МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий Інститут телекомунікаційних систем

Затверджено
вченою радою НН ІТС
протокол № _4___
від “ 24” квітня 2023 року

Технології побудови web-орієнтованих систем

Методичні вказівки
до виконання комп‘ютерного практикуму
для студентів спеціальності
"172 - Електронні комунікації та радіотехніка"

Затверджено Вченою радою НН ІТС КПІ ім. Ігоря Сікорського

Київ – 2023

Технології побудови web-орієнтованих систем [Текст]: метод. вказівки до виконання комп‘ютерного практикуму для студентів спеціальності "Електронні комунікації та радіотехніка " / Уклад.: Суліма С.В., Скулиш М.А.. – К.: КПІ ім. Ігоря Сікорського, 2023. – 54 с.

Гриф надано Вченою радою НН ІТС КПІ ім. Ігоря Сікорського
(Протокол № 4 від 24.04.2023)

Навчальне видання

Технології побудови web-орієнтованих систем

Методичні вказівки
до виконання комп‘ютерного практикуму
для студентів спеціальності
"Електронні комунікації та радіотехніка"

Укладачі: *Суліма Світлана Валеріївна*, канд. техн. наук.
Скулиш Марія Анатоліївна, докт. техн. наук, проф.

Відповідальний редактор: *С.В. Суліма*, канд. техн. наук

Рецензенти: *С.О. Кравчук*, докт. техн. наук, проф.

За редакцією укладачів
Надруковано з оригінал-макета замовника

Зміст

Вступ 4
Загальні методичні вказівки 5
Вимоги до оформлення звіту з лабораторної роботи 6
Інструкція з техніки безпеки 7
Лабораторна робота №1 8
Знайомство з системою контролю версій git 8
I. Підготовка до лабораторної роботи 8
II. Теоретичні відомості 8
1. Про контроль версій 8
2. Системи контролю версій на прикладі git 10
III. Завдання на лабораторну роботу 13
IV. Контрольні питання 15
Лабораторна робота №2 16
Початок роботи з GITHUB 16
I. Підготовка до лабораторної роботи 16
II. Теоретичні відомості 16
III. Завдання на лабораторну роботу 16
IV. Контрольні питання 20
Лабораторна робота №3 Робота з GIT та GITHUB 21
I. Підготовка до лабораторної роботи 21
II. Теоретичні відомості 21
III. Завдання на лабораторну роботу 21
IV. Контрольні питання 27
Лабораторна робота №4 28
Використання сучасних гнучких методологій розроблення інформаційних систем 28
I. Підготовка до лабораторної роботи 28
II. Теоретичні відомості 28
III. Завдання на лабораторну роботу 32
IV. Контрольні питання 32
Лабораторна робота №5 33
Управління ризиками 33
I. Підготовка до лабораторної роботи 33
II. Теоретичні відомості 33
..... 33
III. Завдання на лабораторну роботу 41
IV. Контрольні питання 41
Лабораторна робота №6 42
Тестування програмного забезпечення 42
I. Підготовка до лабораторної роботи 42
II. Теоретичні відомості 42
1. Вступ 42
2. Особливості вимог програмного забезпечення 44
III. Завдання на лабораторну роботу 51
IV. Контрольні питання 51
Перелік літератури 52

Вступ

Сучасні вимоги до інженерів з телекомунікацій ґрунтуються на глибоких науково-технічних знаннях та вміннях розв’язувати практичні задачі, в тому числі задачі щодо створення та роботи з веб-додатками, а саме методологій розробки таких прикладних програм, написання програмного коду програмного продукту, роботи з засобами управління проєктами тощо.

Метою методичних вказівок є надання допомоги студентам в отриманні практичних навичок роботи з системами керування проєктами, контролю версій та тестування. Виконання лабораторних робіт забезпечить закріплення теоретичного та лекційного матеріалу.

Під час виконання лабораторних робіт студенти отримають навички щодо:

- підходів до проектування веб-додактів;
- організації процесу командної розробки програмного продукту;
- тестування веб-додатку на відповідність вимогам замовника.

Загальні методичні вказівки

1. У кожній лабораторній роботі визначено: мету роботи, рекомендації з підготовки до роботи, програму і порядок її виконання.

2. Напередодні кожної лабораторної роботи необхідно:

- вивчити теоретичний матеріал зазначений у розділі "Підготовка до лабораторної роботи";
- усвідомити мету, зміст і порядок виконання;
- у лабораторії перевірити наявне лабораторне устаткування.

3. До виконання лабораторної роботи допускаються тільки підготовлені студенти після тестування (усного чи письмового), що проводиться викладачем перед початком виконання роботи.

Лабораторні роботи виконуються самостійно кожним студентом.

Звіт з роботи акуратно оформлюється і подається під час захисту лабораторної роботи. Схеми, зображені в звіті, мають відповідати вимогам стандартів.

4. Студенти, відсутні на заняттях, виконують роботу у час, погоджений з викладачем і інженером лабораторії після тестування.

5. Перед початком робіт кожному студенту необхідно вивчити правила техніки безпеки, здати залік, за що розписатися в журналі.

Вимоги до оформлення звіту з лабораторної роботи

Звіт оформлюється на аркушах формату А4 і повинен містити:

1. Титульну сторінку з номером та назвою лабораторної роботи, прізвищем студента, номером групи.
2. Особливості завдання.
3. Опис виконаного завдання.
4. Висновки по роботі.

Інструкція з техніки безпеки

1. Вимоги з техніки безпеки перед початком роботи
   1.1 Провести огляд з зовні електророзеток, шнурів, вилок підключення до мережі живлення та заземлення (занулення).
   1.2 Забороняється працювати на несправному устаткуванні.
   1.3 За необхідності отримати додаткове устаткування у викладача та перевірити його справність.

2. Вимоги з техніки безпеки під час виконання робіт
   2.1 Необхідно виконувати лише ту роботу, з якої був проведений інструктаж, забороняється передоручати свою роботу іншим особам.
   2.2 Забороняється:
   - експлуатація кабелів та проводів з пошкодженою ізоляцією або такою, що втратила захисні властивості за час експлуатації;
   - залишати під напругою кабелі та проводи з неізольованими провідниками;
   - застосовувати саморобні подовжувачі, що не відповідають вимогам ПВЕ для переносних електропровідників;
   - користуватися пошкодженими розетками, розгалужувальними та з'єднувальними коробками, вимикачами та іншими електровиробами, а також лампами, скло яких має слід затемнення або випинання;
   - використовувати електроустаткування та прилади в умовах, що не відповідають інструкції з експлуатації підприємств-виробників;
   - залишати пристрої, що працюють без нагляду на тривалий час;
   - переносити пристрої, що підключені до електромережі;
   - забороняється самостійно ремонтувати апаратуру;
   - класти будь-які предмети на апаратуру комп'ютера, напої на клавіатуру або поруч з нею - це може вивести їх з ладу.

