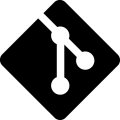
**МІНІСТЕРСТВО ОСВІТИ І НАУКИ,   
МОЛОДІ ТА СПОРТУ УКРАЇНИ**

**ОДЕСЬКА НАЦІОНАЛЬНА АКАДЕМІЯ   
ХАРЧОВИХ ТЕХНОЛОГІЙ**

****

**Мазурок І.Є., Баркар С.О.**

ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ ТА  
СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

******

**Посібник для самостійної роботи**

**Мазурок І.Є., Баркар С.О.** Технології програмування та створення програмного продукту: навчальний посібник для самостійної роботи студентів/ Одеська національна академія харчових технологій, учбово-навчальний інститут холоду кріотехнологій та екоенергетики 2013. – 29 с.

Курс **«**Технології програмування та створення програмних продуктів**»** є нормативно професійно-орієнтованим курсом. Мета курсу – ознайомлення студентів з технологіями розробки програм і програмної продукції для ЕОМ, що відповідають вимогам високої надійності і якості програмного забезпечення з використанням належної методології, засобів і систем програмування. В посібнику розглядається важливий аспект сучасних технологій розробки програмного забезпечення – засоби колективної розробки та контролю версій з використанням хмарових технологій на прикладі системи Git.

Навчальний посібник призначено для студентів напряму підготовки 6.050101 «Комп’ютерні науки».

Рецензент: доктор технічних наук, професор Малахов Є.В.

Розглянуто та рекомендовано до видання на засіданні кафедри інформаційних технологій та кібербезпеки.

Протокол № 8 від 29 березня 2013 р.

Розглянуто та рекомендовано до видання на засіданні науково-методичної комісії з напряму підготовки 6.050101 «Комп’ютерні науки».

Протокол № від 16 червня 2013 р.

**Анотація**

Посібник призначений для самостійної роботи при вивченні курсу «технології програмування та створення програмного продукту» (ТПСПП), який вивчається один семестр студентами напрямку «Комп’ютерна інженерія». У зв'язку з тенденцією збільшення частки часу на самостійну роботу студентів, даний навчальний посібник стане важливою складовою при вивченні студентами курсу ТПСПП. Посібник надає можливість ознайомитися та отримати навики роботи в розподіленій (хмарній) системі колективної розробки Git, призначеної для швидкого та ефективного контролю версій документів, як в невеликих так і в масштабних проектах.

Цілеспрямованість даного посібника:підготовка студента до всіх видів контролю знань з курсу ТПСПП, набуття навичок роботи з системою колективної розробки програмних проектів Git, а також сприяння самостійній творчий праці студентів з метою досягнення високої якості вищої освіти.

**Зміст**

[Вступ 5](#_Toc360300642)

[1. Системи контролю версій 6](#_Toc360300643)

[1.1. Система контролю версій Git 9](#_Toc360300644)

[1.2. Можливості і недоліки Git 10](#_Toc360300645)

[1.3. Особливості системи Git 10](#_Toc360300646)

[2. Склад системи Git 12](#_Toc360300647)

[2.1. Серверний блок 12](#_Toc360300648)

[2.2. Клієнтські блоки 12](#_Toc360300649)

[2.3. Графічний інтерфейс Git GUI 14](#_Toc360300650)

[2.4. Графічний інтерфейс TortoiseGit 14](#_Toc360300651)

[3. Використання GitHub 18](#_Toc360300652)

[3.1. Встановлення SSH-ключа 18](#_Toc360300653)

[3.2. Створення репозиторія 19](#_Toc360300654)

[3.3. Клонування репозиторія 20](#_Toc360300655)

[3.4. Внесення змін в локальний репозиторій 21](#_Toc360300656)

[3.5. Синхронізація змін з головним репозиторієм 23](#_Toc360300657)

[3.6. Робота з розгалуженнями 24](#_Toc360300658)

[3.7. Деякі довідкові дані для роботи з Git 26](#_Toc360300659)

[4. Питання до самопідготовки. 27](#_Toc360300660)

[5. Завдання для самостійної роботи 27](#_Toc360300661)

[5.1. Перелік завдань за послідовним планом 27](#_Toc360300662)

[5.2. Варіанти завдань 28](#_Toc360300663)

[5.3. Окремі завдання по роботі з гілками 28](#_Toc360300664)

[Джерела 29](#_Toc360300665)

# 

# Вступ

Технологія програмування - це технологія розробки програм і програмної продукції для ЕОМ, що відповідає вимогам високої надійності і заданої якості програмного забезпечення з використанням належної методології, засобів і систем програмування. Накопичений до теперішнього часу досвід створення систем програмного забезпечення (ПЗ) показує, що це складна і трудомістка робота, що вимагає високої кваліфікації фахівців, які беруть в ній участь. Курс "технології програмування та створення програмного продукту" (ТПСПП), є однією з найважливіших складових підготовки фахівців цієї категорії. Написаний у рамках цього курсу посібник допоможе студентам освоїти методи структурного аналізу і проектування ПЗ, основні підходи до організації процесу створення і використання програмних засобів. Додатковий матеріал зібраний в цьому посібнику допоможе студентові розібратися в методичних основах технологій створення ПЗ, а також зрозуміти сучасні тенденції в програмній інженерії.

Посібник надає можливість ознайомитися та отримати навики роботи в розподілено системі Git, призначеної для швидкого та ефективного контролю версій, як в невеликих так і в масштабних проектах. Git є системою з відкритим кодом. Кожен клон Git становить собою повнофункціональний репозиторій с повним набором можливостей стеження за редакціями та історією змін. Використання Git для контролю версій файлів походить на використання таких систем, як Mercurial, Subversion, CVS, Perforce і т.п.

# Системи контролю версій

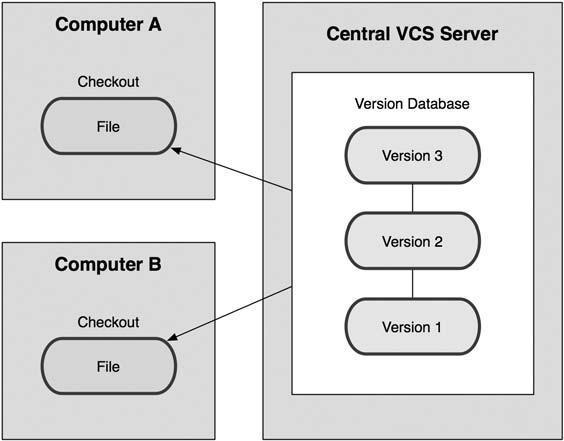
Кожному з нас знайома ситуація, коли втративши важливі дані, ми ставимо питання: а чому я не зробив резервну копію? Ми ж, звичайно, вчимося на своїх помилках, і починаємо робити резервні копії. Тоді випливають інші проблеми: резервних копій стає велика купа, ми не знаємо коли і для чого та чи інша була створена; ми зробили резервну копію давно, і деякі зміни нам треба залишити, а деякі відкинути. В проекті що складається не з одного єдиного файлу, вручну за цим всім стежити стає доволі складно. Це сумно і забирає багато часу. Знову вчимося на своїх помилках: починаємо використати системи контролю версій.

