

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ  
“ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Р.В. ПУГАЧОВ, Н.Ю. ЛЮБЧЕНКО, М.О. СОБОЛЬ

## **СИСТЕМИ КОНТРОЛЮ ВЕРСІЯМИ**

**Навчально-методичний посібник  
для студентів комп'ютерних спеціальностей  
вищих навчальних закладів**

Затверджено редакційно-  
видавничою радою НТУ "ХПІ",  
протокол № 1 від 16.01.2019

Харків  
НТУ “ХПІ”  
2019

УДК 511.2(076)

П88

*Рецензенти:*

*В.Б. Успенський*, д-р техн. наук, доцент, Національний технічний університет "Харківський політехнічний інститут"

*О.М. Резніченко*, канд. техн. наук, старший науковий співробітник, Радіоастрономічний інститут НАН України

Автори: Р.В. Пугачов, к.т.н., доцент

Н.Ю. Любченко, к.т.н., доцент

М.О. Соболев, к.т.н.

**Пугачов Р.В.**

П88 Системи контролю версіями: навч.-метод. посібник / Пугачов Р.В., Любченко Н.Ю. Соболев М.О.– Харків : НТУ "ХПІ", 2019. – 130 с.

Навчально-методичний посібник складається з десяти практичних робіт, які охоплюють першу частину курсу «Технології створення програмних продуктів», присвячені знайомству з найпопулярнішими на сьогоднішній день системами контролю версій Git та SVN. Розглядаються способи роботи у командному рядку (терміналі), за допомогою графічних клієнтів та у спеціалізованих середовищах розробки IntelliJ Idea та Eclipse.

Для студентів комп'ютерних спеціальностей вищих навчальних закладів.

Іл. 53. Табл. 4. Бібліогр. 8 назв.

УДК 511.2(076)

ISBN

© Р.В. Пугачов, Н.Ю. Любченко, М.О. Соболев 2019

## ВСТУП

Уміння писати грамотний код – це лише частина роботи програміста. Йому також необхідно вміти використовувати різні інструменти, що дозволяють оптимізувати, полегшити роботу. Розробка програмного забезпечення (ПЗ) ведеться групами розробників, кожен з яких може як створювати свої файли з вихідним кодом, так і змінювати створені іншими файли. Для контролю змін вихідних файлів, зберігання різних версій ПЗ розробники застосовують системи контролю версіями.

Навчально-методичний посібник складається з десяти практичних робіт, які охоплюють першу частину дисциплін «Інженерія програмного забезпечення» та «Управління ІТ-проектами», присвячені знайомству з найпопулярнішими на сьогоднішній день системами контролю версій Git та SVN. В кінці кожної практичної роботи є список контрольних питань, що дозволяють перевірити ступінь засвоєння матеріалу, який вивчається.

У розробці і тестуванні наведених у посібнику практичних робіт активну участь приймали студенти кафедри інформатики та інтелектуальної власності Сергій Кононов, Роман Ошека, Дмитро Міщенко та Роман Ахромєєв.

## ЗНАЙОМСТВО З СИСТЕМОЮ КОНТРОЛЮ ВЕРСІЙ *Git*

**Мета:** знайомство з порядком застосування, завантаження, встановлення та налагодження системи контролю версій (СКВ) *Git*

### *Теоретичні відомості*

#### *Системи контролю версій*

Уміння писати грамотний код – це лише частина роботи програміста. Йому також необхідно вміти використовувати різні інструменти, що дозволяють оптимізувати, полегшити роботу. Розробка програмного забезпечення (ПЗ) ведеться групами розробників, кожен з яких може як створювати свої файли з вихідним кодом, так і змінювати створені іншими файли. Для контролю змін вихідних файлів, зберігання різних версій ПЗ розробники застосовують системи контролю версіями (СКВ). Основними завданнями, які виконують СКВ, є:

- зберігання файлів в репозиторії;
- підтримка перевірки файлів в репозиторії;
- створення різних варіантів одного документа, так званої гілки, із загальною історією змін до точки розгалуження і з різними – після неї;
- знаходження конфліктів при зміні вихідного коду і забезпечення синхронізації при роботі в середовищі з багатьма користувачами розробки.
- відстеження авторів змін;
- надання інформації про те, хто і коли додав або змінив конкретний набір рядків у файлі;
- ведення журналу змін, в який користувачі можуть записувати пояснення про те, що і чому вони змінили в цій версії;
- контроль прав доступу користувачів, дозволяючи або забороняючи читання або зміну даних, залежно від того, хто запитує цю дію.

Принципово СКВ поділяються на дві великі групи: централізовані (до них відносяться SVN, CVS та ін.) і розподілені (Git, Mercurial та ін.). Основною відмінністю в них є принцип побудови репозиторіїв: у розподілених системах розробник працює з ізольованим локальним репозиторієм, не зачіпаючи головну гілку.

Однією з найбільш популярних СКВ, особливо серед розробників Open-source проєктів, на сьогодні є Git – розподілена система керування версіями файлів. Проєкт був створений Лінусом Торвалдсом для управління розробкою ядра Linux, перша версія випущена 7 квітня 2005 року.

### ***Вимоги до виконання практичної роботи***

Для того щоб приступити до використання системи контролю версій (СКВ) *Git*, а також познайомитися з порядком її застосування, потрібно завантажити та встановити дистрибутив на комп’ютері та виконати налаштування *Git*.

#### ***Завантаження Git***

Для завантаження *Git* треба перейти на сайт <http://git-scm.com/>. На сторінці буде посилання на актуальну на теперішній час версію дистрибутива (рис. 1.1).



Рисунок 1.1 – Вибір версії Git

#### ***Інсталяція Git***

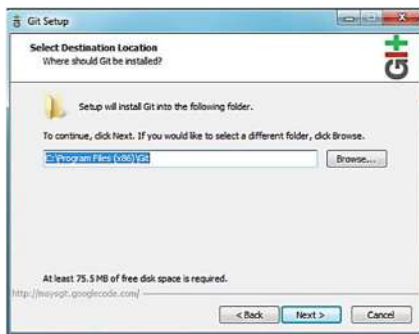
При інсталяції необхідно указати каталог для установки и деякі параметри (рис. 1.2 а – ж).



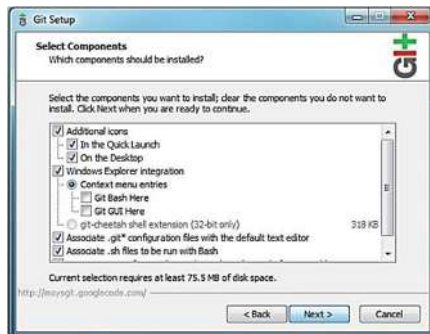
*a*



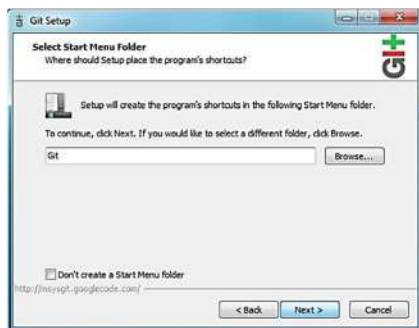
*б*



*в*



*г*



*д*

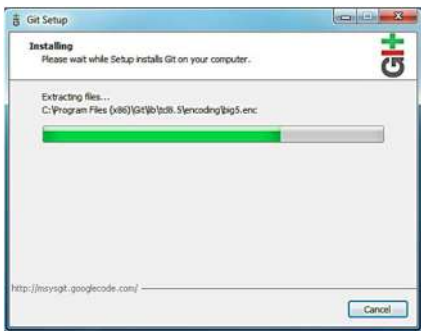


*е*

Рисунок 1.2 – Вибір параметрів для установки *Git*



*e*



*ε*



*ж*

Рисунок 1.2 – Вибір параметрів для установки *Git* (продовження)

### *Налаштування Git*

Після установки на робочому столі повинен з'явитися ярлик *Git Bash*. Після запуску системи контролю версій з'явиться консоль (рис. 1.3).

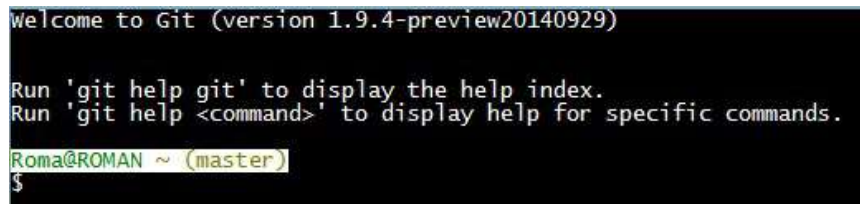


Рисунок 1.3 – Консоль системи контролю версій *Git*

*Короткий словник термінів, які будуть використані у практичних роботах з вивчення Git.*

Робоче дерево (*Working tree*) – будь-яка директорія у файловій системі, яка пов'язана з репозиторієм (що можна бачити за наявності в ній піддиректорії «.git»). Включає в себе всі файли і піддиректорії.

Коміт (*Commit*) – у ролі іменника: «моментальний знімок» робочого дерева в якийсь момент часу. У ролі дієслова: комітити (закомітити) – додавати Коміт в репозиторій.

Репозиторій (*Repository*) – це набір комітів, тобто просто архів минулих станів робочого дерева проекту.

Гілка (*Branch*) – просто ім'я для комітів, також зване посиланням (reference). Визначає походження – «родовід» комітів, і таким чином, є типовим представником «гілки розробки».

*Checkout* – операція перемикавання між гілками або відновлення файлів робочого дерева.

Злиття (*Merge*) – поєднання змін у гілках.

Звичайно, слова на зразок «закомітити» є сленгом, проте у середовищі розробників вони наразі дуже популярні, тому в цих роботах також будуть використовуватися.

### ***Порядок виконання практичної роботи***

У ході роботи ми виконаємо найтипівіші дії з репозиторієм.

1. Створити локальний репозиторій.
2. Додати файл у репозиторій.
3. Змінити файл у репозиторії (*commit*).
4. Видалити файл з репозиторія (*commit*).
5. Виконати *checkout* та створити гілку у репозиторії.
6. З'єднати гілки (операція *merge*).
7. Отримати історію ревізій (*change log*) по будь-якому файлу.

#### ***1. Створити локальний репозиторій***

Для того щоб створити локальний репозиторій, необхідно додати нову директорію для роботи, перейти до неї. Командою ***git init*** треба проініціалізувати порожній репозиторій (рис. 1.4).



```
Roma@ROMAN ~ (master)
$ cd Desktop/gitTest/

Roma@ROMAN ~/Desktop/gitTest (master)
$ git init
Reinitialized existing Git repository in c:/Users/Roma/Desktop/gitTest/.git/

Roma@ROMAN ~/Desktop/gitTest (master)
$
```

Рисунок 1.4 – Ініціалізація порожнього репозиторію

## 2. Додати файл у репозиторій

Створити у новій директорії порожній файл *example.txt*. Перейти у консоль та ввести команду **git status** (рис. 1.5).

```
Untracked files:
(use "git add <file>..." to include in what will be committed)

example.txt
```

Рисунок 1.5 – Створення файлу *example.txt*

Командою **git add example.txt** додається файл під версійний контроль. Якщо потрібно додати усі нові файли у директорії, то використовується команда **git add**.

Після цього треба виконати **commit** (закомітити) (рис. 1.6). за допомогою команди **git commit -m 'Added file'**

```
Roma@ROMAN ~/Desktop/gitTest (master)
$ git add example.txt

Roma@ROMAN ~/Desktop/gitTest (master)
$ git commit -m 'Added file'
[master ab30be2] Added file
1 file changed
create mode 100644 example.txt
```

Рисунок 1.6 – Додавання файлу під версійний контроль

## 3. Змінити файл у репозиторії (commit)

Додати в файл *example.txt* рядок “Hello World” та зберегти файл.

Далі повторюються дії щодо додавання файлу до репозиторію (див. попередній пункт).

#### 4. Видалити файл з репозиторію (*commit*)

Командою ***git rm example.txt*** видаляється файл з репозиторію. Після цього виконуємо *commit* («комітимосся») – ***git commit -m 'removed file'*** (рис. 1.7).



```
Roma@ROMAN ~/Desktop/gitTest (master)
$ git rm example.txt
rm 'example.txt'

Roma@ROMAN ~/Desktop/gitTest (master)
$ git commit -m 'removed file'
[master 9f5935a] removed file
1 file changed, 1 deletion(-)
delete mode 100644 example.txt
```

Рисунок 1.7 – Видалення файлу з репозиторію

#### 5. Виконати *checkout* та створити гілку у репозиторії

Командою ***git checkout -b development*** переключаємося на нову гілку з назвою *development* (рис. 1.8). Принципи створення нових гілок (розгалуження) буде розглянуто далі.



```
Roma@ROMAN ~/Desktop/gitTest (master)
$ git checkout -b development
Switched to a new branch 'development'

Roma@ROMAN ~/Desktop/gitTest (development)
$
```

Рисунок 1.8 – Перемикання між гілками (*checkout*)

#### 6. З'єднати гілки (*merge*)

Командою ***git merge master development*** з'єднуються гілки *development* та *master* (рис. 1.9).

```
Roma@ROMAN ~/Desktop/gitTest (development)
$ git merge master development
Already up-to-date.

Roma@ROMAN ~/Desktop/gitTest (development)
$ git status
On branch development
nothing to commit, working directory clean
```

Рисунок 1.9 – З'єднання гілок

#### 7. Отримати історію ревізій (*change log*) за будь-яким файлом

Командою **git log** отримуємо історію ревізій (*change log*) за фалом *example.txt* (рис. 1.10).

```
Roma@ROMAN ~/Desktop/gitTest (development)
$ git log
commit 659ca09264f64de6c9a531296c46b04a42d7b03c
Author: 
Date: Sun Oct 19 20:02:05 2014 +0300

    delet

commit 3d58227ffa58e0d035ff9e7337c747dc28009747
Author: 
Date: Sun Oct 19 20:01:28 2014 +0300

    changes

commit 57a0e73ae217539b8072b7c9037cd55ff3b8aff0
Author: 
Date: Sun Oct 19 20:00:36 2014 +0300

    test

Roma@ROMAN ~/Desktop/gitTest (development)
$
```

Рисунок 1.10 – Отримання історії ревізій за файлом *example.txt*

### **Висновок**

У ході практичної роботи ознайомилися з основними функціоналом системи контролю версій *Git*, а також виконали найбільш типові дії з репозиторієм.

У звіті не забудьте навести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### **Контрольні запитання**

1. Для чого потрібні системи контролю версіями (СКВ)?
2. Які завдання виконують СКВ?

3. На які групи за архітектурою побудови поділяються СКВ?
4. Що таке «Робоче дерево»?
5. Пояснити термін «коміт».
6. Пояснити терміни Checkout та Merge. У чому їхня різниця?
7. Як створити локальний репозиторій?
8. Як видалити файл з репозиторію?
9. Як отримати історію ревізій за будь-яким файлом у репозиторії?

#### Практична робота 2

### ПОЧАТОК РОБОТИ З GITHUB

**Мета:** навчитися використовувати основні інструменти *GitHub*

#### **Теоретичні відомості**

**GitHub** – один з найбільших веб-сервісів для спільної розробки програмного забезпечення. Існують безкоштовні та платні тарифні плани користування сайтом. Базується на системі керування версіями *Git* і розроблений на *Ruby on Rails* і *Erlang* компанією GitHub, Inc (раніше Logical Awesome).

Сервіс безкоштовний для проектів з відкритим вихідним кодом, з наданням користувачам усіх своїх можливостей, а для окремих індивідуальних проектів пропонуються різні платні тарифні плани. На платних тарифних планах можна створювати приватні репозиторії, доступні обмеженому колу користувачів.

Розробники сайту називають *GitHub* «соціальною мережею для розробників». Окрім розміщення коду, учасники можуть спілкуватись, коментувати редагування один одного, а також слідкувати за новинами знайомих. За допомогою широких можливостей *Git* програмісти можуть поєднувати свої репозиторії – *GitHub* дає зручний інтерфейс для цього і може показувати вклад кожного учасника у вигляді дерева.

Є можливість прямого додавання нових файлів у свій репозиторій через веб-інтерфейс сервісу.

Код проектів можна не лише скопіювати через *Git*, але й завантажити у вигляді архіву. Окрім *Git*, сервіс підтримує отримання і редагування коду через *SVN* та *Mercurial*.

### ***Вимоги до виконання практичної роботи***

Для проходження даної практичної роботи необхідно зареєструватися і створити *GitHub* аккаунт (рис. 2.1).



Рисунок 2.1 – Створення *GitHub* аккаунта

### ***Порядок виконання практичної роботи***

1. Створення репозиторію.
2. Створення гілки.
3. Створення і додавання змін до репозиторію (*commit*).
4. Відкриття вкладки злиття гілок *Pull Request*.
5. Злиття (*Merge*) *Pull Request*.

#### ***1. Створення репозиторію***

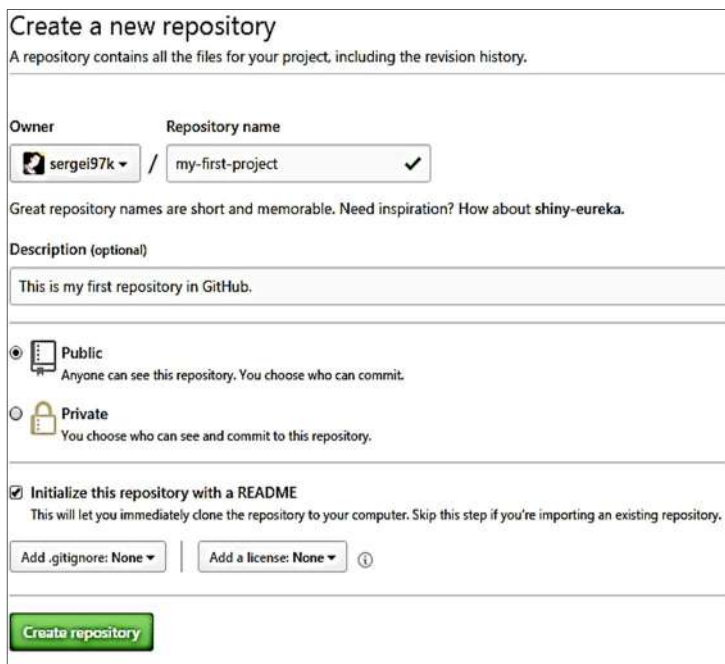
Репозиторій – це сховище, яке використовується для організації проекту. Репозиторій може містити в собі папки і файли (зображення, відео, таблиці з даними).

Сервіс *GitHub* рекомендує додавати файл *README* або інформацію про робочий проект.

Для створення репозиторію (рис. 2.2) необхідно виконати такі дії:

- в правому верхньому кутку натиснути ***New repository***;
- написати назву репозиторія (наприклад, ***my-first-project***);
- написати короткий опис;
- вибрати (поставити галочку) ***Initialize this repository with a README***;

— натиснути *Create repository*.



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and 'A repository contains all the files for your project, including the revision history.' Below this, there are two main sections: 'Owner' and 'Repository name'. The 'Owner' is set to 'sergei97k' with a dropdown arrow. The 'Repository name' is 'my-first-project' with a checkmark. Below these, there is a note: 'Great repository names are short and memorable. Need inspiration? How about shiny-eureka.' Then, there is a 'Description (optional)' section with a text box containing 'This is my first repository in GitHub.' Below the description, there are two radio buttons for visibility: 'Public' (selected) and 'Private'. The 'Public' option has a note: 'Anyone can see this repository. You choose who can commit.' The 'Private' option has a note: 'You choose who can see and commit to this repository.' Below the visibility options, there is a checked checkbox for 'Initialize this repository with a README' and a note: 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.' At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None' with an information icon. Finally, there is a green 'Create repository' button.

Рисунок 2.2 – Створення репозиторію

## 2. Створення гілки

Розгалуження проекту – це спосіб працювати на різних версіях репозиторію в один час.

За замовчуванням сховище має одну гілку *master*, яка вважається кінцевою гілкою. При створенні нової гілки робиться копія гілки *master*.

Діаграма, наведена нижче, добре характеризує роботу розгалуження (рис. 2.3).

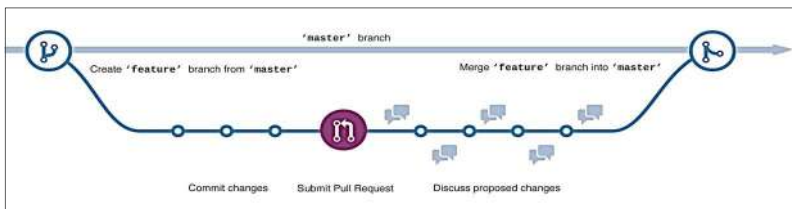


Рисунок 2.3 – Діаграма розгалуження

Для створення нової гілки (рис. 2.4) необхідно виконати такі дії:

- перейти в новий репозиторій (*my-first-project*);
- натиснути верхній список, що випадає, в якому буде написано *branch: master*;
- у текстовому полі ввести назву нової гілки (*edit-readme-file*);
- вибрати пункт *Create branch*.

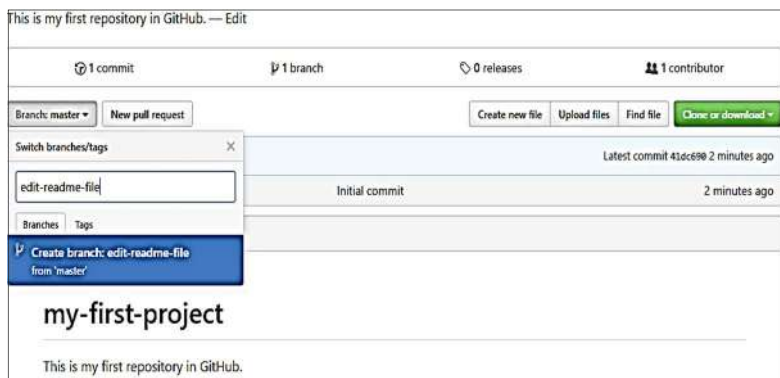


Рисунок 2.4 – Створення нової гілки

Після виконання перерахованих дій в репозиторії знаходяться дві гілки, які поки що нічим не відрізняються одна від одної. При створенні нової гілки вона автоматично стає активною.

### 3. Створення і додавання змін до репозиторію (*commit*)

Для того щоб побачити роботу гілок, необхідно додати зміни в файл. *Commit* – це опис, який пояснює, чому певна зміна було зроблена. Іс-

торія *commit*'ів являє собою послідовність змін. Таким чином, інші учасники проекту можуть зрозуміти, що було зроблено іншими учасниками і чому.

Для створення і додавання змін до репозиторію (рис. 2.5) необхідно виконати такі дії:

- натиснути на файл README.md;
- натиснути на іконку олівця в правому верхньому куті, для того, щоб зробити зміни у файлі;
- у редакторі додати будь-яку інформацію;
- в полі *Commit changes* написати назву поточного *commit*'а. Потім написати опис змін цього *commit*'а;
- натиснути кнопку *Commit changes*.

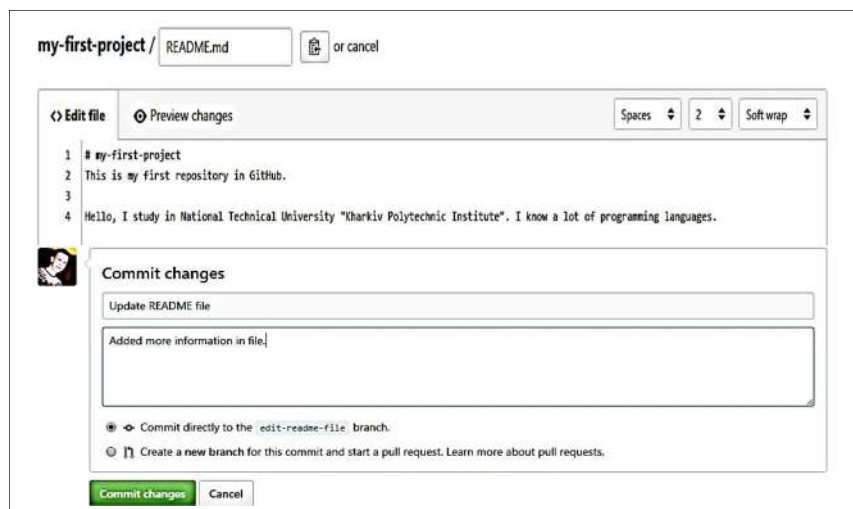


Рисунок 2.5 – Створення та *commit* змін

Ці зміни файлу будуть додані тільки в гілці *edit-readme-file*. Видно, що вміст файлу README.md відрізняється залежно від того, яка з гілок активна.

#### 4. Відкриття вкладки злиття гілок *Pull Request*

У створених гілках є відмінності, тому можна злити всі зміни в основну гілку *master*.



Вміст вкладки *Pull Request* показує відмінність контенту в обох гілках. Зміни, додавання і видалення підсвічені зеленим і червоним кольорами відповідно.

Для роботи з різними гілками необхідно виконати такі дії:

– перейти на вкладку *Pull Request* і натиснути кнопку *New pull request* (рис. 2.6);



Рисунок 2.6 – Вкладка *Pull Request*

– вибрати дві гілки для порівняння змін (рис. 2.7);



Рисунок 2.7 – Вибір гілок

– подивитися на зміни (рис. 2.8);

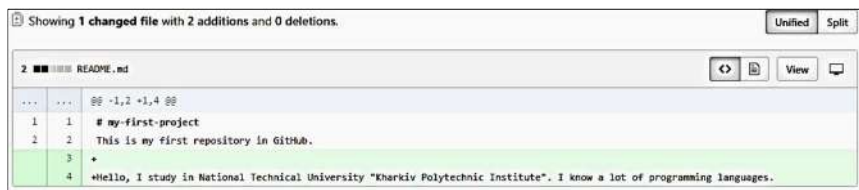


Рисунок 2.8 – Візуалізація змін

– якщо існуючі зміни коректні, натиснути кнопку *Create Pull Request* (рис. 2.9);

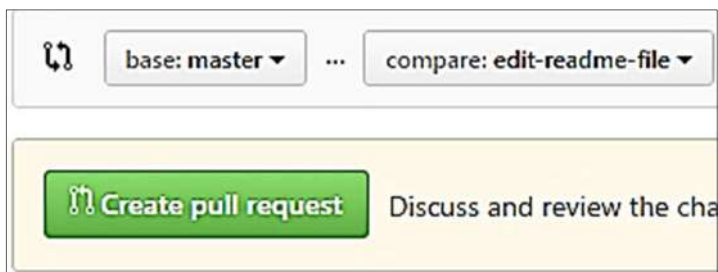


Рисунок 2.8 – Підтвердження змін

– для завершення роботи дати злиття *Pull Request* назву, написати короткий опис поточних змін і натиснути кнопку **Create Pull Request** (рис. 2.9).

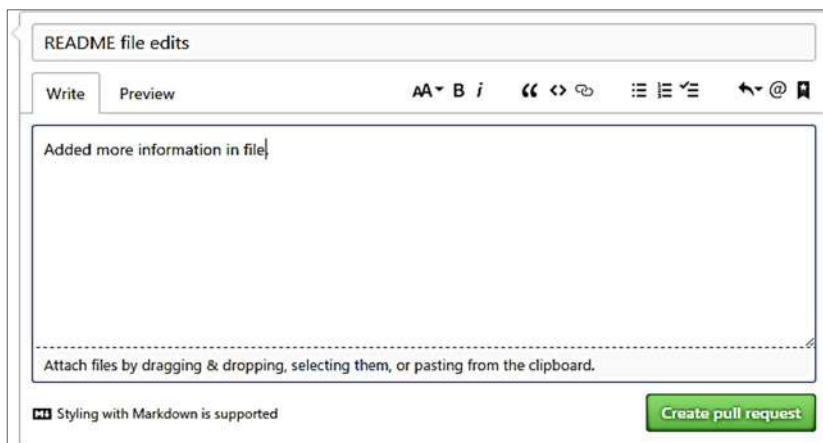


Рисунок 2.9 – Опис змін

### 5. Злиття (Merge) Pull Request

На останньому етапі необхідно зробити злиття всіх змін в основну гілку *master*.

Для цього необхідно виконати такі дії:

– натиснути кнопку **Merge pull request** (рис. 2.10);



Рисунок 2.10 – Початок роботи зі злиття змін

- натиснути кнопку *Confirm merge*;
- видалити гілку, оскільки її зміни вже включені в основну гілку. Для цього натиснути кнопку **Delete branch** (рис. 2.11).

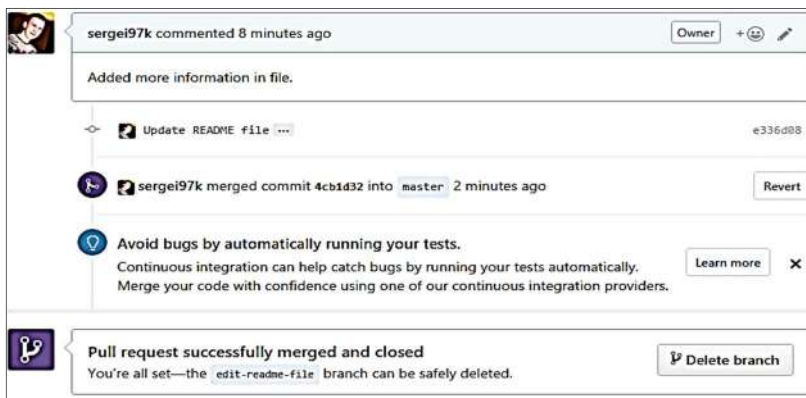


Рисунок 2.11 – Завершення роботи зі злиття змін

### **Висновок**

У ході практичної роботи ознайомилися з основними функціоналом сервісу *GitHub*:

- створення сховища з відкритим вихідним кодом;
- створення і управління новими гілками;
- зміна файлу і *commit* змін в *GitHub*;
- злиття гілок (*Pull Request*).