3. Вимоги з техніки безпеки після закінчення роботи
   3.1. Зберегти необхідні файли на жорсткий диск комп'ютера або на переносний носій.
   3.2 Виключити комп'ютер.
   3.3. Виключити додаткове устаткування та віддати його викладачу.
   3.4 Прибрати робоче місце.

Лабораторна робота №1
Знайомство з системою контролю версій git

I. Підготовка до лабораторної роботи

Ознайомитись з теоретичною та практичною частиною наведеною нижче для виконання даної лабораторної роботи

II. Теоретичні відомості

1. Про контроль версій

Що таке контроль версій, та навіщо він вам потрібен? Контроль версій - це система, котра веде протокол змін в файлі або кількох файлах увесь час для того, щоб ви мали можливість викликати певні версії пізніше.

Якщо ви графічний або веб-дизайнер і ви хочете зберегти кожну версію малюнку або шару, то використання системи контролю версій (надалі - СКВ) буде дуже розсудливим в цьому випадку. СКВ надає можливість робити наступне: повертати файли до попереднього вигляду, повертати цілий проект до попереднього вигляду, переглядати зміни, зроблені за увесь час, дивитися, хто востаннє змінював щось, що могло призвести до проблеми, хто вирішив задачу та коли й багато іншого. Використання СКВ також означає що, якщо ви щось зламаєте або загубите файли, ви зможете легко усе відновити. До того ж, ви все це отримуєте за дуже маленькі накладні витрати.

Якщо декілька розробників працюють над один й тим самим додатком, іноді одним і тим самим файлом або навіть у суміжних строках, Git дозволяє фіксувати зміни паралельно за допомогою декілька бранчей (branch), а потім зливати усі зміні в один бранч, наприклад основну кодову базу. Альтернативою без контролю версій було би збереження змін у кожного розробника локально, а потім змішування змін вручну на девайсі одного із них. Окремою перевагою є можливість порівняти зміни за допомогою утиліти diff (або git diff), що дозволяє бачити лише зміни без порівняння усіх файлів.

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 си- нхронізації 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.

Локальна система контролю версій

Багато хто в якості методу контролю версій вибирає просте копіювання файлів в іншу директорію. Такий підхід дуже розповсюджений, тому що він такий простий, але він, як правило, й призводить до повного краху. Це так легко забути, що ви не в тій директорії, та випадково записати не той файл, або перезаписати не ті файли, котрі хотіли. Для вирішення цієї задачі програмісти вже давно

розробили локальні СКВ, котрі мають просту базу даних для збереження усіх змін файлів, котрі знаходяться під контролем системи (див. рис. 1.1).

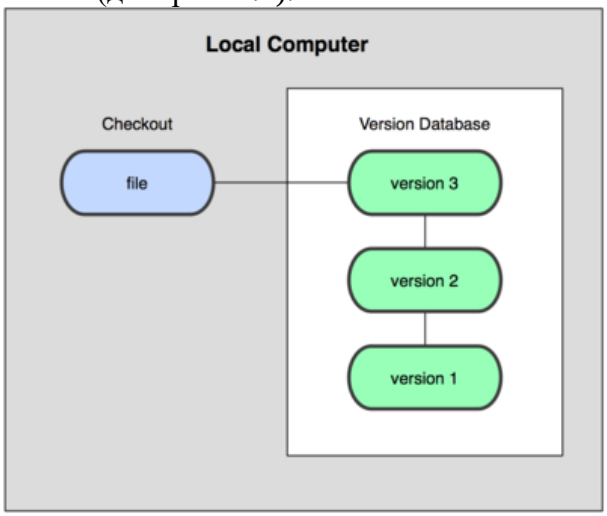


Рисунок 1.1 — Схема локальної СКВ

Одним з найбільш популярних інструментів СКВ є система rcs, котра досі поширена на багатьох ПК нині. Навіть популярна операційна система Mac OS X має утиліту rcs, котра встановлюється разом з Developer Tools. Робота цього інструменту базується на збереженні наборів патчів (це різниця між файлами) від одніє ревізії до іншої в спеціальному форматі на диску; це дає можливість потім відтворити вигляд будь-якого файлу в будь-який момент часу, накладаючи патчі.

Централізовані системи контролю версій

Наступна важлива проблема полягала в потребі співпрацювати з розробниками на інших машинах. Для вирішення цієї проблеми були розроблені централізовані системи контролю версій (ЦСКВ). Ці системи, такі як CVS, Subversion та Perforce, мають єдиний сервер, котрий містить усі версії файлів, а клієнти отримують копії файлів з цього центрального місця. Протягом багатьох років це було стандартом для СКВ. (див. рис. 1.2).

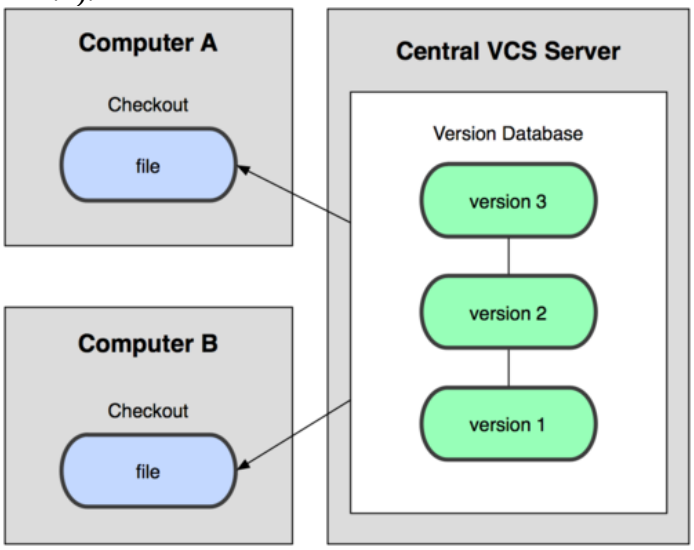


Рисунок 1.2 — Схема централізованого СКВ

Такі системи пропонують багато переваг, особливо над локальними СКВ. Для прикладу, кожен має постійний доступ до інформації: хто що зробив у проекті. Адміністратори мають добре розгалужений контроль над тим, хто що має може робити; і адміністрування ЦСКВ більш легке, ніж надання прав доступу до локальної бази данних кожному користувачу.