Система контролю версій (СКВ) – це програмний комплекс, який веде історію усіх змін у файлі/файлах, дає можливість повернутися до попередніх версій, об'єднувати зміни зроблені одночасно декількома програмістами і так далі. Отже СКВ ­– це потужний, програмний інструмент для керування версіями одиниці інформації: вихідного коду програми, скрипту, веб-сторінки, веб-сайту, 3D моделі, текстового документу, тощо. Ефективна розробка програмного забезпечення в будь-якому сучасному проекті передбачає можливість вести розробку паралельно з іншими учасниками проекту. Система керування версіями дозволяє одночасно, без завад один одному, проводити роботу над груповими проектами та, зазвичай, використовується при розробці програмного забезпечення для відстеження, документування та контролю над поступовими змінами в електронних документах, кресленнях, електронних моделях та інших документах, над змінами яких одночасно працюють декілька людей. Кожна версія позначається унікальною цифрою чи літерою, зміни документу занотовуються. Зазвичай також зберігається автор зробленої зміни та її час. Якщо ви графічний або веб-дизайнер і бажаєте зберігати кожну версію зображення або макету, а це вам, напевно, знадобиться - то користуватися системою управління версіями буде дуже мудрим рішенням. СКВ дозволяє повернути файли до первинного виду, повернути до колишнього стану увесь проект, порівняти зміни з деякого часу, побачити, хто останнім змінював модуль, який дав збій, хто створив проблему, і так далі. Взагалі, якщо ви зіпсували або втратили усі файли, все можна легко відновити, користуючись СКВ, крім того, витрати на це будуть дуже малі.

Інструменти для контролю версій входять до складу багатьох інтегрованих середовищ розробки. Система збереження історії редагувань статей, що застосовується у Вікіпедії є прикладом системи керування версіями.

Системи керування версіями існують двох основних типів: з **централізованим сховищем** та **розподіленим (децентралізованим)**.

Централізовані системи управління версіями (ЦСКВ), як наприклад, CVS, Team Foundation Server, Subversion і Perforce, мають центральний сервер, на якому зберігаються усі відстежувані файли, і ряд клієнтів, які отримують копії файлів з нього. Є можливість створення та роботи з локальними репозиторіями (робочими копіями). Це виглядає приблизно так:



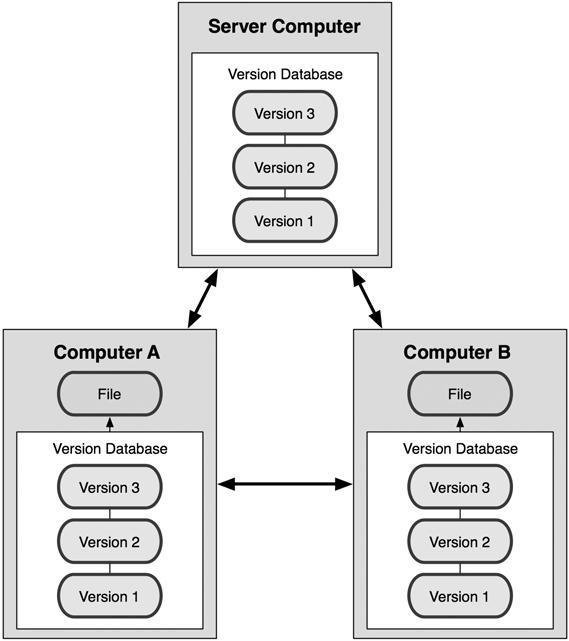
Багато років це був стандарт управління версіями. Такий підхід має безліч переваг, особливо над локальними СКВ.

Переваги:

* Загальна нумерація версій;
* Дані знаходяться на одному сервері;
* Можлива реалізація функції блокування файлів;
* Можливість керування доступом до файлів.

Отже, у адміністраторів є чіткий контроль над тим хто і що може робити, усі знають, хто і чим займається в проекті і, звичайно, адмініструвати ЦСКВ набагато легше, ніж локальні бази. Але є і декілька серйозних недоліків. Найбільш очевидний: потреба в мережевому з'єднанні для оновлення робочої копії чи збереження змін.

Отже, централізований сервер є вразливим місцем усієї системи. Якщо сервер вимикається на годину, то впродовж години розробники не можуть взаємодіяти, і ніхто не може зберегти нові версії. Якщо ж ушкоджується диск з центральною базою даних і не має резервної копіі проекту, ви втрачаєте абсолютно все - усю історію проекту, хіба що, за винятком декількох робочих версій, що збереглися на робочих машинах користувачів. Локальні системи управління версіями схильні до тієї ж проблеми: якщо уся історія проекту зберігається в одному місці, ви ризикуєте втратити все. У такій ситуації виграють розподілені системи управління версіями (РСУВ), це виглядає приблизно так:



В розподілених системах клієнти не просто забирають останні версії файлів, а повністю копіюють репозиторій. Тому у разі, коли не працює сервер через який йшла робота, будь-який клієнтський репозиторій може бути скопійований назад на сервер, щоб відновити базу даних. Кожного разу, коли клієнт забирає свіжу версію файлів, створюється повна копія усіх даних. Крім того, у більшій частині цих систем можна працювати з декількома видаленими репозиторіями, таким чином, можна одночасно працювати по-різному з різними групами людей у рамках одного проекту. Так, в одному проекті можна одночасно вести декілька типів робочих процесів, що не можливо в централізованих системах. До розподілених систем можливо віднести Git, Mercurial, Bazaar, або Darcs, SVK, Monotone, Codeville, BitKeeper. Для порівняння сформулюємо переваги і недоліки цього типу систем.

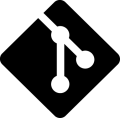
Переваги:

* Кожний з розробників працює зі своїм власним репозиторієм;
* Рішення щодо злиття гілок приймається керівниками проекту;
* Немає потреби в мережевому з'єднанні.

Недоліки:

* Немає можливості контролю доступу до файлів;
* Відсутня загальна нумерація версії файлу;
* Значно більша кількість необхідного дискового простору;
* Немає можливості блокування файлів.