Також можна побачити, що на сторінці профілю з'явився зелений квадрат, який показують активність учасника і кількість зроблених *commit*'ів.

Якщо необхідно більше дізнатися про функціонал *GitHub*, то рекомендується прочитати керівництво *GitHub* (<https://guides.github.com/introduction/flow/>).

У звіті не забудьте навести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### ***Контрольні запитання***

1. Що таке *GitHub*?
2. Які системи контролю версій підтримуються в *GitHub*?
3. Яким чином створюється нова гілка в репозиторії на *GitHub*?
4. Як додати зміни до репозиторію?
5. Як називається основна гілка репозиторію?
6. Яким чином можна перевірити відмінність контенту в гілках?
7. Як називається команда злиття гілок?

## Практична робота 3

### РОБОТА З GIT ТА GITHUB

**Мета:** створення віддаленого сховища та робота з ним з використанням системи контролю версіями *git* і сервісу *GitHub*

#### ***Теоретичні відомості***

У попередніх роботах ми навчилися створювати локальний репозиторій, а також віддалений безпосередньо на сервісі *GitHub*. Настав час навчитися синхронізувати локальний та віддалений репозиторій.

#### ***Вимоги до виконання практичної роботи***

Для виконання даної практичної роботи необхідно мати встановлений локально *git* і аккаунт на *GitHub*.

#### ***Порядок виконання практичної роботи***

1. Створення нового репозиторія на *GitHub*.
2. Клонування віддаленого репозиторія.
3. Створення і зміни файлу.
4. Відправлення зміни у віддалений репозиторій.
5. Зміни та відправка файлів у віддалений репозиторій.
6. Ігнорування файлів.

##### ***1. Створення нового репозиторія на GitHub***

Для того щоб створити новий репозиторій (рис. 3.1), необхідно виконати такі дії:

- зайти в свій аккаунт на *GitHub* і натиснути на кнопку *New repository*;
- написати назву і опис робочого репозиторія;
- вибрати *Initialize this repository with a README*;
- створити репозиторій.

##### ***2. Клонування віддаленого репозиторія***

Для того щоб проводити зміни локально, необхідно отримати дану версію репозиторію.

Для цього необхідно виконати такі дії:


- натиснути кнопку *Clone or download* в правому верхньому кутку;
- вибрати *Clone with HTTPS* і скопіювати посилання (рис. 3.2);

## Create a new repository

A repository contains all the files for your project, including the revision history.

---

**Owner**      **Repository name**

 sergei97k / work-repository ✓

Great repository names are short and memorable. Need inspiration? How about [expert-giggle](#).

**Description (optional)**

This repository describe the work git and GitHub.

---

☒ **Public**  
Anyone can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

---

☒ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾      Add a license: **None** ▾ ⓘ

---

**Create repository**


Рисунок 3.1 – Створення нового репозиторію

This repository describe the work git and GitHub. — Edit

1 commit      1 branch      0 releases      1 contributor

Branch: master ▾      New pull request

Create new file      Upload files      Find file      **Clone or download ▾**

 sergei97k Initial commit	
 README.md	Initial commit
 README.md	

**Clone with HTTPS** ⓘ      Use SSH

Use Git or checkout with SVN using the web URL.

<https://github.com/sergei97k/work-reposito> 

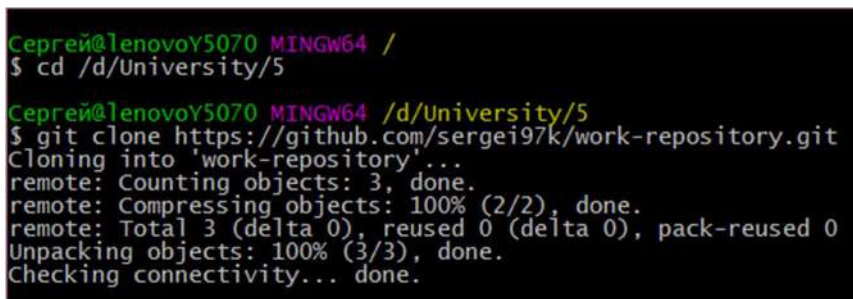
Open in Desktop      Download ZIP

## work-repository

This repository describe the work git and GitHub.

Рисунок 3.2 – Копіювання посилання

- запустити консольну версію *git* – *git-bash* і вибрати директорію, в яку хочемо зберегти робочий репозиторій;
- написати команду *git clone* і вставити збережене посилання (рис. 3.3);



```
Сергей@lenovoY5070 MINGW64 /  
$ cd /d/University/5  
  
Сергей@lenovoY5070 MINGW64 /d/University/5  
$ git clone https://github.com/sergei97k/work-repository.git  
Cloning into 'work-repository'...  
remote: Counting objects: 3, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), done.  
Checking connectivity... done.
```

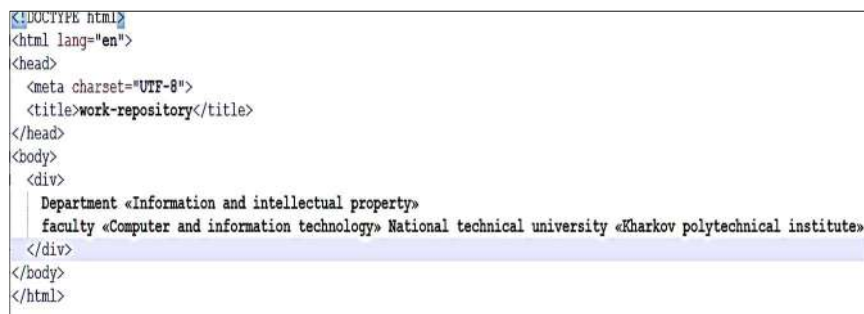
Рисунок 3.3 – Виконання команди *git clone*

- зайти у вибрану вище директорію і переконатися, що в ній з'явилася папка з назвою робочого сховища.

### 3. Створення і зміни файлу

Робоче сховище розміщено локально і можна робити перші зміни. Для цього необхідно створити файл *index.html* і виконати наступні дії:

- зайти в робочу папку і створити файл з таким змістом, як показано на рис. 3.4;



```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <title>work-repository</title>  
  </head>  
  <body>  
    <div>  
      Department «Information and intellectual property»  
      faculty «Computer and information technology» National technical university «Kharkov polytechnical institute»  
    </div>  
  </body>  
</html>
```

Рисунок 3.4 – Зміст файлу *index.html*

– зберегти зміни (рис. 3.5).

```
Сергей@lenovoY5070 MINGW64 /d/University/5
$ cd /d/University/5/work-repository

Сергей@lenovoY5070 MINGW64 /d/University/5/work-repository (master)
$ git add .

Сергей@lenovoY5070 MINGW64 /d/University/5/work-repository (master)
$ git commit -m "added index.html"
[master ab0f239] added index.html
1 file changed, 13 insertions(+)
create mode 100644 index.html
```

Рисунок 3.5 – Збереження змін

#### 4. Відправлення змін у віддалений репозиторій

Файл *index.html* знаходиться локально, але для того, щоб інші користувачі змогли побачити актуальні зміни, необхідно відправити їх в віддалений репозиторій на *GitHub*.

Для цього необхідно виконати такі дії:

– переконатися, що були зроблені всі необхідні зміни – команда *git log* (рис. 3.6);

```
$ git log
commit ab0f2392c3e5ce4e5539106cdb7f00ef6bb8f664
Author: Sergei Kononov <sergei97k@gmail.com>
Date:   Sun Oct 2 18:04:36 2016 +0300

    added index.html

commit 6041aa77d04d6ff2842d8caae82cf14ea93c6db8
Author: Sergei Kononov <sergei97k@gmail.com>
Date:   Sun Oct 2 17:32:47 2016 +0300

Initial commit
```

Рисунок 3.6 – Виконання команди *git log*

– для відправки змін у віддалений репозиторій необхідно написати команду *git push* (рис. 3.7).

*Примітка:* якщо потрібно відправити зміни на конкретну гілку, то необхідно використовувати команду *git push* з прапором *-u* і назвою гілки (наприклад: *git push -u origin master*);



```

$ git push
warning: push.default is unset; its implicit value has changed in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the traditional behavior, use:

    git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

    git config --global push.default simple

When push.default is set to 'matching', git will push local branches
to the remote branches that already exist with the same name.

Since Git 2.0, Git defaults to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 505 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/sergei97k/work-repository.git
 6041aa7..ab0f239 master -> master

```

Рисунок 3.7 – Виконання команди *git push*

- ввести власний логін і пароль на *GitHub*, після чого віддалений репозиторій буде змінений;
- зайти до головного репозиторію на *GitHub*, щоб побачити чи з'явилися зміни в віддаленому репозиторії (рис. 3.8).

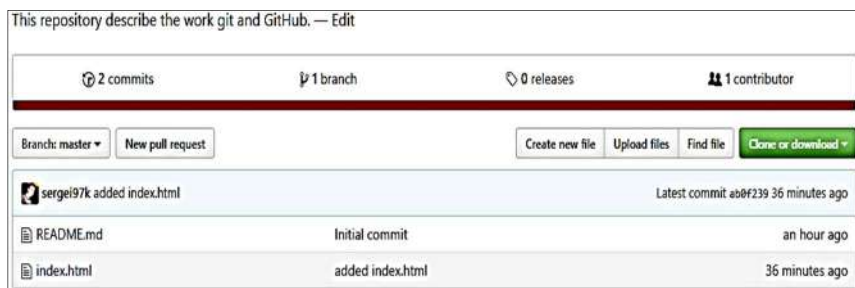


Рисунок 3.8 – Перегляд змін в репозиторії

### 5. Зміни та відправка файлів у віддалений репозиторій

Потрібно додати зміни в файл *index.html* на сторінці сховища. При натисканні на файл, можна побачити кнопку *history*. Тут зберігаються зміни, що стосуються конкретного файлу, а не всього сховища. Адже не в кожному *commit*’і можуть змінюватися усі файли сховища.

Для додавання змін до файлу необхідно зробити такі дії:

- натиснути на іконку олівця в правому верхньому куті, для того щоб зробити зміни у файлі;
- додати новий рядок в файл (рис. 3.9);

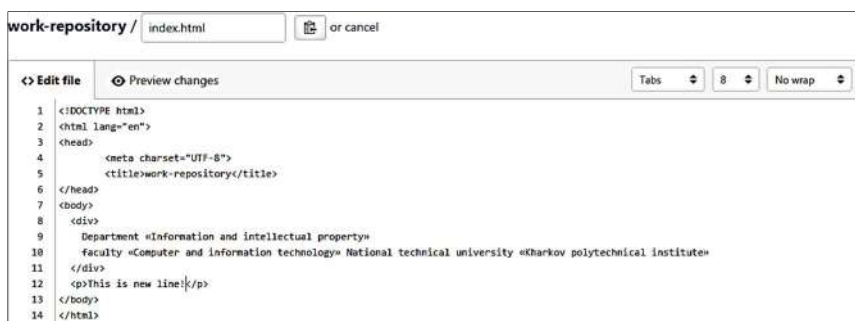


Рисунок 3.9 – Зміни до файлу

- ввести назву *commit*’у і його опис;
- натиснути кнопку **Commit changes** (рис. 3.10);

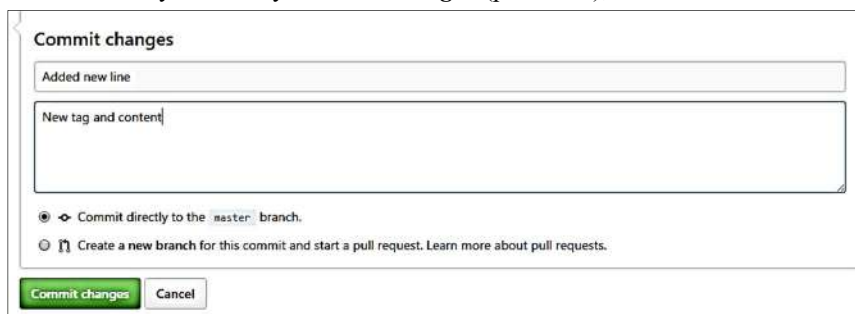


Рисунок 3.10 – Введення назви *commit*’а і його опису

- переконатися, що файл у віддаленому репозиторії змінився;
- перейти на файл і натиснути кнопку **blame**. Тут можна побачити хто і які зміни здійснював в даному файлі (рис. 3.11);

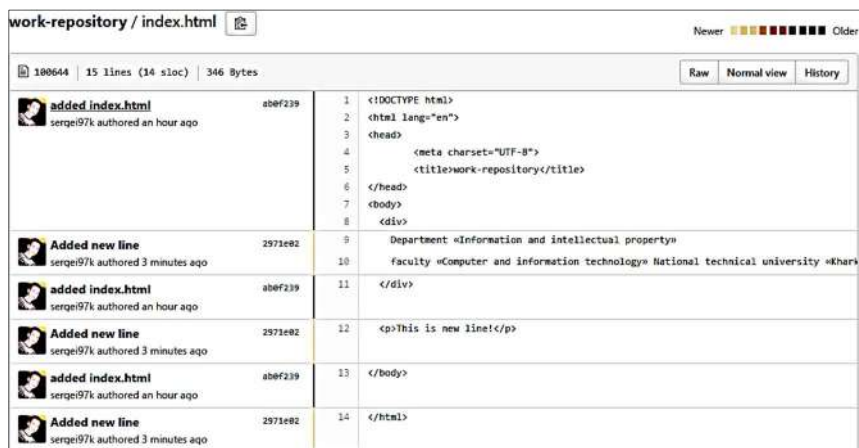


Рисунок 3.11 – Візуалізація змін в файлі

- написати в командному рядку `git pull`, для того щоб злити дані зміни в свій локальний репозиторій (рис. 3.12);

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/sergei97k/work-repository
ab0f239..2971e02 master -> origin/master
Updating ab0f239..2971e02
Fast-forward
 index.html | 7 ++++---
 1 file changed, 4 insertions(+), 3 deletions(-)
```

Рисунок 3.12 – Виконання команди `git pull`

- зайти в текстовий редактор і переконатися, що додано новий рядок (рис. 3.13). Таким чином відбувається синхронізація вашого локального сховища із розміщеним на *GitHub*.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>work-repository</title>
</head>
<body>
  <div>
    Department «Information and intellectual property»
    faculty «Computer and information technology» National technical university «Kharkov polytechnical institute»
  </div>
  <p>This is new line!</p>
</body>
</html>

```

Рисунок 3.13 – Візуалізація змін в файлі

## 6. Ігнорування файлів

Ігноровані файли не обов'язково розміщувати у віддалений репозиторий. По суті ці файли необхідні в процесі розробки, але можуть не знадобитися в основний гілці.

Для ігнорування файлів необхідно виконати такі дії:

- створити файл з розширенням **.gitignore**. Це можна зробити при створенні сховища або локально (рис. 3.14);

Рисунок 3.14 – Створення файлу

- створити файл **.gitignore** локально і текстовий файл, який треба ігнорувати (рис. 3.15);





 .gitignore	14.06.2016 17:54	Текстовый докум...	1 КБ
 hide-file.txt	02.10.2016 19:17	Текстовый докум...	0 КБ
 index.html	02.10.2016 19:02	JetBrains WebStorm	1 КБ
 README.md	02.10.2016 17:38	Файл "MD"	1 КБ

Рисунок 3.15 – Створення файла **.gitignore**

– відкрити файл **.gitignore** і записати назву файлу, який необхідно ігнорувати (рис. 3.16);

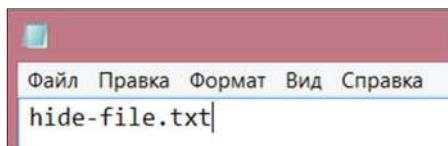


Рисунок 3.16 – Запис назви файлу для ігнорування

– зайти в командний рядок і переконатися, що файли змінилися (рис. 3.17), використовуючи команду **git status**;

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Рисунок 3.17 – Використання команди **git status**

– зберегти всі зміни командою **git add** (рис. 3.18).

```
$ git add .
Сергей@lenovoY5070 MINGW64 /d/University/5/work-repository (master)
$ git commit -m "added gitignore file"
[master 0f07db0] added gitignore file
1 file changed, 1 insertion(+)
create mode 100644 .gitignore
```

Рисунок 3.18 – Використання команди **git add**

– відправити зміни у віддалений репозиторій командою **git push** (рис. 3.19);

```

$ git push
warning: push.default is unset; its implicit value has changed in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the traditional behavior, use:

    git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

    git config --global push.default simple

When push.default is set to 'matching', git will push local branches
to the remote branches that already exist with the same name.

Since Git 2.0, Git defaults to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 331 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/sergei97k/work-repository.git
   2971e02..0f07db0  master -> master

```

Рисунок 3.19 – Використання команди *git push*

– переконатися, що у віддаленому репозиторії немає файлу *hide-file.txt* (рис. 3.20), хоча він є локально. Це зменшує розмір вашого сховища і звільняє його від непотрібних файлів.

Branch: master ▾ New pull request		Create new file Upload files Find file Clone or download ▾	
sergei97k added gitignore file		Latest commit 0f07db0 4 minutes ago	
📄 .gitignore	added gitignore file	4 minutes ago	
📄 README.md	Initial commit	2 hours ago	
📄 index.html	Added new line	37 minutes ago	

Рисунок 3.20 – Результат роботи по ігнорування файлу

### Висновок

У ході практичної роботи навчилися працювати з віддаленим репозиторієм, клонувати його і відправляти зміни. Було розібрано використання файлу *.gitignore* в віддаленому репозиторії.

У звіті треба не забути привести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### ***Контрольні запитання***

1. Як створити копію віддаленого репозиторія у локальній директорії?
2. Як перевірити наявність змін у файлі?
3. Яким чином відбувається відправлення змін у віддалений репозиторий?
4. Як відправити зміни на конкретну гілку у віддаленому репозиторії?
5. Яким чином можна відокремити файли у локальному репозиторії, які не підлягають синхронізації із віддаленим репозиторієм?
6. Яку функцію у репозиторії відіграє файл *.gitignore*?

## ГРАФІЧНИЙ ІНТЕРФЕЙС GIT

**Мета:** познайомитися з одним з графічних клієнтів для роботи з *git* репозиторієм

### **Теоретичні відомості**

Рідне «середовище проживання» *Git* – це термінал. Але текстовий інтерфейс – не найкращий вибір для всіх завдань. Іноді графічне представлення більш зручне, а деякі користувачі відчують себе комфортніше, користуючись мишкою та кнопками меню, а не командним рядком.

Також варто розуміти, що різні інтерфейси служать різним цілям. Деякі *Git*-клієнти обмежуються лише тим функціоналом, який їх автор вважає найбільш затребуваним або ефективним. З огляду на це, жоден з представлених нижче інструментів не може бути «краще» інших: вони просто призначені для різних завдань. Також варто пам'ятати, що все, що можна зробити за допомогою графічного інтерфейсу, може бути виконано і з консолі. Командний рядок, як і раніше, – місце, де є найбільше можливостей щодо контролю над репозиторієм.

Після установки *Git*, стають доступними два графічних інструменти: ***gitk*** – це графічний переглядач історії (подібно до ***git log***). Це той інструмент, який буде використовуватися для пошуку подій і візуалізації історії;

***git-gui*** – це інструмент редагування окремих *commit*'ів.

Інструменти ***gitk*** і ***git-gui*** – це приклади інструментів, орієнтованих на завдання. Кожен з них налаштований для вирішення певної задачі (перегляд історії або створення комітів, відповідно) і не підтримує функції *Git*, непотрібні для цього завдання.

Існує величезна кількість інших графічних інструментів для роботи з *Git* (ви можете навіть написати свій!), починаючи від спеціалізованих, що виконують одну задачу, закінчуючи «комбайнами» покривають весь функціонал *Git*. На офіційному сайті *Git* підтримується в актуальному стані список найбільш популярних оболонок.



### ***Вимоги до виконання практичної роботи***

Для проходження даної практичної роботи необхідно встановити популярний крос-платформний клієнт – *SmartGit*.

### ***Порядок виконання практичної роботи***

1. Установка *SmartGit*.
2. Робота з репозиторієм.
3. Відміна змін коду.
4. Робота з гілками.

#### ***1. Установка SmartGit***

Для установки *SmartGit* необхідно завантажити з офіційного сайту *Git* версію для своєї операційної системи (рис. 4.1).

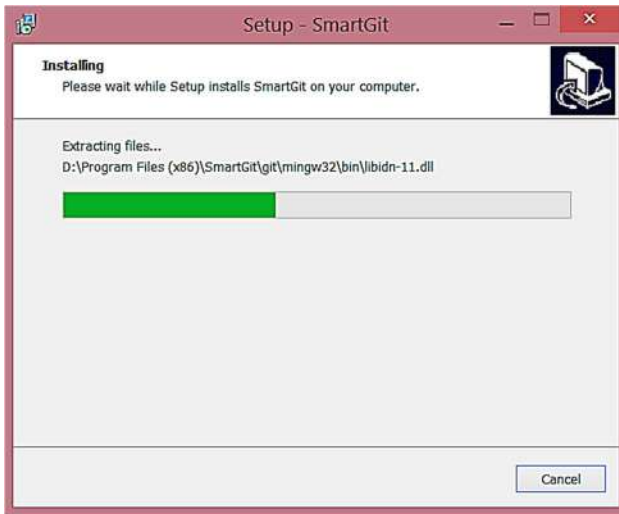


Рисунок 4.1 – Установка *SmartGit*

Далі необхідно запустити програму установки і вибрати пункт некомерційного використання (рис. 4.2).



Рисунок 4.2 – Вибір пункту некомерційного використання

Після проходження всіх етапів установки потрапляємо в робоче вікно *SmartGit* (рис. 4.3),

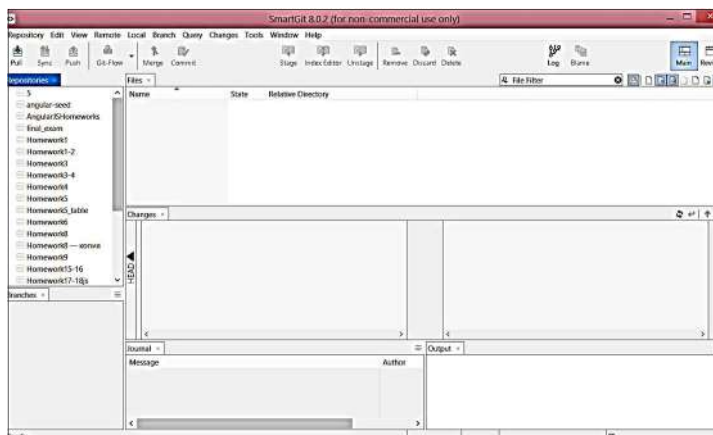


Рисунок 4.3 – Робоче вікно *SmartGit*

## 2. Робота з репозиторієм

Спочатку необхідно створити новий репозиторій на *GitHub* (див. попередню роботу) і скопіювати в буфер обміну посилання ***https*** (рис. 4.4).



Рисунок 4.4 – Створення нового репозиторію на *GitHub*

Для клонування сховища натиснути комбінацію кнопок **Ctrl+Shift+O** (рис. 4.5).

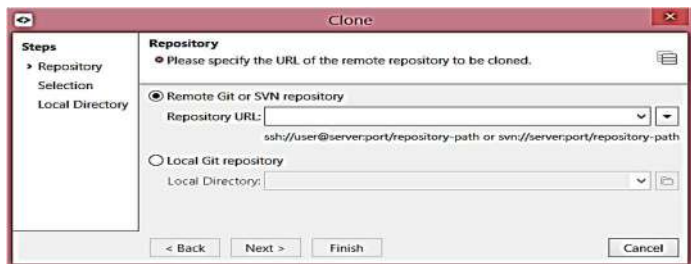


Рисунок 4.4 – Вікно клонування репозиторію

Далі необхідно вставити посилання, натиснути **Next** і вибрати директорію, в якій буде знаходитися робочий репозиторій (рис. 4.5).



Рисунок 4.5 – Вибір директорії для репозиторію

Для того, щоб робочий репозиторій з'явився в списку репозиторіїв необхідно натиснути **Finish** (рис. 4.6).

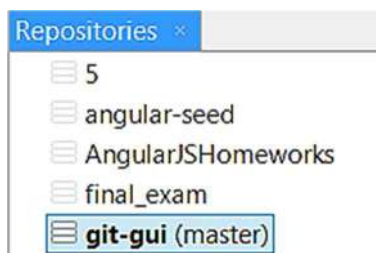


Рисунок 4.5 – Список репозиторіїв

Після створення нового файлу в папці репозиторію, з'являється новий файл (рис. 4.6).



Рисунок 4.6 – Створення нового файлу в репозиторії

*Untrack* – означає, що цього файлу ще немає в репозиторії. Якщо натиснути на файл *index.html*, то можна побачити зміст цього файлу (рис. 4.7).

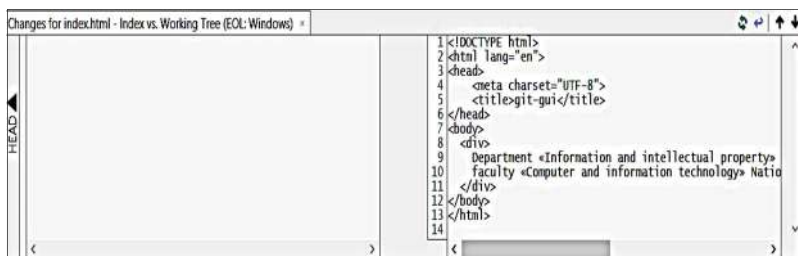


Рисунок 4.7 – Зміст файлу в *index.html*

Для збереження змін необхідно натиснути кнопку **Commit** (рис. 4.8).



Рисунок 4.8 – Збереження змін

Після натискання кнопки **Commit** з'являється новий *commit* (рис. 4.9).



Рисунок 4.9 – Створення нового *commit*'у

### 3. Відкат змін коду

Для прикладу потрібно створити новий текстовий файл (рис. 4.10).

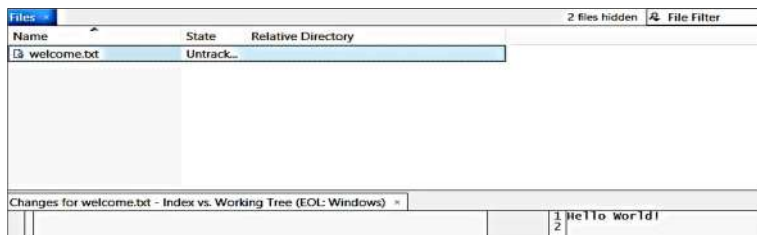


Рисунок 4.10 – Створення нового текстового файлу

Далі необхідно зробити коміт змін (рис. 4.11).



Рисунок 4.11 – *Commit* змін

При розробці бувають випадки, коли останній комміт або серія коммітів не потрібні, або були зроблені помилково. Тому можна зробити відкат змін коду.

Для цього необхідно натиснути правою кнопкою миші на той комміт який більше не потрібен і вибрати пункт **Revert**.

Далі потрібно натиснути **Revert and Commit** і подивитися в журнал коммітів. Видно, що створений файл пропав з папки.

Також, бувають ситуації, що треба видалити серію коммітів. Для цього необхідно виділити останній потрібний комміт і натиснути праву кнопку миші, і зробити **Reset**.

Приведемо наш репозиторій в первинний стан.

Вибираємо команду **initial commit** (рис. 4.12) і робимо **Reset** (рис. 4.13).

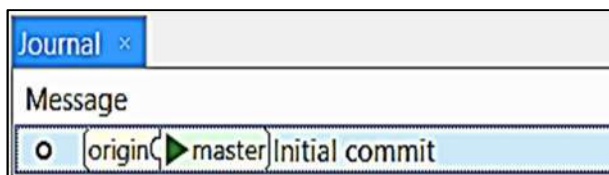


Рисунок 4.12 – Вибір команди *initial commit*

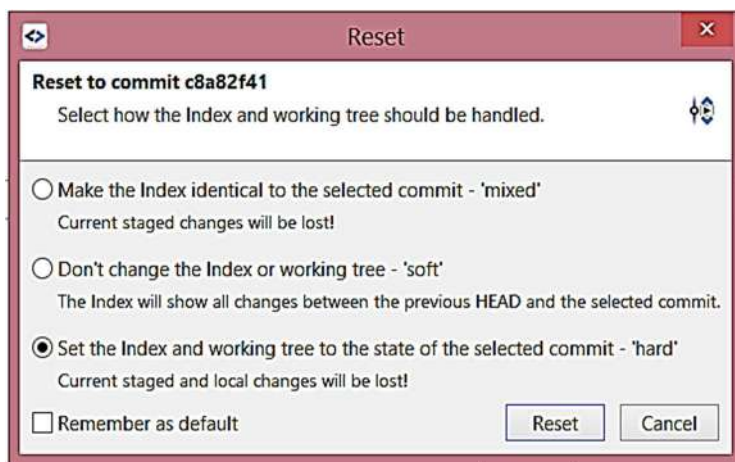


Рисунок 4.13 – Видалення серії коммітів

#### 4. Робота з гілками

Логіка роботи з гілками нічим не відрізняється від консольної версії. Зробимо кілька довільних *commit*'ів. (рис. 4.14).