Однак, такий підхід також має деякі серйозні недоліки. Найбільш очевидний: централізований сервер є місце вразливості системи. Якщо цей сервер вийде з ладу на годину, то протягом цієї години

ніхто не зможе співпрацювати з іншими або зберегти зміни у файлах нової версії до будь-якого проекту, над котрим працює. Якщо жорсткий диск з централізованою базою даних вийде з ладу та не буде збережено резервної копії, ви втратите абсолютно усе - повну історію проекту, за виключенням тих окремих знімків, котрі збереглися на машинах користувачів. Локальні СКВ також схильні до тієї ж проблеми: коли повна історія проекту зберігається в одному місці, існує ризик втратити усе.

Розподілені системи контролю версій

І тут настає черга розподілених систем контролю версій (РСКВ). В РСКВ (таких як Git, Mercurial, Bazaar або Darcs) клієнт не лише отримує останній знімок файлів, а й повне дзеркало репозиторія. В цьому випадку, якщо будь-який сервер "помре" та інші системи, котрі працювали з ним, любий клієнтський репозиторій може бути використаний для відновлення його копії на сервері. Кожен знімок є насправді цілою копією усіх даних (див. рис. 1.3).

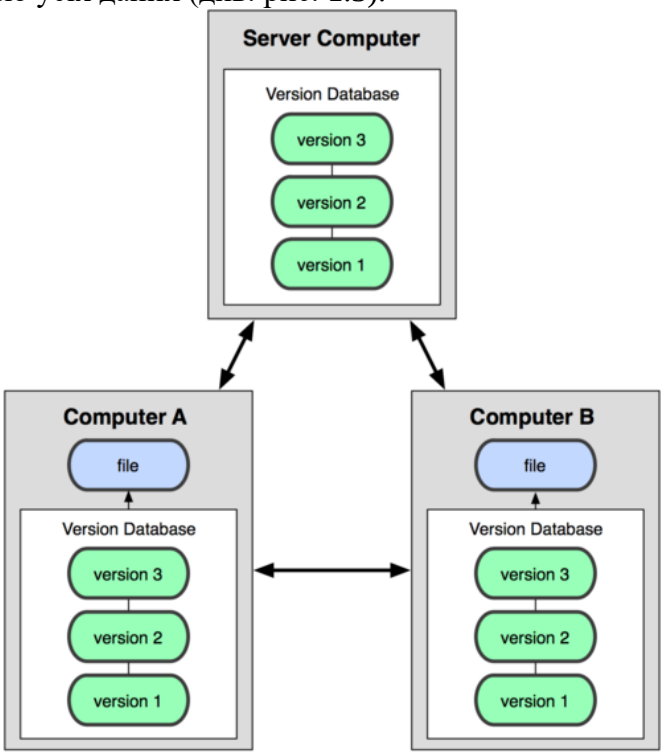


Рисунок 1.3 — Схема розподілених СКВ

Крім того, багато цих систем дуже добре надають доступ до декількох існуючих віддалених репозиторіїв, з котрими можна працювати, таким чином можна одночасно співпрацювати з різними групами розробників в різний спосіб над одним й тим же проектом. Це дає можливість використати декілька типів робочих проектів, що неможливо в ЦСКВ як в ієрархічній моделі.

2. Системи контролю версій на прикладі git

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

Структура

Git не є централізованим сервером як файлові сервера. Замість цього він є розподіленим (distributed), тобто він може зберігатися одночасно локально та на серверах сервісів таких як GitHub, BitBucket, GitLab тощо. Виглядає він як звичайна директорія з файлам, за виключенням наявності директорії “.git”, що зберігає зміни за допомогою різниць тобто diff-ів. Тобто якщо в історії змін є великий файл, то Git не зберігає його копії, а тільки різницю між його станом до й після зміни. Ця директорія називається репозиторій.

Після деяких змін стан файлів фіксується за допомогою комітів (commit). Коміт це контрольна точка у історії змін, наприклад коміт включає у себе зміну 3 файлів, створення 2 інших файлів, видалення одного файлу.

Для організації комітів використовуються бранчі та теги. Бранчі - це вказівники на певний коміт, що змінюється при додавання інших. Вони використовуються при розробці версії або певної фічі. Коли наближається час релізу, розробник створює тег, що на відміну від бранча не може бути зміненим. Теги звичайно мають маркування як і версії ПЗ.

Операції

Git дозволяє полегшити життя розробника за допомогою повністю або частково автоматичних операцій:

– операції commit/push/pull/fetch забезпечують додавання та синхронізацію змін між локальним та віддаленими репозиторіями;

– операції merge/rebase/cherry-pick надають можливість змішувати зміни між бранчами;

– операції status/log надають можливість слідкувати за змінами у проекті;

– операції checkout/revert/reset забезпечують відміну змін та навігацію між комітами, тегами або бранчами.

Додаткові функції Git сервісів

Крім звичайних команд, Git сервера такі як GitHub, GitLab тощо мають можливості:

– створення та зберігання релізів, тобто файлів для використання ПЗ (наприклад .exe файл для встановлення на Windows);

– створення issues, тобто тікетів на будь-яку проблему або нову фічу для ПЗ;

– створення pull request (merge request для GitLab), що дозволяє перед додаванням змін у основний бранч надати іншому розробнику можливість перевірити ваш код, запропонувати поліпшення тощо.

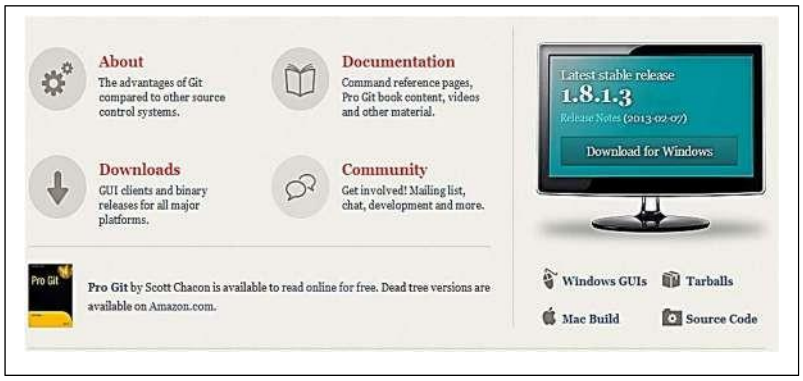
3. Початок роботи з Git

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. 1.4).