## Система контролю версій Git

Git призначена для контрольованого внесення змін до процесу розробки програмного забезпечення, а також при написанні статей і іншому роді діяльності, пов'язаному з редагуванням текстових файлів (текстів програм). Проект був створений Лінусом Торвальдсом для управління розробкою ядра [Linux](http://ru.wikipedia.org/wiki/Linux_%28%D1%8F%D0%B4%D1%80%D0%BE%29), перша версія вийшла 7 квітня [2005 року](http://ru.wikipedia.org/wiki/2005_%D0%B3%D0%BE%D0%B4). На сьогоднішній день підтримується [Джуніо Хамано](http://ru.wikipedia.org/wiki/%D0%A5%D0%B0%D0%BC%D0%B0%D0%BD%D0%BE,_%D0%94%D0%B6%D1%83%D0%BD%D0%B8%D0%BE)[[1]](#footnote-2). Як говориться в документації: "Git — це швидка, масштабна, розподілена система керування версіями з незвичайно багатим набором команд, які забезпечують як операції верхнього рівня, так і повний доступ к внутрішнім механізмам". Прикладами проектів, що використовують Git, є [ядро Linux](http://ru.wikipedia.org/wiki/Linux_%28%D1%8F%D0%B4%D1%80%D0%BE%29), [Android](http://ru.wikipedia.org/wiki/Android), [Drupal](http://ru.wikipedia.org/wiki/Drupal), [Cairo](http://ru.wikipedia.org/wiki/Cairo), [GNU Core Utilities](http://ru.wikipedia.org/wiki/GNU_Coreutils), [Mesa](http://ru.wikipedia.org/wiki/Mesa), [Wine](http://ru.wikipedia.org/wiki/Wine), [Chromium](http://ru.wikipedia.org/wiki/Chromium), Compiz Fusion, [FlightGear](http://ru.wikipedia.org/wiki/FlightGear), [jQuery](http://ru.wikipedia.org/wiki/JQuery), [PHP](http://ru.wikipedia.org/wiki/PHP), [NASM](http://ru.wikipedia.org/wiki/NASM), [MediaWiki](http://ru.wikipedia.org/wiki/MediaWiki) та деякі дистрибутіви [Linux](http://ru.wikipedia.org/wiki/Linux).

Ядро Git є набором утиліт командного рядка з параметрами. Усі налаштування зберігаються в текстових файлах конфігурації. Маємо сенс згадати відмінності конфігураційних файлів Git: існує системний конфігураційний файл, який використовується для всіх, це */etc/gitconfig.* Також у кожного користувача є свій конфігураційний файл – */.gitconfig,* налаштування якого перекривають системні, та у кожного проекту є свій конфігураційний файл *.git/config*, який перекриває налаштування користувача. Така реалізація робить Git легким для перенесення на будь-яку платформу і дає можливість легко інтегрувати Git в інші системи (зокрема, створювати графічні Git- клієнти з будь-яким бажаним інтерфейсом).

Репозиторій Git є каталогом файлової системи, в якому знаходяться файли конфігурації репозиторія, файли журналів, які зберігають операції, що виконуються над репозиторієм, індекс, що описує розташування файлів і сховище, що містить власне файли. Структура сховища файлів не відбиває реальну структуру файлового дерева, що зберігається в репозиторії, вона орієнтована на підвищення швидкості виконання операцій з репозиторієм. Коли ядро обробляє команду зміни (неважливо, при локальних змінах або при отриманні патча[[2]](#footnote-3) від іншого вузла), воно створює в сховищі нові файли, що відповідають новим станам змінених файлів. Істотно, що ніякі операції не змінюють вмісту вже існуючих в сховищі файлів.

Внесені зміни можна контролювати за допомогою протоколу, який веде система. Крім того, система контролю версій в автоматичному режимі відстежує коректність змін, що вносяться, при одночасній розробці і запобігає ситуацій, що суперечать одна одної при виправленні коду.

## Можливості і недоліки Git

Контроль версій програм:

* автоматичне резервне копіювання поточної версії програми і відповідних файлів з кодів;
* зручне зведення внесених змін – система дозволяє відстежити зміни, що відбуваються, в текстах програм, що робить зручною спільну розробку;
* інтелектуальне відстежування змін, що суперечать одна одній.

Спільна розробка програми групою програмістів:

* контроль відповідальності: хто, коли і які зміни вніс в тексти;
* статистика вкладу учасників в розробку проекту.

Створення версій розробки:

* проведення експериментів, не зачіпаючи основного коду програми. Механізм гілок дозволяє з легкістю проводити експерименти над проектом, не зачіпаючи основного коду;
* спільна модульна розробка: окремі модулі, які надалі включаються в загальний проект;
* легке управління версіями: злиття версій в основний проект, переміщення версій, видалення версій.

Використання системи контролю версій для написання статей:

* спільне написання з контролем відповідальності і розділенням обов'язків;
* зручність відстежування змін.

До недоліків можна віднести:

* великі витрати часу на формування історії конкретного файлу;
* відсутність окремої команди перейменування/перенесення файлу;
* система не має можливості відстежувати порожні каталоги.

## Особливості системи Git

Головна відмінність Git від будь-яких інших СКВ (наприклад, Subversion і їй подібних) – це те, як Git «дивиться» на дані. Як правило, більшість інших систем зберігають інформацію як список змін (патчів) для файлів. Ці системи (CVS, Subversion, Perforce, Bazaar і інші) відносяться до даних, що зберігаються, як до набору файлів і змін, зроблених для кожного з цих файлів в часі. Git не зберігає свої дані у такому вигляді. Замість цього Git вважає дані, що зберігаються набором зліпків невеликої файлової системи. Кожного разу, коли ви фіксуєте поточну версію проекту, Git, по суті, зберігає зліпок того, як виглядають усі файли проекту на поточний момент. Заради ефективності, якщо файл не змінювався, Git не зберігає файл знову, а робить посилання на раніше збережений файл. Це важлива відмінність Git від практично усіх інших систем управління версіями. Git більше схожий на невелику файлову систему з неймовірно потужними інструментами, працюючими поверх неї, чим на просто СКВ.

Для здійснення більшості операцій в Git потрібні тільки локальні файли і ресурси, тобто зазвичай інформація з інших комп'ютерів в мережі не потрібна. Якщо ви користувалися централізованими системами, де практично на кожну операцію накладається мережева затримка, ви, будете здивовані, оскільки в Git більшість операцій виглядають практично миттєвими завдяки тому, що уся історія проекту зберігається локально у вас на диску. Приміром, щоб показати історію проекту, Git не треба викачувати її з сервера, він просто «читає» її прямо з вашого локального репозиторія. Тому історію ви побачите практично миттєво. Якщо вам треба проглянути зміни між поточною версією файлу і версією, зробленою місяць тому, Git може узяти файл місячної давнини і обчислити різницю на місці, замість того щоб запросити різницю у сервера СКВ, або брати з нього стару версію файлу і робити локальне порівняння.

Git стежить за цілісністю даних. Перед збереженням будь-якого файлу Git обчислює контрольну суму, і вона стає індексом цього файлу. Тому неможливо змінити вміст файлу або каталогу так щоб Git не дізнався про це. Ця функціональність вбудована в сам фундамент Git і являється важливій складовій його філософії. Якщо інформація загубиться при передачі або ушкодиться на диску, Git завжди це виявить. Механізм, використовуваний Git для обчислення контрольних сум, називається SHA-1 хеш. Це рядок з 40 шістнадцятирічних знаків (0-9 і a-f), який обчислюється на основі вмісту файлу або структури каталогу Git, що зберігається. SHA-1 хеш виглядає приблизно так: 4b9da6552252987aa493b52f8696cd6d3b00373. Обчислити хеш-функцію SHA-1 можна дуже швидко. Працюючи з Git, ви постійно зустрічатимете ці хеши, оскільки вони широко використовуються. Фактично, у своїй базі даних Git зберігає все не по іменах файлів, а по хешам їх вмісту. Таким чином, репозиторій Git – це просто колекція об’єктів, об’єднаннях посиланнями один на одного. Кожний об’єкт - це файл спеціального формату. У кожного об’єкта є «ім’я», яке обчислюється як SHA1 хеш. Якщо вміст двох файлів співпадає – то їх хеши теж співпадають, якщо вміст двох файлів відрізняється, то їх хеши теж різні. Таким чином у репозиторії ніколи не зберігаються два однакових файла.