Рисунок 4.14 – Створення довільних *commit*'ів.

Для того щоб створити нову гілку, необхідно натиснути правою кнопкою миші на поле **Local Branches** (рис. 4.15).

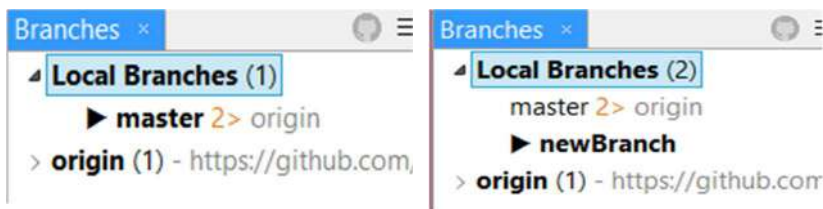


Рисунок 4.15 – Створення нової гілки

Далі на новій гілці потрібно створити новий файл і зробити *Commit* (рис. 4.16)

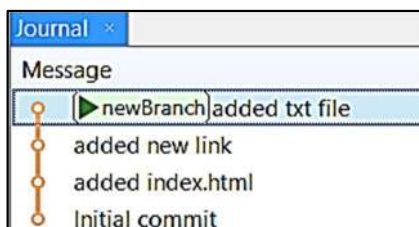


Рисунок 4.16 – Створення нового файлу і *commit*

Тепер необхідно перейти на основну гілку і оновити файл *index.html*. Після цього, структура нашого проекту виглядає наступним чином (рис. 4.17).

Commits (8)			
Message		Date	Author
◀ master 3>	update file	22:09	Sergei Kononov
newBranch	added txt file	22:06	Sergei Kononov
	added new link	21:59	Sergei Kononov
	added index.html	21:58	Sergei Kononov
	added file txt (reverted from commit f2b1c0d2f5f0ad15c9675eff...	21:49	Sergei Kononov
	added file txt	21:46	Sergei Kononov
	Added new file	21:40	Sergei Kononov
origin/master	Initial commit	20:11	Sergei Kononov

Рисунок 4.17 – Структура проекту після змін

Тепер необхідно зробити злиття гілок. Для цього натиснути на гілку **newBranch** правою кнопкою миші і виконати команду **Merge** (рис. 4.18).

Commits (6)	
Message	
▶ newBranch	Merge branch 'master' into newBranch
	added txt file
master 3>	update file
	added new link
	added index.html
origin/master	Initial commit

Рисунок 4.18 – Злиття гілок

Для того щоб видалити гілку, треба натиснути правою кнопкою миші на гілку і вибрати пункт **Delete**.

Для відправки всіх змін на *GitHub*, необхідно натиснути кнопку **Push** (рис. 4.19).





Рисунок 4.19 – Збереження змін.

### ***Висновок***

Перераховані інструменти відмінно вирішують поставлені завдання. З їх допомогою розробники (і не тільки) можуть розпочати спільну роботу над проектами в лічені хвилини, причому з налаштованим робочим процесом. Але якщо дотримуватися інших підходів до використання *Git*, або якщо потрібно більше контролю над тим, що відбувається, рекомендується працювати з командним рядком.

У звіті не забудьте привести посилання на репозиторій, який був використаний при виконанні даної практичної роботи.

### ***Контрольні запитання***

1. Навіщо потрібні *Git*-клієнти?
2. Яку комбінацію кнопок використовують для клонування сховища?
3. Як клонувати віддалений репозиторій на свій комп'ютер?
4. Як створити новий файл в папці репозиторія?
5. Яку кнопку необхідно натиснути для збереження змін у репозиторії?
6. Яким чином можна виконати відкат змін у файлі?
7. Як створити нову гілку у репозиторії?
8. Як виконати злиття гілок?

## ГРАФІЧНИЙ КЛІЄНТ ДЛЯ ВИКОРИСТАННЯ GIT – TORTOISEGIT

**Мета:** познайомитися з графічним клієнтом *TortoiseGit* і його основним функціоналом

### *Теоретичні відомості*

*TortoiseGit* – візуальний клієнт системи управління вихідними кодами програм *git* для ОС Microsoft Windows (рис. 5.1). Розповсюджується по ліцензії GNU GPL. Написана на C++ та C, перший реліз (випуск) у 2008 році.



Рисунок 5.1 – Офіціальний сайт *TortoiseGit*

Однією з найбільш важливих особливостей *TortoiseGit* є наявність значків, які з'являються на файлах в робочій директорії. Вони показують, які файли були змінені (рис. 5.2).



Рисунок 5.2 – Іконки файлів у репозиторії,  
зміни до яких внесені в *TortoiseGit*

Всі команди *TortoiseGit* викликаються з контекстного меню *Windows Explorer*. Більшість з них видно після натискання правою кнопкою миші на файлі або папці. Різні команди доступні в залежності від того, чи є файл або папка під управлінням системи *TortoiseGit* чи ні (рис. 5.3).

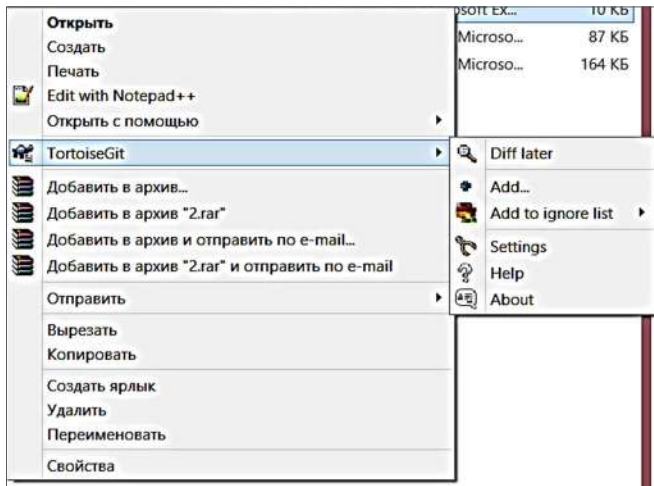


Рисунок 5.3 – Меню *TortoiseGit*

### ***Вимоги до виконання практичної роботи***

Для виконання практичної роботи необхідно завантажити файл інсталяції з офіційного сайту (див. рис. 5.1).

### ***Порядок виконання практичної роботи***

1. Створення нового репозиторію.
2. Виконання основних команд.
3. Створення гілок і їх злиття (*merge*).

#### ***1. Створення нового репозиторію***

Створюємо новий репозиторій на *GitHub* (див. попередні роботи).

Далі необхідно клонувати створений репозиторій. Для цього натиснути правою кнопкою миші та вибрати пункт ***Git Clone***. Вставити адресу *https* в рядок ***URL*** (рис. 5.3).

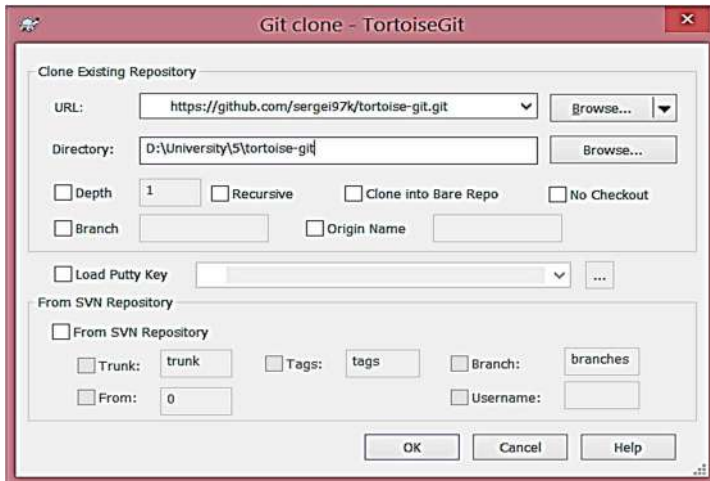


Рисунок 5.3 – Створення нового репозиторія

Після цього потрапляємо в меню завантаження (рис. 5.4).

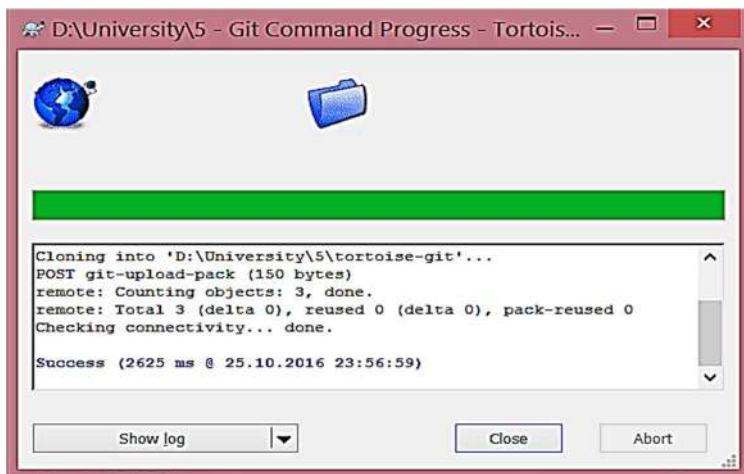


Рисунок 5.4 – Меню завантаження

Тепер видно, що в поточній директорії створилася нова папка з репозиторієм.

## 2. Виконання основних команд

Команда *commit* – важлива складова будь-якої системи контролю версій. Щоб додати файл до головного сховища необхідно виконати такі дії:

– створити в цій папці файл з будь-яким вмістом (рис. 5.5);

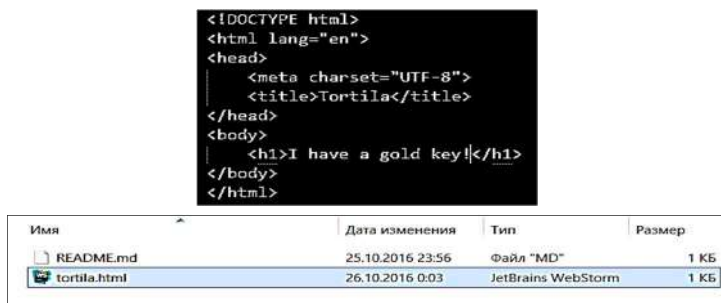


Рисунок 5.5 – Створення довільного файлу

– натиснути правою кнопкою миші і вибрати пункт **Git Commit** → **master**. Заповнити всі обов’язкові поля і натиснути кнопку **Commit** (рис. 5.6);

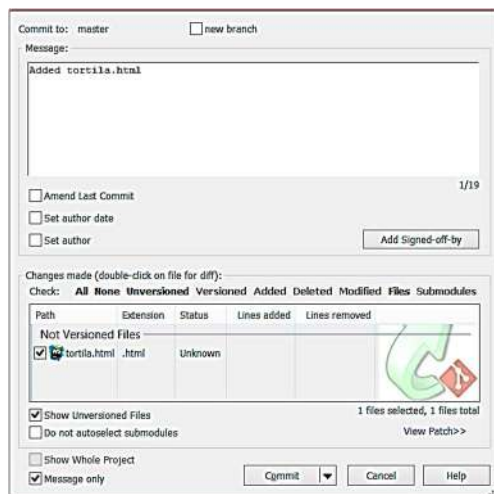


Рисунок 5.6 – Заповнення обов’язкових атрибутів

– далі бачимо вікно завантаження, що означає, що зміни збережені (рис. 5.7);

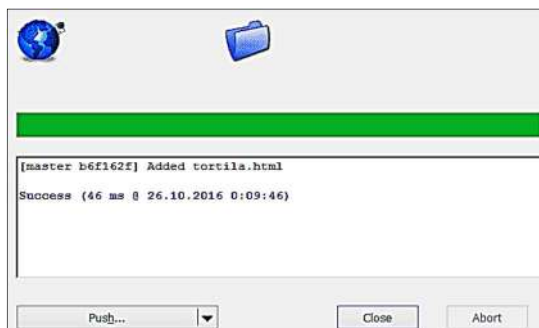


Рисунок 5.7 – Вікно завантаження

– додати зміни у віддалений репозиторій. Для цього необхідно виконати контекстну команду **push** (рис. 5.8);

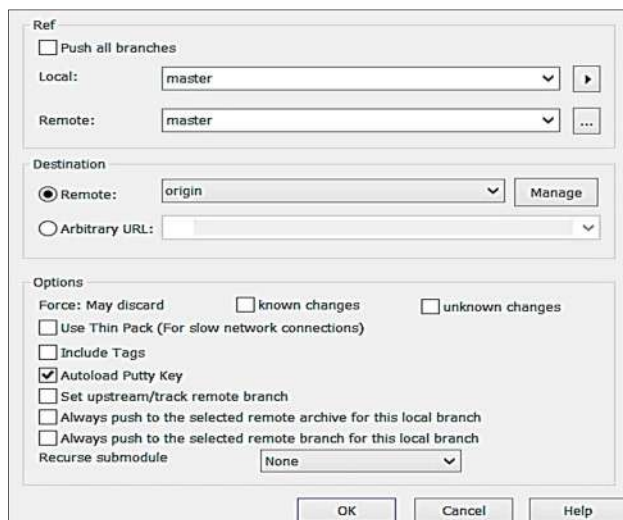


Рисунок 5.8 – Додавання змін у віддалений репозиторій

– заповнити поля і натиснути кнопку ОК (рис. 5.9 – 5.10);

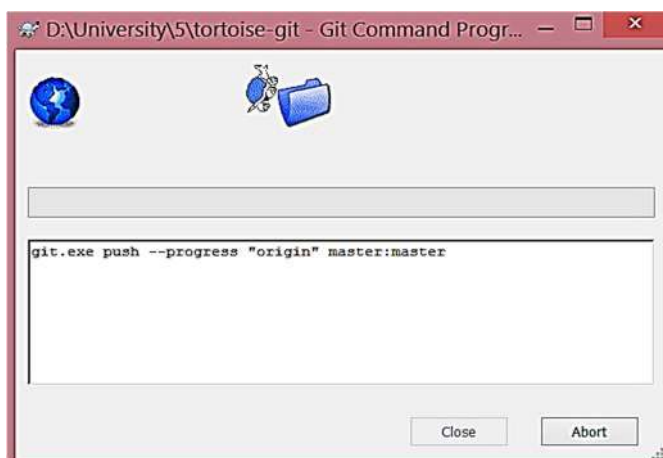


Рисунок 5.9 – Виконання команди *push*

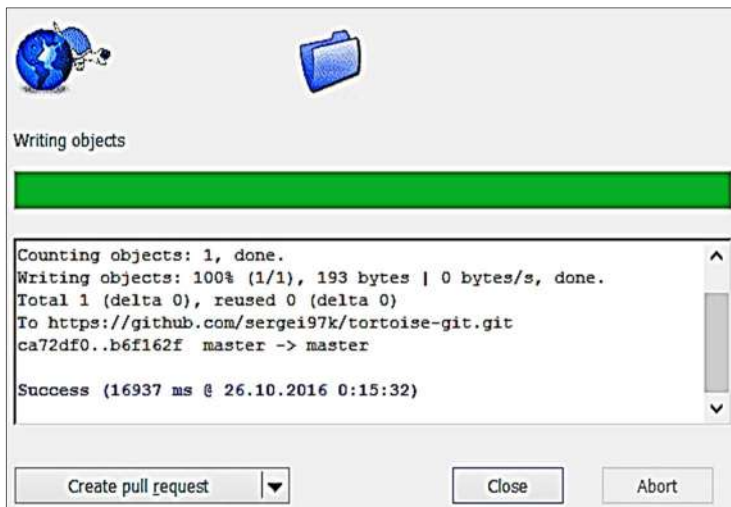


Рисунок 5.10 – Завершення роботи

– для перевірки наявності змін в віддаленому репозиторії зайти на репозиторій *GitHub* (рис. 5.11).

README.md	Initial commit	26 minutes ago
tortila.html	Added tortila.html	38 seconds ago

Рисунок 5.11 – Перевірка змін в репозиторії

### 3. Створення гілок і їх злиття (*merge*)

По суті даний клієнт має весь основний функціонал стандартного *git* клієнта. Відповідно робота з гілками відмінно реалізована в даному інтерфейсі.

Для створення гілок необхідно виконати такі дії:

– у випадяючому списку вибрати команду **Create Branch**, щоб створити нову гілку (рис. 5.12);



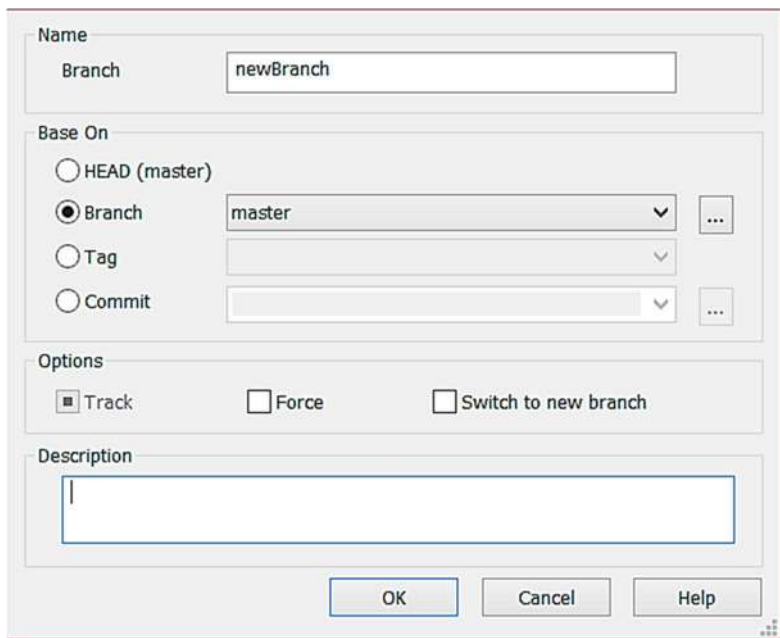


Рисунок 5.12 – Створення нової гілки

– додати новий файл або змінити вже існуючий, для того, щоб гілки мали відмінності (рис. 5.13);




 describe-tortila.txt	26.10.2016 0:27	Текстовый докум...	0 КБ
 README.md	25.10.2016 23:56	Файл "MD"	1 КБ
 tortila.html	26.10.2016 0:03	JetBrains WebStorm	1 КБ

Рисунок 5.13 – Додавання нового файлу

– зберегти зміни в даній гілці (рис. 5.14);

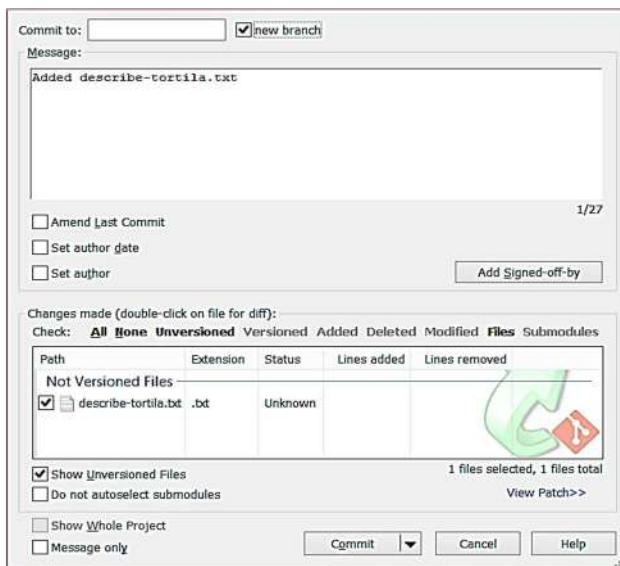


Рисунок 5.14 – Збереження змін в гілці

– зробити злиття (*merge*) гілок (рис. 5.15);

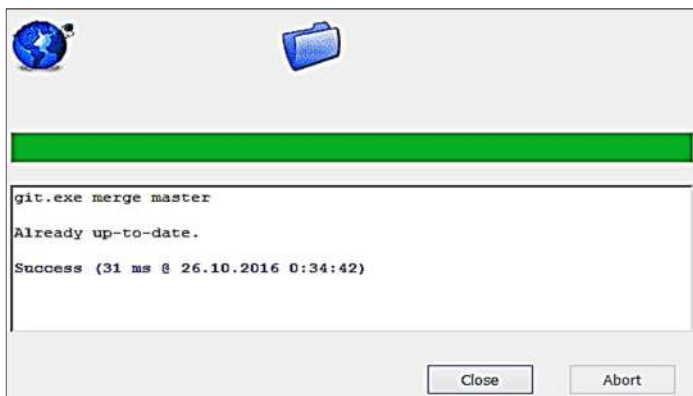


Рисунок 5.15 – Злиття гілок

– відправити зміни в віддалений репозиторій (рис.5.16);

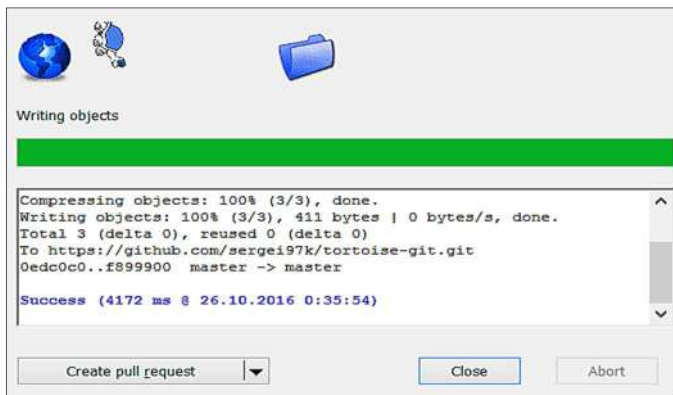


Рисунок 5.16 – Відправка змін в віддалений репозиторій

– зайти на *GitHub* і переконатися, що зміни збереглися і є в віддаленому репозиторії.

### **Висновок**

У ході даної практичної роботи було розглянуто один із зручних інструментів для роботи з *Git* репозиторіями на *OC Windows* і базові команди. Але не варто забувати, що при повсякденній роботі функціонал репозиторія може значно змінитися або розширитися. Тому рекомендовано ознайомитися з документацією *TortoiseGit*.

У звіті треба не забути привести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### **Контрольні запитання**

1. Навіщо потрібен клієнт *TortoiseGit*?
2. Яким чином команди викликаються *TortoiseGit*?
3. Як клонувати віддалений репозиторій на свій комп'ютер?
4. Як зберегти зміни у файлі в головній гілці репозиторія?
5. Як додати зміни у віддалений репозиторій?
6. Як створити нову гілку у репозиторії?
7. Як виконати злиття гілок?
8. Як переконатися, що зміни збереглися і є в віддаленому репозиторії?

## РОБОТА З GITHUB ТА ІНТЕГРОВАНІМ СЕРЕДОВИЩЕМ РОЗРОБКИ ДОДАТКІВ INTELIJ IDEA

**Мета:** забезпечення практичних навичок по використанню *GitHub* і *IntelliJ Idea*, спрощення використання ресурсу *GitHub* для проектів, створених за допомогою *IntelliJ Idea*

### **Теоретичні відомості**

*IntelliJ IDEA* – інтегроване середовище розробки програмного забезпечення для багатьох мов програмування, зокрема *Java*, *JavaScript*, *Python*, розроблена компанією *JetBrains*. Починаючи з версії 9.0, середовище доступна в двох редакціях: *Community Edition* і *Ultimate Edition*. *Community Edition* є повністю вільною версією, доступною під ліцензією *Apache 2.0*, в ній реалізована повна підтримка *Java SE*, *Kotlin*, *Groovy*, *Scala*, а також інтеграція з найбільш популярними системами управління версіями. У даній роботі ми будемо працювати саме з такою версією.

В редакції *Ultimate Edition*, доступною під комерційною ліцензією, реалізована підтримка *Java EE*, *UML*-діаграм, підрахунок покриття коду, а також підтримка інших систем управління версіями, мов та фреймворків.

### **Вимоги до виконання практичної роботи**

Для виконання практичної роботи знадобиться аккаунт на *GitHub*. Якщо аккаунта немає, потрібно його створити, використовуючи інструкцію до практичної роботи «Початок роботи на *GitHub*» (див. попередню роботу). Далі встановити *IntelliJ Idea* будь-якої версії, скачати та встановити інструмент *JDK*, а також встановити *Git* на своєму комп'ютері.

### **Порядок виконання практичної роботи**

1. Налаштування *Git* для командного рядка.
2. Підключення *Git* для поточного проекту.
3. Створення репозиторію.
4. Відправлення проекту на локальний репозиторій.
5. Підготовка проекту для відправки у віддалений репозиторій.
6. Створення першого коміту.
7. Створення наступних комітів.
8. Зворотня синхронізація.

## 1. Налаштування Git для командного рядка

Для налаштування *Git* для командного рядка необхідно виконати такі дії:

– зайти в папку з файлами *Git* (за замовчуванням попередній шлях: *C:\Program Files\Git*. Зайти в папку **bin** і скопіювати повний її шлях (рис. 6.1);

– зайти в пункт меню *Властивості системи* → *Додаткові параметри системи* → *Параметри середовища*. Якщо змінної **Path** немає, то необхідно створити її і скопіювати шлях до потрібної папки. Якщо змінна **Path** є, то до існуючого дописати через знак «;» шлях до створеної папки (рис. 6.2);

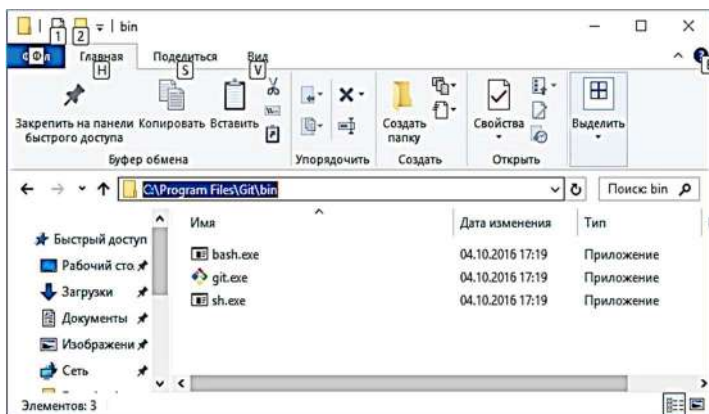


Рисунок 6.1 – Папка з файлами *Git*

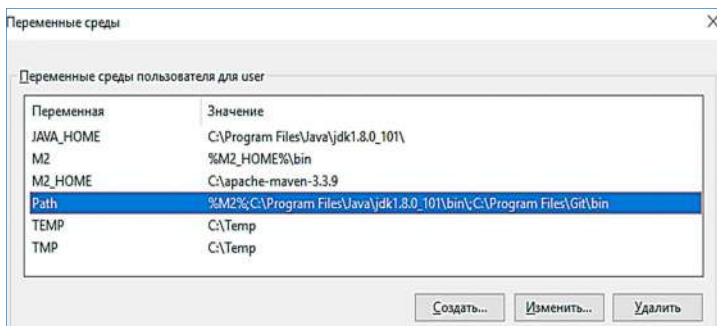
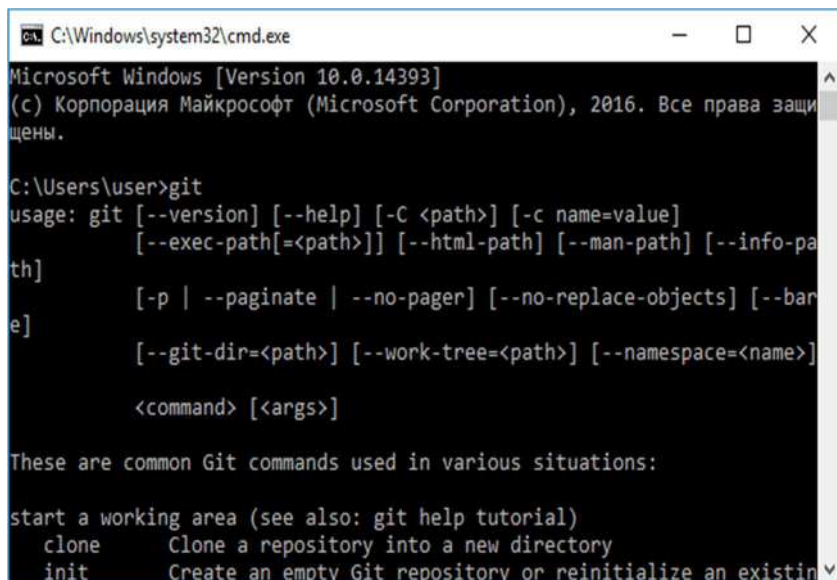


Рисунок 6.2 – Змінна середовища *Path*

– для перевірки правильності виконаних дій необхідно відкрити командний рядок і виконати команду *git* (рис. 6.3);



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) Корпорация Майкрософт (Microsoft Corporation), 2016. Все права защищены.