Рисунок 1.4 – Вибір версії Git



Інсталяція Git

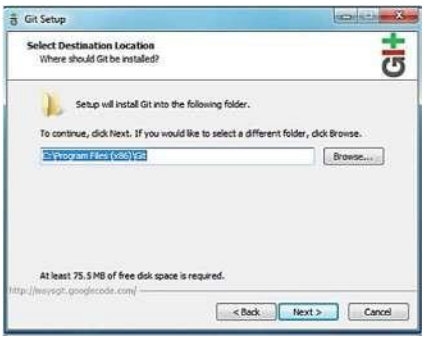
При інсталяції необхідно указати каталог для установки й деякі параметри (рис. 1.5 а – ж).


Git Setup
Welcome to the Git Setup Wizard
This will install Git version 1.7.9-preview20120201 on your computer.
It is recommended that you close all other applications before continuing.
Click Next to continue, or Cancel to exit Setup.
Next >
Cancel

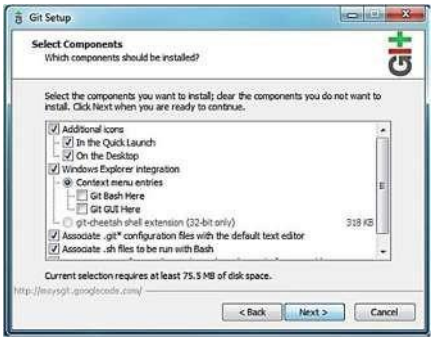
а


Git Setup
Information
Please read the following important information before continuing.
When you are ready to continue with Setup, click Next.
GNU General Public License
Version 2, June 1991
Preamble
The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public
< Back
Next >
Cancel

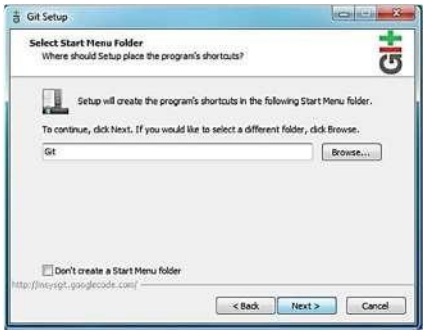
б


Git Setup
Select Destination Location
Where should Git be installed?
Setup will install Git into the following folder.
To continue, click Next. If you would like to select a different folder, click Browse.
C:\Program Files (x86)\Git
Browse...
At least 75.5 MB of free disk space is required.
< Back
Next >
Cancel

в


Git Setup
Select Components
Which components should be installed?
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.
Additional icons
In the Quick Launch
On the Desktop
Windows Explorer integration
Context menu entries
Git Bash Here
Git GUI Here
git-cheetah shell extension (32-bit only)
318 KB
Associate .git* configuration files with the default text editor
Associate .sh files to be run with Bash
Current selection requires at least 75.5 MB of disk space.
< Back
Next >
Cancel

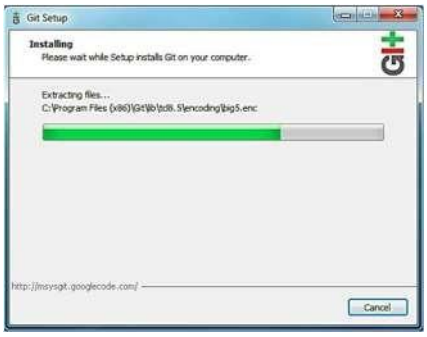
г


Git Setup
Select Start Menu Folder
Where should Setup place the program's shortcuts?
Setup will create the program's shortcuts in the following Start Menu folder.
To continue, click Next. If you would like to select a different folder, click Browse.
Git
Browse...
Don't create a Start Menu folder
< Back
Next >
Cancel

д


Git Setup
Adjusting your PATH environment
How would you like to use Git from the command line?
Use Git Bash only
This is the most conservative choice if you are concerned about the stability of your system. Your PATH will not be modified.
Run Git from the Windows Command Prompt
This option is considered safe and no conflicts with other tools are known. Only Git will be added to your PATH. Use this option if you want to use Git from a Cygwin Prompt (make sure to not have Cygwin's Git installed).
Run Git and included Unix tools from the Windows Command Prompt
Both Git and its accompanying Unix tools will be added to your PATH.
Warning: This will override Windows tools like find.exe and sort.exe. Select this option only if you understand the implications.
< Back
Next >
Cancel

е


Git Setup
Installing
Please wait while Setup installs Git on your computer.
Extracting files...
C:\Program Files (x86)\Git\lib\tcl8.5\encoding\big5.enc
Cancel

є

ж

Рисунок 1.5 – Вибір параметрів для установки *Git*

Налаштування Git

Після установки на робочому столі повинен з’явитися ярлик Git Bash. Після запуску системи контролю версій з’явиться консоль (рис. 1.6).

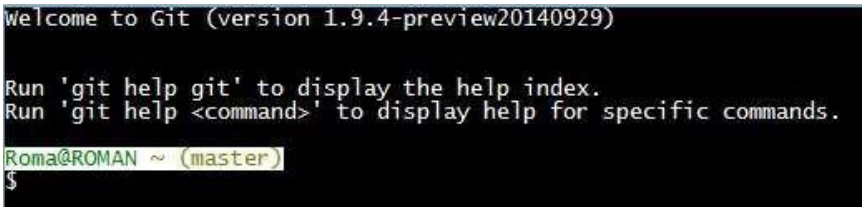


Рисунок 1.6 – Консоль системи контролю версій *Git*

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 перемикання 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 дуже популярні, тому в цих роботах також будуть використовуватися.

III. Завдання на лабораторну роботу

У ході роботи виконаємо найтиповіші 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.

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. 1.8).

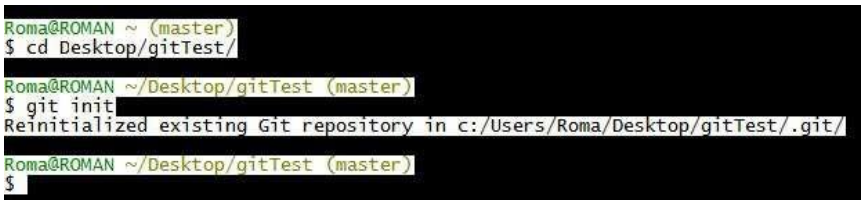


Рисунок 1.8 – Ініціалізація порожнього репозиторію

Додати файл у репозиторій

Створити у новій директорії порожній файл example.txt. Перейти уконсоль та ввести команду git status (рис. 1.9).