Отже, практично, усі дії, які ви здійснюєте в Git, тільки додають дані у базу. Дуже складно змусити систему видалити дані або зробити щось таке, що потім неможливо буде відмінити. Можна, як і у будь-який іншій СКВ, втратити дані, які ви ще не зберегли, але як тільки вони зафіксовані – їх дуже складно втратити.

Легка у використанні і одночасно багата функціонально, система контролю версій Git покликана стати незамінним інструментом будь-якого розробника.

# Склад системи Git

Використання системи складається з декілька суттєвих блоків і в кожному потрібно особливе програмне забезпечення. Умовно блоки можна поділити на серверні та клієнтські.

## Серверний блок

Серверний блок – дещо нагадує Web-hosting для розміщення сайтів. Тобто для організації спільної роботи команди над проектом необхідно мати сервер з відповідним програмним забезпеченням. Під проектом розуміється сукупність текстових та інших файлів, що будуть створюватися та змінюватися незалежно декількома користувачами.

Сервер Git може бути розміщений у локальній мережі, або в Інтернет. Розміщення в локальній мережі є більш безпечним. Останній, хмаровий (cloud) варіант на наш погляд є більш гнучким та (при використанні готового сервісу) не потребує додаткового адміністрування та налагодження. Найбільш поширені сервіси, що надають хостінг для Git-репозиторієв: GitHub, Codebase, SourceForge, Gitorious, Google Code, Bitbucket.

Розглянемо варіант використання хостінгу Github. Сайт *Github.com* э веб-сервіс хостінгу проектів з використанням системи контролю версій Git, а також як соціальна сіть для розробників. Користувачі мають змогу створювати необмежене число репозиторієв, для кожного з яких надається wiki[[3]](#footnote-4), система issue tracking-а (відстежування), є можливості проводити code review (перегляд коду). Також на GitHub можливо обговорювати програмні проекти, організовувати суспільства програмістів, ділитися кодом тощо; складати та публікувати документацію к програмним продуктам, і ця функція на GitHub зроблена зручніше, чим у будь-якого з конкуруючих сервісів. GitHub на теперішній час є самим популярним сервісом такого роду, обігнавши Sourceforge, Google Code та Bitbucket. Для open-souce та навчальних проектів використання сайту безкоштовно.

## Клієнтські блоки

Клієнтських блоків зазвичай декілька. Зазвичай до їх складу входять:

* служба роботи з Git;
* програма графічного інтерфейсу або інтеграція в провідник операційної системи;
* програмне забезпечення розробки (IDE, текстові та інші редактори, SDK мов програмування).

Основна робота розробника проходить саме серед цих блоків. Безпосередньо до Git відносяться два перші розділи які можна завантажити зі сторінки ***https://Github.com*** після реєстрації.

Якщо ви не хочете або просто бракує часу детально розібратися з консольними командами Git, то для роботи с Git під Windows існують графічні інтерфейси. Окрім вбудованого, що встановлюється разом з системою, існує декілька досить зручних і популярних, наприклад, TortoiseGit, SmartGit (кросплатформовий). [Перш за все розглянемо вбудований](https://www.google.com.ua/search?biw=1022&bih=770&q=%D0%9F%D0%B5%D1%80%D1%88+%D0%B7%D0%B0+%D0%B2%D1%81%D0%B5+%D1%80%D0%BE%D0%B7%D0%B3%D0%BB%D1%8F%D0%BD%D0%B5%D0%BC%D0%BE+%D0%B2%D0%B1%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9&spell=1&sa=X&ei=KrjJUZCyCMe7O4LogZAI&ved=0CCcQvwUoAA) клієнт Git GUI, потім зупинимось на TortoiseGit.

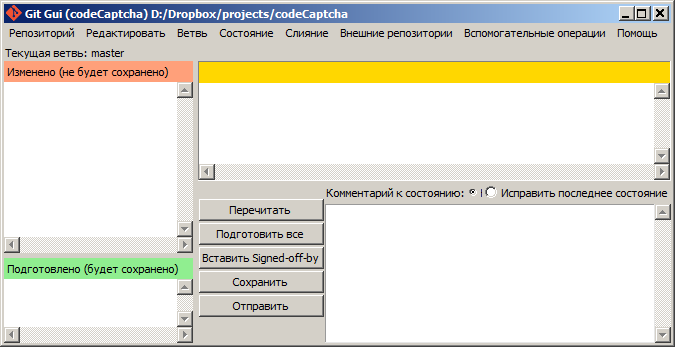
## Графічний інтерфейс Git GUI

Цей графічний клієнт не є самим популярним, але досить прийнятний і, якщо не дуже прискіпуватись, може здатися цілком прийнятним. Після встановлення системи в системному меню з’являється рядок для його запуску. Перше вікно клієнта дає змогу зробити початковий вибір:

Створити новий репозиторій відповідає відкриттю нового проекту. Перше, що буде запропоновано у цьому режимі – обрати папку на диску, що містить (чи буде містити) файли проекту. Тобто ми легко можемо скорегувати проекти Git та обраної системи розробки IDE (наприклад, MS Visual Studio, Eclipse та інше).

Клонування існуючого проекту допомагає створити так званий форк (від англійського **fork** – відгалуження). Тобто можна заснувати новий проект як відгалуження від вже існуючої розробки.

Обрати існуючий проект – ви можете поновити роботу з вже створеним раніше репозиторієм. Для зручності останній з них вказується нижче. Якщо репозиторій ще не містить робочих документів (файлів програм, документації, тощо), то ми побачимо таке вікно:



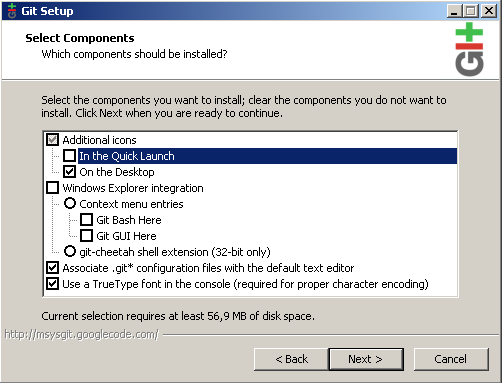
Опції меню дозволяють виконати усі основні операції Git.