C:\Users\user>git
usage: git [--version] [--help] [-C <path>] [-c name=value]
         [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
         [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
         [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
         <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one
```

Рисунок 6.3 – Результат виконання команди *git*

– зробити схожі налаштування для середовища *IntelliJ Idea* (якщо це не відбулося автоматично). Для перевірки в командному рядку *IntelliJ Idea* необхідно викликати команду *git*. Для цього потрібно натиснути комбінацію клавіш **Ctrl + Alt + S**, щоб викликати вікно налаштувань. У пункті *Version Control* → *Git* вибрати файл *git.exe* в по шляху, який прописаний в *Path* (рис. 6.4).

## 2. Підключення *Git* для поточного проекту

Для підключення *Git* для поточного проекту необхідно вибрати пункт меню *VCS* → *Enable Version Control Integration* → *Git* (рис. 6.5).

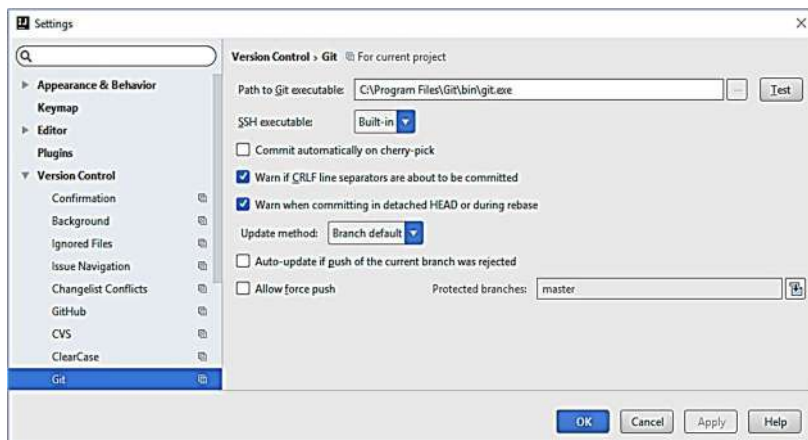


Рисунок 6.4 – Вікно налаштувань

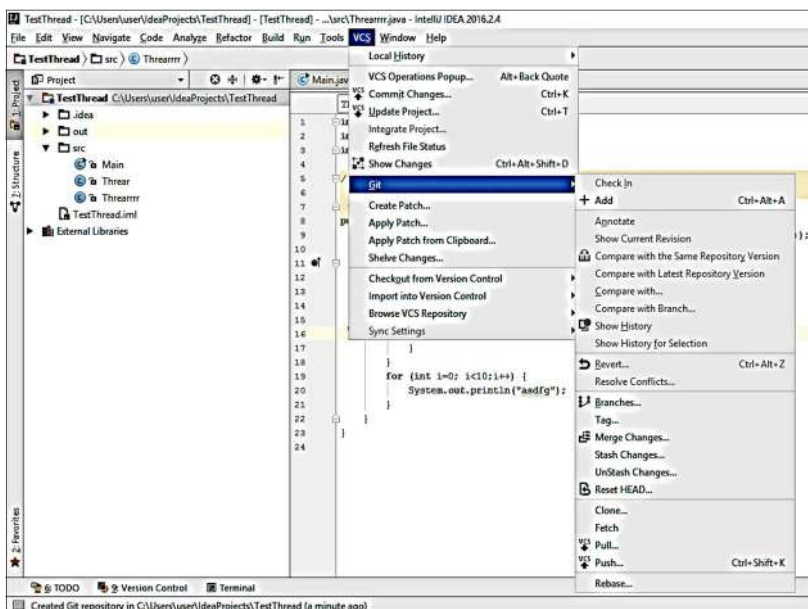


Рисунок 6.5 – Підключення *Git* для поточного проекту

### 3. Створення репозиторію

Необхідно зайти на *GitHub* і створити репозиторій з потрібним ім'ям (рис. 6.6).

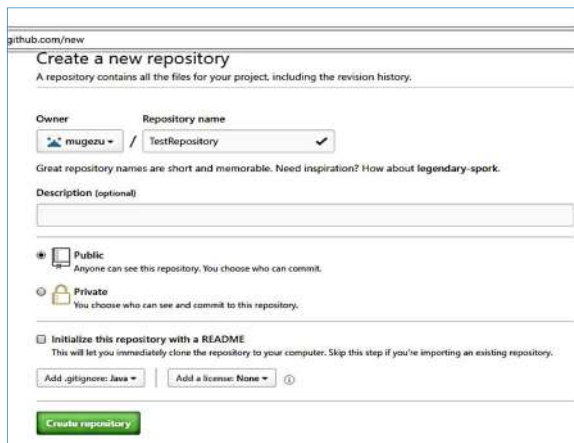


Рисунок 6.6 – Створення нового репозиторія

### 4. Відправка проекту на локальний репозиторій

Для відправки проекту на локальний репозиторій необхідно виконати такі дії:

– вибрати *Version Control* (в закладці *Changes*, в нижньому лівому кутку *IntelliJ IDEA* або за допомогою комбінації клавіш **Alt + 9**), натиснути правою кнопкою миші на файлах *Unversioned Files*. Далі в контекстному меню вибрати *Add to VCS* (рис.6.7);

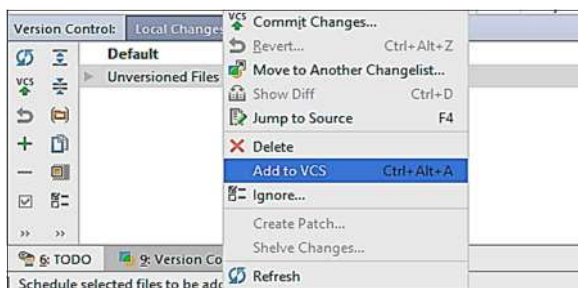


Рисунок 6.7 – Вибір команди *Add to VCS*



– натиснути правою кнопкою миші на пункті *Default* і в контекстному меню вибрати *Commit Changes* (рис. 6.8);

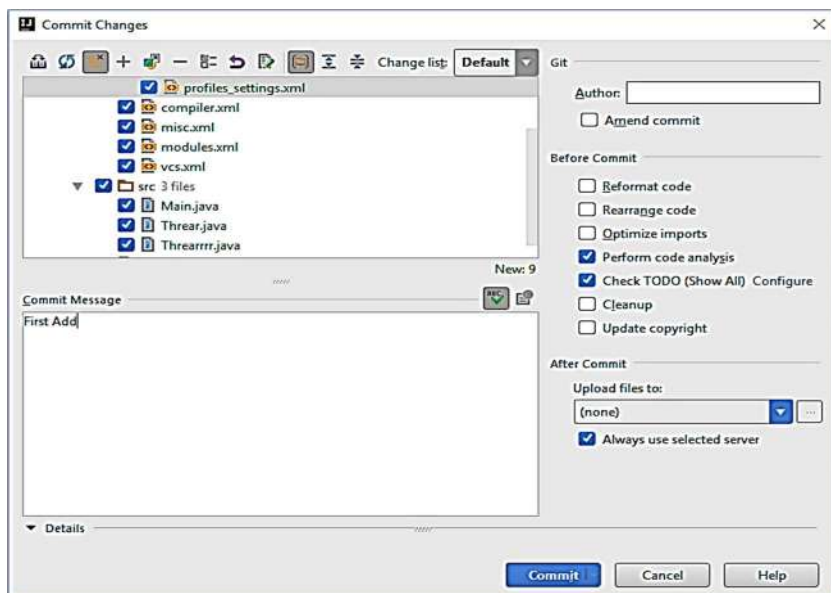


Рисунок 6.8 – Вибір команди *Commit Changes*

– у вікні *Commit Message* написати коментар про зміни в порівнянні з попередньою версією. Якщо висвічується вікно попередження про помилки (рис. 6.9), необхідно проаналізувати помилки і виправити їх, якщо потрібно. Ці попередження можна проігнорувати.

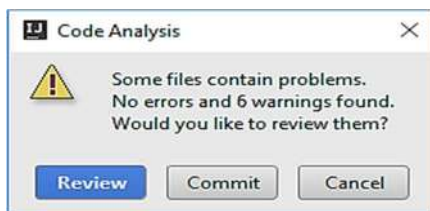


Рисунок 6.9 – Вікно попередження про помилки

### 5. Підготовка проекту для відправки у віддалений репозиторій

Для відправки у віддалений репозиторій необхідно зайти в репозиторій на *GitHub* і скопіювати посилання з репозиторієм, створеним в п. 3 (рис. 6.10).

Посилання має бути виду: ***https://github.com/username/reponame***,  
де *username* – логін на *GitHub*;  
*reponame* – назва репозиторію.

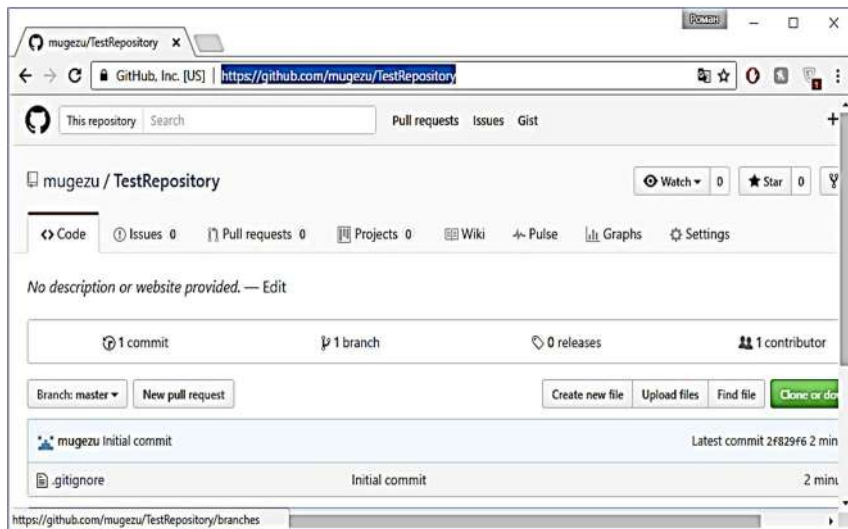


Рисунок 6.10 – Відправка проекту у віддалений репозиторій

### 6. Створення першого комміта

Перший комміт трохи складніше, ніж інші. Для цього в *IntelliJ Idea* у командному рядку (внизу під назвою *Terminal*) треба написати такі команди (рис. 6.11):

```
git remote add origin https://github.com/username/reponame  
git pull origin master --allow-unrelated-histories  
git push origin master
```

```
Terminal
+ C:\Users\user\IdeaProjects\TestThread>git remote add origin https://github.com/mugezu/TestRepository

C:\Users\user\IdeaProjects\TestThread>git pull origin master --allow-unrelated-histories
warning: no common commits
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/mugezu/TestRepository
 * branch          master       -> FETCH_HEAD
 * [new branch]     master       -> origin/master
Merge made by the 'recursive' strategy.
.gitignore | 12 ++++++++
1 file changed, 12 insertions(+)
create mode 100644 .gitignore

C:\Users\user\IdeaProjects\TestThread>git push origin master
Counting objects: 16, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (16/16), 2.68 KiB | 0 bytes/s, done.
Total 16 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/mugezu/TestRepository
2f829f6..6308d1e  master -> master
```

Рисунок 6.11 – Результат виконання команд

Далі повинна з'явитися форма з паролем і логіном на *GitHub*. Заповнити її своїми даними і перевірити *GitHub* (рис. 6.12).

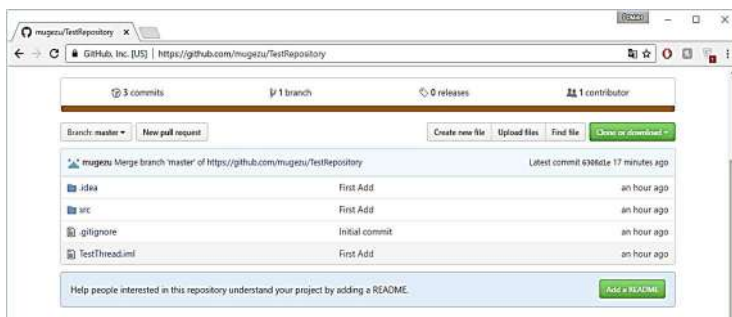


Рисунок 6.12 – Перевірка *GitHub*

## 7. Створення наступних *commit*'ів

Після змін проекту просто робимо *Commit* заново, але необхідно вибрати кнопку *Commit and Push* (рис. 6.13) і натиснути *Push* (рис. 6.14).

Далі необхідно здійснити перевірки на *GitHub* (рис. 6.15) і в *IntelliJ Idea* (рис. 6.16).

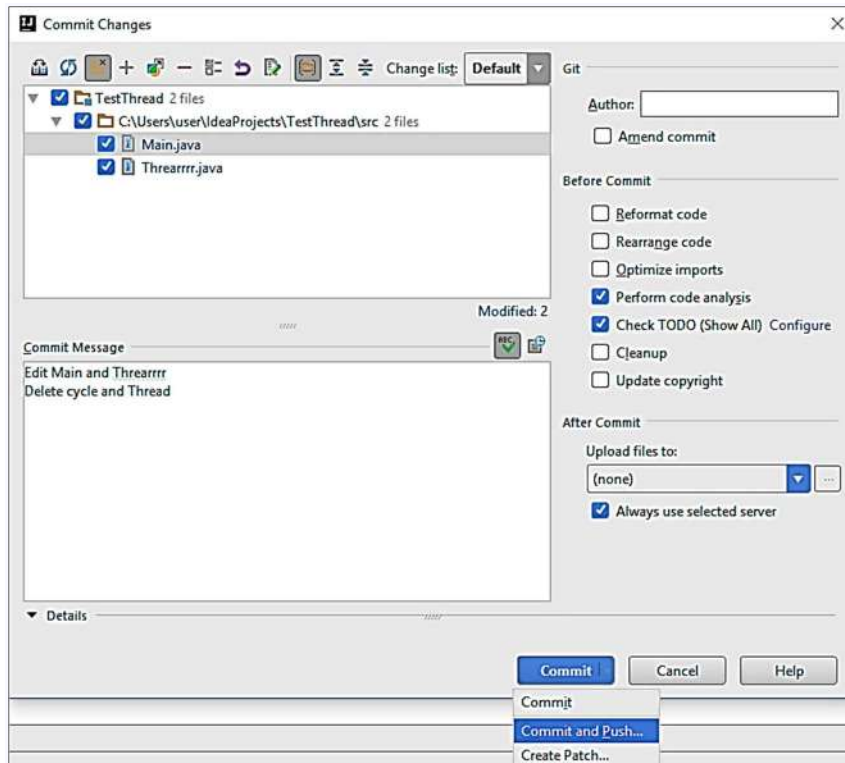


Рисунок 6.13 – Вибір кнопки *Commit and Push*

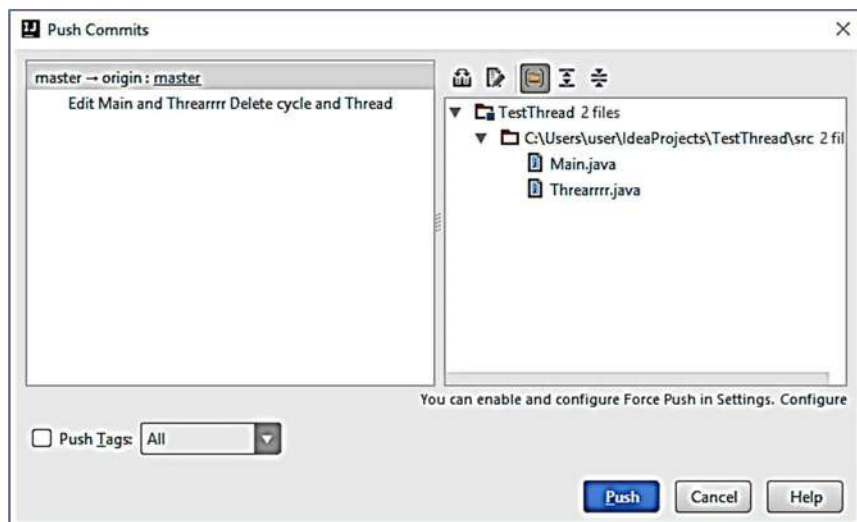


Рисунок 6.14 – Вибір кнопки *Push*

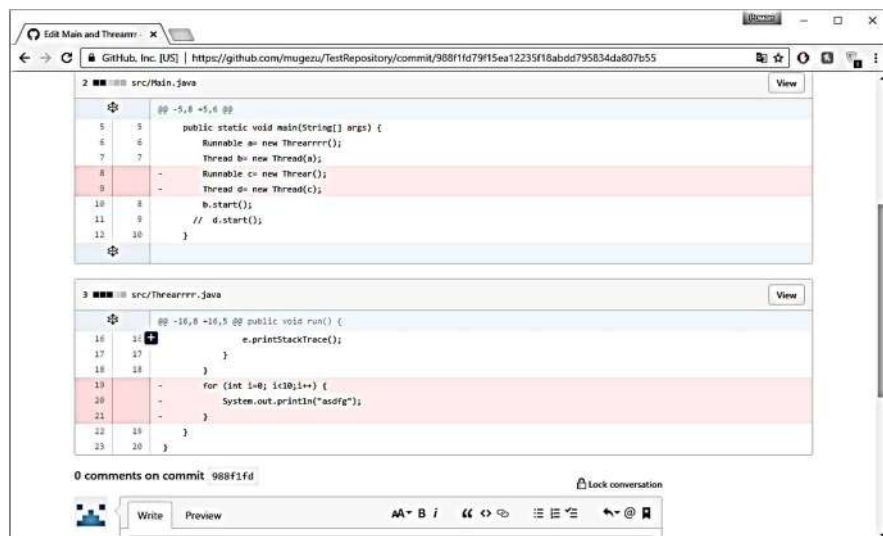


Рисунок 6.15 – Перевірка в *GitHub*

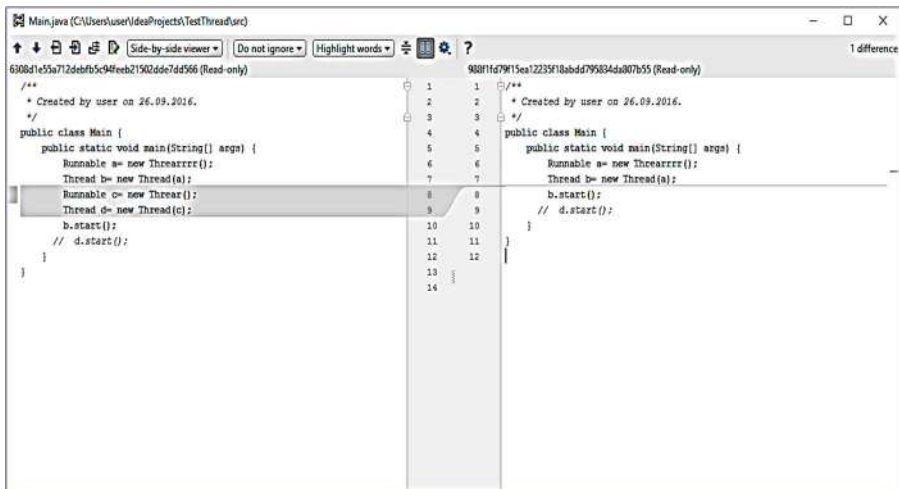


Рисунок 6.16 – Перевірка в *IntelliJ Idea*

## 8. Зворотня синхронізація

Для отримання останньої актуальної версії, яка знаходиться на *GitHub* достатньо (рис. 6.17) ввести в командний рядок *IntelliJ Idea* команду (тільки для першого з'єднання):

```
git branch --set-upstream master origin/master
```

```
C:\Users\user\IdeaProjects\TestThread>git branch --set-upstream master origin/master
The --set-upstream flag is deprecated and will be removed. Consider using --track or --set-upstream-to
Branch master set up to track remote branch master from origin.
```

Рисунок 6.17 – Результат виконання команди

Далі, коли потік налаштований, потрібно синхронізуватися. У *IntelliJ Idea* в панелі управління вибираємо *VCS* → *Update Project*.

## Висновок

Були забезпечені практичні навички з використання *GitHub* і *IntelliJ Idea*, спрощення використання ресурсу *GitHub* для проектів, створених за допомогою *IntelliJ Idea*.

У звіті треба не забути привести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### ***Контрольні запитання***

1. Яку інформацію слід додати до змінної ***Path***, щоб налаштувати запуск *Git* з командного рядка?
2. Які налаштування слід провести у середовищі *IntelliJ Idea* для коректної роботи з *Git*?
3. Як підключити *Git* для поточного проекту?
4. Як створити репозиторій на *GitHub*?
5. Яку інформацію слід заносити у вікні *Commit Message*?
6. Чим відрізняється створення першого комміта від наступних?
7. Як здійснити зворотню синхронізацію із віддаленим репозиторієм?

## РОБОТА С СИСТЕМОЮ КОНТРОЛЮ ВЕРСІЙ GIT В СЕРЕДОВИЩІ РОЗРОБКИ ECLIPSE

**Мета.** Забезпечення практичних навичок по використанню *GitHub* і *Eclipse*, спрощення використання ресурсу *GitHub* для проєктів, створених за допомогою *Eclipse*

### **Теоретичні відомості**

*Eclipse* – інтегроване середовище розробки модульних кроссплатформних додатків, що вільно розповсюджується. Розвивається і підтримується *Eclipse Foundation*.

*Eclipse JDT (Java Development Tools)* – найбільш відомий модуль, націлений на групову розробку: середовище інтегроване з системами керування версіями - *CVS*, *GIT* в основний постачання, для інших систем (наприклад, *Subversion*, *MS SourceSafe*) існують плагіни. Також пропонує підтримку зв'язку між *IDE* і системою управління завданнями (помилками). В основний постачання включена підтримка трекера помилок *Bugzilla*, також є безліч розширень для підтримки інших трекерів (*Trac*, *Jira* та ін.). В силу безкоштовності і високої якості, *Eclipse* в багатьох організаціях є корпоративним стандартом для розробки додатків.

### **Вимоги до виконання практичної роботи**

Для виконання роботи необхідно переконатися, що встановлений *Git* (керівництво по установці дивись у першій роботі). Також необхідно мати базові теоретичні знання про те, що таке система контролю версій і розуміння її роботи. В тому числі, необхідно мати навички роботи з середовищем розробки *Eclipse*. Всі дії, описані нижче, актуальні для версії *Eclipse Luna*.

### **Порядок виконання практичної роботи**

1. Створення *Git*-репозиторію.
2. Відправка проєкту на локальний репозиторій.
3. Створення нової гілки.
4. Видалення, додавання, злиття (*merge*) гілок, зміна файлів всередині гілок.



## 1. Створення Git-репозиторію

Для створення сховища необхідно виконати такі дії:

– відкрити середовище *Eclipse* з встановленим *EGit*-плагіном і в перспективі *Java* в меню **File** вибрати команду **New** → **Other** → **Java** → **Java Project**, натиснути кнопку **Next**, ввести ім'я проекту **HelloGit** і натиснути кнопку **Finish** (рис. 7.1);

– у вікні *Package Explorer* натиснути правою кнопкою мишки на вузлі проекту і в контекстному меню вибрати команду **New** → **Other** → **Java** → **Class**, натиснути кнопку **Next**, ввести ім'я пакету **main**, ім'я класу **Main**, відзначити прапорець **public static void main (String[] args)** і натиснути кнопку **Finish** (рис. 7.2);

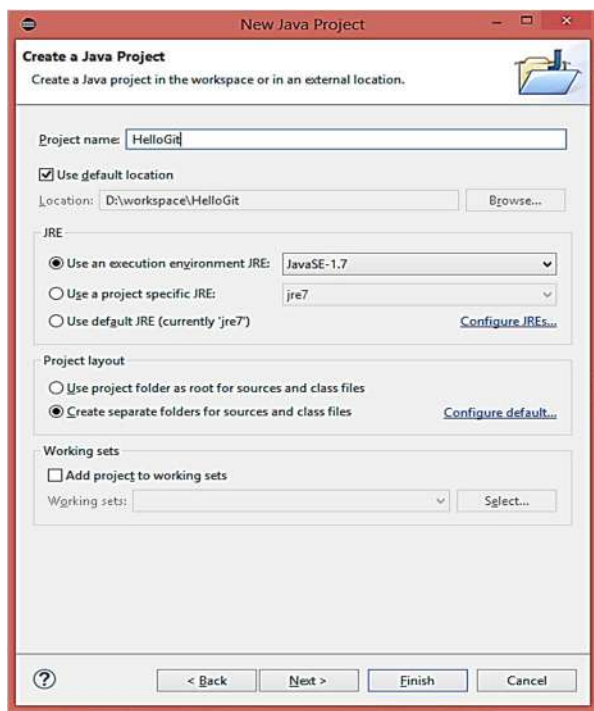


Рисунок 7.1 – Створення нового *Java* проекту

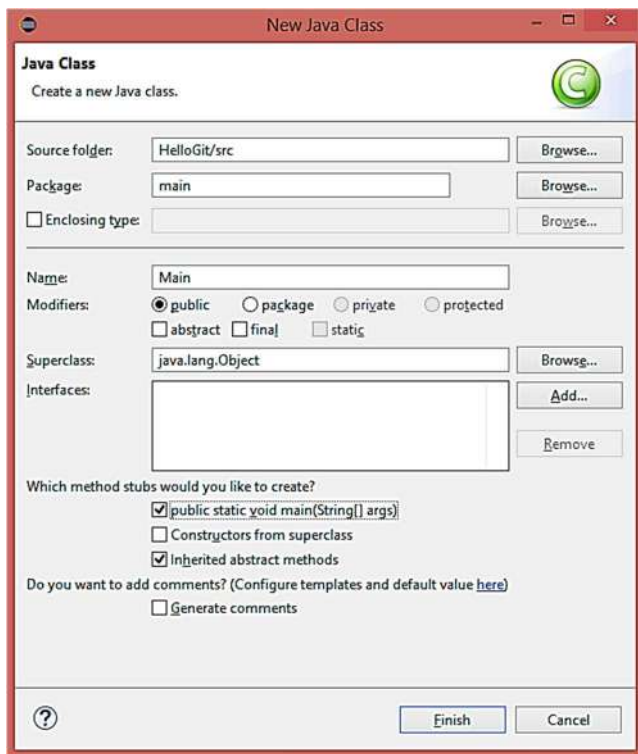


Рисунок 7.2 – Створення нового *Java* класу

– у вікні *Package Explorer* натиснути правою кнопкою мишки на вузлі проекту і в контекстному меню вибрати команду **Team** → **Share Project**. У вікні майстра *Share Project* вибрати *Git* і натиснути кнопку **Next** – з’явиться вікно *Configure Git Repository*, в якому пропонується створити *Git*-репозиторій в папці каталогу проекту (що не рекомендується) або створити окремий *Git*-репозиторій. В списку *Repository*: натиснути кнопку **Create**, ввести ім’я репозиторію **hellogit** і натиснути кнопку **Finish** (рис. 7.3);

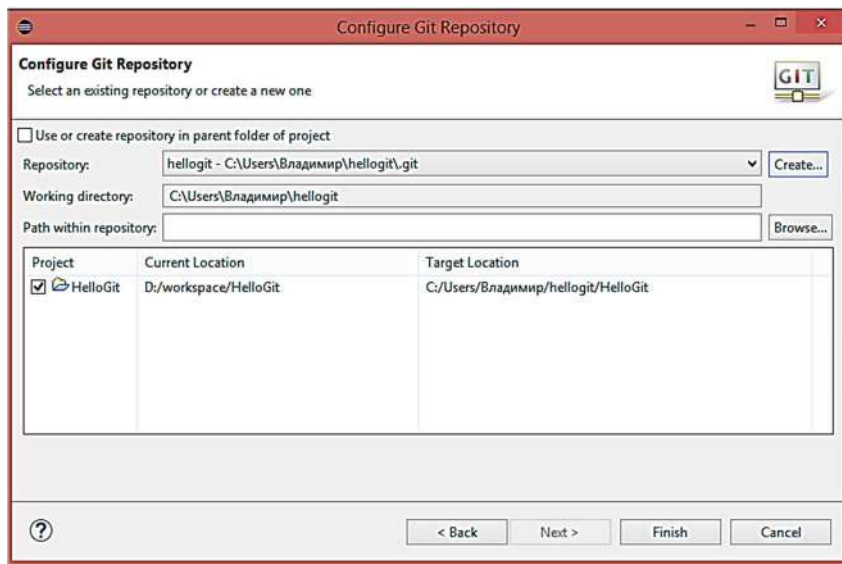


Рисунок 7.3 – Конфігурація *Git*-репозиторію

– в результаті проект ***HelloGit*** буде переміщений з каталогу *workspace* в каталог ***git/hellogit***, в якому також буде створена папка ***git*** для зберігання історії і метаданих проекту в системі *Git*.

## 2. Відправка проекту на локальний репозиторій

Для відправки проекту на локальний репозиторій необхідно виконати такі дії:

– за допомогою команди ***Show View*** → ***Other*** → ***Git*** меню ***Window*** відкрити перспективу (тобто представлення) *Git Staging*, в якому видно, що ресурси проекту не додано в область підготовлених файлів (рис. 7.4);

– у вікні *Package Explorer* натиснути правою кнопкою мишки на вузлі проекту і в контекстному меню вибрати команду ***Team*** → ***Add to Index***. При необхідності оновити представлення *Git Staging* кнопкою ***Refresh*** панелі інструментів і переконатися, що файли проекту перемістилися в *stage*-область (рис. 7.5);

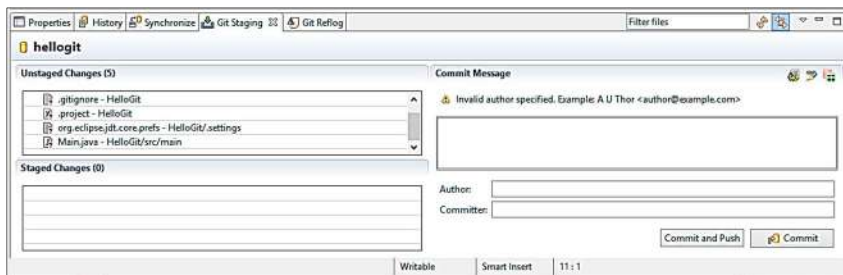


Рисунок 7.4 – Перспектива *Git Staging*

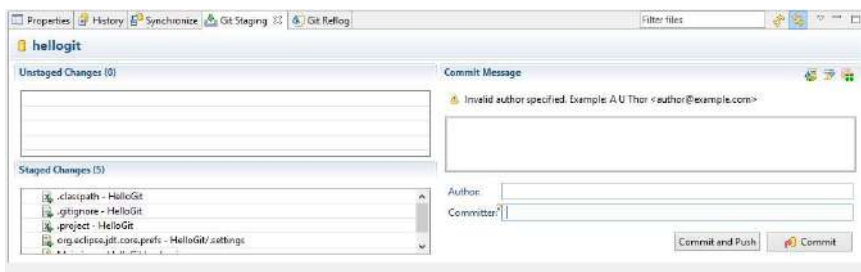


Рисунок 7.5 – Додавання файлів проекту в *stage*-область

– для створення першої фіксації в полі **Commit message** ввести коментар фіксації і натиснути кнопку **Commit** представлення *Git Staging*. В результаті вікно *Git Staging* очиститься, тому що файли з *stage*-області перемістяться в область фіксацій (рис. 7.6). Тепер локальний репозиторій містить потрібні файли.

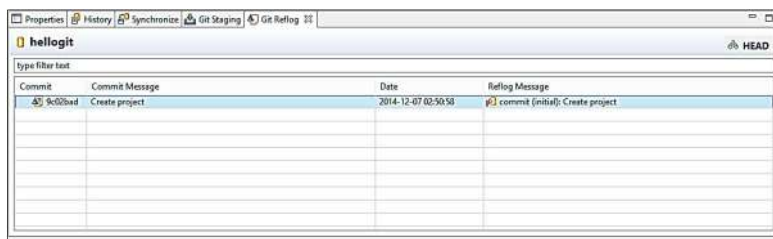


Рисунок 7.6 – Переміщення файлів в область фіксацій

### 3. Створення нової гілки

Далі необхідно створити нову гілку. У перспективі *Java* клікнути по вузлу проекту і вибрати команду **Team/Switch to**. Обов'язково поставити галочку в полі **Checkout new branch** (рис. 7.7). Це автоматично зробить активної щойно створену гілку.

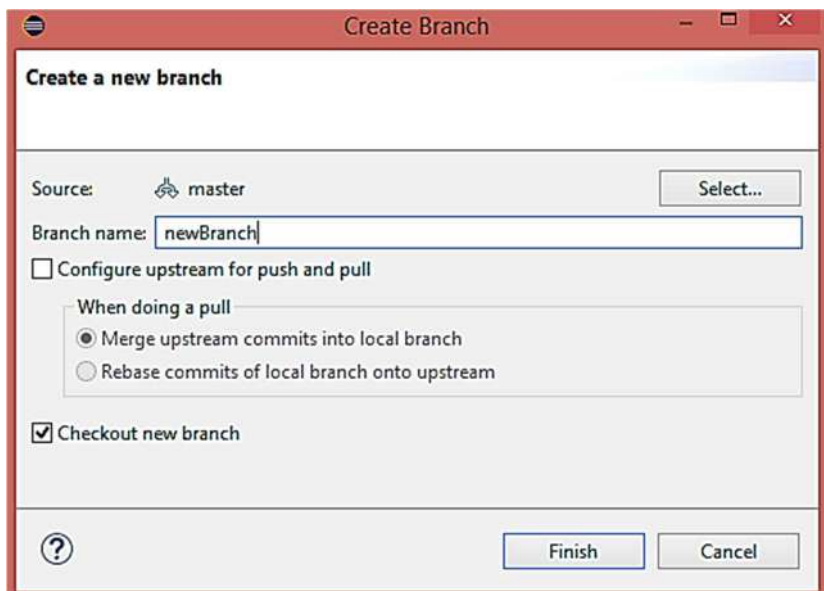


Рисунок 7.7 – Створення нової гілки

### 4. Видалення, додавання, злиття (merge) гілок, зміна файлів всередині гілок

Всі ці функції системи контролю версій вже відомі з попередніх практичних робіт. При роботі з середовищем розробки *Eclipse* принцип роботи такий же.

Для виконання цих функцій необхідно виконати такі дії:

- додати рядок коду в новій гілці в клас *Main* (рис. 7.8);
- після збереження в *Git Staging* помітні зміни в робочому файлі (рис. 7.9);

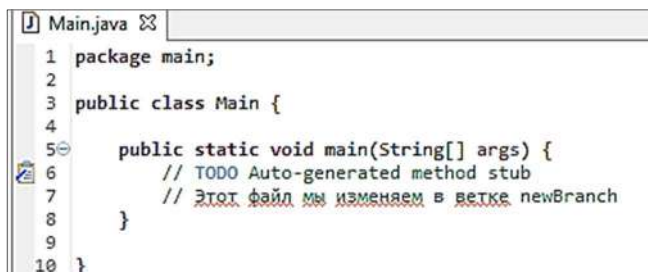


Рисунок 7.8 – Зміна файлу проекту

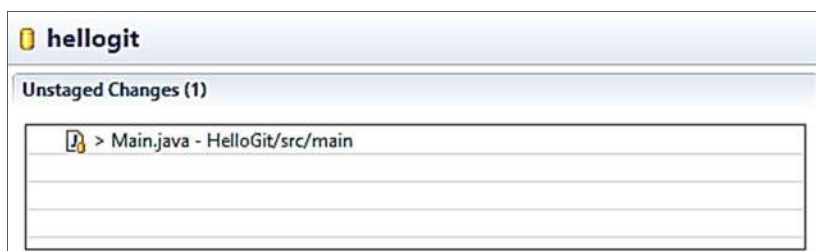


Рисунок 7.9 – Зміни в *stage*-області

– у перспективі *Java* натиснути правою кнопкою мишки на вузлі проекту і в контекстному меню вибрати команду **Team/Add to Index**. У перспективі *Git* зробити **Commit**;

– переключитися на гілку *master* (як це зробити, описано в п. 3). Видно, що в гілці *master* доданий рядок відсутній (рис. 7.10);

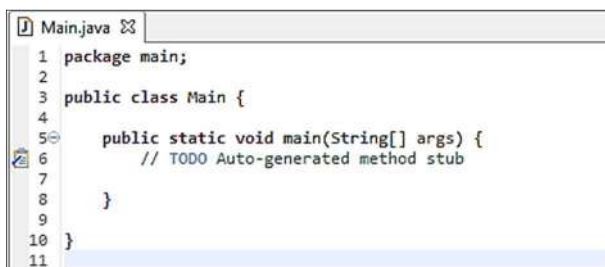


Рисунок 7.10 – Зміна у файлі в гілці *master*

— для виконання злиття (*merge*) в перспективі *Java* потрібно виконати команду **Team/Merge**. У вибраному списку показуються доступні гілки для *merge*. Цікавим є злиття з гілки **newBranch** (рис. 7.11);

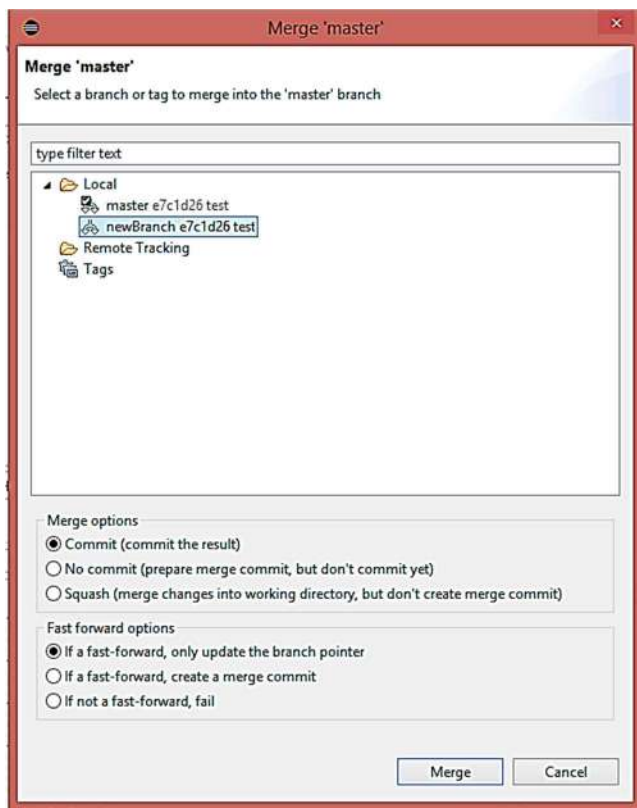
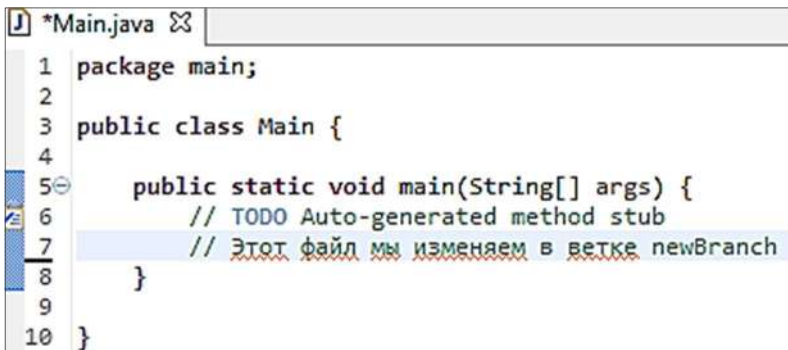


Рисунок 7.11 – Злиття з гілки *newBranch*

— після натискання кнопки **Merge** з'являється повідомлення про успішну зміну файлів. Подивившись тепер на файл класу (рис. 7.12), можна побачити, що зміни додалися в існуючий файл. Тепер необхідно використувати *commit*.