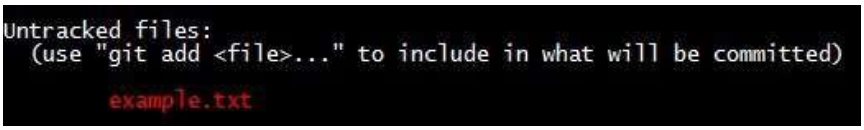


Рисунок 1.9 – Створення файлу example.txt

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. 1.10). за допомогою команди git commit -m 'Added file'

Рисунок 1.10 – Додавання файлу під версійний контроль

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).

Видалити файл з репозиторію (commit)

Командою git rm example.txt видаляється файл з репозиторію. Після цього виконуємо commit («комітимося») – git commit -m 'removed file' (рис. 1.11).

Рисунок 1.11 – Видалення файлу з репозиторію

Виконати checkout та створити гілку у репозиторії

Командою git checkout -b development переключаємося на нову гілку з назвою development (рис. 1.12). Принципи створення нових гілок (розгалуження) буде розглянуто далі.

Рисунок 1.12 – Перемикання між гілками (checkout)

З'єднати гілки (merge)

Командою git merge master development з'єднуються гілки development та master (рис.1 13).

Рисунок 1.13 – З'єднання гілок

Отримати історію ревізій (change log) за будь-яким файлом

Командою git log отримуєм історію ревізій (change log) за файлом example.txt (рис. 1.14).

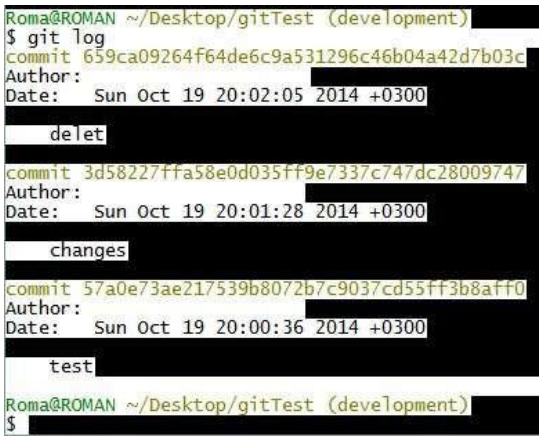


Рисунок 1.14 – Отримання історії ревізій за файлом example.txt

IV. Контрольні питання

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?

Лабораторна робота №2
Початок роботи з GITHUB

I. Підготовка до лабораторної роботи

Для підготовки до лабораторної роботи слід проробити відповідний теоретичний матеріал, що вказаний в цьому розділі.

Також для виконання цієї лабораторної роботи необхідною умовою є виконана лабораторна робота №1.

II. 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.

III. Завдання на лабораторну роботу

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.

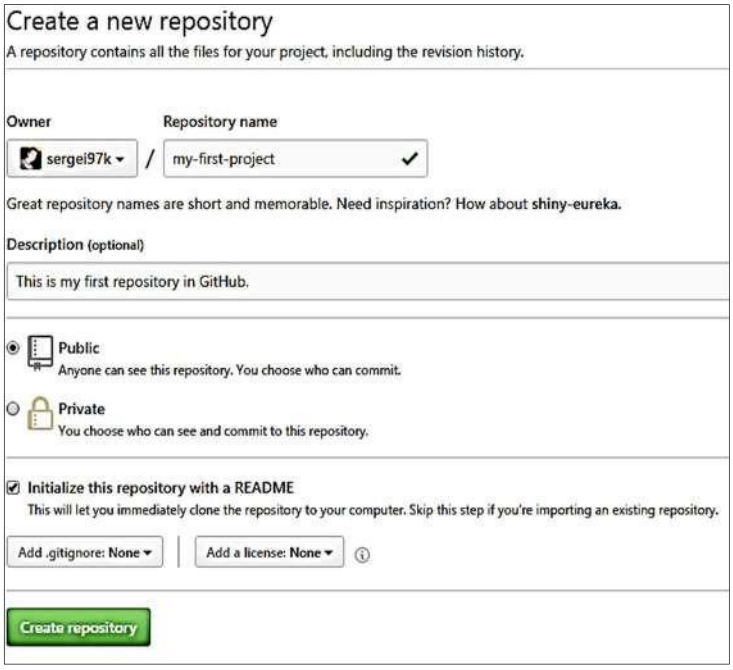


Рисунок 2.2 – Створення репозиторію

2. Створення гілки

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

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

Діаграма, наведена нижче, ілюструє роботу розгалуження(рис. 2.3).

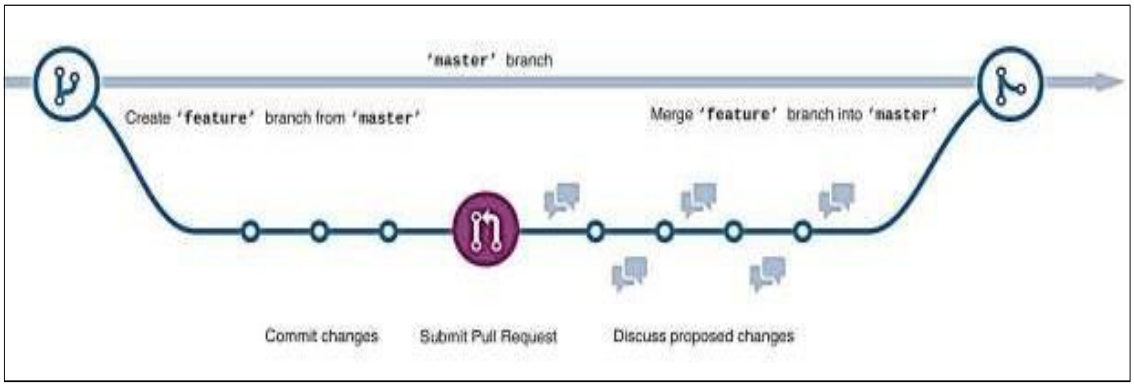


Рисунок 2.3 – Діаграма розгалуження

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.

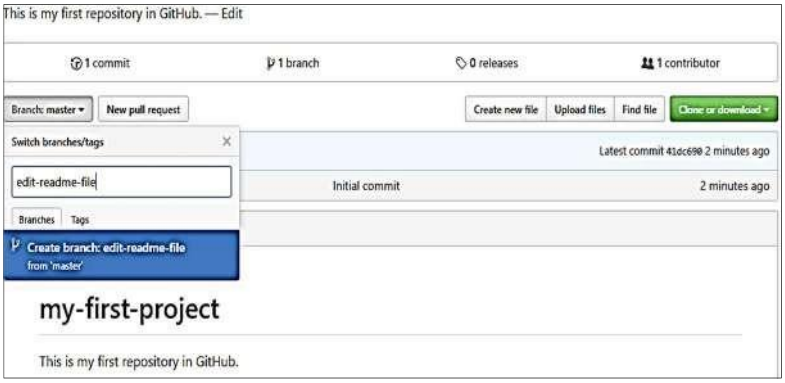


Рисунок 2.4 – Створення нової гілки

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. Історія 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.