## Графічний інтерфейс TortoiseGit

TortoiseGit – це візуальний клієнт системи управління текстами програм Git для ОС Microsoft Windows. По суті, TortoiseGit після натискання команди з контекстного меню запускає консольну команду із Git (MSysGit) та виносить в віконний інтерфейс її вивід. TortoiseGit інтегрується в оболонку Windows (тобто в провідник). Це означає, що ви маєте можливість працювати з інструментами, якими ви вже знайомі і вам не прийдеться переходити на іншу прикладну програму кожного разу, коли виникає потреба в функціях контролю версій. Всі Git команди доступні з контекстного меню провідника, TortoiseGit додає своє особисте підменю там. Контекстне меню TortoiseGit працює в багатьох інших файлових менеджерах, однак ви повинні мати на увазі, що TortoiseGit спеціально розроблений як розширення для Windows Explorer.

Розглянемо особливості встановлення Git з цим клієнтом. Це дещо складніше ніж установка і використання вбудованого графічного клієнта.

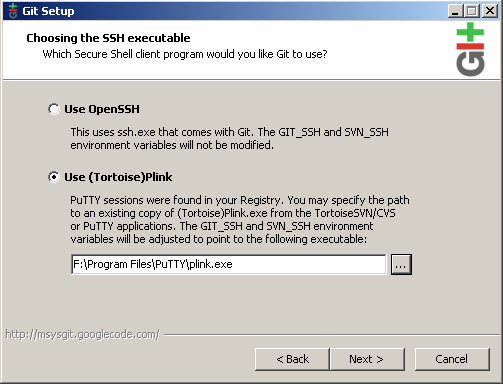
Для 32-х бітної системи Windows, оберемо для завантаження, наприклад, msysgit – Git-1.8.1.0-preview20120710.exe. Запускаємо інсталлер. В пункті «Select Components» треба зняти галочку з «Windows Explorer integration» (отже цим буде займатися TortoiseGIT), також зняти галочку з «Addititional icons/Quick Launch»:



При завантаженні буде запропоновано вибрати тип запуску Git:

* Use Git bash only: Git ставиться і викликається командою контекстного меню “Git bash here”/”Git gui here”
* Run Git from the Windows Command Prompt: встановлює Git і прописує шлях до консольної версії в PATH. Команду 'Git Bash here' все одно можна використати
* Run Git and included Unix from the Windows Command Prompt: те ж що попередній варіант, але додатково прописує в Windows шлях до різних Unix- утиліт типу find і sort. Git попереджає нас що при цьому замість windows-застосувань з відповідними іменами викликатимуться unix-аналоги.

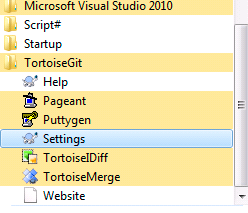
Перший варіант більш прийнятий. Продовжуємо установку. В наступному кроці нам пропонують обрати SSH клієнта. Обираємо (Tortoise) *plink* та вказуємо директорію, куди був встановлений *PuTTY*:



Putty – це декілька окремих програм, призначених для роботи с unix-сервером по протоколам SSH1, SSH2, Telnet, Rlogin, Raw. Автором Putty є Simon Tatham[[4]](#footnote-5). Повний комплект програм, під загальною назвою Putty, складається з декількох утиліт, їх призначення:

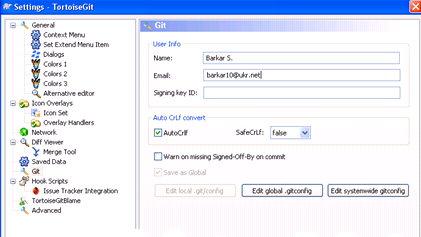
* putty.exe – клієнт для з'єднання з сервером по протоколах telnet, ssh, raw, rlogin;
* puttytel.exe – тільки telnet клієнт;
* puttygen.exe – генератор rsa/dsa ключів;
* pagent.exe – агент аутентифікації, зберігає ключі в пам'яті, при його використанні не треба самостійно вводити ключову фразу паролю;
* plink.exe – інтерфейс командного рядка для putty;
* pscp.exe – безпечне копіювання файлів;
* psftp.exe – безпечний ftp клієнт для копіювання, перегляду, перейменування файлів, тощо.

Наприкінці Git запропонує проглянути файл приміток до релізу, на цьому установка закінчується. Після установлення Git треба перевантажитися або завершити сеанс користувача і знову увійти, щоб застосувалися зміни в системній змінній PATH. Отже, після установлення Git, почнемо інсталювати клієнт для Git, який можна знайти на сайті: *http://code.google.com/p/TortoiseGit*. В процесі установлення програми, нам буде запропонований вибір SSH[[5]](#footnote-6) клієнта, вибираємо «Putty» – перший пункт (адже ми хочемо безпечно передавати наші тексти в мережі.).

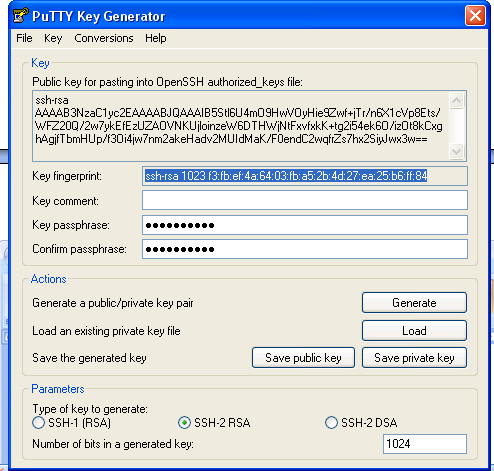
Після інсталяції, одразу ж зробимо налаштування деяких параметрів. Для цього заходимо в *Tortoise*→*Settings:*

Заходимо в пункт Git и задаємо своє ім’я та e-mail (потім ця інформація буде відображатися в наших commit-ах),також, ставимо галочку у пункті «*AutoCrlf*»:

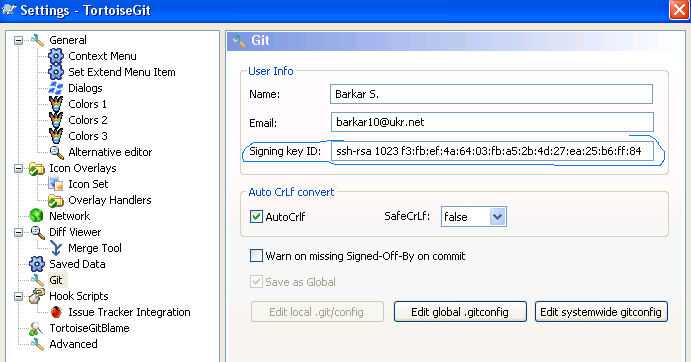
Клієнт ми налаштували, тепер необхідно генерувати SSH-ключи для безпечного передавання даних. Для цього заходимо в *Tortoise* → *Puttygen*.



