    - make test
  dependencies:
    - build_program
```

Continuous integration & delivery

Docker images

- When using a Docker container based GitLab Runner, the base *Docker image* can be specified:

```
image: ubuntu:22.04
```

image name

tag name

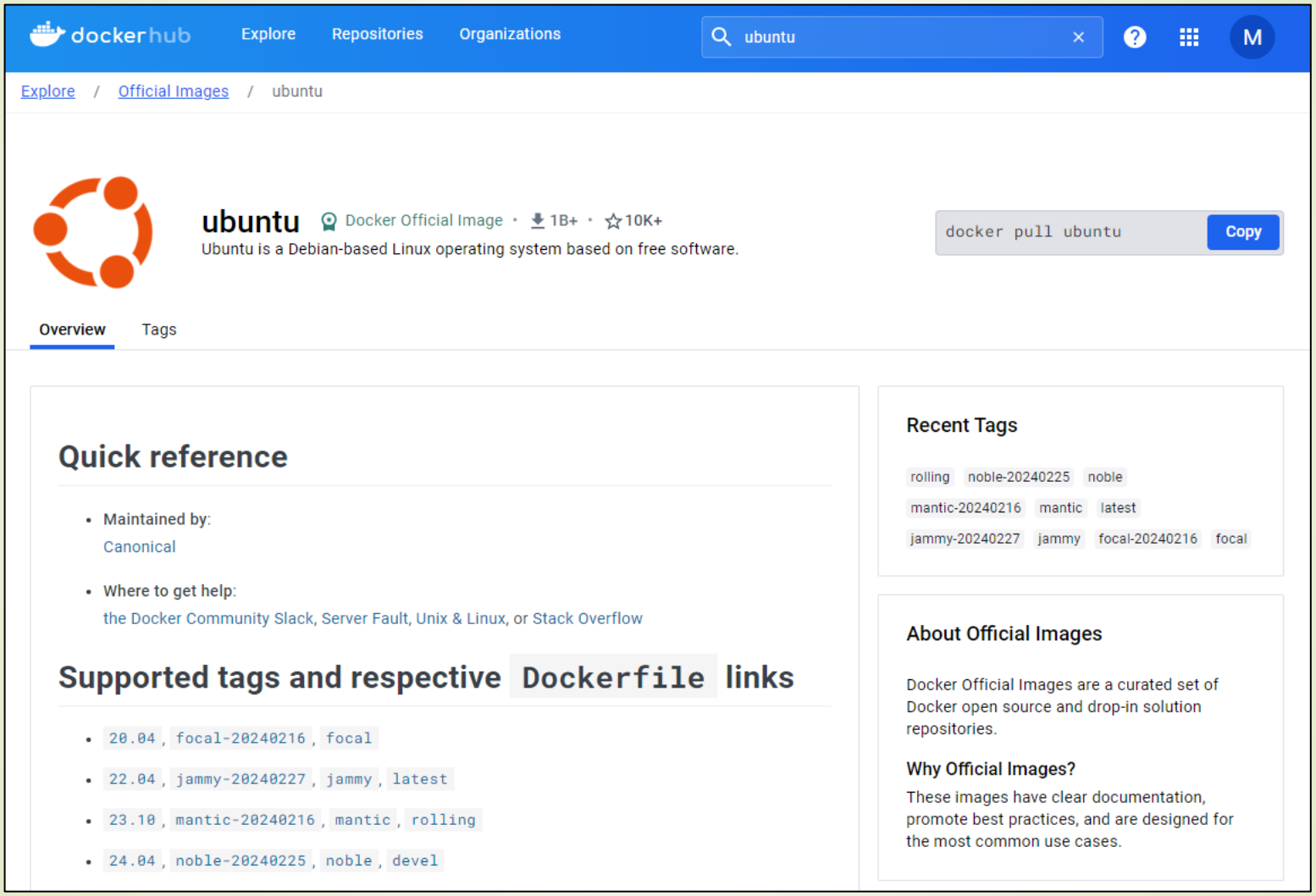
- A *Docker image* can be requested from a *Docker registry*:
  - By default, the public Docker Hub is used, where custom images can also be uploaded: https://hub.docker.com/
  - A private docker registry can also be used (e.g., in an enterprise environment)

```
image: mycompany.com:5000/custom:latest
```

- If not specified, the runner configuration determines the image to be used
  - the runners at szofttech.inf.elte.hu are configured to use `ubuntu:22.04`
  - For Windows runners, the default image is: `mcr.microsoft.com/windows/servercore:1809`

Continuous integration & delivery

Docker Hub



Continuous integration & delivery

GitLab CI/CD

- Additional options (not exhaustive):
  - `only, except:` conditional execution of CI jobs (e.g., run only on the *master* branch)
  - `when:` conditional execution of CI jobs (manual vs. automatic execution)

```
deploy_program:
  stage: deploy
  script:
    - ...
  only:
    - master
  when: manual
```

GitLab CI/CD

- `rules` provides advanced configuration options compared to `only` and `except`
  - `only, except` are considered deprecated, they are not developed with further capabilities
  - e.g. run only on master branch:

```
rules:
  - if: $CI_COMMIT_BRANCH == "master"
```

  - e.g. run only when *Dockerfile* changes:

```
rules:
  - changes:
    - Dockerfile
```

  - these rules can also be combined arbitrarily

Continuous integration & delivery

GitLab CI/CD

- `variables:` defining variables.
  Many variables are predefined for the runtime environment:
  https://docs.gitlab.com/ee/ci/variables/
- `services:` running services (e.g., database engines) in a separate Docker container (*docker-compose*)

```
services:
    - mysql:latest
variables:
    MYSQL_DATABASE: my_db
    MYSQL_USER: my_user
    MYSQL_PASSWORD: very_secret_password
```

  - the hostname will be `mysql` (can be changed with the `alias` option)

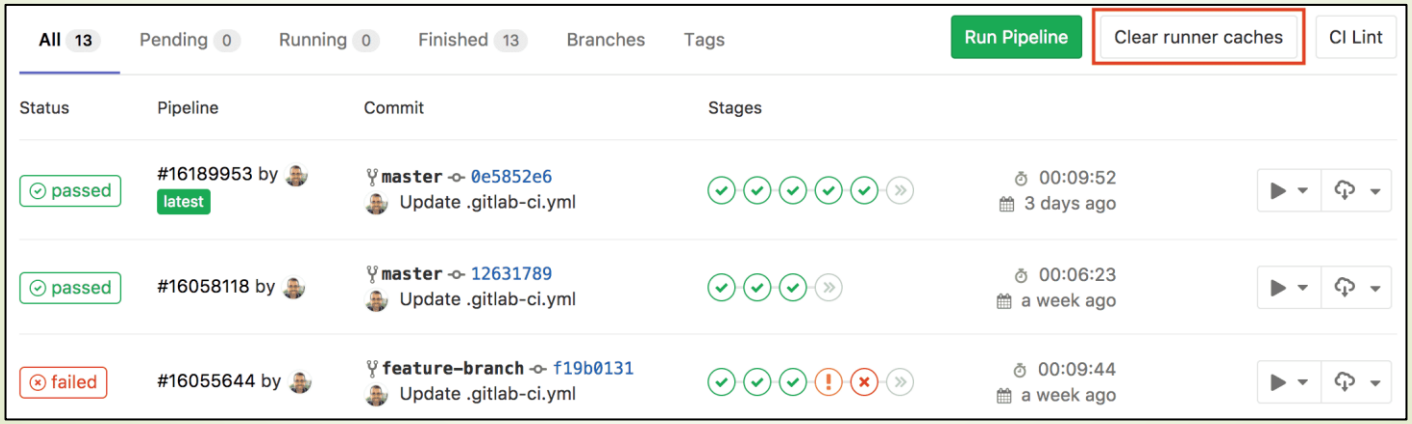
Continuous integration & delivery

GitLab CI/CD

- `cache:` preserving files and directories (typically dependencies) between CI jobs and pipelines
  - example:

```
cache:
  paths:
    - vendor/
```

  - the cache can be cleared manually:



Continuous integration & delivery

GitLab CI/CD

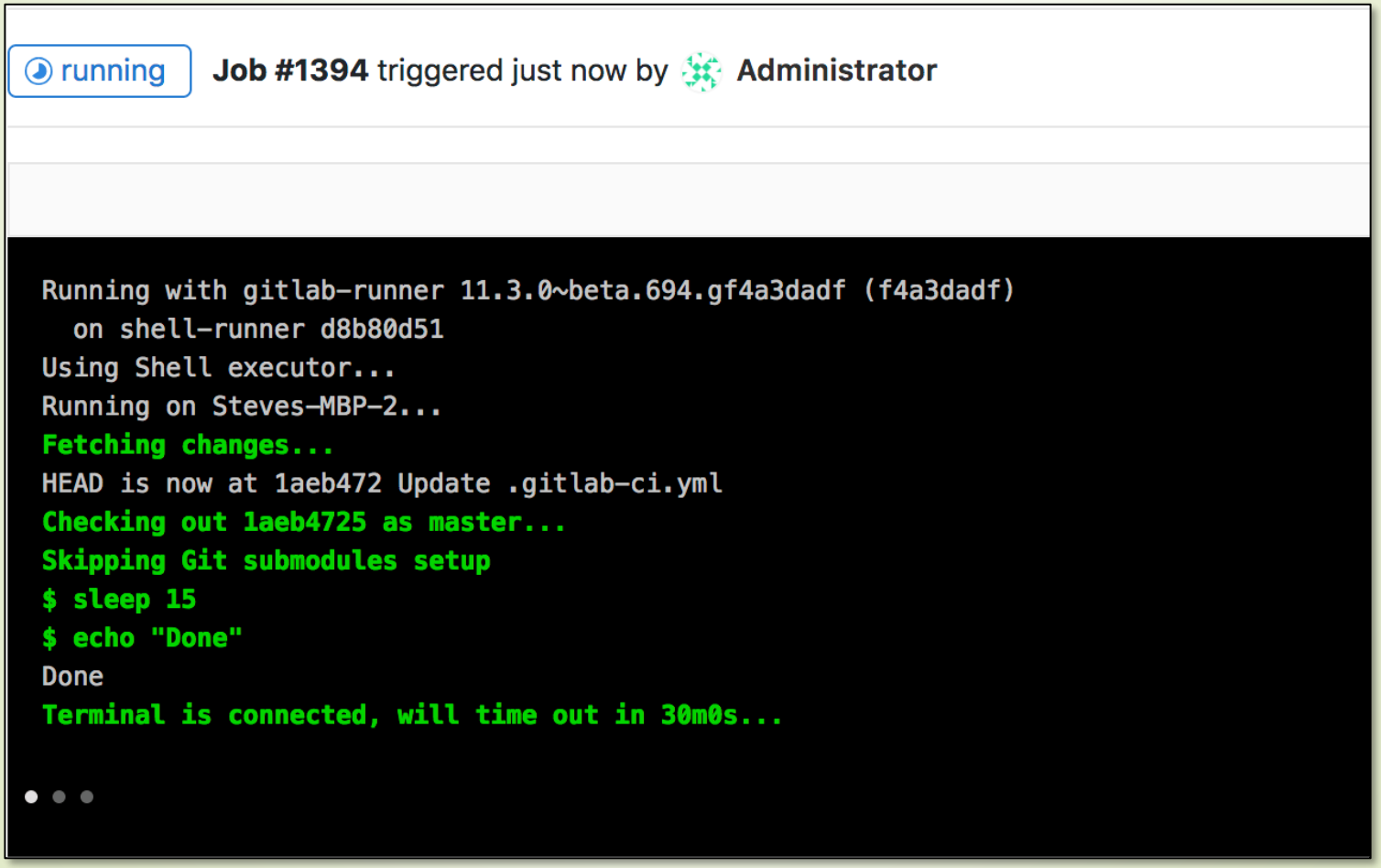
- A `key` can also be specified e.g. to allow different caches per CI *job* or *branch*
- For example, *caching* downloaded NuGet packages for a .NET application:

```
cache:
  # Separate cache per job and branch
  key: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
  paths:
    - .nuget/

build_program:
  script:
    # Download dependencies into the .nuget
    # directory in the project root.
    - dotnet restore --packages .nuget
    - dotnet build --no-restore
```

Continuous integration & delivery

GitLab CI/CD: terminals

- CI *job* execution can be monitored through an online terminal window in the GitLab web interface



Continuous integration & delivery

GitLab CI/CD: example projects

- Tic-tac-toe implementation in C++/Qt: https://szofttech.inf.elte.hu/mate/tictactoe-qt

```yaml
image: ubuntu:20.04

stages:
  - build
  - test

before_script:
  - apt-get update -yqq
  - apt-get install -yqq build-essential cmake
  - apt-get install -yqq qt5-default

# Build
build_game:
  stage: build
  script:
    - cd TicTacToeGame
    - qmake
    - make

# Test
test_model:
  stage: test
  script:
    - cd TicTacToeTest
    - qmake
    - make
    - make check
```

Continuous integration & delivery

GitLab CI/CD: example projects

- Tic-tac-toe implementation in C#/.NET: https://szofttech.inf.elte.hu/mate/tictactoe-dotnet

```yaml
image: mcr.microsoft.com/dotnet/sdk:6.0

stages:
  - build
  - test

before_script:
  - dotnet --version

# Build
build_model:
  stage: build
  script:
    - dotnet build TicTacToeGame.Model.Basic
    - dotnet build TicTacToeGame.Persistence.Text
    - dotnet build TicTacToeGame.Persistence.Binary

build_view:
  stage: build
  image: mcr.microsoft.com/dotnet/sdk:6.0-windowsservercore-ltsc2019
  tags: [windows]
  script:
    - dotnet build TicTacToeGame.sln

# Test
test_model:
  stage: test
  script:
    - dotnet test TicTacToeGame.Test

```

Continuous integration & delivery

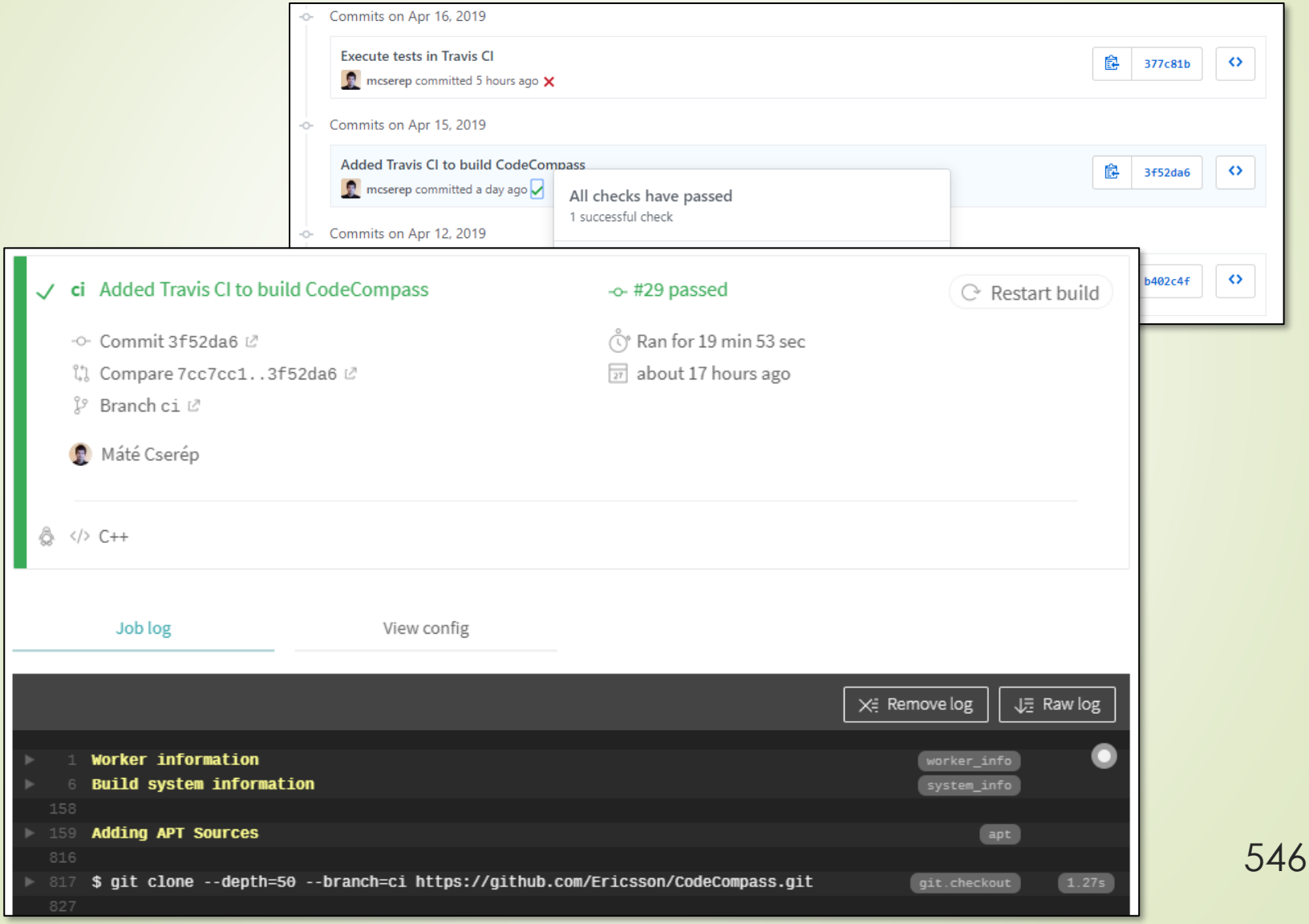
Travis CI

- Continuous integration service for GitHub projects
  - previously popular and free for open-source projects, but not anymore since 2021
  - https://travis-ci.com/
- Supports Linux, Windows, and macOS with multiple preconfigured environments (*image*)
  - the environment can be set up based on the used programming language, with common compiler tools and libraries available
- CI configuration is defined in the `.travis.yml` file

Continuous integration & delivery

ELTE | IK
INFORMATIKAI KAR

Travis CI – GitHub integration



Continuous integration & delivery

Travis CI

- For example:

```
os: linux
dist: xenial
language: cpp

stages:
  - compile
  - test
  - deploy

# ...

jobs:
  include:
    - stage: compile
        - cd $TRAVIS_BUILD_DIR
        - mkdir build && cd build
        - cmake ..
        - make

# ...
```

Continuous integration & delivery

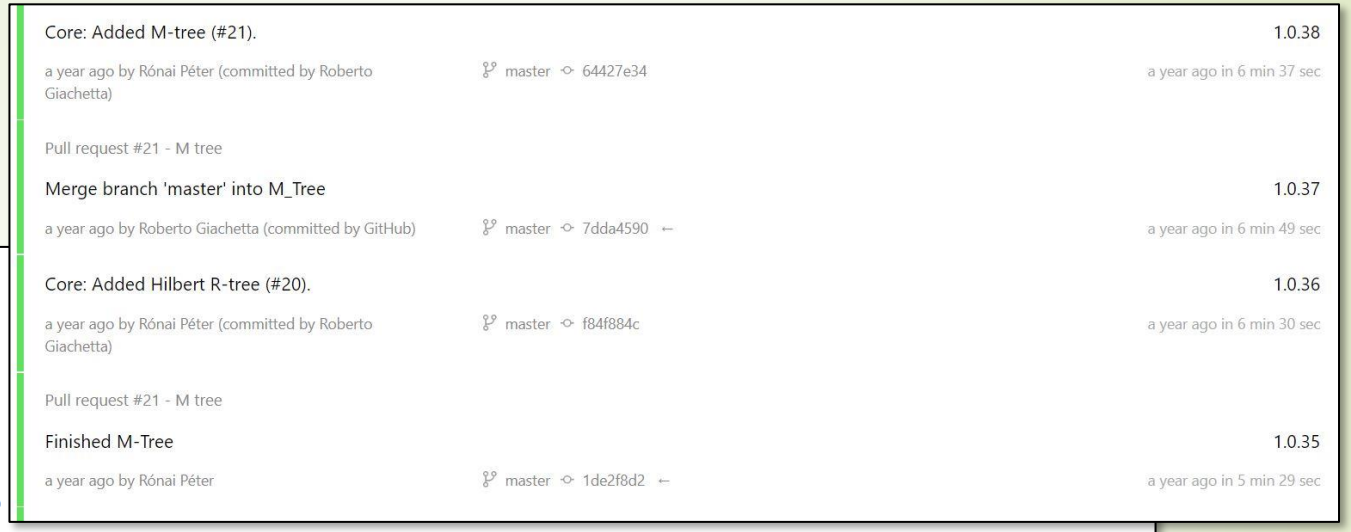
AppVeyor CI

- Continuous integration service
  - integrates with GitHub, GitLab, BitBucket, and Visual Studio Team Services
  - free for open-source projects
  - https://www.appveyor.com/
- Supports Linux, Windows, and macOS with multiple preconfigured environments (*image*)
  - strong .NET and Visual Studio support
- CI configuration is defined in the `appveyor.yml` file

Continuous integration & delivery

ELTE | IK
INFORMATIKAI KAR

AppVeyor CI – web interface



aegis

Current build History

Core: Added M-tree (#21).

a year ago by Rónai Péter (committed by Roberto Giachetta)

Console Messages 34 Tests Artifacts