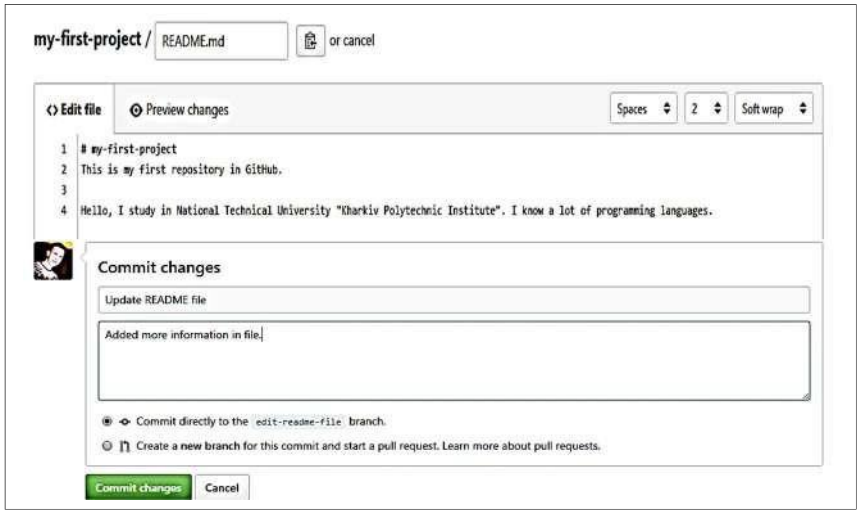


Рисунок 2.5 – Створення та commit змін

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);

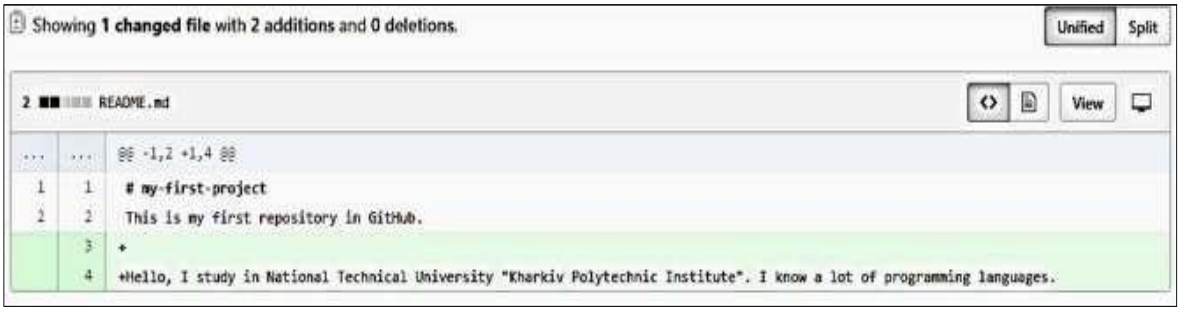


Рисунок 2.8 – Візуалізація змін

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

Рисунок 2.9 – Підтвердження змін

– для завершення роботи дати злиттю Pull Request назву, написати короткий опис поточних змін і натиснути кнопку Create Pull Request (рис. 2.10).

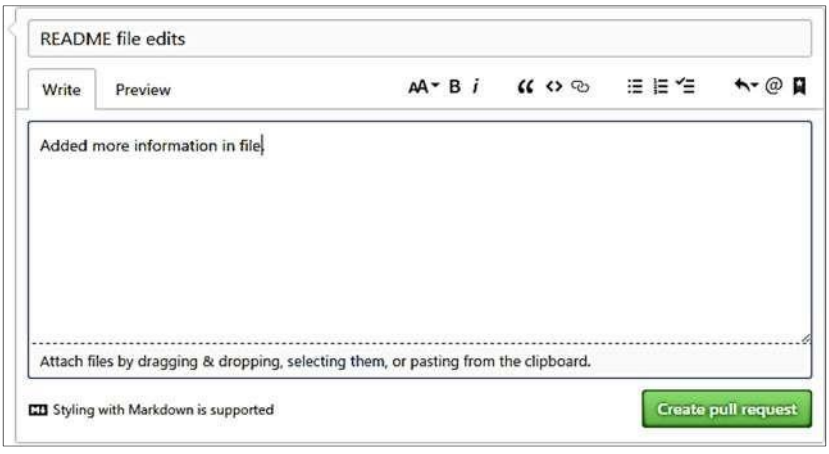


Рисунок 2.10 – Опис змін

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. 2.11);

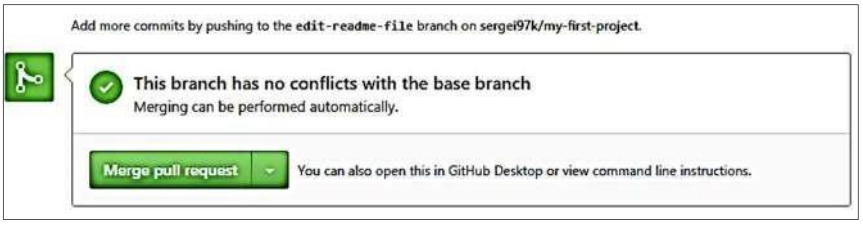


Рисунок 2.11 – Початок роботи зі злиття змін

– натиснути кнопку Confirm merge;

– видалити гілку, оскільки її зміни вже включені в основну гілку. Для цього натиснути кнопку Delete branch (рис. 2.12).

Рисунок 2.12 – Завершення роботи зі злиття змін

IV. Контрольні питання

1. Що таке GitHub?
2. Які системи контрою 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?

Лабораторна робота №3
Робота з GIT та GITHUB

I. Підготовка до лабораторної роботи

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

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

II.Теоретичні відомості

Git – це інструмент, що дозволяє реалізувати розподілену систему контролю версій.

GitHub - сервіс онлайн-хостингу репозиторіїв, що має всі функції розподіленого контролю версій і функціональність управління вихідним кодом - все, що підтримує Git і навіть більше. Також GitHub може похвалитися контролем доступу, багтрекінгом, управлінням завданнями та вікі для кожного проекту.