Натискаємо кнопку «*Generate*» та хаотично проводимо мишкою по екрану (для генерування ключа). Після цього, записуємо пароль (для захисту ключа) в поле «*Key passpharase*» і підтверджуємо його в наступному полі, (щоб не забути пароль, можливо записати підказ в полі «*Key comment*»). Зберігаєямо наші ключі, кнопки: «*Save public key / Save private key*»:



Копіюємо наш генерований публічний ключ, далі заходимо в налаштування акаунта Git та вставляємо його в відповідне поле:



# Використання GitHub

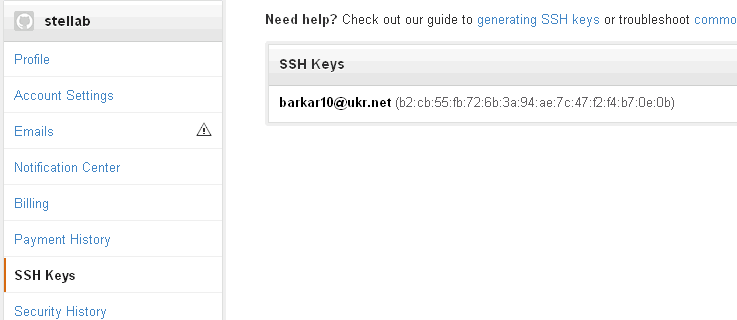
Використання системи ми будемо розглядати на прикладі рішення від GitHub.com як одне з найбільш цілісних і зручних.

## Встановлення SSH-ключа

Робота системи цілком якісно проходить з використанням протоколу http. Це не потребує ніяких додаткових дій. Але для промислових систем зазвичай використовують SSH доступ. Через це корисно буде розглянути процедуру створення ключів і використання відповідних протоколів аутентифікації та безпечного обміну інформацією.

Відразу після реєстрації необхідно прописати в системі GutHub свій публічний ключ шифрування (відкритий SSH-ключ). Для додавання ключа, потрібно в таблиці вибрати розділ «SSH-keys».

У вікні, що відкрилося, треба кликнути на посилання «Add SSH Keys». З'являться два поля – назва ключа (Title) та вміст ключа (Key). В полі «Title» можна написати назву комп’ютера, на якому сгенерован публічний ключ. Можливо записати російською. В полі «Key» потрібно вставити вміст файлу id\_rsa.pub, котрий знаходиться в каталозі *C:\Documents and Settings\username\.ssh*. Переходимо в цей каталог, відкриваємо будь-яким текстовим редактором файл *id\_rsa.pub* (саме з розширенням *.pub*). Виділяємо увесь текст, копіюємо і вставляємо на сторінці GitHub в поле Key. Після додавання ключа, комп'ютер може з'єднуватися з GitHub через програму Git.



## Створення репозиторія

Щоб мати можливість спільної роботи над яким-небудь Git- проектом, необхідно знати, як управляти репозиторіями на далечіні. Самі по собі репозиторіями на далечіні – це модифікації проекту, які зберігаються в інтернет-мережі. Їх може бути декілька, кожен з яких, як правило, доступний для вас або тільки для читання, або для читання і редагування. Спільна робота вміщує управління репозиторіями на далечіні та розміщення (push) і отримання (pull) даних з них тоді, коли потрібно обмінятися результатами роботи. Управління включає уміння додавати репозиторії, видаляти ті з них, які більше не діють, уміння управляти різними гілками на далечіні визначати їх як відстежувані (tracked) або ні та інше. Репозиторій можливо розглядати просто як директорію, в якій лежатимуть файли, що синхронізуються, і піддиректорії. Важливо розуміти, що git-репозиторій це набор файлів, папка «.git» і нічого більше. Створювати репозиторії потрібно в web-інтерфейсі GitHub, а наповнювати його файлами і працювати з ним можна буде вже за допомогою програми Git на своєму комп'ютері.

Для створення репозиторія, треба в правому верхньому кутку натиснути значок-посилання «Create a Repository*»,* який відкриє діалог додавання нового репозиторія. У діалозі додавання нового репозиторію треба заповнити, як найменш, поле назви проекту «Project Name». У назві проекту краще не використовувати кирилицю, оскільки ім'я проекту - це по факту ім'я директорії. Для уникнення проблем краще, щоб ім'я проекту містило тільки латиницю. Після натиснення кнопки «Create Repository», репозиторій буде створено.

  Діюче посилання на репозиторій в системі GitHub формується наступним чином. Якщо ви зареєструвалися під ім'ям «*username*», і ваш репозиторій називається «*repositname»*, то для доступу до цього репозиторію можна використати наступні посилання:

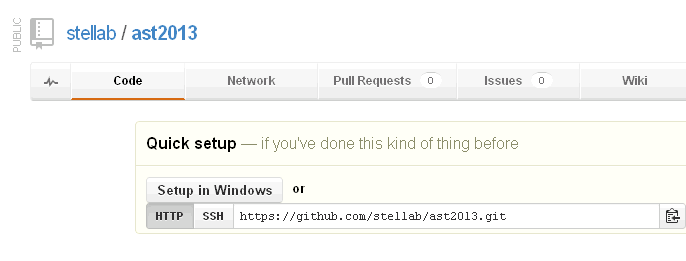
у синтаксисі Git : *Git@Github.com:username/repositname.Git*

у синтаксисі *Https:https://username@Github.com/username/reponame.Git*

Тепер ваш проект розміщений на GitHub'е, та ви можете дати послання на нього будь-кому, з ким ви бажаєте розділити проект.

## Клонування репозиторія

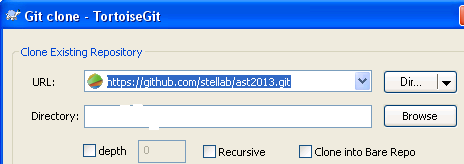
**Щоб зробити клон репозиторія заходимо на сторінку проекту, і копіюємо у буфер адресу:**



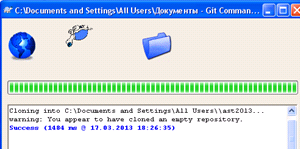
Далі перемикаємося на роботу з TortoiseGit. Обираємо директорію на своєму жорсткому диску де ми зберігатимемо тексти та**, в**икликаємо контекстне меню і обираємо пункт «**Git Clone»:**

http://www.webdelphi.ru/wp-content/uploads/2010/07/tts15-300x60.png

У вікні, що відкрилося, в полі *URL* вставляємо скопійовану адресу, поле *Directory* заповнюється автоматично, натиснемо "Ok".



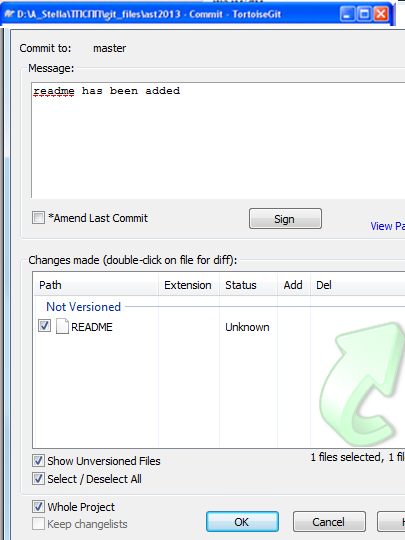
Почнеться процес клонування репозиторію.