```
Build started
git clone -q --branch=master https://github.com/robertogiachetta/aegis.git C:\projects\aegis
git checkout -qf 64427e34f338989da19925565681efdb67e46c17
nuget restore src\AEGIS.sln
MSBuild auto-detection: using msbuild version '15.5.180.51428' from 'C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\MSBuild\15.0\bin'.
Restoring NuGet package NUnit.3.6.1.
Restoring NuGet package Shouldly.2.8.2.
Restoring NuGet package StyleCop.Analyzers.1.0.0.
Restoring NuGet package Castle.Core.4.1.0.
Restoring NuGet package Moq.4.7.63.
Restoring NuGet package NUnit.ConsoleRunner.3.6.1.
Restoring NuGet package OpenCover.4.6.519.
Restoring NuGet package SimpleInjector.4.0.7.
Restoring NuGet package Microsoft.Win32.Primitives.4.3.0.
Restoring NuGet package System.Diagnostics.DiagnosticSource.4.3.0.
Adding package 'Microsoft.Win32.Primitives.4.3.0' to folder 'C:\projects\aegis\src\packages'
Adding package 'System.Diagnostics.DiagnosticSource.4.3.0' to folder 'C:\projects\aegis\src\packages'
Added package 'System.Diagnostics.DiagnosticSource.4.3.0' to folder 'C:\projects\aegis\src\packages'
```

Continuous integration & delivery

AppVeyor CI

- For example:

```
version: 1.0.{build} # Version format
image: Visual Studio 2017 # Build worker image
platform: Any CPU # Build platform
configuration: Debug # Build Configuration

# Execute script before build
before_build:
  - dotnet restore src\MyProject.sln

# Execute build script
build_script:
  - dotnet build src\MyProject.sln

# Execute test script
test_script:
  - dotnet test src\MyProject.sln
```

Continuous integration & delivery

Jenkins

- Open-source continuous integration service
  - integrates with GitHub, GitLab, and other project management platforms
  - does not provide hosting, but third-party services are available (e.g. CloudBees)
  - https://jenkins.io/
- CI configuration is defined in the `Jenkinsfile`, for example:

```
pipeline {
  agent { docker { image 'ubuntu:18.04' } }
  stages {
    stage('build') {
      steps {
        sh 'apt-get install build-essential'
        sh 'make'
      }
    }
  }
}
```

Continuous integration & delivery

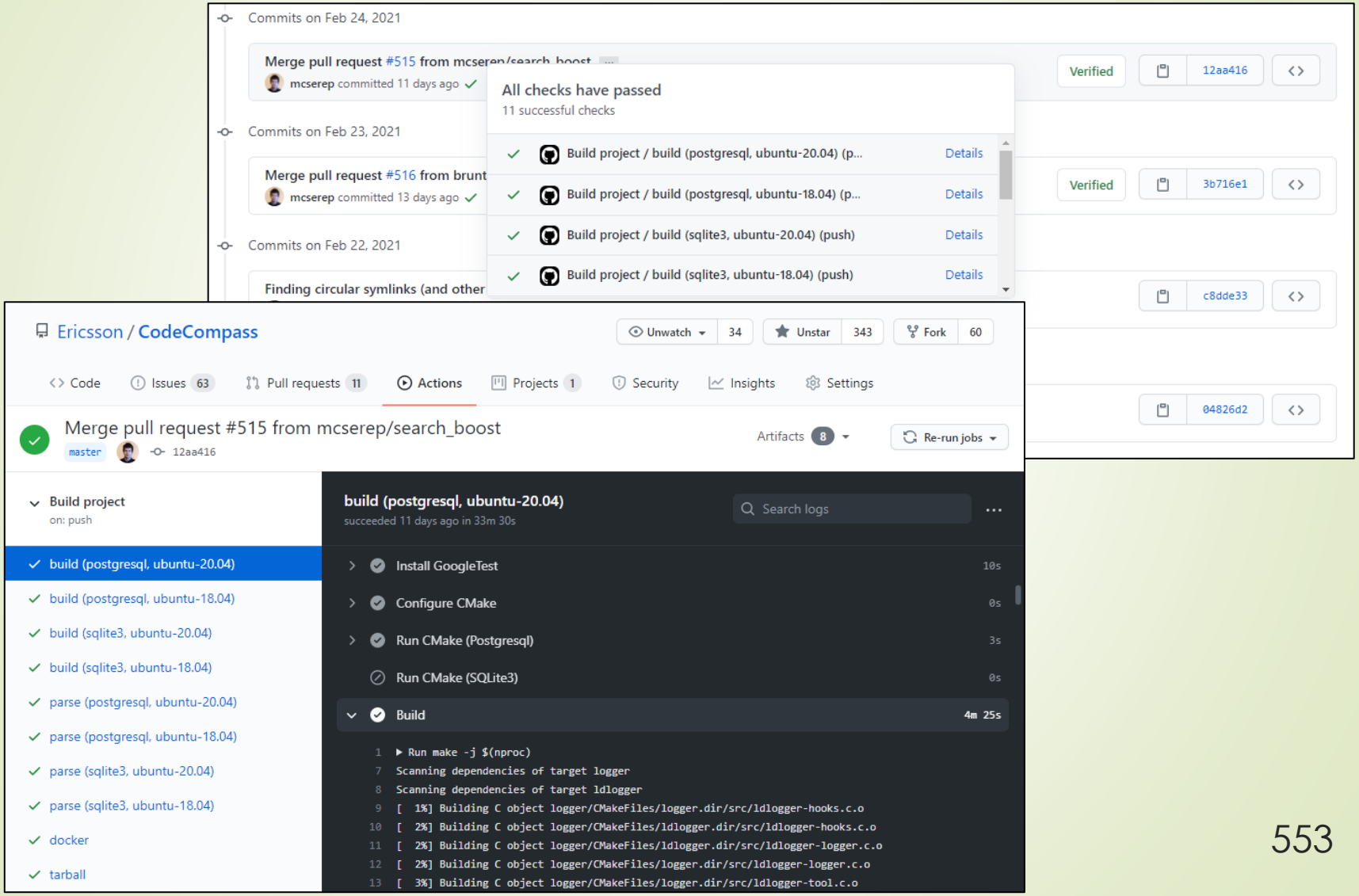
GitHub Actions

- GitHub's relatively new, proprietary CI/CD service
  - free with limitations for free accounts (currently 2000 runtime minutes/month), extendable with a subscription
  - https://docs.github.com/en/actions
- Supports Linux (Ubuntu), Windows, and macOS based virtual machines, as well as Docker
  - virtual machines run on Microsoft's cloud (*Azure*)
  - *self-hosted* GitHub Actions Runner can also be installed
- CI configuration is defined in the `.github/workflows/` directory using YAML files
  - in addition to shell commands, we can use predefined *actions* created by others (or our own custom ones) as complex building blocks

Continuous integration & delivery

ELTE | IK
INFORMATIKAI KAR

GitHub Actions



Continuous integration & delivery

GitHub Actions

- For example:

```
name: Build project
on: [push, pull_request]
jobs:
  build:
    runs-on: [windows-latest] # or ubuntu-latest
    steps:
    - uses:  actions/checkout@v4
    - name:  Setup .NET SDK
      uses:  actions/setup-dotnet@v4
      with:
        dotnet-version: '6.0.x'
    - name:  Restore NuGet packages
      run:   dotnet restore AEGIS.sln
    - name:  Build the solution
      run:   dotnet build AEGIS.sln -c Release
    - name:  Run unit tests
      shell: powershell
      run:   dotnet test AEGIS.sln
```

Software Technology

Continuous integration & delivery

Specialization C

Dr. Máté Cserép

ELTE Faculty of Informatics

2025.

Continuous integration & delivery

Continuous integration

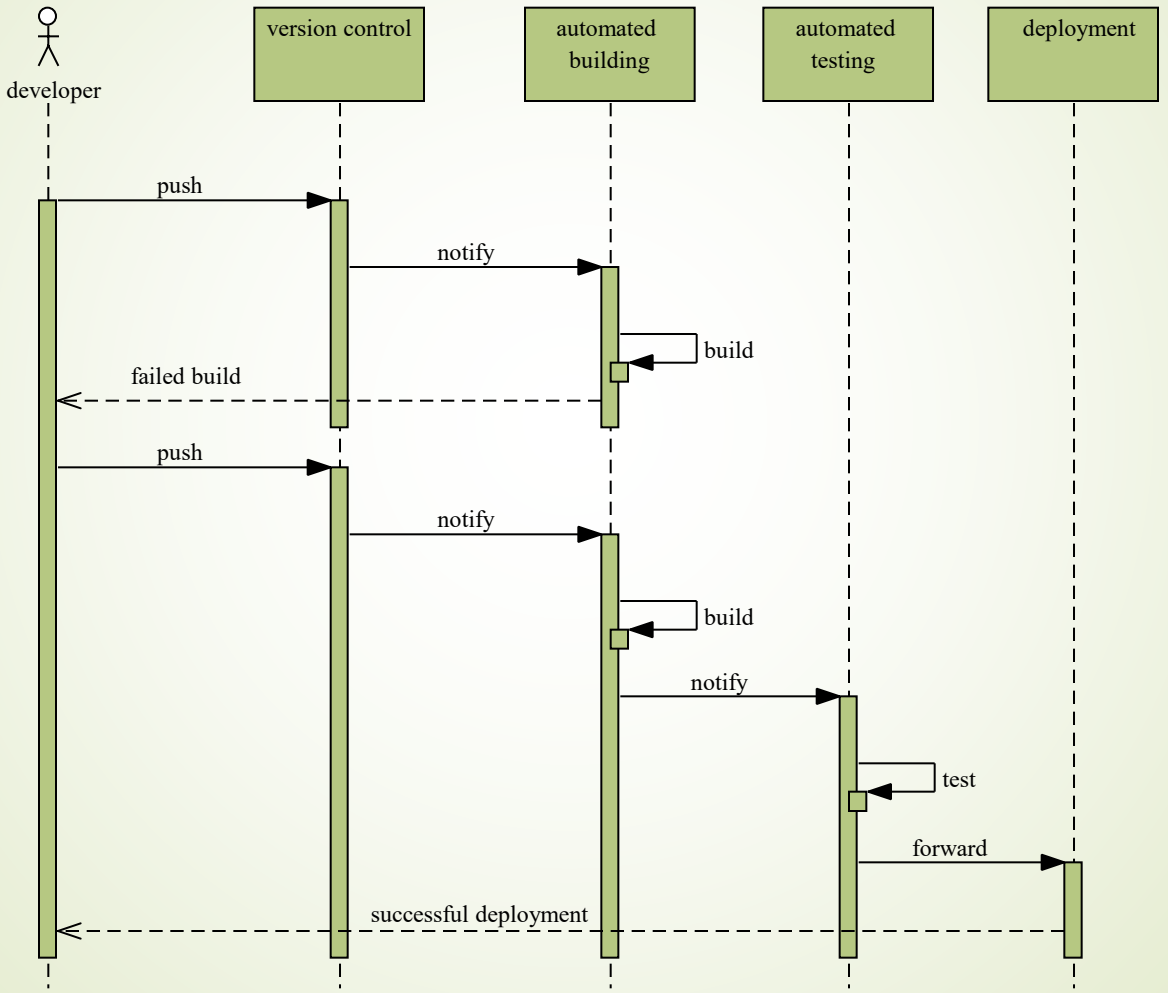
- *Continuous integration* (*CI*) is a practical method that accelerates the verification and testing of program code
  - its goal is to immediately and automatically detect potential errors and integration issues, providing feedback to developers
  - the source code is stored in a central repository using a version control system and is updated multiple times a day
  - after each modification, the repository's content is automatically built (*build automation*), and coded tests are executed along with the build process
  - the verified code can then undergo further testing

Continuous integration & delivery

Continuous delivery

- *Agile software development* aims to achieve rapid application development on an incremental basis
  - the software is continuously developed and released (*continuous delivery*), maintaining a steady pace while allowing changes to be incorporated at every step (*welcome changes*)
  - working software serves as the primary measure of progress, emphasizing simplicity while maintaining a focus on proper design and optimization
  - development is typically carried out by small, self-organizing teams with shared responsibility, continuous interaction, and rapid feedback cycles
  - continuous releases can be automated, which is known as *continuous deployment*

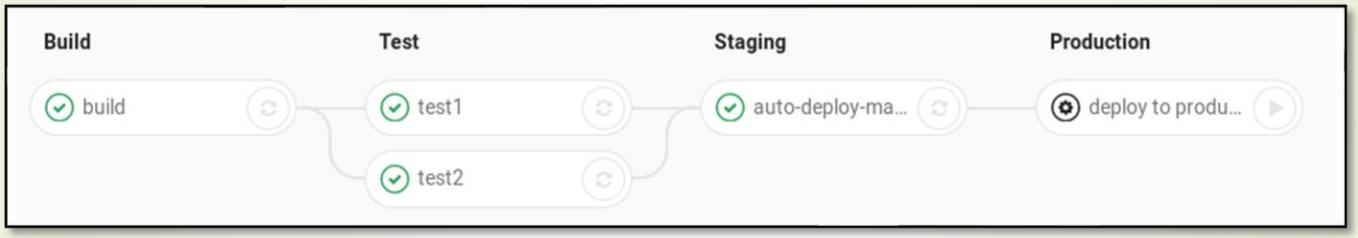
Continuous integration & delivery

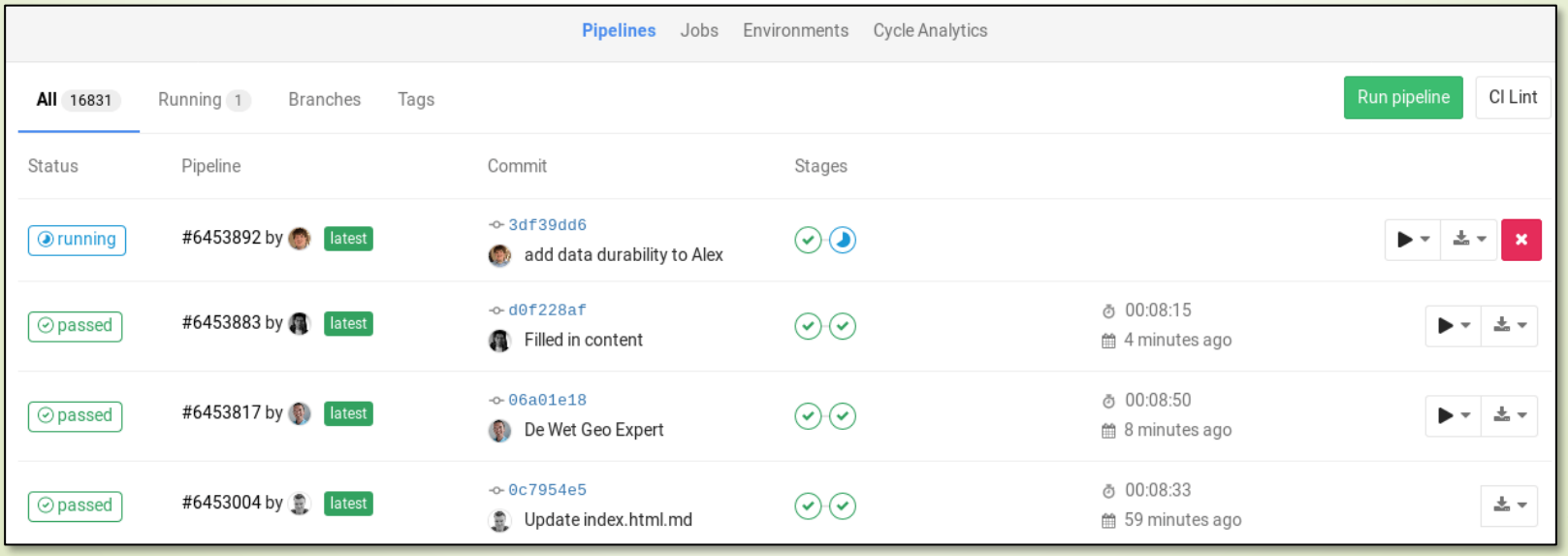


Continuous integration & delivery

Jobs

- The steps of continuous integration and delivery can be defined as *pipelines,* a chain of interdependent *jobs*

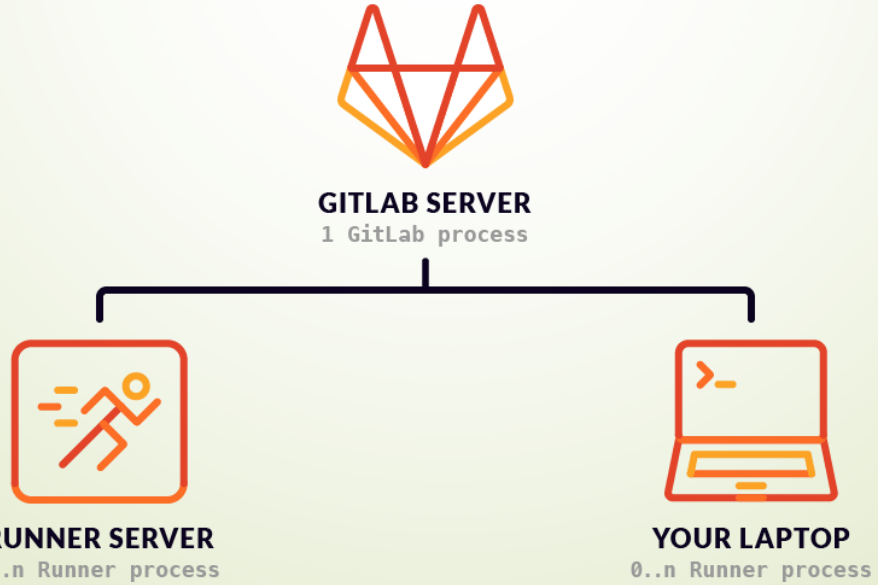




Continuous integration & delivery

GitLab Runners

- GitLab provides an integrated, built-in solution to support continuous integration and delivery
  - *jobs* are executed by so-called *GitLab Runner* instances, independent of the GitLab server
    - Shell, SSH, VirtualBox, Docker, Kubernetes, etc.
  - *runners* can be individually configured and can either be shared or assigned to specific projects



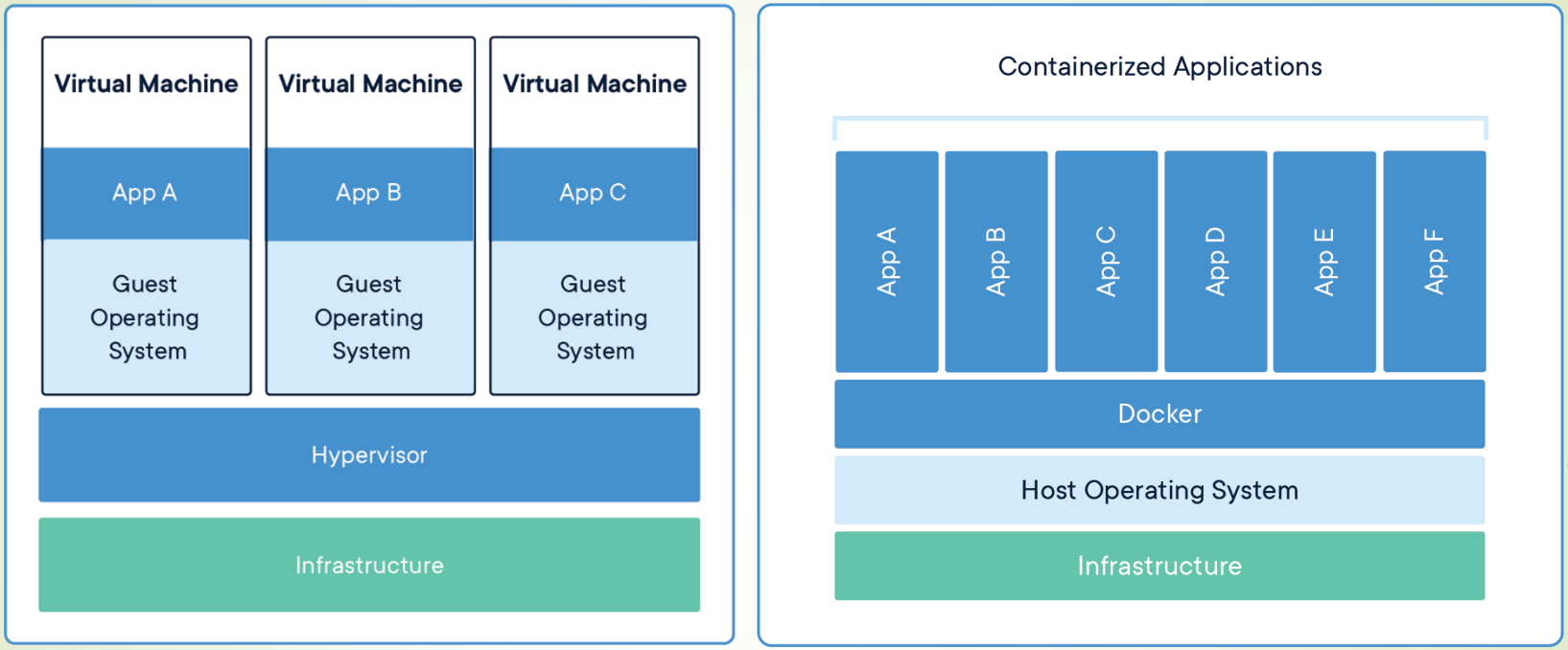
Continuous integration & delivery

Docker

- Continuous integration is best performed in an isolated, reproducible environment
- Docker is currently the most widely used *container framework* system
  - a *container* is similar to a virtual machine (*VM*) in that it provides a fully isolated, virtualized environment where the host machine supplies resources
  - the key difference between *containers* and virtual machines is that all *containers* share the host's kernel, virtualized hardware and an operating system are not included; instead, only the necessary libraries, binaries, and user space are part of the *container*
  - as a result, *containers* have significantly lower *overhead* compared to VMs, making them a more lightweight virtualization solution

Continuous integration & delivery

Virtual machines and *containers*



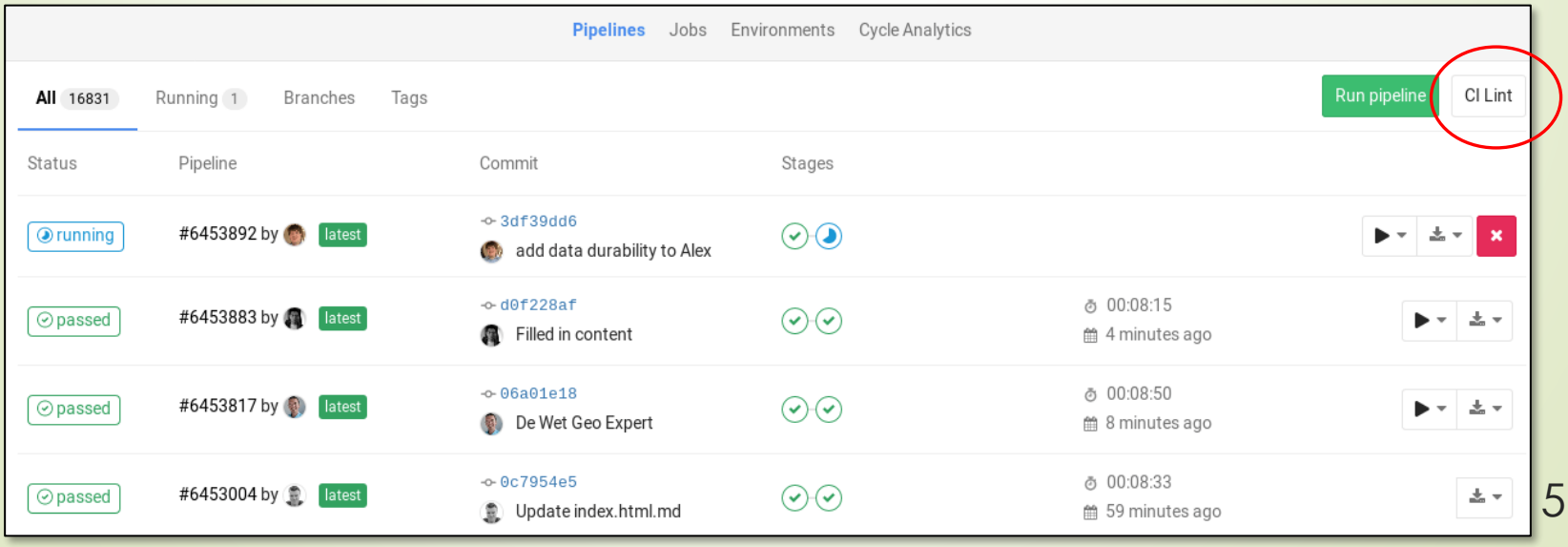
Source: docker.com

- *container frameworks* enable the creation and management of portable applications
- applications can be modularized and scaled, with components running in separate *containers*

Continuous integration & delivery

GitLab CI/CD

- The configuration for continuous integration is stored in the `.gitlab-ci.yml` file, written in YAML format
  - YAML (*YAML Ain't Markup Language*) is a structured descriptive language that is easily (or more easily) readable by humans
  - https://yaml.org/spec/1.2.2/
- GitLab's web interface includes a CI Lint tool for validating the format before submission



Continuous integration & delivery

YAML syntax

- Example YAML code:

```yaml
name: John Doe # key-value pairs
age: 42
details: # embedded collection
  givenname: John
  familyname: Doe
  birthyear: 1978
languages: # list of values (array)
  - Hungarian
  - English
  - German
# multi-line text
intro_multi: |
  multiple line
  introduction
intro_single: >
  single line
  introduction
```

GitLab CI/CD: jobs

- In the `.gitlab-ci.yml` file, we can define *jobs* specifying which commands they should execute (`script`)
- Example:

```
build_program:
  script:
    - apt-get update -qq
    - apt-get install -yqq openjdk-11-jdk ant
    - ant compile
    - ant jar
```

Continuous integration & delivery

GitLab CI/CD: multiple jobs

- Multiple *jobs* can be defined, and a global `before_script` can be set to run before every *job*. (This can also be overridden within individual *jobs*.)

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant junit

build_program:
  script:
    - ant compile
    - ant jar

test_program:
  script:
    - ant compile-test
    - ant test
```

GitLab CI/CD: stages

- Continuous integration *jobs* can be divided into sequential *stages*
  - by default, there are 3 *stages*: *build*, *test*, *deploy*
  - this can be redefined as needed

```
stages:
  - lint
  - build
```

- *Jobs* within a *stage* can be run in parallel independently (using multiple *runners*)
  - *stages* are built upon each other: if a *job* in a *stage* fails, the dependent *stages* will not be executed

Continuous integration & delivery

GitLab CI/CD: stages

- Splitting the process into *stages:*

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant junit

build_program:
  stage: build
  script:
    - ant compile
    - ant jar

test_program:
  stage: test
  script:
    - ant compile-test
    - ant test
```

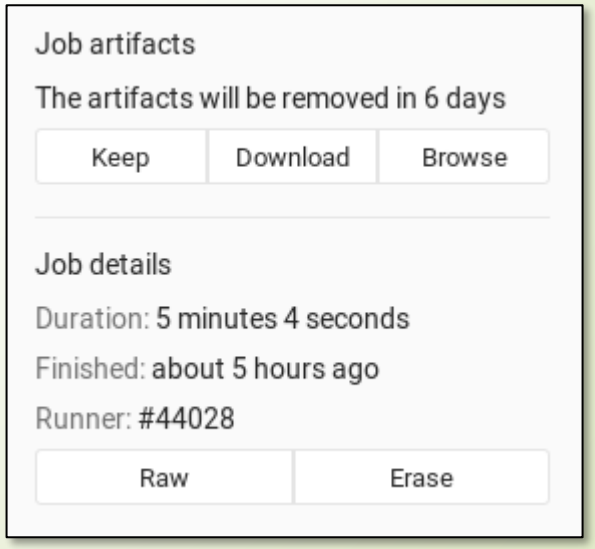
Continuous integration & delivery

GitLab CI/CD: artifacts

- Binaries or other files generated as a part of CI *jobs* can be preserved (*artifact*)

```
pdf:
  script:
    - pdftex paper.tex
  artifacts:
    paths:
      - paper.pdf
    expire_in: 1 month
```

- *Artifacts* can be easily downloaded from the GitLab web interface



Continuous integration & delivery

GitLab CI/CD: artifacts

- Defining *artifacts:*

```
before_script:
  - apt-get update -qq
  - apt-get install -yqq openjdk-11-jdk ant

build_program:
  stage: build
  script:
    - ant compile
    - ant jar
  artifacts:
    paths:
      - dist/program.jar
    expire_in: 1 week
```

Continuous integration & delivery

GitLab CI/CD: dependencies

- The verification of interdependent programs (modules) can easily lead to redundant execution:

```
before_script: ...

build_program_A:
  stage: build
  script:
    - cd module_A
    - ant jar

build_program_B: # depends on program_A.jar file
  stage: build
  script:
    - cd module_A
    - ant jar
    - cd ..
    - cd module_B
    - ant jar
```

Continuous integration & delivery

GitLab CI/CD: dependencies

- Passing *artifacts* between *jobs:*

```
before_script: ...

build_program_A:
  stage: build
  script:
    - cd module_A
    - ant jar
  artifacts:
    paths:
      - module_A/dist/program_A.jar

build_program_B: # depends on program_A.jar file
  stage: build
  script:
    - cd module_B
    - ant jar
  dependencies:
    - build_program_A
```

Continuous integration & delivery

Docker images

- When using a Docker container based GitLab Runner, the base *Docker image* can be specified:

```
image: ubuntu:22.04
```

image name

tag name

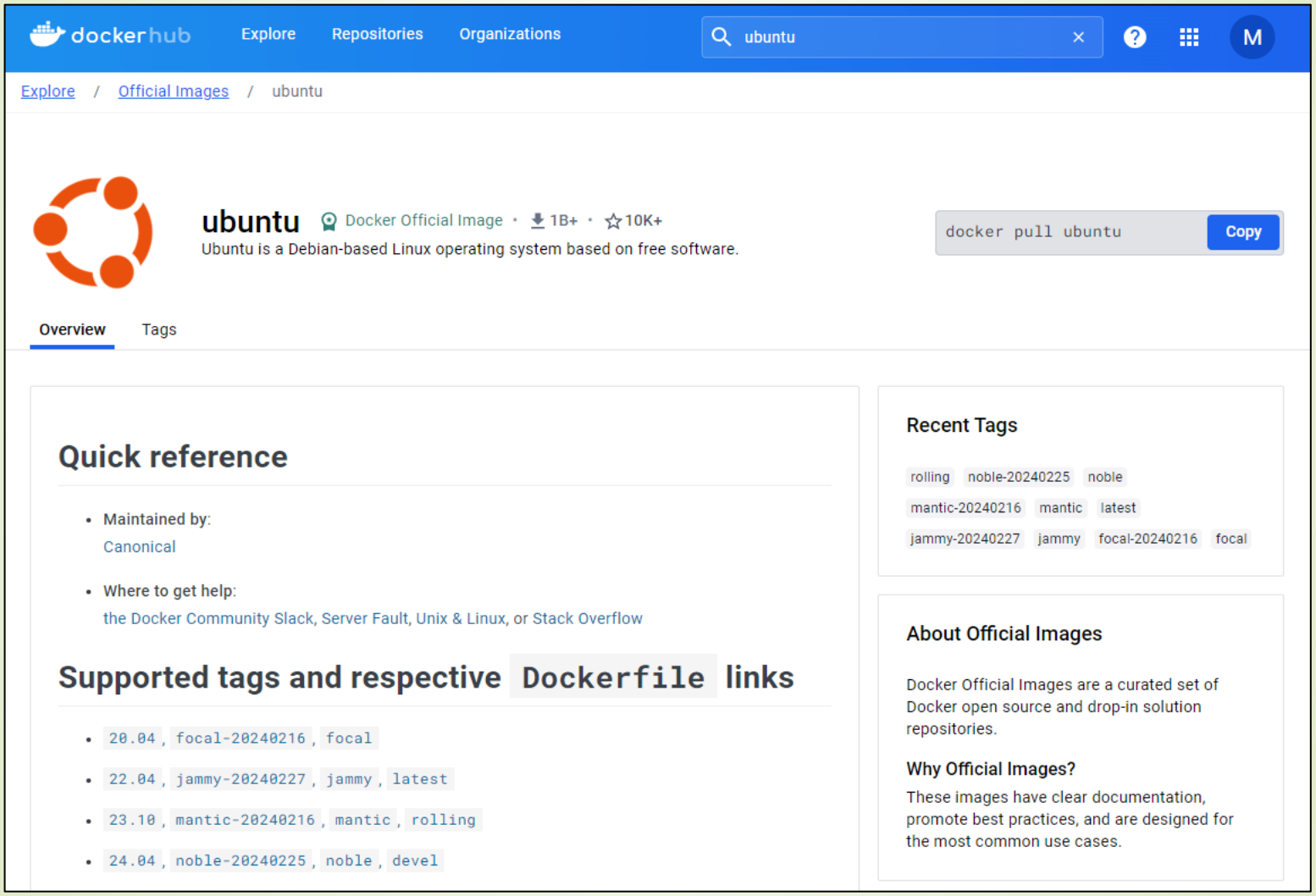
- A *Docker image* can be requested from a *Docker registry*:
  - By default, the public Docker Hub is used, where custom images can also be uploaded: https://hub.docker.com/
  - A private docker registry can also be used (e.g., in an enterprise environment)

```
image: mycompany.com:5000/custom:latest
```

- If not specified, the runner configuration determines the image to be used
  - the runners at szofttech.inf.elte.hu are configured to use `ubuntu:22.04`
  - For Windows runners, the default image is: `mcr.microsoft.com/windows/servercore:1809`

Continuous integration & delivery

Docker Hub



Continuous integration & delivery

GitLab CI/CD

- Additional options (not exhaustive):
  - `only, except:` conditional execution of CI jobs (e.g., run only on the *master* branch)
  - `when:` conditional execution of CI jobs (manual vs. automatic execution)

```
image: maven:latest
build_program:
  stage: build
  script:
    - mvn compile
test_program:
  stage: test
  script:
    - mvn test
deploy_program:
  stage: deploy
  script:
    - mvn deploy
  only:
    - master
  when: manual
```

Continuous integration & delivery

GitLab CI/CD

- `rules` provides advanced configuration options compared to `only` and `except`
  - `only, except` are considered deprecated, they are not developed with further capabilities
  - e.g. run only on master branch:

```
rules:
  - if: $CI_COMMIT_BRANCH == "master"
```

  - e.g. run only when *Dockerfile* changes:

```
rules:
  - changes:
    - Dockerfile
```

  - these rules can also be combined arbitrarily

Continuous integration & delivery

GitLab CI/CD

- `variables:` defining variables.
  Many variables are predefined for the runtime environment:
  https://docs.gitlab.com/ee/ci/variables/
- `services:` running services (e.g., database engines) in a separate Docker container (*docker-compose*)

```
services:
    - mysql:latest
variables:
    MYSQL_DATABASE: my_db
    MYSQL_USER: my_user
    MYSQL_PASSWORD: very_secret_password
```

  - the hostname will be `mysql` (can be changed with the `alias` option)

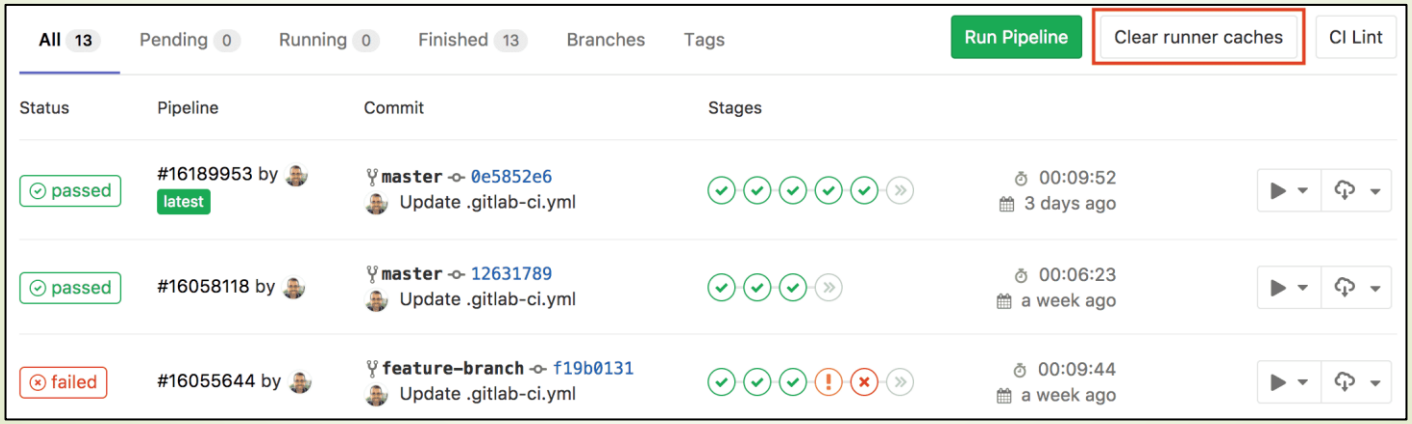
Continuous integration & delivery

GitLab CI/CD

- `cache:` preserving files and directories (typically dependencies) between CI jobs and pipelines
  - example:

```
cache:
  paths:
    - vendor/
```

  - the cache can be cleared manually:



Continuous integration & delivery

GitLab CI/CD

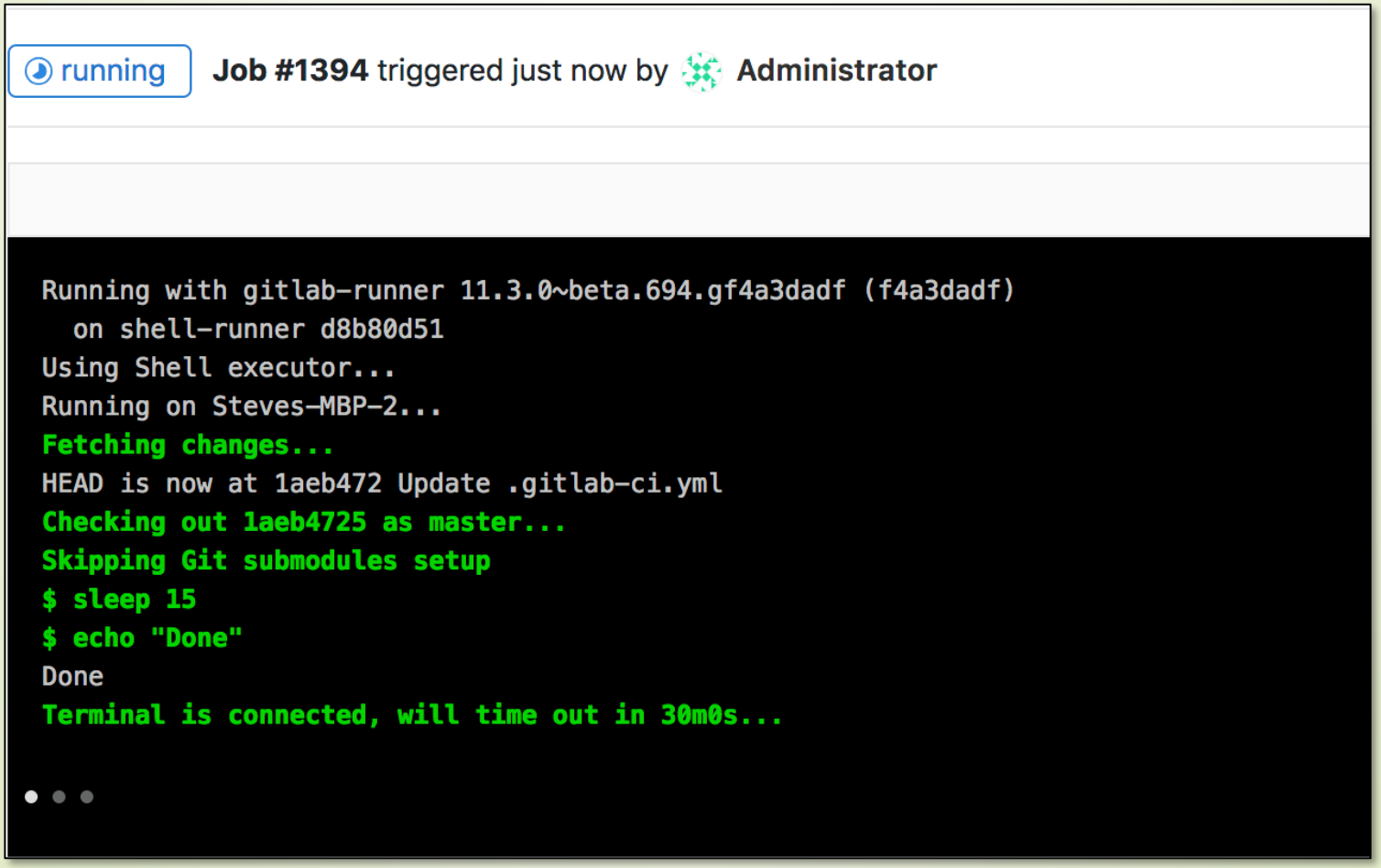
- A `key` can also be specified e.g. to allow different caches per CI *job* or *branch*
- For example, *caching* downloaded packages in a Maven application:

```
variables:
  MAVEN_OPTS: "-Dmaven.repo.local=
               $CI_PROJECT_DIR/.m2/repository"

cache:
  # Separate cache per job and branch
  key: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
  paths:
    - .m2/repository

build_program:
  script:
    - mvn compile
```

Continuous integration & delivery

GitLab CI/CD: terminals

- CI *job* execution can be monitored through an online terminal window in the GitLab web interface



Continuous integration & delivery

GitLab CI/CD: example project

- Network Pacman game with Java implementation: https://szofttech.inf.elte.hu/mate/pacman-java/

```yaml
image: openjdk:11

stages:
  - build
  - test

before_script:
  - apt-get update -yqq
  - apt-get install -yqq ant junit4

# Build
build_game:
  stage: build
  script:
    - ant compile
    - ant jar

# Test
test_model:
  stage: test
  script:
    - >
      ant test
      -Dlibs.junit_4.classpath=/usr/share/java/junit4.jar
      -Dlibs.hamcrest.classpath=/usr/share/java/hamcrest-core.jar
```

- With Ant build system: `ant` development branch

Continuous integration & delivery

GitLab CI/CD: example project

- With Maven build system: `maven` development branch

```yaml
image: maven:latest

variables:
  MAVEN_OPTS: "-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository"

cache:
  key: maven-build
  paths:
    - .m2/repository

build:
  stage: build
  script:
    - mvn compile
test:
  stage: test
  script:
    - mvn test
```

- The cache can be identified with a key, allowing e.g. control over which development branches should share it

Continuous integration & delivery

GitLab CI/CD: example project

- With Gradle build system: `gradle` development branch

```yaml
image: gradle:latest

variables:
  GRADLE_USER_HOME: "$CI_PROJECT_DIR/.gradle"

# Cache and keep downloaded dependencies between CI pipelines
cache:
  key: gradle-build
  paths:
    - .gradle

build:
  stage: build
  script:
    - gradle assemble

test:
  stage: test
  script:
    # build also executes test
    - gradle build

```

Continuous integration & delivery

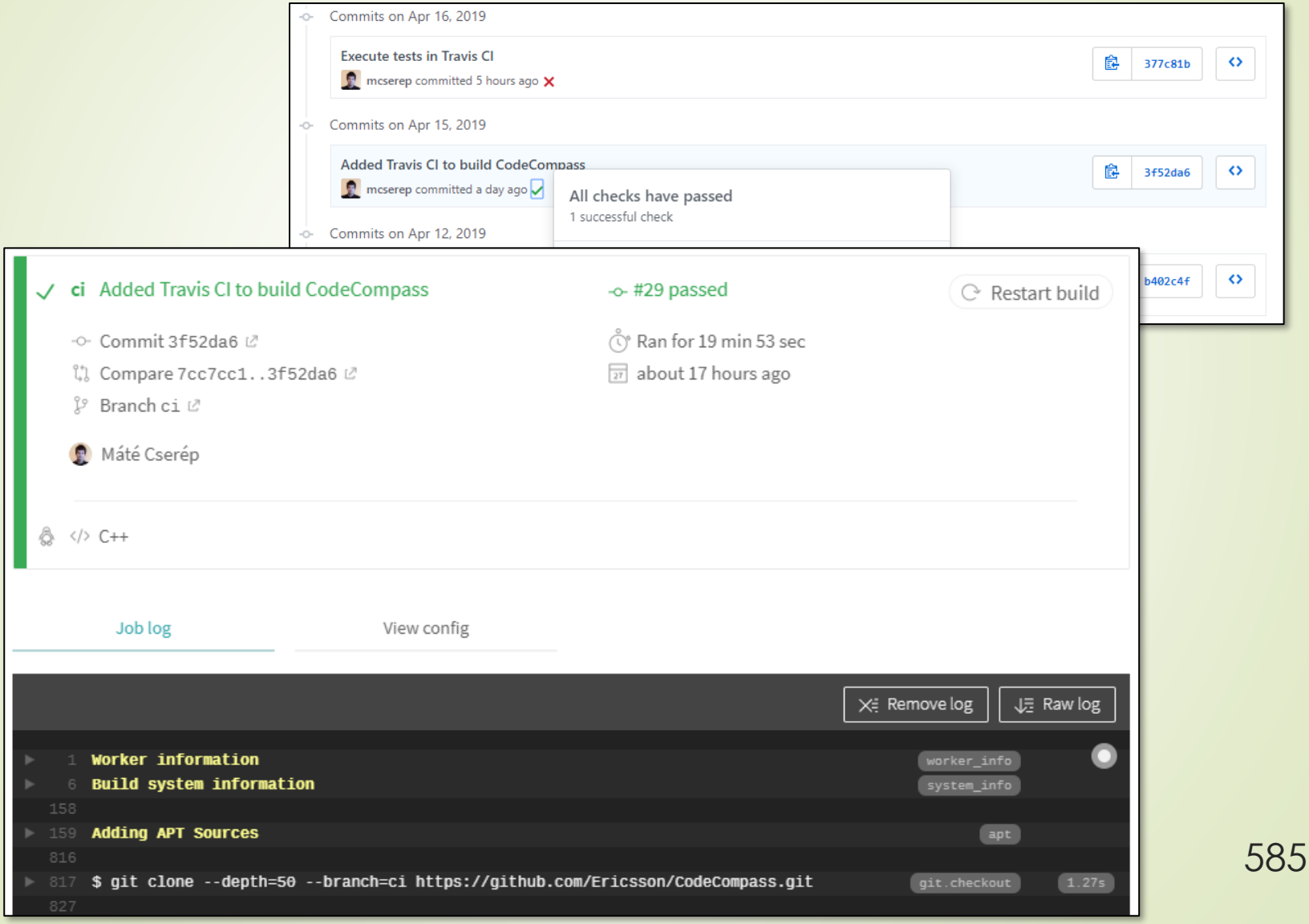
Travis CI

- Continuous integration service for GitHub projects
  - previously popular and free for open-source projects, but not anymore since 2021
  - https://travis-ci.com/
- Supports Linux, Windows, and macOS with multiple preconfigured environments (*image*)
  - the environment can be set up based on the used programming language, with common compiler tools and libraries available
- CI configuration is defined in the `.travis.yml` file

Continuous integration & delivery

ELTE | IK
INFORMATIKAI KAR

Travis CI – GitHub integration



Continuous integration & delivery

Travis CI

- For example:

```
os: linux
dist: xenial
language: java

stages:
  - compile
  - test
  - deploy

# ...

jobs:
  include:
    - stage: compile
        - mvn compile
    - stage: test
        - mvn test