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

III. Завдання на лабораторну роботу

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.

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.

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);

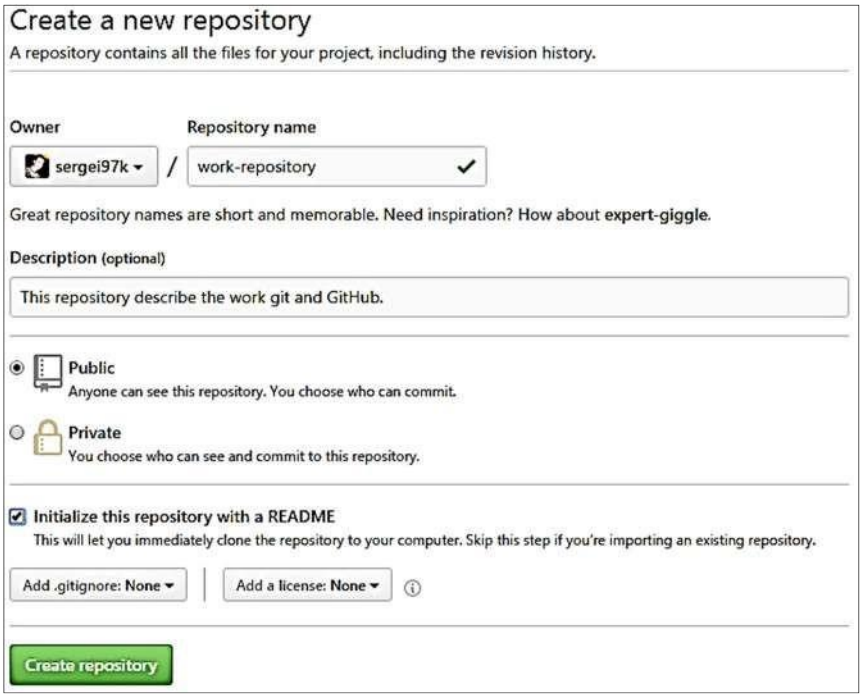


Рисунок 3.1 – Створення нового репозиторію

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);

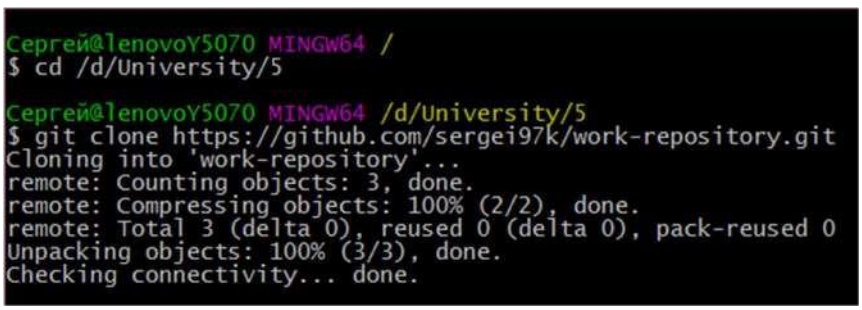


Рисунок 3.3 – Виконання команди git clone

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

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 файла 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

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 не обхідно використовувати команду git push з прапором -u і назвою гілки (наприклад: git push -u origin master);

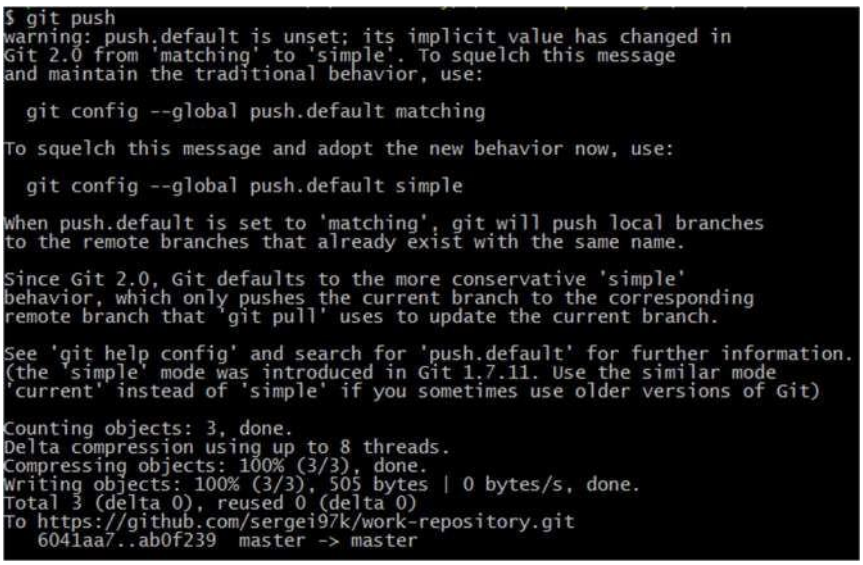


Рисунок 3.7 – Виконання команди git push

- 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).

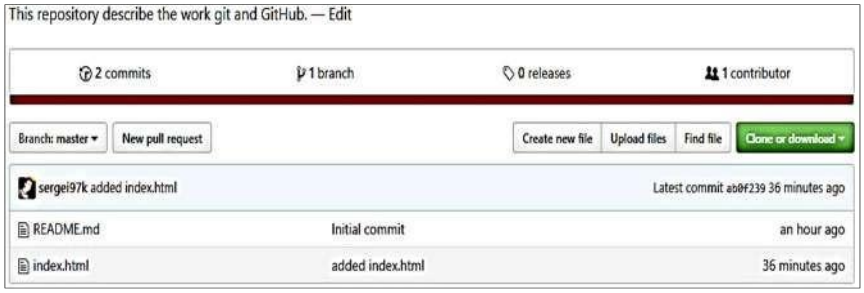


Рисунок 3.8 – Перегляд змін в репозиторії

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);

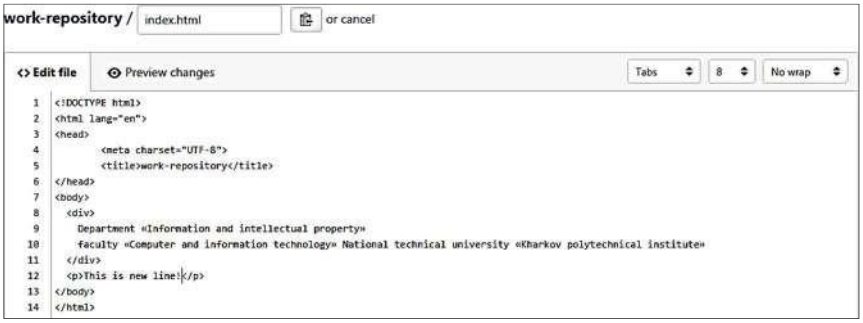


Рисунок 3.9 – Зміни до файлу

- ввести назву commit'у і його опис;