```
1 package main;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         // Этот файл мы изменяем в ветке newBranch
8     }
9
10 }
```

Рисунок 7.12 – Змінений файл класу

### **Висновок**

Дана практична робота вчить використовувати систему контролю версій *Git* з середовища розробки *Eclipse*, використовуючи графічний інтерфейс і не вдаючись до команд консолі. Практично у всіх *IDE* є попередньо встановлена система контролю версій. Працювати в графічному середовищі зручно, хоча використовуються не всі можливості, для успішної роботи цього достатньо.

### **Контрольні запитання**

1. Як відправити проект в *Eclipse* на локальний репозиторій *Git*?
2. Як створити нову гілку у репозиторії?
3. Якою командою відбувається переключення між гілками?
4. Якою командою відбувається злиття двох гілок?
5. Як змінити файл усередині заданої гілки?



## РОБОТА З КОНФЛІКТАМИ. СТВОРЕННЯ SSH КЛЮЧА

**Мета:** створення і використання *SSH* ключа при клонуванні віддаленого репозиторію. Робота з гілками і вирішення конфліктів

### *Теоретичні відомості*

Клонування віддаленого репозитарію можна проводити двома способами: за допомогою *https* або *ssh*.

**SSH** (англ. *Secure Shell* — «безпечна оболонка») — мережевий протокол прикладного рівня, що дозволяє виробляти віддалене управління операційною системою і тунелювання *TCP*-з'єднань (наприклад, для передачі файлів). По суті, *SSH* — це файл, з великою кількістю різних символів, які дуже важко підробити. Клонування через *SSH* вважається хорошим тоном.

Основна відмінність від *https* — це високий рівень безпеки передачі даних.

Це зв'язка між призначеним для користувача віддаленим акаунтом на *GitHub* і локальним комп'ютером. При спробі клонування через *SSH* ключ, *git* бере *SSH* ключ користувача, який генерується локально і порівняти його з *SSH* ключем, який прив'язаний до акаунту на *GitHub*.

Також важливим плюсом є те, що при коммітах або інших діях не потрібно постійно авторизуватися.

### *Вимоги до виконання практичної роботи*

Для виконання даної практичної роботи необхідно мати встановлений локально *git* і акаунт на *GitHub*.

### *Порядок виконання практичної роботи*

1. Створення *SSH* ключа.
2. Додавання *SSH* ключа в акаунт на *GitHub*.
3. Робота с гілками.

#### *1. Створення SSH ключа*

Для створення *SSH* ключа необхідно виконати такі дії:

— запустити консоль керування *Git* за допомогою команди *git-bash*;

– вставити текст, наведений нижче, замінивши адресу електронної пошти на потрібну і натиснути **Enter**:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

– далі рекомендується прописати пароль (в цілях безпеки):

```
Enter passphrase (empty for no passphrase): [Type a passphrase]  
Enter same passphrase again: [Type passphrase again]
```

– після генерації *SSH* ключа система видасть набір символів;

– додати *SSH* ключ в утиліту **ssh-agent**. Для цього потрібно переконатися, що **ssh-agent** включений:

```
eval "$(ssh-agent -s)"    Agent pid 59566
```

– додати ключ в **ssh-agent**:

```
ssh-add ~/.ssh/id_rsa
```

## 2. Додавання *SSH* ключа в аккаунт на *GitHub*

Для додавання *SSH* ключа в аккаунт на *GitHub* необхідно виконати такі дії:

– копіювати ключ і прописати його в командному рядку в консолі (копіює в буфер обміну ключ):

```
clip < ~/.ssh/id_rsa.pub
```

*Примітка: якщо команда **clip** не працює, можна знайти приховану папку *.ssh*, відкрити файл в текстовому редакторі, і скопіювати його в буфер обміну;*

– зайти на *GitHub* в розділ **Settings** (рис. 8.1);

– в меню **SSH and GPG keys** натиснути на кнопку **New SSH key** (рис. 8.2);

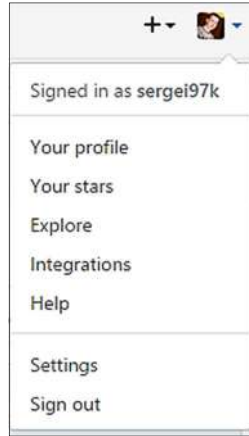


Рисунок 8.1 – Розділ *Settings* на *GitHub*

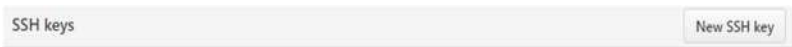


Рисунок 8.2 – Кнопка *New SSH key*

– у полі ***Title*** написати назву ключа, а в поле ***Key*** вставити скопійований ключ (рис. 8.3);

A form for adding a new SSH key. It has two main sections: "Title" and "Key". The "Title" section has a text input field with the placeholder text "My First SSH". The "Key" section has a large text area with a placeholder text that reads: "Begins with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'". At the bottom of the form is a green button labeled "Add SSH key".

Рисунок 8.3 – Вставка ключа

– натиснути *Add SSH key* і підтвердити свій пароль у спливаючому вікні. Далі можна побачити потрібний *SSH* ключ у списку ключів. Іконка ключа повинна бути зеленою – це означає, що даний ключ активний на даний момент. Можна мати необмежену кількість ключів;

– перевірити роботу ключа. Для цього клонуємо з його допомогою репозиторію (рис. 8.4);

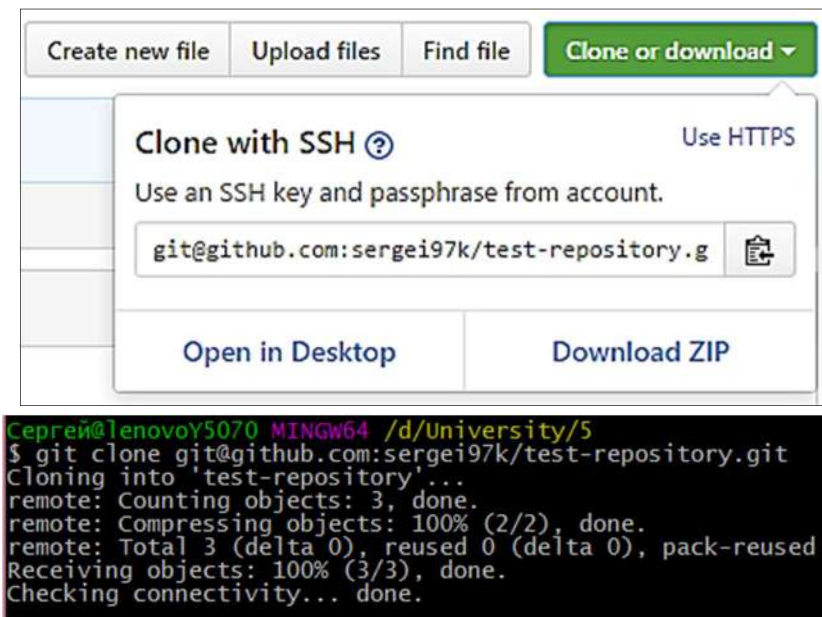


Рисунок 8.4 – Клонування ключа

– переконатися, що папка з ім'ям репозиторію з'явилася на комп'ютері.

### 3. Робота з гілками

Базові команди для роботи з гілками були розглянуті раніше, так що просто створимо два файли з різним вмістом в різних гілках. Для цього треба виконати такі дії:

- створити файл *index.html* з довільним контентом;
- зробити *commit* цього файлу (рис. 8.5);

```

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        index.html

nothing added to commit but untracked files present (use "git add" to track)
Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git add .
Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git commit -m "added file index.html"
[master b248869] added file index.html
1 file changed, 15 insertions(+)
create mode 100644 index.html

```

Рисунок 8.5 – Commit файлу

– створити нову гілку і перейти в неї (рис. 8.6);

```

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git branch list-branch
Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git branch
      list-branch
* master
Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git checkout list-branch
Switched to branch 'list-branch'
Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch)
$

```

Рисунок 8.6 – Перехід в нову гілку

– зайти в потрібний файл і зробити невеликі зміни (наприклад, додати рядок, як на рис. 8.7);

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>test-repository</title>
6  </head>
7  <body>
8    <h1>To Do List</h1>
9    <ul>
10     <li>Wash Up</li>
11     <li>Have a breakfast(eggs and bread)</li>
12     <li>Drink tea(green)</li>
13     <li>Read the newspaper</li>
14   </ul>
15 </body>
16 </html>

```

Рисунок 8.7 – Зміни в файлі

– зберегти зміни (рис. 8.8) ;

```
Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch)
$ git add .

Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch)
$ git commit -m "updated content in file"
[list-branch 546023c] updated content in file
1 file changed, 3 insertions(+), 2 deletions(-)
```

Рисунок 8.8 – Збереження змін

– перейти на гілку *master* і зробити злиття гілок (рис. 8.9).

```
Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git merge list-branch
Updating b248869..546023c
Fast-forward
 index.html | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)

Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git add .

Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git commit -m "merge branch"
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

Рисунок 8.9 – Злиття гілок

Як видно, все вийшло і потрібний файл оновлено. Але бувають випадки, коли виникають конфлікти при злитті, якщо в проєкті велика кількість гілок. Розглянемо такий випадок:

– перейти на гілку *list-branch* (рис. 8.10) і додати зміни у файлі (рис. 8.11) ;

```
Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch)
$ git add .

Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch)
$ git commit -m "swap the list element"
[list-branch 898c2ba] swap the list element
1 file changed, 1 insertion(+), 1 deletion(-)
```

Рисунок 8.10 – Перехід на гілку *list-branch*

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>test-repository</title>
6  </head>
7  <body>
8      <h1>To Do List</h1>
9      <ul>
10         <li>Wash Up</li>
11         <li>Have a breakfast(eggs and bread)</li>
12         <li>Read the newspaper</li>
13         <li>Drink coffee</li>
14     </ul>
15 </body>
16 </html>

```

Рисунок 8.11 – Зміни у файлі

– створити нову гілку *list-branch2* (рис. 8.12) і зробити інші зміни (рис. 8.13);

```

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git checkout -b list-branch2
Switched to a new branch 'list-branch2'

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch2)
$ git add .

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (list-branch2)
$ git commit -m "change the last element"
[list-branch2 c7f3c92] change the last element
1 file changed, 2 insertions(+), 2 deletions(-)

```

Рисунок 8.12 – Створення нової гілки *list-branch2*

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>test-repository</title>
6  </head>
7  <body>
8      <h1>To Do List</h1>
9      <ul>
10         <li>Wash Up</li>
11         <li>Have a breakfast(eggs and bread)</li>
12         <li>Drink tea(black)</li>
13         <li>Watch the TV</li>
14     </ul>
15 </body>
16 </html>

```

Рисунок 8.13 – Зміни у файлі

– перейти на гілку **master** і зробити злиття спочатку з гілкою **list-branch2**, а потім с **list-branch** (рис. 8.14);

```
Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git merge list-branch2
Updating 546023c..c7f3c92
Fast-forward
 index.html | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

Cepрей@lenovoY5070 MINGW64 /d/University/5/test-repository (master)
$ git merge list-branch
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Рисунок 8.14 – Злиття гілок

– в результаті виник конфлікт. Для того щоб його вирішити потрібно відкрити файл (рис. 8.15);

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>test-repository</title>
6  </head>
7  <body>
8    <h1>To Do List</h1>
9    <ul>
10     <li>Wash Up</li>
11     <li>Have a breakfast(eggs and bread)</li>
12     <<<<<<< HEAD
13     <li>Drink tea(black)</li>
14     <li>Watch the TV</li>
15     =====
16     <li>Read the newspaper</li>
17     <li>Drink coffee</li>
18     >>>>>> list-branch
19   </ul>
20 </body>
21 </html>
```

Рисунок 8.15 – Робочий файл

– *Git* позначає конфлікт таким чином, що від стрілок <<< до ==== – це зміни з *першої* гілки яку зливали, а от === до стрілки >>> зміни з *другої* гілки. Для того щоб вирішити конфлікт, необхідно прибрати зайві знаки (<,,>) і залишити тільки ту частину коду, яка необхідна. Наприклад, можна зробити так (рис. 8.16):



```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>test-repository</title>
6  </head>
7  <body>
8    <h1>To Do List</h1>
9    <ul>
10     <li>Wash Up</li>
11     <li>Have a breakfast(eggs and bread)</li>
12     <li>Watch the TV</li>
13     <li>Read the newspaper</li>
14     <li>Drink coffee</li>
15   </ul>
16 </body>
17 </html>

```

Рисунок 8.16 – Рішення конфлікту в файлі

– зберегти зміни, тим самим вирішивши конфлікт (рис. 8.17);

```

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master|MERGING)
$ git add .