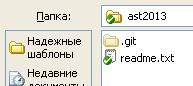


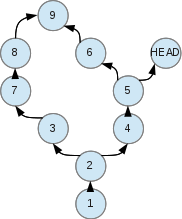
Після завершення в вашій директорії з’явиться нова папка-клон вашого проекту з цім же ім’ям.

## Внесення змін в локальний репозиторій

Змінимо локальний репозиторій: додамо в нього файл readme (файл з ім’ям *README* обробляється github спеціальним чином – його вміст буде відображатися в якості опису репозиторія на відповідній сторінці). Создаємо файл *redme.txt*, в який занесемо деяку інформацію про наш проект. Далі зробимо «комміт» змін в локальний репозиторій наступним чином: в контекстному меню обираємо *Commit* → *master, де master* – головна гілка проекту.



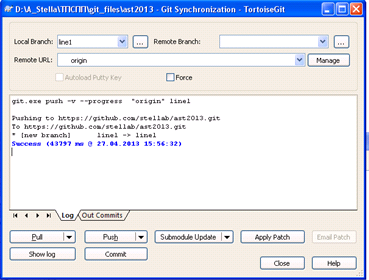
Заповнюємо поле з описом, ставимо відповідні «галочки», натискаємо «*Оk*». Якщо ви бажаєте одразу відправити ці зміни в репозиторій, то в вікні натисніть кнопку «*push*». Якщо ж вас не влаштовує відправляти по-одному файлу, або зміни зовсім незначні, що нема сенсу їх відправляти на сервер, то продовжуєте подальшу роботу над проектом, а «*push*» виконуєте пізніше. Тільки ця команда передає зміни в інший репозиторій, тому не бійтеся ставити експерименти, видаляти, змінювати. Ви маєте змогу робити все що завгодно, виконувати будь-які команди, головне не виконуйте команду «*push*», тільки так можна щось «зламати». Але, суворо кажучи, навіть невдалий «*push*» можна виправити.

Кожний комміт може мати декілька коммітів-предків та декілька дочірніх-коммітів. Ми маємо змогу переходити в будь-яку точку (відновлювати будь-який попередній стан) цього дерева, а точніше- графа. Одже, гілкою дерева є покажчик на який-небудь комміт. Таким чином, для того щоб створити гілку, треба просто дати ім’я будь-якому комміту. Щоб злити дві гілки, одна з яких починається з кінця другої, можна просто передвинути покажчик другої гілки на новий комміт (це називається Fast-Forward). Для того щоб підтримувати «плоску» основну master гілку, використовується техніка rebase’а перед злиттям і злиття без

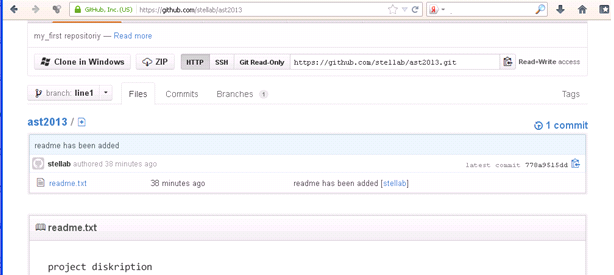
fast-forward'а. Детальніше про це говориться в п.**3.6**.

## Синхронізація змін з головним репозиторієм

Переходимо в папку з локальним репозиторієм, обираємо в контекстному меню команду «*Git sync»*. У вікні в полі «*Local Branch»* обираємо з списку головну, або іншу гілку до якої приєднуємо файли, натискаємо «*push*». В процесі синхронізації буде запрошено ваш пароль та логін на github.



Після цього всі додані файли ви побачите в головному репозиторії.



## Робота з розгалуженнями

Уся робота в Git побудована з урахуванням можливості утворення гілки розвитку проекту з будь-якого місця. Якщо потрібно додати в проект нову функціональність або виправити помічений недолік, то треба дотримуватися такої схеми:

1. Створити гілку від мастер-гілки;
2. Внести зміни;
3. Впевнитися в робочому стані нового проекту;
4. Злити гілку з master-гілкою.

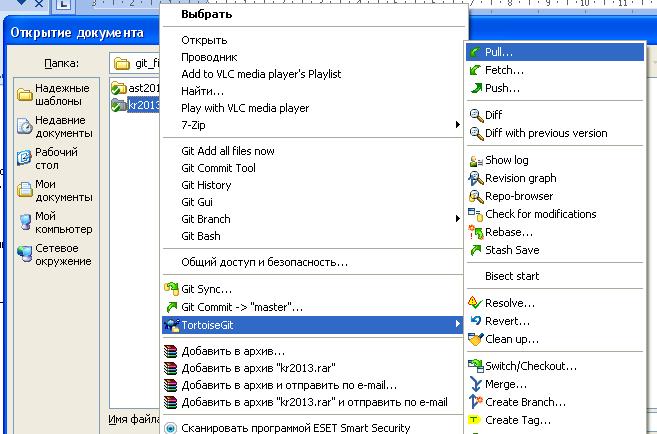
При цьому в процесі особливу роль відіграють команди розгалуження та злиття.

В Git створення гілки («*branch*») — це створення нового вузла дерева з відповідним місцем розгалуження (*parent*). Наприклад, після клонування репозиторія ми за замовченням знаходимося в гілці *master*, створюємо нову гілку *test1* (в яку буде все злито з *master*), далі вносимо деякі зміни, робимо комміт, а потім переключимося назад на гілку *master*. В кожній з гілок, можна буде продивитися всі комміти. В гілці *master* ви побачите, що ваш комміт з *test* на гілку *master* не розповсюджується. Таким чином гілку можна вважати «сучасним станом» вашої розробки.

Представимо більш детальний алгоритм роботи над проектом двох розробників:

1. Створюється гілка, що ґрунтується на останній копії master гілки. Радимо в назву нової гілки включати клас завдання, номер завдання та короткий опис.
2. Усі зміни проводяться у середині цієї гілки. При кожному елементарному, внесенні змін створюється свій комміт (наприклад, додали плагін – «закоммітили» додавання; зробили уточнення API одній функції в усіх місцях – «закоммітили» і т.п.). Це дозволяє розділяти всі зроблені зміни, спрощується читання і перевірка на помилки коду і всього процесу розробки.
3. Після того, як робота над кодом в гілці закінчена і все відлагоджено, усі зміни «закоммічені», дана гілка ***rebase***’иться відносно останньої *master* гілки, і робиться «***push***» в центральний репозиторій. ***Rebase*** – означає зміну предка гілки на новий комміт. При ***rebase***’і усі зміни зроблені в цій гілці «відтворюються назад» і зберігаються у вигляді змін, внесених кожним коммітом. Після чого покажчик гілки переноситься на нове начало, і з нього послідовно починають застосовуватися зміни. Якщо конфліктів немає, то зміни накладаються автоматично, після чого гілка є набором змін відносно нового начала. Якщо тепер зробити злиття цієї гілки з початковою, покажчик голови початкової гілки буде просто пересунутий на нове місце, і ми втратимо інформацію про те, що взагалі існувала нова гілка. Саме тому використовується злиття без «fast-forward» (дивиться п.3.4).
4. Інша людина, працююча над тим же проектом, робить до себе «***pull***» з центрального репозиторія. Якщо треба, видаляє свою локальну копію гілки, після чого перемикається на вказану гілку. Прочитує код, перевіряє його працездатність, після чого, якщо там виявлені проблеми, доробляє, або одразу робить ще раз «***rebase***» поверх *master*, і злиття гілки з *master* гілкою.
5. Після злиття з *master* вже закінченного фрагменту коду, розробник (можливо рецензент) робить «***push***» нової *master* гілки в центральний репозиторій та видаляє у себе локальну гілку завдання.
6. Перший розробник видаляє локальну гілку задачі після того як задача була закрита та внесені зміни були додані в *master*.