# ...
```

Continuous integration & delivery

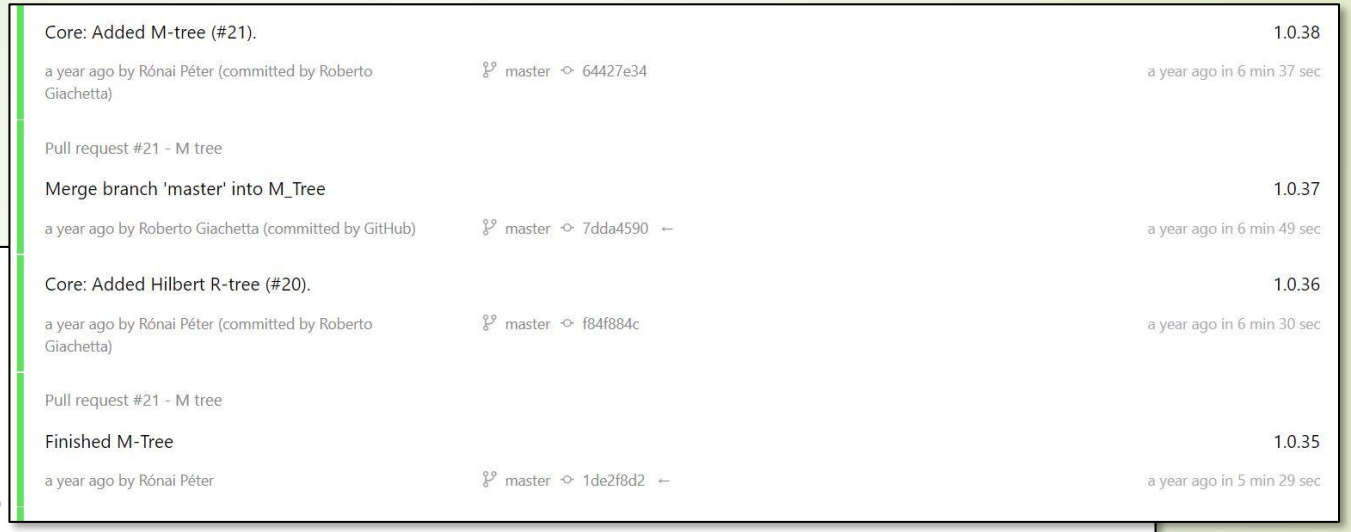
AppVeyor CI

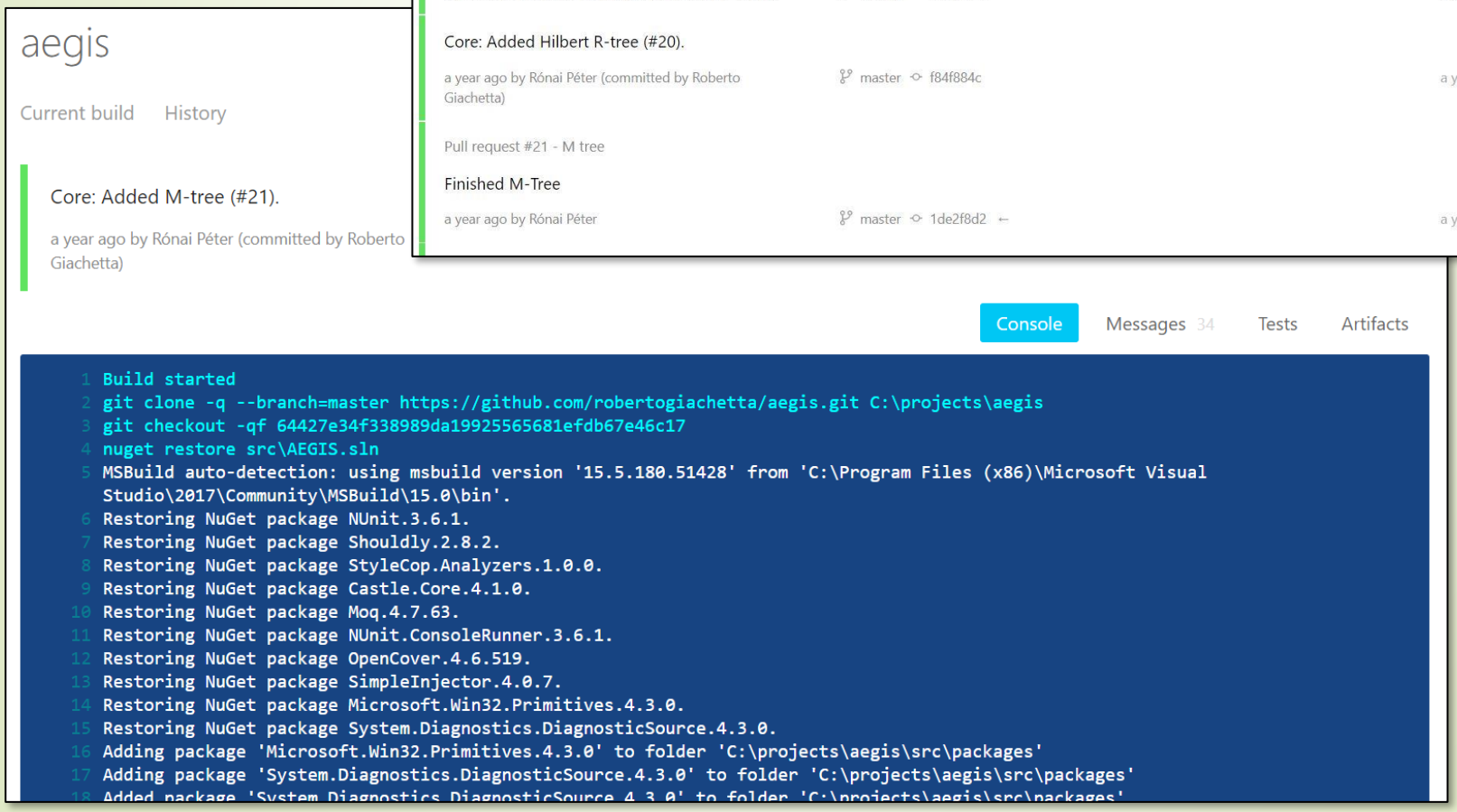
- Continuous integration service
  - integrates with GitHub, GitLab, BitBucket, and Visual Studio Team Services
  - free for open-source projects
  - https://www.appveyor.com/
- Supports Linux, Windows, and macOS with multiple preconfigured environments (*image*)
  - strong .NET and Visual Studio support
- CI configuration is defined in the `appveyor.yml` file

Continuous integration & delivery

ELTE | IK
INFORMATIKAI KAR

AppVeyor CI – web interface





Continuous integration & delivery

AppVeyor CI

- For example:

```
version: 1.0.{build} # Version format
image: Visual Studio 2017 # Build worker image
platform: Any CPU # Build platform
configuration: Debug # Build Configuration

# Execute script before build
before_build:
  - dotnet restore src\MyProject.sln

# Execute build script
build_script:
  - dotnet build src\MyProject.sln

# Execute test script
test_script:
  - dotnet test src\MyProject.sln
```

Continuous integration & delivery

Jenkins

- Open-source continuous integration service
  - integrates with GitHub, GitLab, and other project management platforms
  - does not provide hosting, but third-party services are available (e.g. CloudBees)
  - https://jenkins.io/
- CI configuration is defined in the `Jenkinsfile`, for example:

```
pipeline {
    agent { docker { image 'maven:3.3.3' } }
    stages {
        stage('build') {
            steps {
                sh 'mvn compile'
            }
        }
    }
}
```

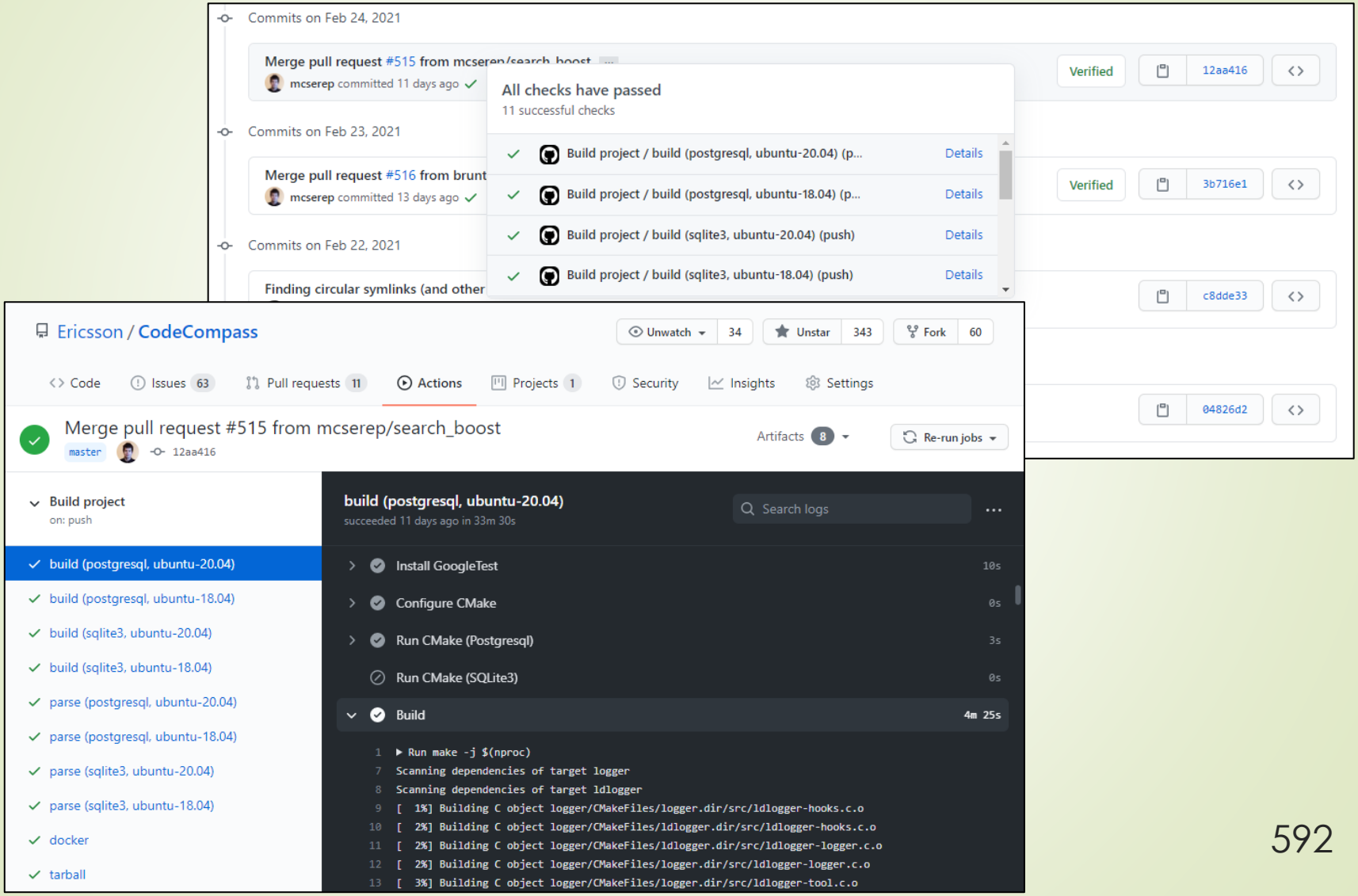
Continuous integration & delivery

GitHub Actions

- GitHub's relatively new, proprietary CI/CD service
  - free with limitations for free accounts (currently 2000 runtime minutes/month), extendable with a subscription
  - https://docs.github.com/en/actions
- Supports Linux (Ubuntu), Windows, and macOS based virtual machines, as well as Docker
  - virtual machines run on Microsoft's cloud (*Azure*)
  - *self-hosted* GitHub Actions Runner can also be installed
- CI configuration is defined in the `.github/workflows/` directory using YAML files
  - in addition to shell commands, we can use predefined *actions* created by others (or our own custom ones) as complex building blocks

Continuous integration & delivery

GitHub Actions



Continuous integration & delivery

GitHub Actions

- For example:

```
name: Build project
on: [push, pull_request]
jobs:
  build:
    runs-on: [ubuntu-latest]
    steps:
    - uses: actions/checkout@v4

    - name: Set up JDK 1.21
      uses: actions/setup-java@v4
      with:
        distribution: zulu # Azul Zulu OpenJDK
        java-version: 21

    - name: Build the project
      run:  mvn compile

    - name: Run the unit tests
      run:  mvn test
```