Сергей@lenovoY5070 MINGW64 /d/University/5/test-repository (master|MERGING)
$ git commit -m 'resolve conflict'
[master fe57cd5] resolve conflict

```

Рисунок 8.17 – Збереження змін

– переконатися, що виконані всі комміти і виконати **push** зміни в репозиторій на *GitHub* (рис. 8.18);




<b>resolve conflict</b> sergei97k committed 3 minutes ago	 fe57cd5	
<b>change the last element</b> sergei97k committed 18 minutes ago	 c7f3c92	
<b>swap the list element</b> sergei97k committed 21 minutes ago	 898c2ba	
<b>updated content in file</b> sergei97k committed 35 minutes ago	 546023c	
<b>added file index.html</b> sergei97k committed 41 minutes ago	 b248869	
<b>Initial commit</b> sergei97k committed 2 hours ago	 e58eb20	

Рисунок 8.17 – Зміни в репозиторію

### ***Висновок***

В ході практичної роботи було розглянуто питання щодо створення *SSH* ключа і клонування репозиторію з його допомогою, була реалізована робота з гілками і вирішення конфлікту злиття гілок.

Для подальшої роботи з гілками, можна ознайомитися однієї з популярних методологій, наприклад, *GitFlow*.

У звіті до практичної роботи вказати посилання на репозиторій, який був створений у ході виконання роботи.

### ***Контрольні запитання***

1. Як створити *SSH* ключ?
2. Як додавання *SSH* ключа в аккаунт на *GitHub*?
3. Як вирішити конфлікт при злитті гілок?
4. Як зберегти зміни у репозиторії після вирішення конфлікту злиття гілок?
5. Як у файлі позначається конфлікт при злитті гілок?

## СИСТЕМИ УПРАВЛІННЯ ВЕРСІЯМИ: TORTOISESVN

**Мета:** познайомитися зі ще однією розповсюдженою системою контролю версій та визначити схожість та відмінність з *GIT*

### *Теоретичні відомості*

#### *Що таке Subversion*

Це система, яка використовується для спільного управління файлами і каталогами (в т.ч. по мережі), а також зробленими в них змінами в часі. Вона дозволяє відновити попередні версії даних, дає можливість вивчити історію всіх змін. Тому багато хто вважає систему управління версіями свого роду «машиною часу», яка дозволяє не побоюватися, що допущені помилкові зміни даних якось негативно вплинуть на роботу в цілому, адже завдяки збереженню історії змін, завжди можна повернутися до попереднього стану.

Одна з головних переваг даної системи полягає в тому, що вона є системою загального призначення, тобто дозволяє маніпулювати будь-якими типами даних. Спочатку придумана для полегшення роботи розробників, система контролю версій полюбилася і прижилася і в інших професіях, де саме файли є кінцевим результатом праці: письменницька діяльність, наукові роботи та ін.

#### *Загальний принцип роботи Subversion*

В основі *Subversion* використовується модель «Копіювання-Зміна-Злиття»:

- для підключення до сховища *Subversion* використовують *svn*-клієнт;
- створення локальної копії файлів і каталогів на локальному комп'ютері (робоча копія) – етап «**Копіювання**»;
- подальша робота з проектом проводиться вами саме в рамках збереженої копії – етап «**Зміна**»;
- після внесення необхідних змін особисті копії відправляються назад в сховище і зливаються в нову версію – етап «**Злиття**».

*Примітка 1.* Якщо при спробі злиття чиеїсь робочої копії і даних сховища буде виявлений конфлікт (наприклад, декількома користувачами був

відредагували один і той же файл так, що ці зміни «перекрили» один одного), *Subversion* сповістить про це і запропонує внести зміни так, щоб вирішити цей конфлікт. При цьому злиття конфліктує копії з даними сховища буде скасовано.

*Примітка 2.* Модель «Копіювання-Зміна-Злиття» ґрунтується на припущенні про те, що файли контекстно-поєднувані, і воно справедливо, якщо більшість файлів в сховище – текстові файли (наприклад, вихідні коди програм). Для файлів же бінарних форматів (графічні, аудіо-файли) існує проблема неможливості об’єднання конфліктуючих змін. Для вирішення цієї проблеми *Subversion* наділена спеціальним функціоналом, який передбачає доступ до певного файлу відкритим для редагування для єдиного користувача в конкретний момент часу.

### ***Вимоги до виконання практичної роботи***

Для взаємодії з *svn*-репозиторієм необхідно встановити *svn*-клієнт – графічний або консольний. *Svn*-клієнт – це спеціальна програма, яка підтримує певний набір команд, що дозволяють виконувати різні дії над даними в репозиторії. Далі розглядається робота з графічним *svn*-клієнтом *TortoiseSVN*.

### ***Порядок виконання практичної роботи***

Налаштування графічного *svn*-клієнта під ОС сімейства *Windows* – *TortoiseSVN* складається з таких пунктів.

1. Установка *TortoiseSVN*.
2. Завантаження файлів в сховище і управління його структурою заходами *TortoiseSVN*.
3. Створення робочої копії на локальному комп’ютері.
4. Типовий робочий цикл репозиторію.

#### ***1. Установка TortoiseSVN***

Основна перевага цього популярного графічного клієнтського додатка до *Subversion* – багатий функціонал і зручність використання. Щоб почати роботу, необхідно:

- завантажити потрібну програму;
- встановити її на робочий комп’ютер;
- перезавантажити комп’ютер.

Тепер *TortoiseSVN* працює аналогічно плагіну програми **Провідник** у *Windows*. Доступ до нього можна отримати, натиснувши на потрібній директорії правою кнопкою миші – в контекстному меню в списку опцій будуть представлені додаткові пункти (рис. 9.1).

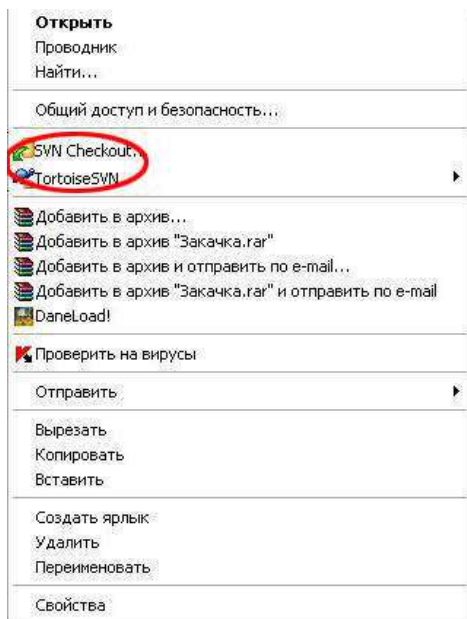


Рисунок 9.1 – Додаткові пункти в контекстного меню

## 2. Завантаження файлів в сховище і управління його структурою засобами *TortoiseSVN*

Для завантаження файлів в сховище і управління його структурою засобами *TortoiseSVN* необхідно зробити такі дії:

- щоб перенести будь-яку директорію (наприклад, *project*) з локального комп'ютера в сховище *Subversion* для подальшого зберігання та управління засобами *svn*, відкрийте **Провідник Windows**, натисніть правою кнопкою миші по директорії *project* і виберіть пункт меню **TortoiseSVN** → **Import** (рис. 9.2);

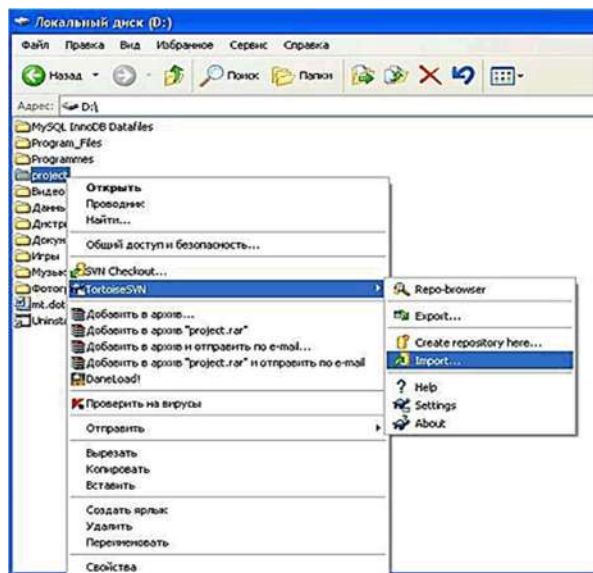


Рисунок 9.2 – Перенесення директорії *project* в сховище *Subversion*

– для звернення до сховища необхідно використовувати URL <http://your-domain.ru/svn>, де *your-domain.ru* – ваш домен. У віконці, що з’явилося в полі URL потрібно ввести повний шлях до каталогу в сховищі, в якому планується зберігати файли (у прикладі це каталог *project\_1*). Якщо зазначеного каталогу в сховищі немає, він буде створений. У текстовому полі нижній частині вікна можна ввести свій пояснювальний коментар до завантажуваних даних (рис. 9.3);

– після натискання кнопки ОК і введення вірних логіна і пароля відкриється вікно, що показує завершення етапу імпорту директорії *project* в сховище *Subversion* (рис. 9.4).

*Примітка.* Вважається загальноприйнятим для зручності всередині кожного каталогу в сховищі, відповідного окремого проекту, створювати три каталогу: **trunk** (поточна робоча версія), **branches** (сюди рекомендується поміщати готові, але ще не протестовані версії проекту) і **tags** (сюди потрапляють, як правило, готові рішення, доступні кінцевим користувачам). Це лише рекомендована, але зовсім не обов’язкова структура, якої потрібно дотримуватися;

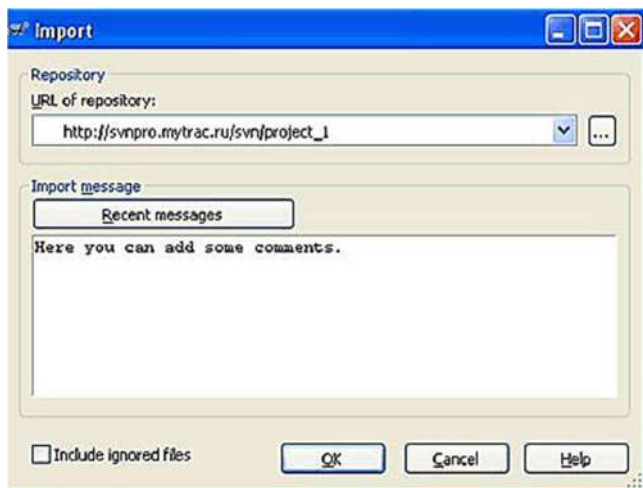


Рисунок 9.3 – Використання URL

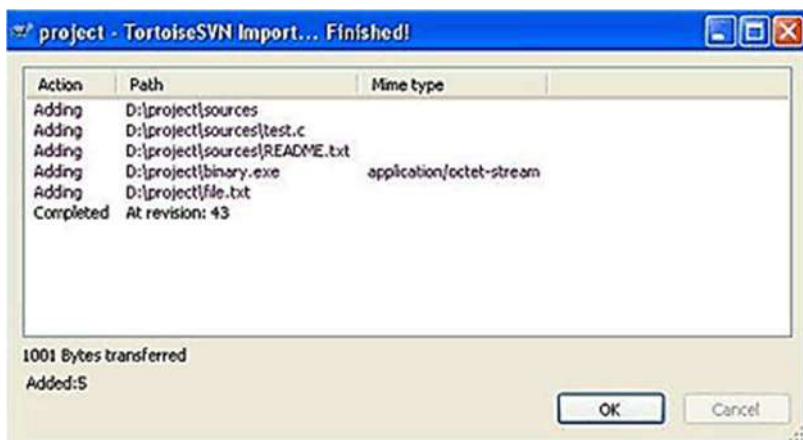


Рисунок 9.4 – Завершення процесу імпорту директорії

– подивитися вміст сховища, натиснувши правою кнопкою миші по директорії і вибравши **Repo-browser** (рис. 9.5);

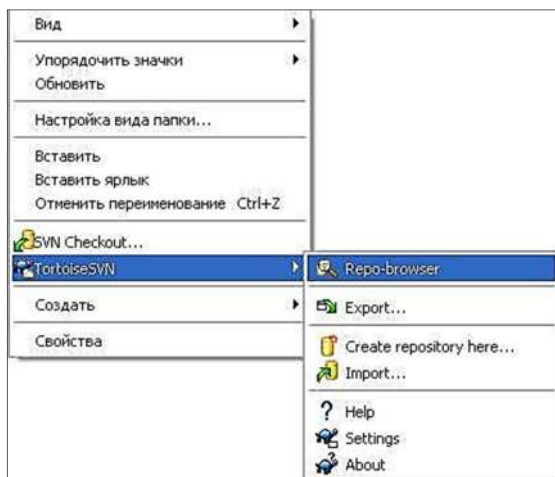


Рисунок 9.5 – Перегляд вмісту сховища

– у вікні ввести шлях до кореневого каталогу потрібного сховища (<http://your-domain.ru/svn>). В результаті з’явиться вікно, в якому вміст всього репозиторію буде представлено у вигляді дерева (рис. 9.6).

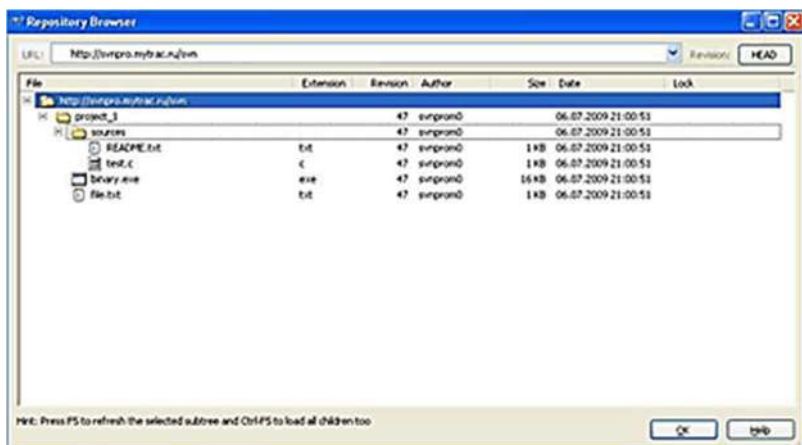


Рисунок 9.6 – Вміст всього репозиторію



Натисканням правої кнопки миші відкривається можливість управління сховищем (додавати і видаляти елементи; завантажувати елементи в свою локальну робочу копію; копіювати елементи як звичайні файли або каталоги).

Крім структури сховища, у вікні **Repo-browser** відображається декілька стовпців, які описують властивості елементів даної структури:

**Extension** – якщо елемент є файлом, в цьому стовпці навпроти нього буде відображатися його розширення, якщо елемент є каталогом – нічого не відображатиметься;

**Revision** – номер правки відповідного елемента сховища. виправлення є «знімок» сховища в конкретний момент часу. При створенні сховища *Subversion* воно починає своє існування з правки 0, і кожна наступна фіксація збільшує номер правки на одиницю. У будь-який момент часу, якщо потрібно буде посылитися на будь-яку правку, можна зробити це, натиснувши на кнопку **«HEAD»** у верхньому правому кутку. У віконці, що з'явилося потрібно переставити прапорець на **«Revision»** і ввести в поле праворуч номер правки, яку потрібно подивитися (рис. 9.7).

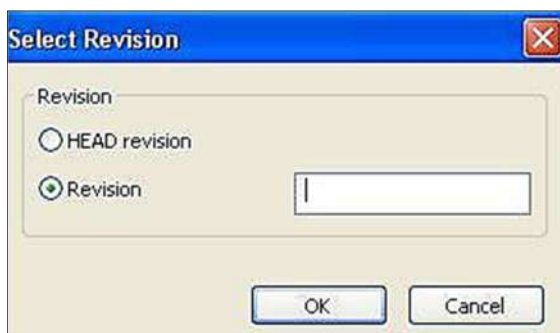


Рисунок 9.7 – Вікно вибору ревізій

Атрибути поточної ревізії:

*Author* – автор поточної правки;

*Size* – розмір елемента;

*Date* – час модифікації елемента, що відповідає номеру правки, зазначеної в стовпці *Revision*;

*Lock* – показує, чи заблокований файл для редагування.

*Примітка.* При будь-яких змінах, що стосуються додавання/видалення елементів у сховище, вироблених описаним вище способом, тобто безпосередньо через **Repo-browser**, це ніяк не відобразиться в локальних робочих копіях користувачів. Для того щоб ці зміни стали помітні в робочій копії, користувач повинен зв'язатися зі сховищем і завантажити оновлену версію на комп'ютер. Про те, як це зробити, буде розказано далі.

### 3. Створення робочої копії на локальному комп'ютері

Для того щоб своєчасно і коректно вносити зміни в сховище, а також отримувати звіди зміни, зроблені іншими користувачами, вам необхідно зробити такі дії:

– створити так звану «робочу копію». Для цього потрібно вибрати директорію, в якій вона буде зберігатися, і натиснути по ній правою кнопкою миші. Далі вибрати пункт **SVN Checkout** (рис. 9.8);

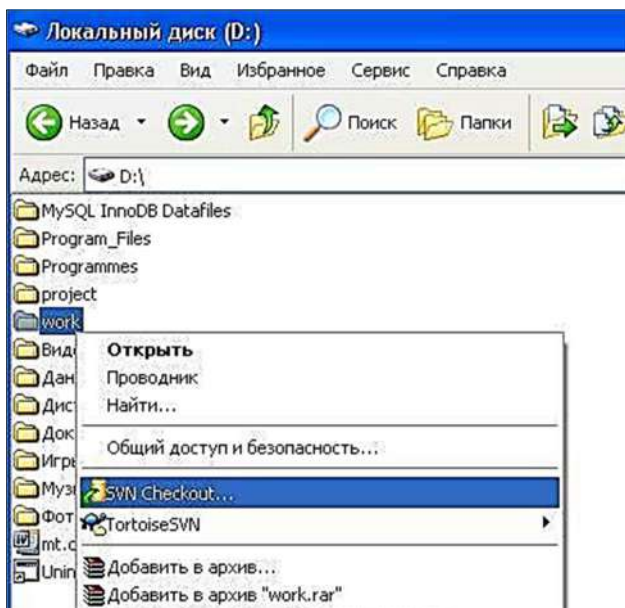


Рисунок 9.8 – Створення «робочої копії»

– в діалоговому вікні необхідно вказати шлях до даних, які потрібно завантажити в робочу копію; також є можливість змінити шлях до робочого каталогу, якщо це необхідно (рис. 9.9);

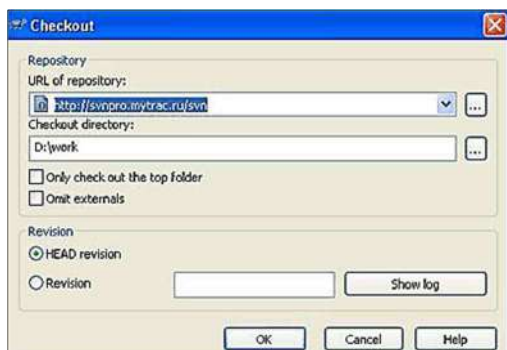


Рисунок 9.9 – Вказівка шляху до даних

– в результаті буде представлено вікно, яке показує, що процес закінчений (рис. 9.10);

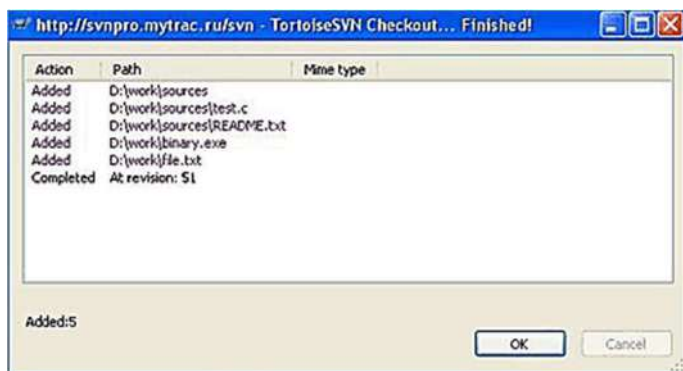


Рисунок 9.10 – Вікно закінчення процесу

– після натискання правої кнопки миші на каталозі, що містить робочу копію, а також на її елементах (вони відзначені «галочкою»), будуть представлені додаткові опції (рис. 9.11).

*Примітка.* Крім файлів робочої копії, в робочому каталозі створюється директорія `.svn` (за замовчуванням прихована). Вона містить в собі службову інформацію, тому видаляти її вкрай не рекомендується – це може порушити коректність зв'язку зі сховищем.

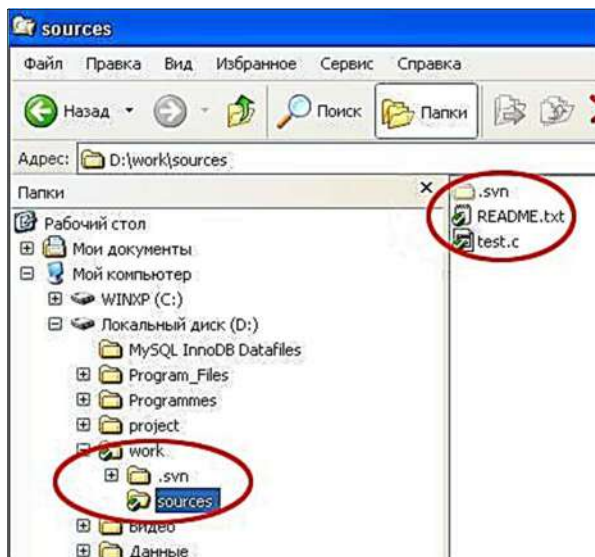


Рисунок 9.11 – Додаткові опції

#### 4. Типовий робочий цикл

##### 4.1. Операція отримання даних зі сховища:

– відкрити у **Провіднику Windows** каталог, що містить робочу копію даних, натиснути правою кнопкою миші по каталогу з робочою копією та вибрати пункт меню *SVN Update* (рис. 9.12). Після того як операція буде завершена, можна побачити віконце зі списком оновлених файлів.

*Примітка.* Передбачається, що всі наступні операції, аж до відправки змінених даних в сховищі, застосовуються до робочої копії, розташованої на локальному комп'ютері. Потрібно пам'ятати, що можна виконувати всі ці операції безпосередньо всередині сховища, через опцію **Repo-browser** (про це було описано докладно вище). Однак в цьому випадку, щоб побачити зміни в робочій копії, потрібно буде виконувати команду *SVN Update*.

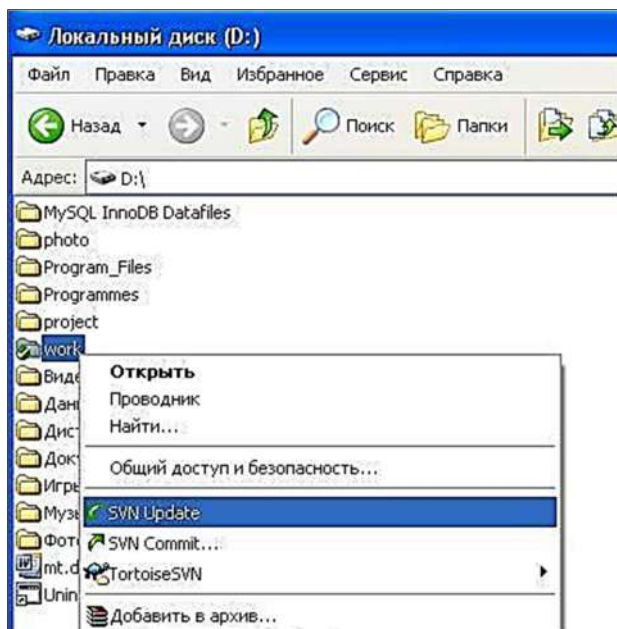


Рисунок 9.12 – Відкриття каталогу з робочою копією даних

#### 4.2. Внесення будь-яких змін всередині самих файлів робочої копії

Цю операцію проводиться в звичайному режимі, тобто просто потрібно відкрити файли робочої копії.

#### 4.3. Внесення змін до структури проекту

Ці операції найзручніше проводити через контекстне меню:

- додати файл; для цього вибрати пункт меню **Add...** (рис. 9.13);
- в діалоговому вікні зазначити вибрані для додавання файли (рис. 9.14). Аналогічним чином в робочу копію додаються каталоги, при цьому за замовчуванням всі елементи, що знаходяться всередині копії, програма також запропонує додати;

*Примітка.* svn-клієнт може тільки додавати або видаляти елементи, створювати файли самостійно він не може.

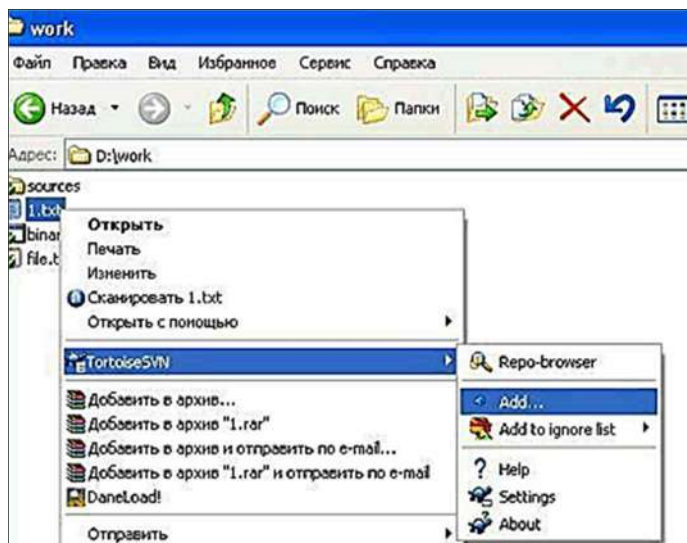


Рисунок 9.13 – Пункт меню для добавления файла

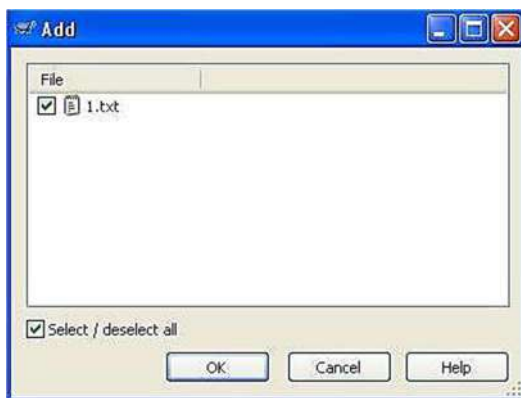


Рисунок 9.14 – Вікно вибору файлів для додавання

– видалити файл. Для видалення файлу вибрати пункт меню **Delete**

(рис. 9.15), після чого файл буде видалений з робочої копії (аналогічно видаляються каталоги);

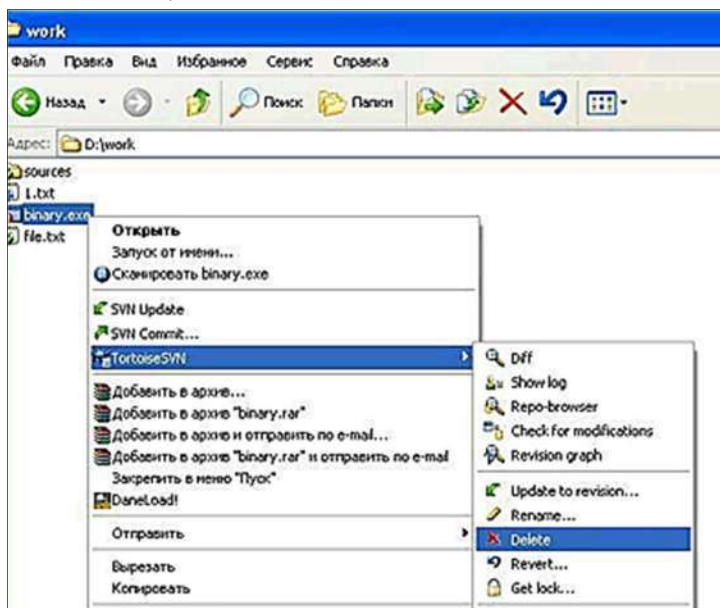


Рисунок 9.15 – Вибір команди видалення файлів

– перейменувати файл. Для цього потрібно вибрати пункт меню **TortoiseSVN → Rename**. При цьому створиться копія вихідного файлу з таким же змістом, але з новим ім'ям. Якщо вихідний файл не потрібен, видалити його. Дії з каталогами виробляються аналогічно.

#### 4.4 Аналіз змін

– команда **svn status** дає всю потрібну інформацію щодо того, що змінилося в робочій копії, при цьому не потрібно звертатися до сховища і зливати нові зміни, опубліковані іншими користувачами. Детальніше про дану команду можна дізнатися на сторінці інструкції з *Subversion*.

*TortoiseSVN* також надає кошти для перегляду подібної інформації (яких локальних змін зазнали елементи копії, не пов'язуючись зі сховищем) – пункти **show log** (рис. 9.16) та **check for modifications** (рис. 9.17);

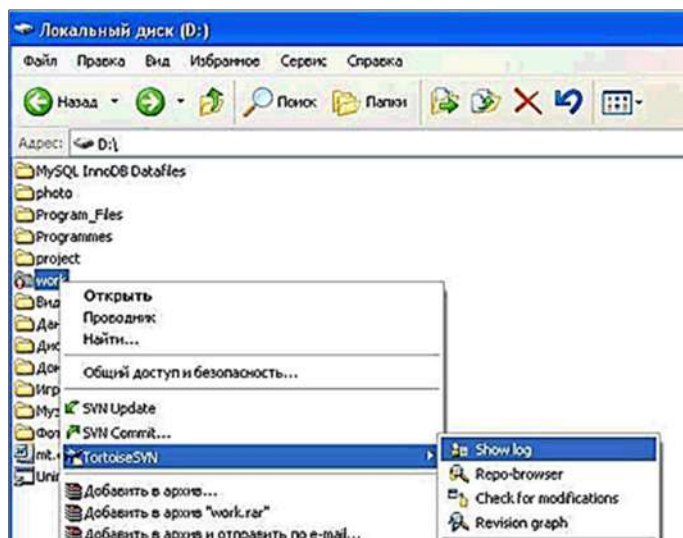


Рисунок 9.16 – Пункт *show log*

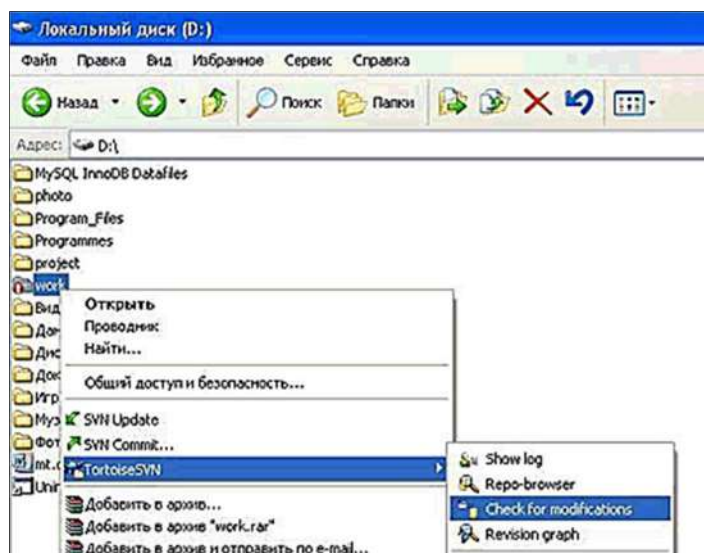


Рисунок 9.17 – Пункт *check for modifications*



– результати роботи команд *show log* (рис. 9.18) та *check for modifications* (рис. 9.19);

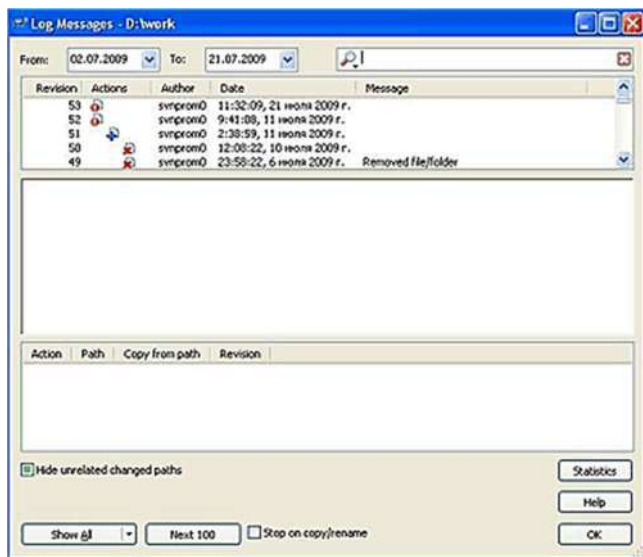


Рисунок 9.18 – Результати роботи команди *show log*

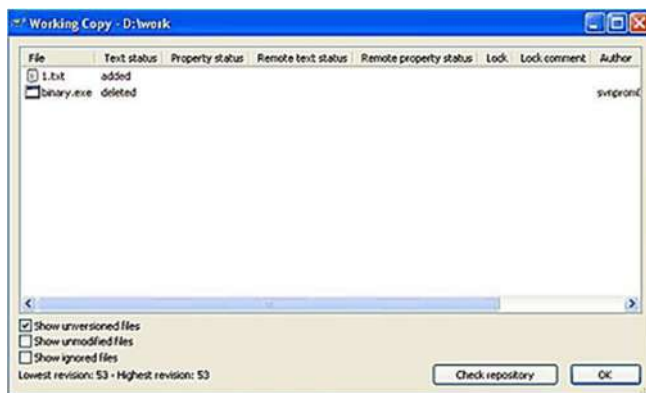


Рисунок 9.19 – Результати роботи команди *check for modifications*

— команда **svn diff** формує свій висновок, порівнюючи робочі файли з кешованими «недоторканими» копіями з **.svn**. Весь текст запланованих для додавання файлів показується як доданий, а весь текст запланованих для видалення файлів показується як видалений. *TortoiseSVN* пропонує для подібного аналізу опцію **Create patch**.

За допомогою даної опції порівнюються робочі файли з кешованими «недоторканими» копіями з **.svn**, формуючи висновок в окремий файл з розширенням **.patch(.diff)**. Крім того, вона дозволяє зробити подібний аналіз тільки для окремих змінених елементів (рис. 9.20);

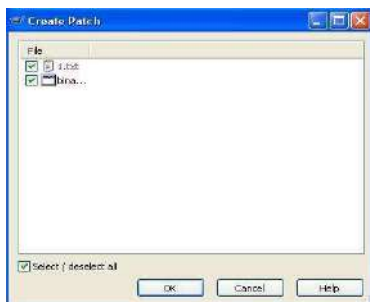


Рисунок 9.20 – Вибір файлів

— в результаті буде отриманий файл виправлень (рис. 9.21). У рядку з написом **Index** міститься ім'я аналізованого файлу. Плюсами відзначені додані рядки, мінусами – видалені. Крім того, якщо з файлами проводилися які-небудь дії, то в дужках буде вказано номер відповідної цим діям ревізії;

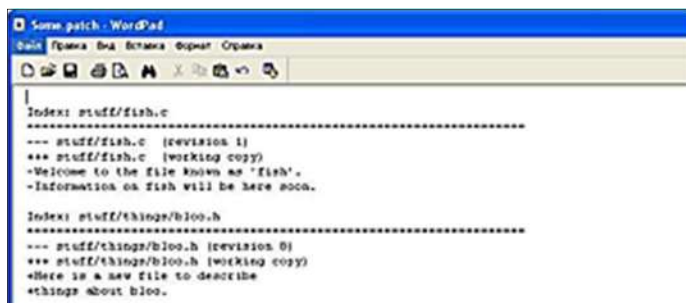


Рисунок 9.21 – Файл виправлень

– команда **svn revert** дозволяє повернути файл в стан, що передувє модифікації, шляхом заміни файлу його кешованою «первісною» копією з **.svn**-області. Крім того, консольна версія даної команди дозволяє повернутися на певну ревізію. Детальніше про цю команду в керівництві по *Subversion*.

*TortoiseSVN* має в своєму арсеналі дві функції – **Revert** та **Update to revision**. Перша просто повертає файл в стан, що передувє модифікації. Друга дозволяє повернутися на певну ревізію. Нижче представлено приклад команди **Revert** (повертаємо файл **binary.exe** і скасовуємо додавання файлу **1.txt** в *Subversion*) (рис. 9.22);

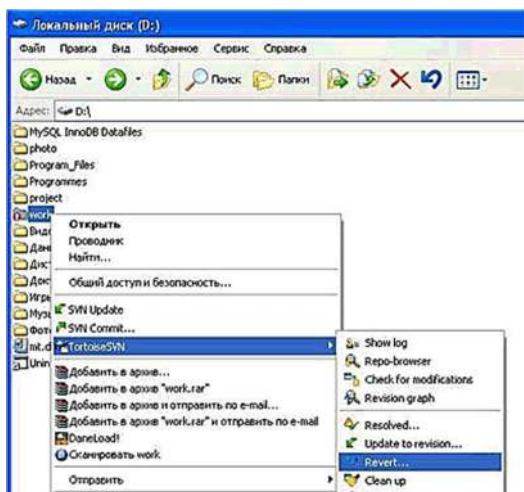


Рисунок 9.22 – Вибір команди *Revert*.

– позначити файли, які потрібно повернути (рис. 9.23).

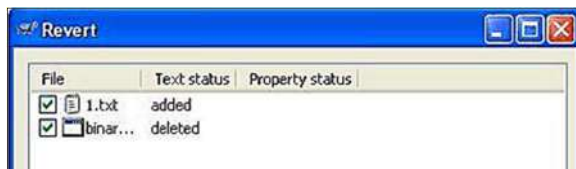


Рисунок 9.23 – Вибір необхідних файлів

#### 4.5. Злиття поточних змін зі змінами в сховище

Завершивши зміни робочої копії, необхідно узгодити їх зі змінами в сховище. Для цього:

– виконати команду **SVN Update**. Якщо ніяких попереджень в ході забору змін зі сховища отримано не було, значить команда була завершена успішно. Розглянемо випадок, коли одночасно з одним користувачем інший користувач редагує файл, наприклад, **sources/README.txt**. При виконанні **SVN Update** буде отримано наступне повідомлення (рис. 9.24).

Це означає, що зміни, внесені користувачем, перекрилися зі змінами, внесеними в сховище раніше. У цьому випадку буде створено кілька примірників файлу **README.txt**:

**README.txt** – в ньому будуть перебувати зміни всіх користувачів, помічені спеціальними маркерами;

**README.txt.mine** – файл робочої копії користувача зі змінами;

**README.txt.r53** – файл, відповідний попередньої ревізії (53 в прикладі);

**README.txt.r54** – файл, відповідний новій ревізії (54 в прикладі);

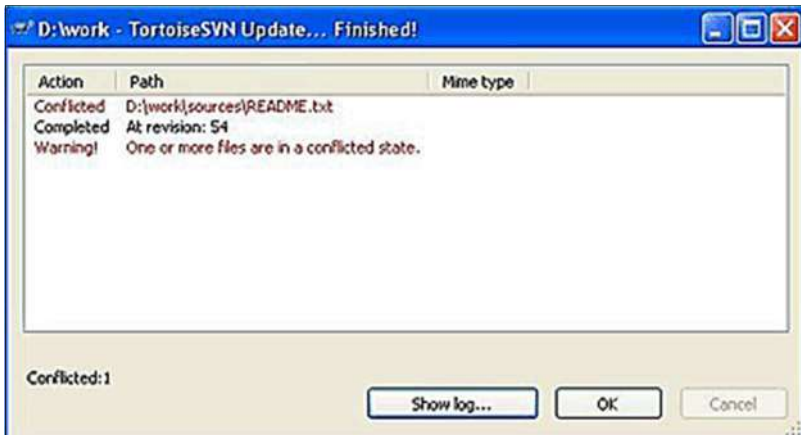


Рисунок 9.24 – Повідомлення про одночасне редагування файлу

– необхідно вирішити цей конфлікт і повідомити про це *Subversion*. Можна або скасувати свої зміни, або скасувати чужі, або якимось чином скомбінувати свої і чужі зміни (*Subversion* тільки повідомляє про конфлікт, процедура дозволу же повністю під контролем користувача, який працює

зі сховищем в даний момент). Після того як було закінчено вирішення конфлікту, користувач повідомляє про це *Subversion* виконанням команди **Resolved** (рис. 9.25);

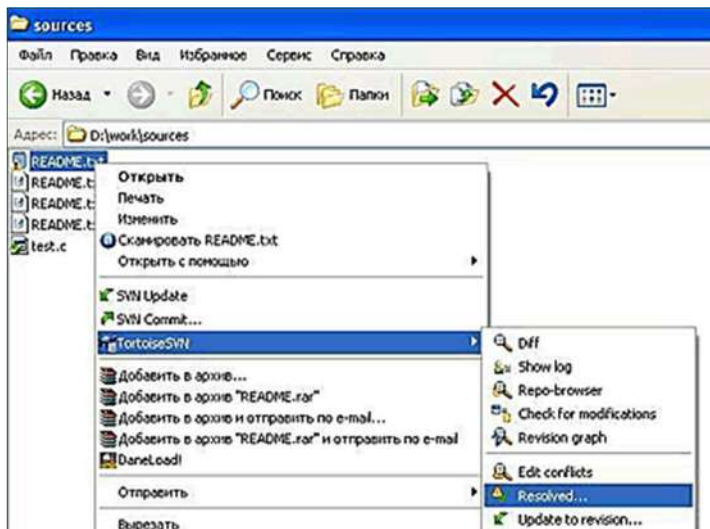


Рисунок 9.25 – Вибір команди *Resolved*

– відмітити файли, в яких конфлікт дозволений, і натиснути **OK** (рис. 9.26).

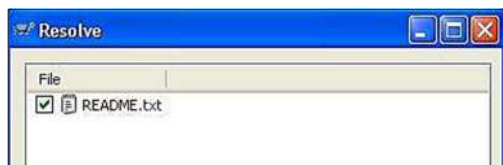


Рисунок 9.26 – Вибір необхідних файлів

#### 4.6. Відправка змін в сховище

Для відправки змін в сховище необхідно виконати такі дії:

– виконати команду **SVN Commit** (рис. 9.27);

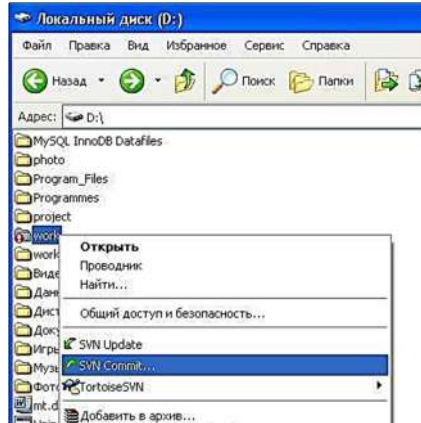


Рисунок 9.27 – Вибір команди *SVN Commit*

– додати необхідні коментарі (рис. 9.28);

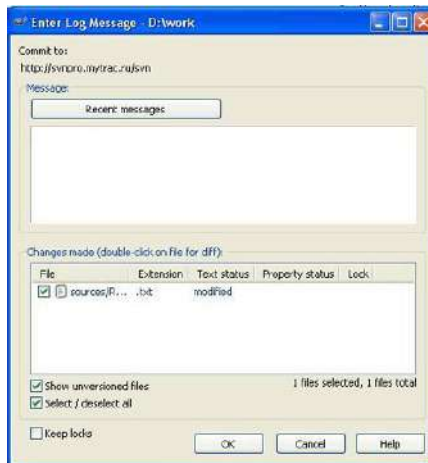


Рисунок 9.28 – Додавання коментарів

– найпростіший робочий цикл на цьому закінчено (рис. 9.29).

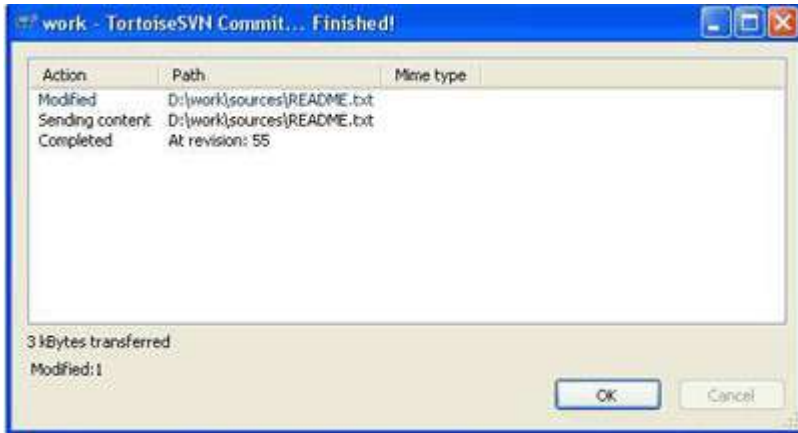


Рисунок 9.29 – Закінчення найпростішого робочого циклу

### ***Висновок***

У ході практичної роботи навчилися працювати з системою контролю версій *SVN*, та виконувати найбільш типові дії з файлами у репозиторії.

У звіті треба не забути привести посилання на репозиторій, який був використаний при написанні даної практичної роботи.

### ***Контрольні запитання***

1. Пояснити суть моделі «Копіювання-Зміна-Злиття».
2. Як отримати доступ до клієнта *TortoiseSVN*?
3. Як завантажити файли в сховище засобами *TortoiseSVN*?
4. Як подивитися вміст сховища?
5. Як створити «робочу копію» репозиторію на локальному комп'ютері?
6. Як внести зміни до структури проекту?

## МЕТОДОЛОГІЯ РОЗРОБКИ ПРОЕКТІВ З ВИКОРИСТАННЯМ GIT – GIT-FLOW

**Мета:** познайомитися з паттерном роботи з гілками у *git*. Освоїти *SourceTree* – ще один інструмент роботи з *git*

### **Теоретичні відомості**

**Git-flow** – це набір розширень *git* надає високорівневі операції над репозиторієм для підтримки моделі розгалуження *Vincent Driessen* (оригінальна стаття від *Vincent*: <http://nvie.com/posts/a-successful-git-branching-model/> , переклад: <https://habrahabr.ru/post/106912/>).

Сам *git* не прив'язує нас до якогось певного способу розробки, і кожен розробник, в теорії, може працювати з контролем версій так, як він хоче. Щоб в таких умовах не занурити репозиторій в хаос, потрібно придумати і донести до всіх розробників якийсь єдиний стандарт для роботи з контролем версій в проекті. **Git-flow** дає готовий стандарт, перевірений часом, і вже відомий багатьом розробникам.

У той же час, потрібно розуміти, що методологія **Git-flow** не є єдиною і на 100% універсальною. У поточному проекті може існувати власний підхід до роботи з *git*. Однак, якщо необхідно працювати з *git* в команді, то варто знати про те, що таке **Git-flow** і в чому його особливості.

### **Суть розгалуження в Git-flow.**

Всього в *git-flow* існує п'ять типів гілок, кожна з яких несе певну функціональне навантаження (рис. 10.1).

#### **Гілка master**

Гілка **master** (рис. 10.2) створюється при ініціалізації репозиторію, що має бути знайомим кожному користувачеві *Git*. В *git-flow* – це гілка, куди надходять найстабільніші зміни, які йдуть в реліз. Існує протягом усього процесу розробки. В гілці *master* зливаються тільки зміни з гілок *release* і *hotfix*. На кожне таке злиття створюється тег з ім'ям версії.

По суті, в самій гілці не повинно бути ніяких *commit'ів*, крім *commit'ів* злиття, кожен з яких повинен бути відзначений тегом версії.



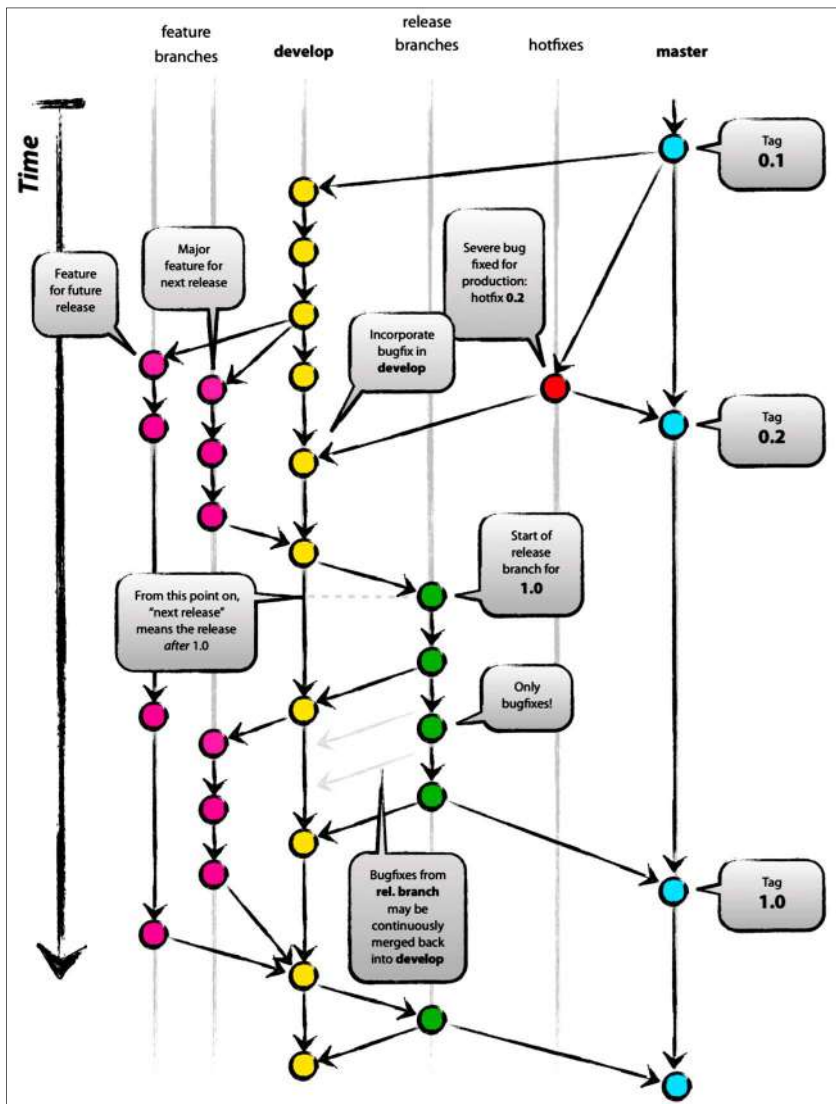


Рисунок 10.1 – Суть розгалуження в *Git-flow*

### Гілка *develop*

Паралельно гілці **master** також створюється гілка для розробки під назвою **develop** (см. рис. 10.2).

Гілка **develop** вважається головною гілкою для розробки. Код, що зберігається в ній, в будь-який момент часу повинен містити найостанніші видані зміни, необхідні для наступного релізу.

Суть в тому, що вся робота відбувається в гілці **develop**, а в гілці **master** зміни зливаються, коли є якась готова версія продукту – реліз.

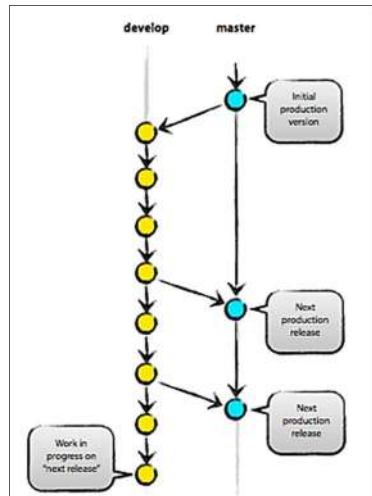


Рисунок 10.2 – Гілки *master* у *develop*

### Гілки *feature*-\*

**Feature** гілки використовуються для розробки нових функцій, які повинні з'явитися в поточному або майбутньому релізах. Кожна гілка **feature** відділяється від **develop** і зливаються назад в неї (рис. 10.3). Після початку роботи над функціональністю (фичей) може бути ще невідомо, в який саме реліз вона буде додана. Сенса існування **feature** гілки полягає в тому, що вона живе так довго, скільки триває розробка цієї функціональності. Коли робота в гілці завершена, остання вливається назад в **develop** (що означає, що функціональність буде додана в майбутній реліз) або ж видаляється (в разі невдалого експерименту).

Ім'я гілки повинна відповідати імені розробляється функціональності, наприклад «*feature-statistic-integration*».

У деяких проектах префікс «*feature-*» опускається (тоді будь-яка гілка без префікса – це гілка *feature*).

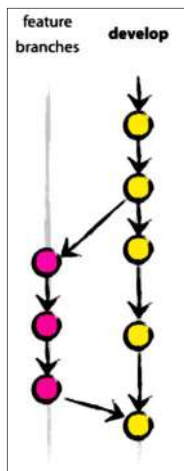


Рисунок 10.3 – Гілка *feature*

#### *Гілки release-\**

**Release** гілки використовуються для підготовки до випуску нових версій продукту. Вони дозволяють розставити фінальні крапки над і перед випуском нової версії. Крім того, в них можна додавати мінорні виправлення, а також готувати метадані для чергового релізу (номер версії, дата складання та ін.). Коли вся ця робота виноситься в гілку релізів, головна гілка розробки (*develop*) очищається для додавання наступних змін (які увійдуть в наступний великий реліз).

Як тільки робота над версією закінчується, відбувається фінальне злиття гілки в *master* і *develop*, після чого гілка видаляється, а *commit* у в *master* присвоюється тег нової версії.

Ім'я гілки має відповідати версії, що випускається, наприклад «*release-1.4*».

### Гілки *hotfix*-\*

**Hotfix** гілки вельми схожі на **release** гілки, так як вони теж використовуються для підготовки нових випусків продукту, хіба лише незапланованих. Вони породжуються необхідністю негайно виправити небажану поведінку виробничої версії продукту. Коли у виробничій версії знаходиться помилка (баг), що вимагає негайного виправлення, з відповідного даної версії тега головної гілки (**master**) породжується нова гілка для роботи над виправленням.

Гілка відходить від **master** (рис. 10.4) і по завершенню правок зливається назад в **master** і в **develop**. Сама гілка після цього видаляється, а *commit* у злиття в **master** присвоюється тег нової версії.

Сенс її існування полягає в тому, що робота команди над гілкою розробки (**develop**) може спокійно тривати, в той час як хтось один готує швидко виправлення виробничої версії.

Іменем гілки зазвичай є нова версія з урахуванням версії правок, наприклад «*hotfix-1.4.1*» (перша правка версії 1.4).

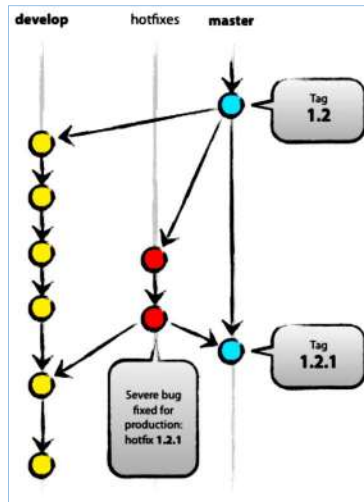


Рисунок 10.4 – Гілка *hotfix*

Так само, **git-flow** підтримується деякими популярними графічними оболонками такими як *SourceTree* і *SmartGit* у базовій версії («з коробки»).

### ***Вимоги до виконання практичної роботи***

З *git-flow* можна працювати в консолі –

[https://danielkummer.github.io/git-flowcheatsheet/index.ru\\_RU.html](https://danielkummer.github.io/git-flowcheatsheet/index.ru_RU.html).

Однак, набагато зручніше працювати в графічній оболонці, тому в даній практичній роботі буде використовуватися *SourceTree*. На даний момент (2018) це найпопулярніша і зручна графічна оболонка для *Git* и *Mercurial*.

### ***Порядок виконання практичної роботи***

1. Створення нового репозиторію.
2. Створення першого *commit*'у.
3. Створення гілки *develop*.
4. Створення гілки *release*.
5. Створення гілки *hotfix*.
6. Перенесення на віддалений репозиторій.

#### ***1. Створення нового репозиторію***

Для створення нового репозиторію необхідно виконати такі дії:

- натиснути ***New Repository*** і вибати ***Create Local Repository*** (рис. 10.5);

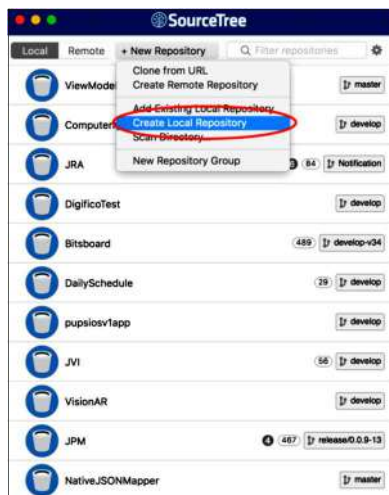


Рисунок 10.5 – Вибір команди *Create Local Repository*

– створити порожню папку для репозиторію, натиснути **Create** (рис. 10.6);

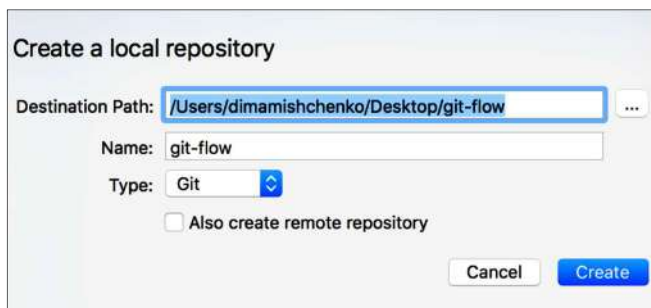


Рисунок 10.6 – Створення порожньої папки

В результаті отримали порожній репозиторій (рис. 10.7)

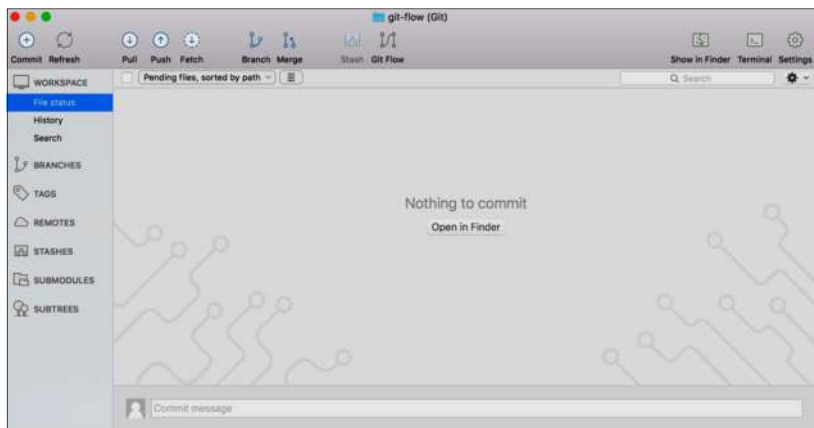


Рисунок 10.7 – Створення порожнього репозиторію

## 2. Створення першого commit'у

На початку проекту часто роблять якийсь «*init commit*», тим самим створюючи відправну точку для роботи. Він є першим *commit*'ом в гілці **master**. Для цього:

– створити **README.md** файл і записати в нього будь-яку інформацію (рис. 10.8);