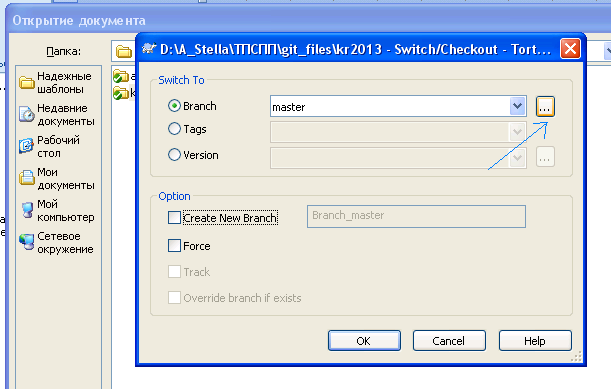
Далі роз’яснимо роботу з використаними командами. Для оновлення поточної гілки з центрального репозиторія в контекстному меню вашої папки з файлами обираємо *TortoiseGit->Pull*



Для відправлення поточної гілки в центральний репозиторій в тому ж контекстному меню обираємо *TortoiseGit→Push*. Далі обираємо в ***Local*** спочатку *master*, потім нашу поточну гілку. ***Remote*** заповнюється при цьому автоматично. Назва ***remote*** гілки повинна бути ідентична ***Local***.

Для переключення на будь-яку гілку в контекстному меню обираємо *TortoiseGit→Switch/Checkout*. В вікні, що відкриється, *Switch/Checkout* в полі «*Branch»*, обираємо з списку потрібну гілку remotes/origin/<назва гілки>. Далі встановити галочки на опціях «Create new branch» і «Track», та ще раз додаємо в полі назву потрібної гілки.

Для видалення гілок відкриваємо меню *TortoiseGit→Switch/Checkout*, заходимо в «*Browse Reference*»:



В розділі «*heads*» розташовані локальні гілки, в розділі «*remotes/origin*» – гілки розташовані на далечині. Обираємо потрібну та в контексному меню обираємо команду «*Delete Remote Branch*», для локальної гілки («*Delete Branch*»).

Для злиття гілки с поточною, обираємо контекстне меню *TortoiseGit → Merge*, далі обираємо «*Branch*», який потрібно злити з поточною гілкою, ставимо галочку на опції «*No fast forward*», «*Merge message*» залишаємо без змін.

У якості самостійного завдання спробуйте ознайомитися з іншими функціями *Check for modifications, Stash Save, Create Tag* за інтернет посиланнями, що надаються в переліку використаних джерел.

## Деякі довідкові дані для роботи з Git

**pull** – отримання змін з центрального репозиторія; ця дія забирає зміни та проводить ь злиття з активною гілкою;

**push** –  **д**одавання змін в центральний репозиторій;

**init** – створення, репозиторія;

**add и rm** – індексація змін в репозиторії;

**status** – відображення стану проекту, змінені але не додані файли, індексовані файли, активна гілка;

**commit** – зміни, відправленні в репозиторій (зафіксовані);

**reset** – повертання к певному комміту (від англ. ***commit*** – фіксувати), повертання змін;

**revert** – скасування змін, зроблених раніше окремим ***commit***;

**log** – різноманітна інформація о коммітах в цілому;

**diff** – відмінність між гілками проекту, коммітами і т.д.;

**checkout** – переключення між гілками (тобто взяти з репозиторія будь- яке його становище;

**merge** – злиття гілок;

**rebase** – побудування рівної лінії коммітів;

**gitignore** – файл, який **с**лужить для роз’яснення файлів, що були ігноровані.

**index** – область зафіксованих змін, тобто все що, ви підготували к збереженню в репозиторій;

**head**– покажчик на commit, в якому ми знаходимося;

**master** – им’я гілки за замовченням (це також покажчик на визначений комміт;

**origin** – им’я репозиторія на далечині за замовченням.

# Завдання

Самостійна робота за денною темою передбачає групову роботу з використанням мережі Інтернет. Студенти, що надходять до однієї навчальної групи разом виконують роботу над спільним проектом. Але частково завдання повинні виконуватися індивідуально. В завданнях використане таке маркування:

⬩ – завдання виконується окремо кожним студентом

❖ – завдання виконується спільно усією групою

## Перелік завдань за послідовним планом

1. Реєстрація на сайті github.com (⬩)
2. Загрузка та встановлення програмного забезпечення (⬩)
3. Створення репозиторія та відкриття проекту (❖)
4. Розробка спільного технічного завдання та
5. Розробка спільної специфікації програми (❖)
6. Утворення гілки з власним варіантом завдання (описом методу) (⬩)
7. Періодичні оновлення та узгодження версій текстів опису.
8. Розробка програмного забезпечення (❖)
9. Утворення гілки з власним варіантом програми (⬩)
10. Періодичні оновлення та узгодження версій текстів програм
11. Злиття з головною гілкою (⬩)

1. Джунио Хамано ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Junio Hamano) – програміст з 2005 р., координатор проекту Git. Приймає активну участь в розробці ядра [Linux](http://ru.wikipedia.org/wiki/Linux),та утіліти [GNU](http://ru.wikipedia.org/wiki/GNU) [Tar](http://ru.wikipedia.org/wiki/Tar). Закінчив [Токійський універсітет](http://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BA%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%83%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%82%D0%B5%D1%82), проживає в штаті [Каліфорнія](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BB%D0%B8%D1%84%D0%BE%D1%80%D0%BD%D0%B8%D1%8F). [↑](#footnote-ref-2)
2. патч – файл, що описує відмінність між файлами [↑](#footnote-ref-3)
3. Можливість редагувати сторінку сайту або створювати нові сторінки на вікі-сайти, з використанням звичайного веб-браузера. [↑](#footnote-ref-4)
4. англійський професійний програміст та розробник вільного програмного забезпечення. [↑](#footnote-ref-5)
5. мережевий протокол, що дозволяє проводити віддалене управління [комп'ютером](http://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80) і передачу [файлів](http://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB). Схожий за функціональністю з протоколом [Telnet](http://uk.wikipedia.org/wiki/Telnet) і [rlogin](http://uk.wikipedia.org/wiki/Rlogin), проте використовує алгоритми [шифрування](http://uk.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) інформації, що передається. SSH-клієнти і SSH-сервери доступні для більшості мережевих операційних систем. [↑](#footnote-ref-6)