```
MacBook-Pro-Dima:~ dimamishchenko$ cd /Users/dimamishchenko/Desktop/git-flow
MacBook-Pro-Dima:git-flow dimamishchenko$ touch README.md
MacBook-Pro-Dima:git-flow dimamishchenko$ echo "Description of project." >> README.md
```

Рисунок 10.8 – Створення **README.md** файлу

– написати назву *commit*'ів (рис. 10.9);

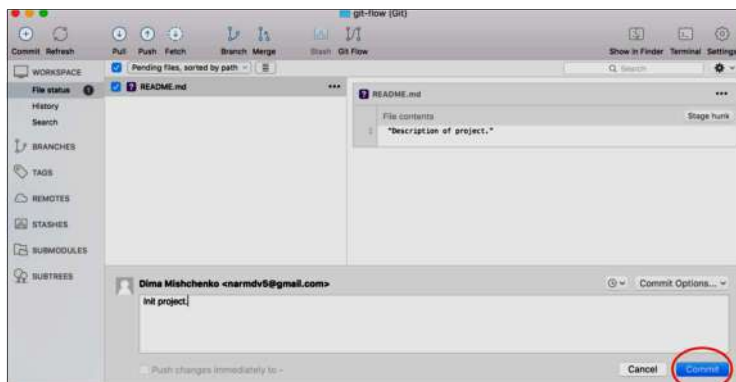


Рисунок 10.9 – Написання назви *commit*'у

– натиснути **Git Flow** кнопку зверху в панелі інструментів. Якщо її немає, то натиснути правою кнопкою миші на панель інструментів, і вибрати **Customize Toolbar...** (рис. 10.10);

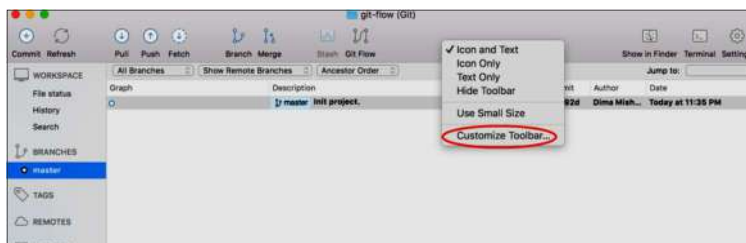


Рисунок 10.10 – Вибір пункту *Customize Toolbar...*

– перетягнути кнопку **Git Flow** на панель інструментів і натиснути **Done** (рис. 10.11);

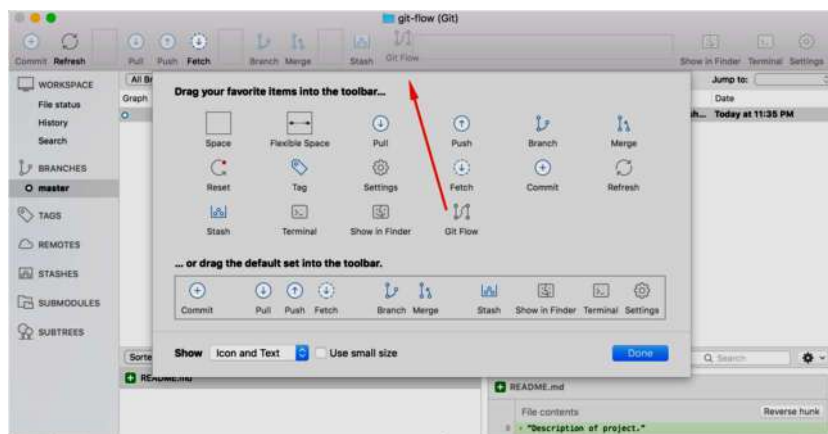


Рисунок 10.11 – Кнопка *Git Flow* на панелі інструментів

– натиснути на **Git Flow** і кнопку **OK** (рис. 10.12). *SourceTree* створи-  
вши гілку **develop**, в якій тепер будемо працювати.

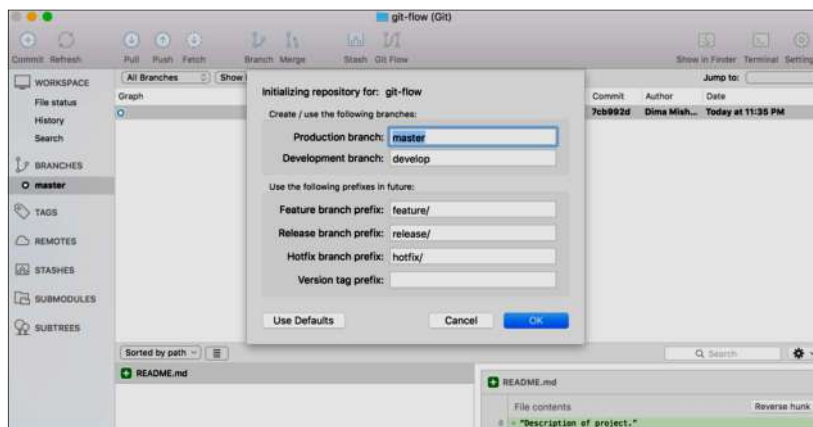


Рисунок 10.12 – Натискання кнопки *Git Flow*



### 3. Створення гілки *develop*

Для прикладу була взята частина коду <http://coding.dp.ua/jquery/1335-prostoy-tekstovyyiy-slayder.html> і необхідно виконати такі дії:

– в корені папки репозиторію створити папку **Source**, в ній створити файл **index.html** (рис. 10.13);

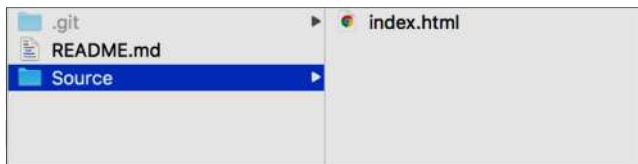


Рисунок 10.13 – Створення файлу *index.html*

– поруч створити папку **css**, в ній – файл **style.css** (рис. 10.14);



Рисунок 10.14 – Створення файлу *style.css*

– створити папку **js**, а в ній файл **script.js** (рис. 10.15);



Рисунок 10.15 – Створення файлу *script.js*

– в файл **index.html** вставити наступний код:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://code.jquery.com/jquery-1.9.1.min.js"
      integrity="sha256-
        wS9gmOZBqsqWxgIVgA8Y9WcQOa7PgSIX+rPA0VL2rbQ="
```

```

        crossorigin="anonymous">
    </script>
    <script type="text/javascript" src="js/script.js">
    </script>
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
    <h1>Hello Git-Flow!</h1>
    <span id="heading1">I</span>
    <span id="heading2">Love</span>
    <span id="heading3">Git Flow</span>
    <span id="heading4">♥ </span>
</body>
</html>

```

– в файл ***style.css*** вставить наступный код:

```

#heading1, #heading2, #heading3, #heading4{
    position: absolute;
    left: 0px;
    opacity: 0;
}

```

– в файл ***script.js*** вставить наступный код:

```

var heading_cur=0;
function showHeading(){
    $('#heading'+(heading_cur+1)).css({opacity: 0}).
    animate({opacity:1.0,left: "50px"}, 500);
    setTimeout(hideHeading, 1000);
}
function hideHeading(){
    $('#heading'+(heading_cur+1)).css({opacity: 1}).
    animate({opacity:0,left: "-50px"},
    500,function(){showHeading();});
    heading_cur=(heading_cur+1)%4;
}
$(document).ready(function() {
    showHeading();}

```

- зберегти всі файли;
- відкрити файл *index.html* (рис. 10.16) і перевірити його працездатність (анімація буде працювати, якщо є підключення до інтернету);



Рисунок 10.16 – Відкриття файлу *index.html*

- *commit*'ути зміни (рис. 10.17);

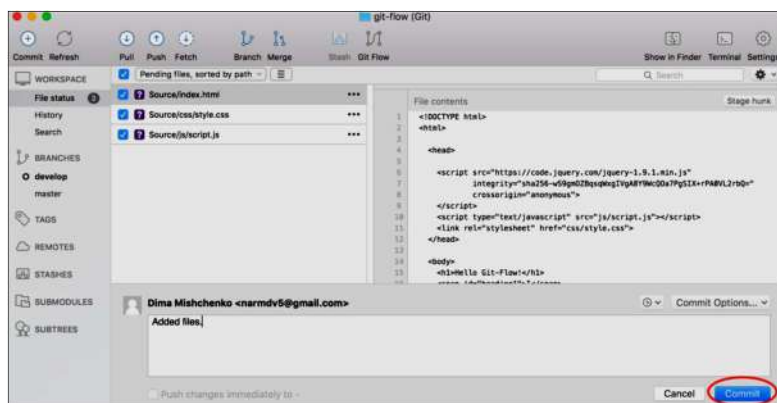


Рисунок 10.17 – *Commit* змін

- зробити якісь зміни (наприклад, змінити стилі). Для цього натиснути на кнопку *Git Flow* і вибрати пункт *Start a New Feature* (рис. 10.18);

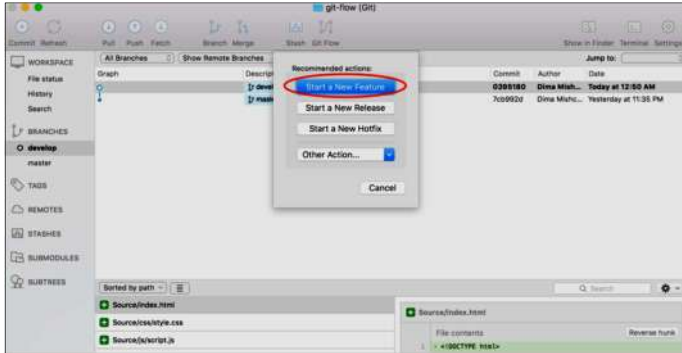


Рисунок 10.18 – Вибір пункту *Start a New Feature*

– назвати нову версію ***update-style*** и натиснути **OK**. Далі додати якихось рандомних стилів. Змінити файл ***style.css*** наступним чином:

```
#heading1, #heading2, #heading3, #heading4{
    position: absolute;
    left: 0px;
    opacity: 0;
    font-size: 92px;
    letter-spacing: .15em;
    text-shadow:
        1px -1px 0 #8f8e8d,
        -1px 2px 1px #949392,
        -2px 4px 1px #999897,
        -3px 6px 1px #9e9c9c,
        -4px 8px 1px #a3a1a1,
        -5px 10px 1px #a8a6a6,
        -6px 12px 1px #adabab,
        -7px 14px 1px #b2b1b0,
        -8px 16px 1px #b7b6b5,
        -9px 18px 1px #bcbbba,
        -10px 20px 1px #c1bfbf,
        -11px 22px 1px #c6c4c4,
        -12px 24px 1px #cbc9c8,
        -13px 26px 1px #cfdcdcd,
        -14px 28px 1px #d4d2d1,
        -15px 30px 1px #d8d6d5,
```

```

-16px 32px 1px #dbdad9,
-17px 34px 1px #dfdddc,
-18px 36px 1px #e2e0df;
}
#heading1 {
    color: red;
}
#heading2 {
    color: green;
}
#heading3 {
    color: blue;
}

```

– *commit*'ути зміни ( рис. 10.19);

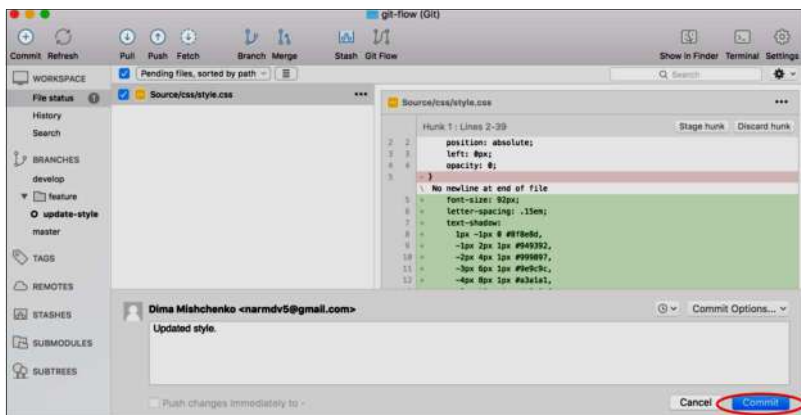


Рисунок 10.19 – *Commit* змін

– закінчити внесення змін. Для цього натиснути на **Git Flow** и вибрати **Finish Current** та **OK** (рис. 10.20).

Таким чином закінчили внесення змін і виконали *Commit* в гілку *develop*.

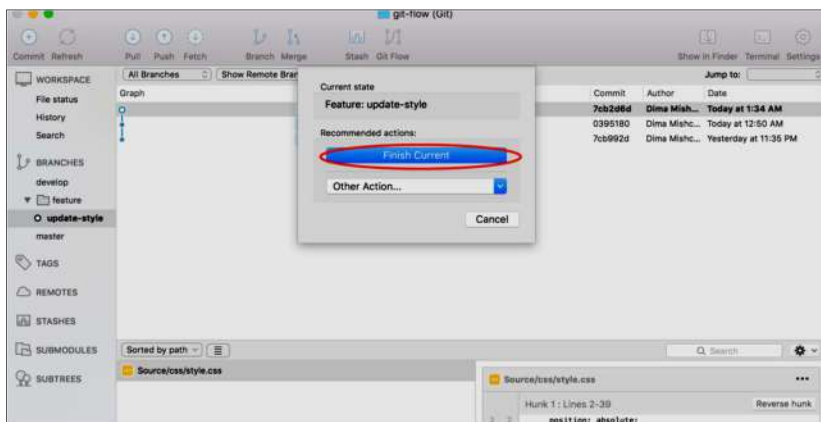


Рисунок 10.20 – Закінчення внесення змін

#### 4. Створення гілки *release*

Для того, щоб створити гілку *release* необхідно виконати такі дії:

- натиснути на **Git Flow** та вибрати **Start a New Release**, написати версію 1.0.0 і натиснути **OK** (рис. 10.21);

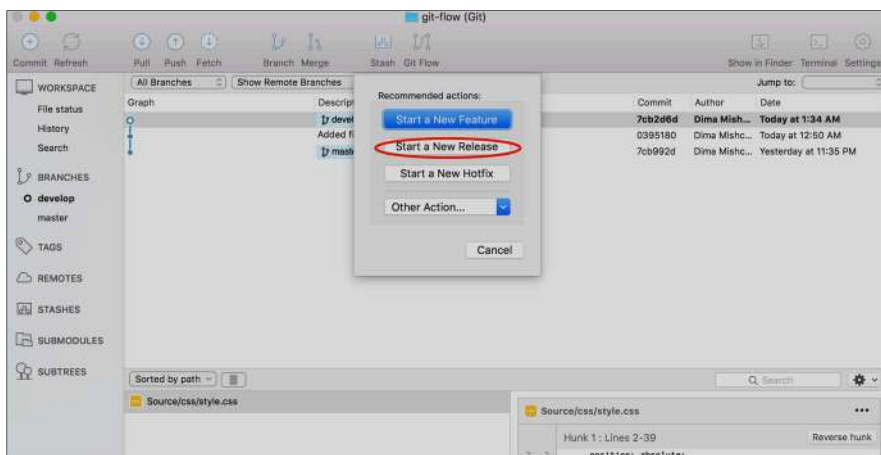


Рисунок 10.21 – Вибір пункту меню *Start a New Release*

– зробити будь-який *Commit*, який буде останнім в цій версії і на якому в *Git* буде стояти тег версії, щоб в майбутньому легко можна було знайти *commit* з версією і подивитися, що там було.

Для цього у файлі **README.rm** можна написати поточну версію. Додати в нього рядок **Version:1.0.0** та виконати *commit* (рис. 10.22);

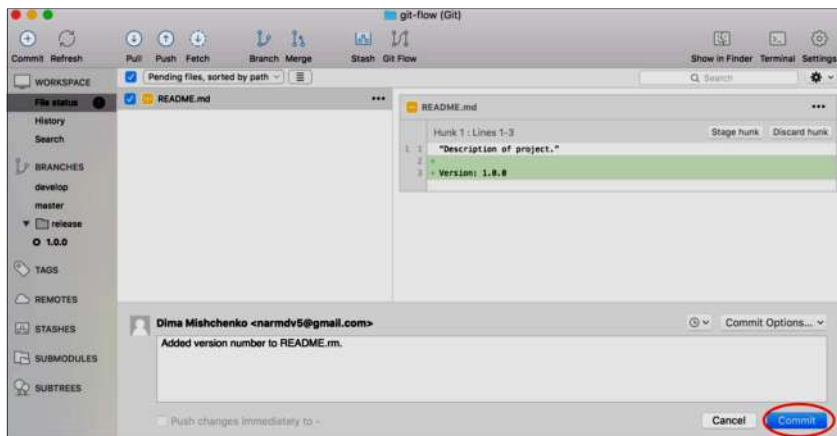


Рисунок 10.22 – *Commit* змін

– закінчити реліз. Для цього натиснути на **Git Flow** і вибрати **Finish Current**, написати тег 1.0.0 і натиснути кнопку **OK**.

В результаті виконали злиття цієї гілки, **develop** і **master** і був поставлений тег.

### 5. Створення гілки *hotfix*

Для того, щоб щось виправити в поточному релізі, потрібно просто виправити файл **README.rm**. Для цього:

– натиснути на **Git Flow** і вибрати **Start a New Hotfix**. Написати версію 1.0.1 і натиснути **OK** (рис. 10.23);

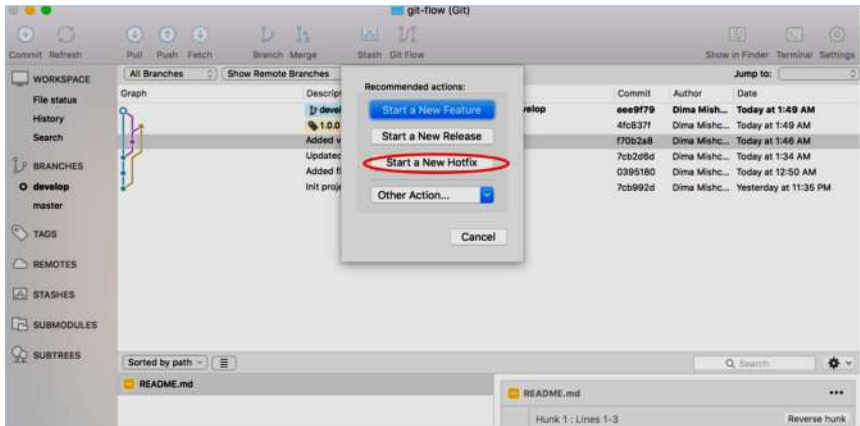


Рисунок 10.23 – Вибір пункту *Start a New Hotfix*

– редагувати файл **README.rm**, записати що-небудь і змінити версію: *Project to practice Git-Flow. Version: 1.0.1*. Далі зберегти і виконати *commit* (рис. 10.24);

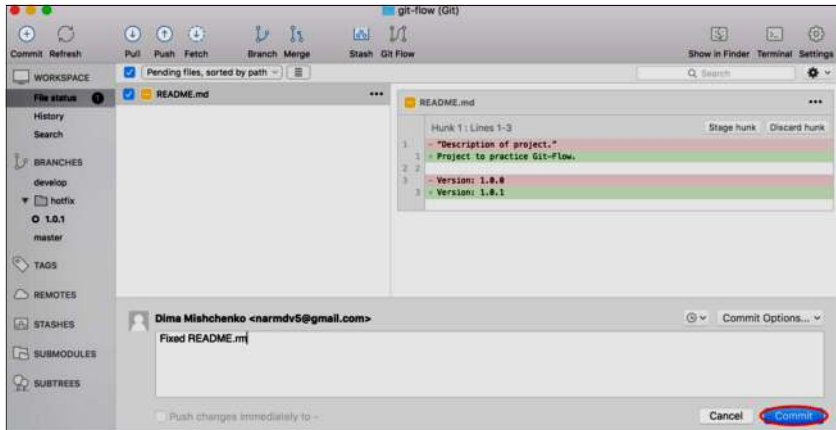


Рисунок 10.24 – *Commit* змін

– закінчити *hotfix*, натискаючи на **Git Flow** і вибираючи **Finish Current**. Потім записати тег 1.0.1 і натиснути **OK**. В результаті гілка злита в *develop* і *master* і був поставлений тег (рис. 10.25);



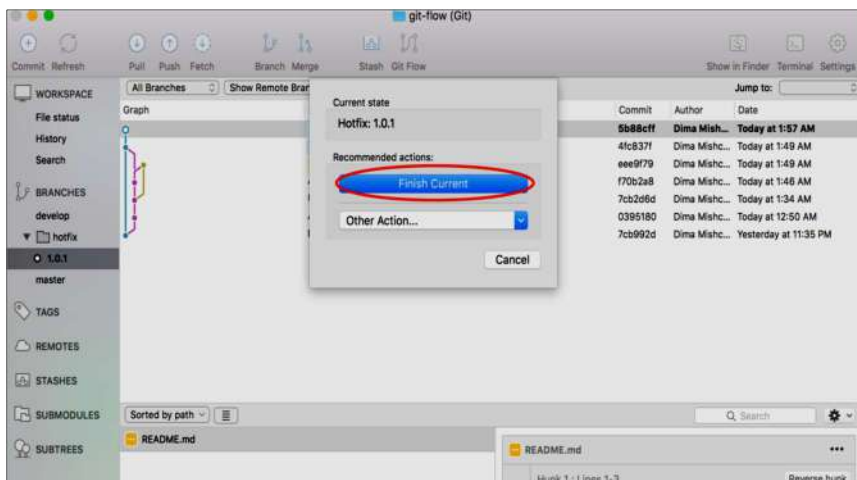


Рисунок 10.25 – Завершення *hotfix*

– було отримано результат (рис. 10.26).

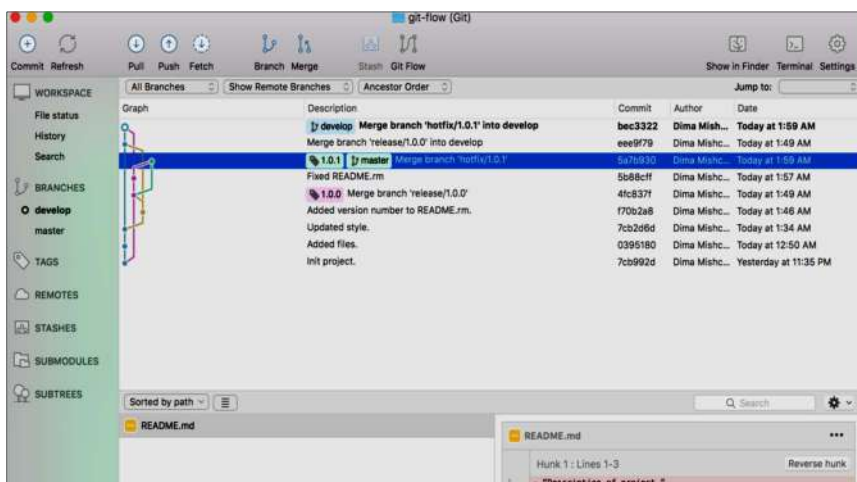


Рисунок 10.26 – Результат виконаної роботи

## 6. Перенесення на віддалений репозиторій

Для того, щоб перенести локальний репозиторій на віддалений (на *GitHub*) необхідно виконати такі дії:

– прив’язати *GitHub* аккаунт до *SourceTree*. Для цього зайти в налаштування або натиснути на іконку налаштувань в списку репозиторіїв і вибрати пункт *Accounts* (рис. 10.27);



Рисунок 10.27 – Вибір пункту *Accounts*

– натиснути *Add* і додати аккаунт (рис. 10.28). Інтерфейс в різних версіях *SourceTree* може відрізнятись. Документація щодо прив’язки аккаунта – <https://confluence.atlassian.com/get-started-with-sourcetree/connect-your-bitbucketor-github-account-847359096.html>;

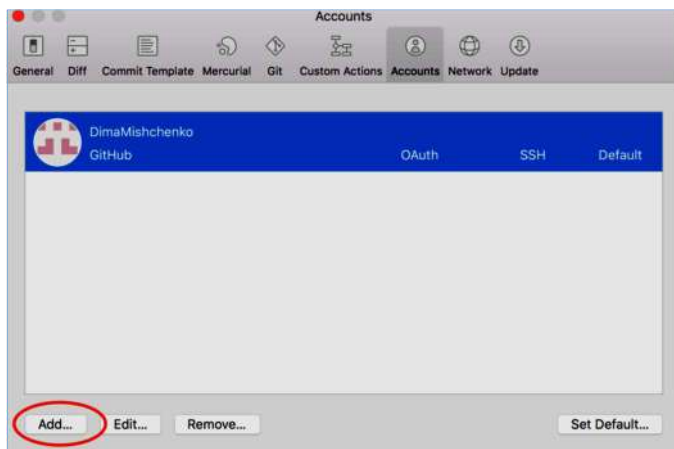


Рисунок 10.27 – Додавання акаунту

– натиснути правою клавiшею миші на потрібний репозиторiй в списку і вибрати ***Publish to Remote...*** (рис. 10.28);



Рисунок 10.28 – Вибір *Publish to Remote...*

– вибрати аккаунт, написати ім'я та опис, прибрати прапорець ***This is a private repository*** (тому що приватні репозиторії на *GitHub* – платні) і натиснути ***Create*** (рис. 10.29);

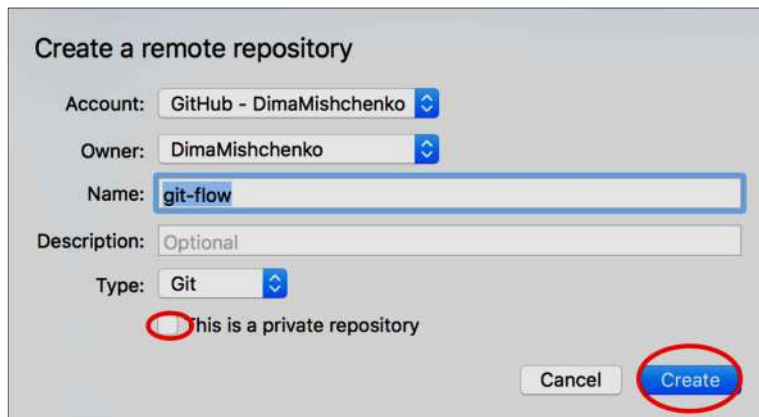


Рисунок 10.29 – Створення репозиторію

– після того, як *SourceTree* створить репозиторій, вибрати все гілки і виконати ***Push*** (рис. 10.30);

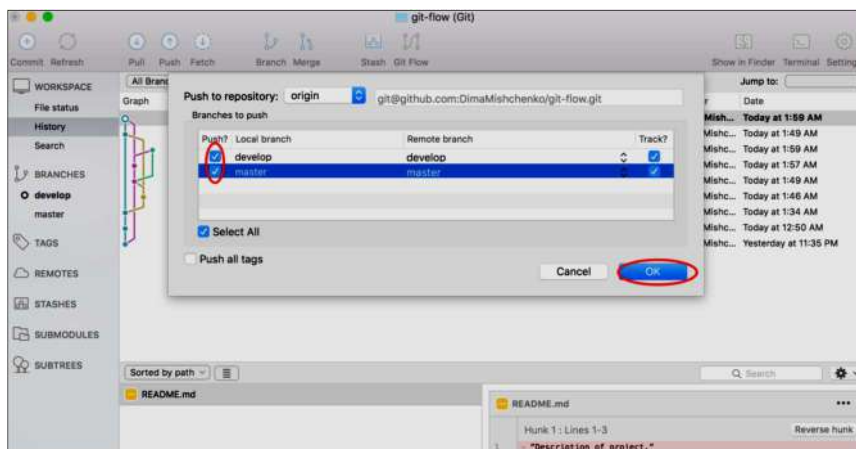


Рисунок 10.30 – Вибір гілок

– зайти на *GitHub* і перевірити, що всі зміни збережені (рис. 10.31).

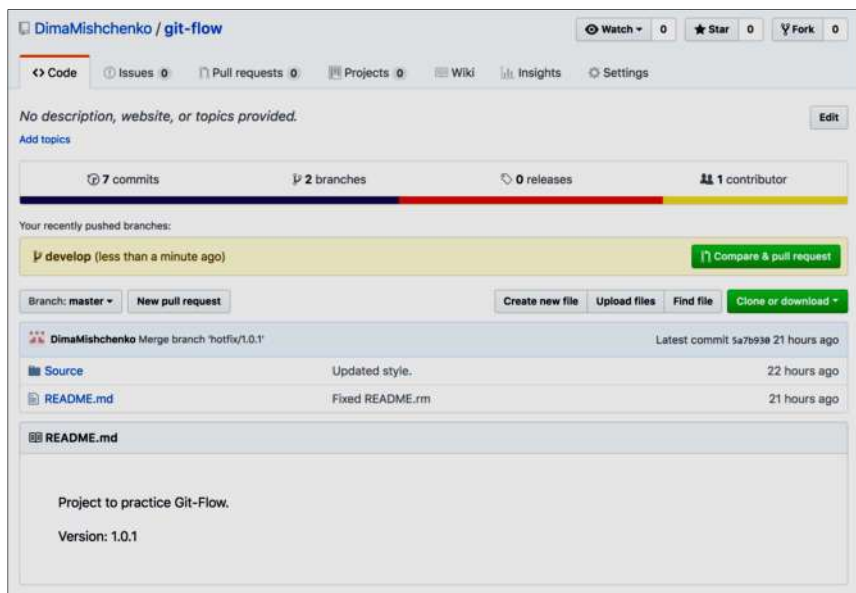


Рисунок 10.30 – Завершення роботи

### ***Висновок***

У ході роботи познайомилися з паттерном роботи з гілками у *git* та освоїли ще один інструмент роботи з *git* – *SourceTree*.

У звіті до практичної роботи не забудьте вказати посилання на віддалений репозиторій, який був створений у ході роботи.

### ***Контрольні запитання***

1. Скільки типів гілок у *git-flow* існує?
2. Для чого призначена гілка *master*?
3. Для чого призначена гілка *develop*?
4. У чому різниця між гілками *feature* та *hotfix*?
5. Для чого застосовується команда ***Start a New Release***?
6. Для чого застосовується команда ***Publish to Remote...***?

## РОЗРАХУНКОВО-ГРАФІЧНЕ ЗАВДАННЯ

### **Вимоги до виконання розрахунково-графічного завдання**

Написати простий калькулятор, розмістити в репозиторії *GitHub* в **Project\_Name** – основний каталог (*Name* – фамілія студенту), що містить:

**img** – каталог, що містить графічні файли;

**scc** – каталог, що містить файли стилів;

**src** – каталог, що містить файли скриптів;

**report** – каталог, що містить файл звіту;

**index.html** – основний файл *html*-сторінки;

**readme.md** – службовий файл.

Репозиторій повинен містити гілки **master**, **develop**, **feature**.

Гілка **master** повинна містити вихідний стан проєкту.

Гілка **develop** створюється з гілки **master** (операція *checkout*).

Гілка **feature** створюється з гілки **develop** (операція *checkout*) і містить **не менше п'яти змін** («commit»), з **обов'язковим коментуванням внесених змін**.

Після цього створюється **PullRequest** (з описом деталей), виконується злиття **feature** и **develop** (операція *merge*) і видалється гілка **feature**.

Після цього створюється **PullRequest**, виконується злиття **feature**, **develop** і **master** (операція *merge*) і гілці **master** присвоюється **тег v.1.0**.

Файл звіту повинен містити наступні пункти.

1. Завдання.
2. Опис ходу виконання з ілюстраціями (*print screen*).
3. Опис структури репозиторію із зазначенням імен файлів і описом їх вмісту.
4. Приклад виконання програми.
5. Посилання на діючий репозиторій на *GitHub*.

### ***Порядок виконання розрахунково-графічного завдання***

В якості основи можна використовувати наведений нижче приклад. Для цього потрібно «рознести» по різних файлах відповідні структурні елементи (скрипт, стилі, *html*-сторінка). Крім того, необхідно додати файли з зображенням.

Приклад калькулятора, написаного у вигляді *html*-сторінки:

```
<html>
<head>
<meta charset="utf-8" />
<title>Calculator</title>
<style type="text/css">
    #calculator * {font-size: 16px;}
    #calculator table {border: solid 3px silver; border-
spacing: 3px; background-color: #EEE; }
    #calculator table td {border-spacing: 3px;}
    input.display {width: 166px; text-align: right;}
    td.buttons {border-top: solid 1px silver;}
    input[type= button] {width: 40px; height: 30px;}
</style>
</head>
<!--http://www.4stud.info/web-programming/simple-js-
calculator.html -->
<body>
<form name="calc" id="calculator">
    <table>
    <tr>
    <td>
        <input type="text" name="input" size="16"
class="display">
    </td>
    </tr>
    <tr>
    <td class="buttons">
        <input type="button" name="one" value="1"
OnClick="calc.input.value += '1'">
        <input type="button" name="two" value="2"
OnClick="calc.input.value += '2'">
```

```

        <input type="button" name="three" value="3"
OnClick="calc.input.value += '3'">
        <input type="button" name="add" value="+"
OnClick="calc.input.value += '+'">
        <br>
        <input type="button" name="four" value="4"
OnClick="calc.input.value += '4'">
        <input type="button" name="five" value="5"
OnClick="calc.input.value += '5'">
        <input type="button" name="six" value="6"
OnClick="calc.input.value += '6'">
        <input type="button" name="sub" value="-"
OnClick="calc.input.value += '-'">
        <br>
        <input type="button" name="seven" value="7"
OnClick="calc.input.value += '7'">
        <input type="button" name="eight" value="8"
OnClick="calc.input.value += '8'">
        <input type="button" name="nine" value="9"
OnClick="calc.input.value += '9'">
        <input type="button" name="mul" value="x"
OnClick="calc.input.value += '*'">
        <br>
        <input type="button" name="clear" value="c"
OnClick="calc.input.value = ''">
        <input type="button" name="zero" value="0"
OnClick="calc.input.value += '0'">
        <input type="button" name="doit" value="="
OnClick="calc.input.value = eval(calc.input.value)">
        <input type="button" name="div" value="/"
OnClick="calc.input.value += '/'">
    </td>
</tr>
</table>
</form>
</body>
</html>

```



## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. <https://www.ibm.com/developerworks/ru/library/l-git-subversion-1/index.html> - Git для пользователей Subversion.
2. <https://try.github.io/> - Resources to learn Git.
3. <https://guides.github.com/activities/hello-world/> - The Hello World project.
4. <https://githowto.com/ru> - Git How To.
5. <http://gitimmersion.com/> - Git Immersion. A guided tour.
6. <https://www.eclipse.org/documentation/> - Eclipse Documentation.
7. <https://www.jetbrains.com/idea/documentation/> - IntelliJ IDEA. Learn and support.
8. <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> - Gitflow Workflow.
9. <https://nvie.com/posts/a-successful-git-branching-model/> - A successful Git branching model

## ЗМІСТ

Вступ	3
.....	4
Практична робота 1 Знайомство з системою контролю версій Git.....	
Практична робота 2 Початок роботи з <i>GitHub</i> .....	12
Практична робота 3 Робота з <i>git</i> та <i>GitHub</i> .....	20
Практична робота 4 Графічний інтерфейс <i>Git</i> .....	31
Практична робота 5 Графічний клієнт для використання <i>Git</i> – <i>TortoiseGit</i> .....	41
Практична робота 6 Робота з <i>GitHub</i> та інтегрованим середовищем розробки додатків <i>IntelliJ Idea</i> .....	51
Практична робота 7 Робота с системою контролю версій <i>GIT</i> в середовищі розробки <i>Eclipse</i> .....	63
Практична робота 8 Робота з конфліктами. Створення <i>SSH</i> ключа.....	72
Практична робота 9 Системи управління версіями: <i>TortoiseSVN</i> .....	82
Практична робота 10 Методологія розробки проєктів з використанням <i>git</i> – <i>Git-flow</i> .....	103
Розрахунково-графічне завдання .....	125
Список використаних джерел .....	128

Навчальне видання

ПУГАЧОВ Роман Володимирович  
ЛЮБЧЕНКО Наталія Юріївна  
СОБОЛЬ Максим Олегович

## **СИСТЕМИ КОНТРОЛЮ ВЕРСІЯМИ**

Навчально-методичний посібник  
для студентів комп'ютерних спеціальностей  
вищих навчальних закладів

Роботу рекомендував до видання проф. Заполовський М.Й.  
Відповідальний за випуск проф. Солощук М.М.

Дизайн обкладинки Пугачов Р.В.

В авторській редакції

План 2018 р., поз. 23

Підписано до друку 25.02.19. Формат 60х84 1/16. Папір офсет.

Друк – ризографія. Гарнітура Times New Roman. Ум. друк. арк.

Наклад 20 прим. Зам №.

Ціна договірна

---

Видавничий центр НТУ "ХПІ" 61002, Харків, вул. Кирпичова, 2.

Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.

---