– натиснути кнопку Commit changes (рис. 3.10);

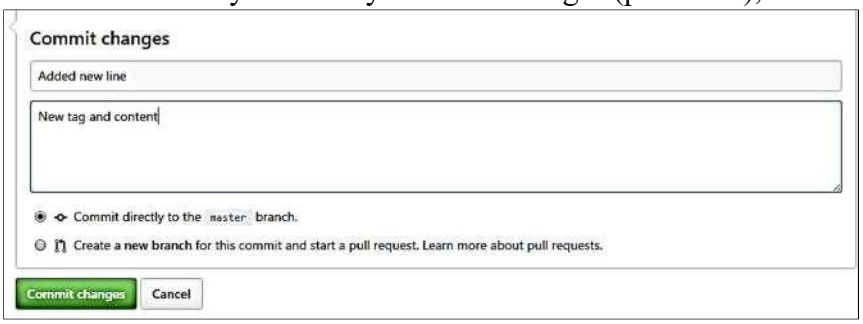


Рисунок 3.10 – Введення назви commit'а і його опису

– 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);

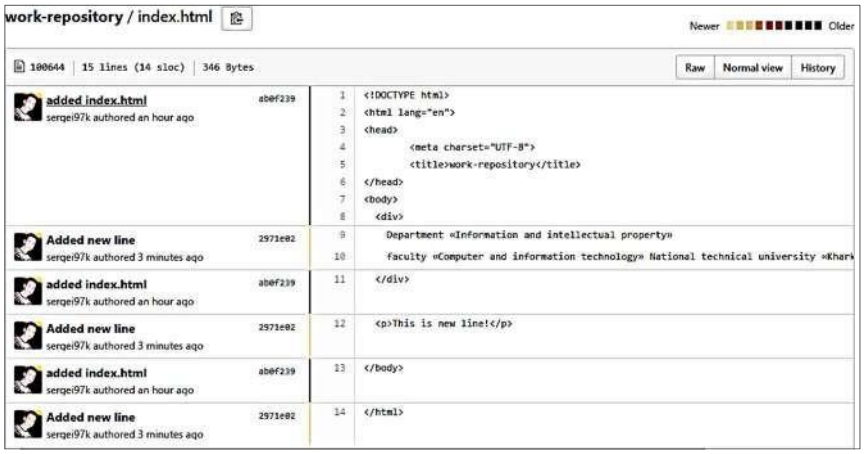


Рисунок 3.11 – Візуалізація змін в файлі

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

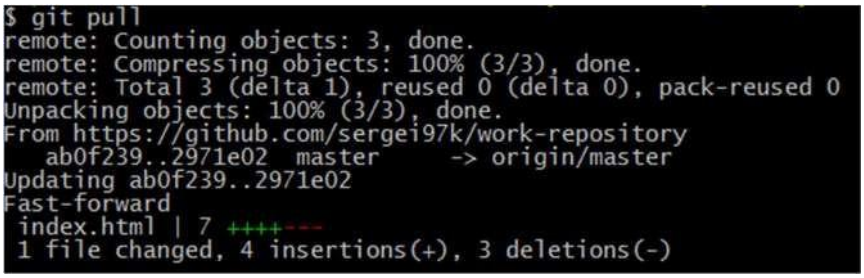


Рисунок 3.12 – Виконання команди git pull

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

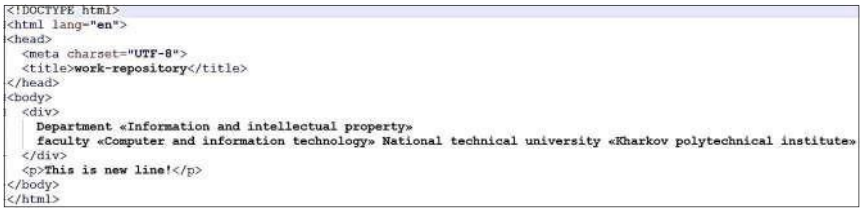


Рисунок 3.13 – Візуалізація змін в файлі

Ігнорування файлів

Ігноровані файли не обов’язково розміщувати у віддалений репозиторій. 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);

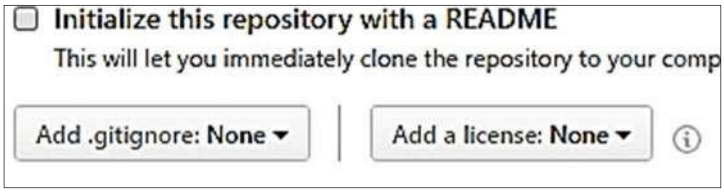


Рисунок 3.14 – Створення файлу

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

Рисунок 3.15 – Створення файла .gitignore

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

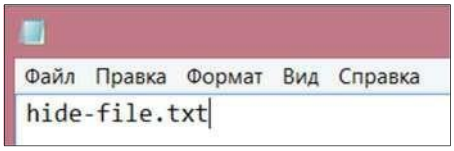


Рисунок 3.16 – Запис назви файлу для ігнорування

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

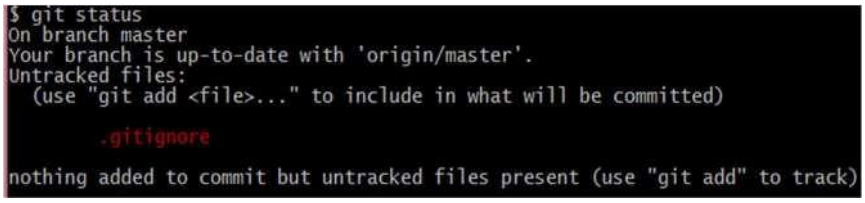


Рисунок 3.17 – Використання команди git status

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

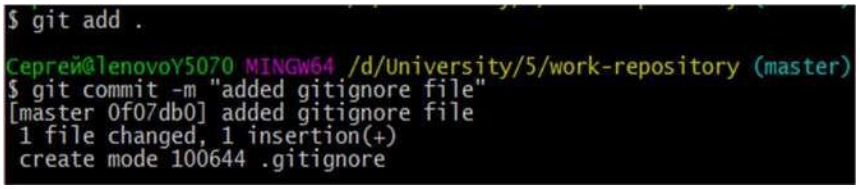


Рисунок 3.18 – Використання команди git add

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

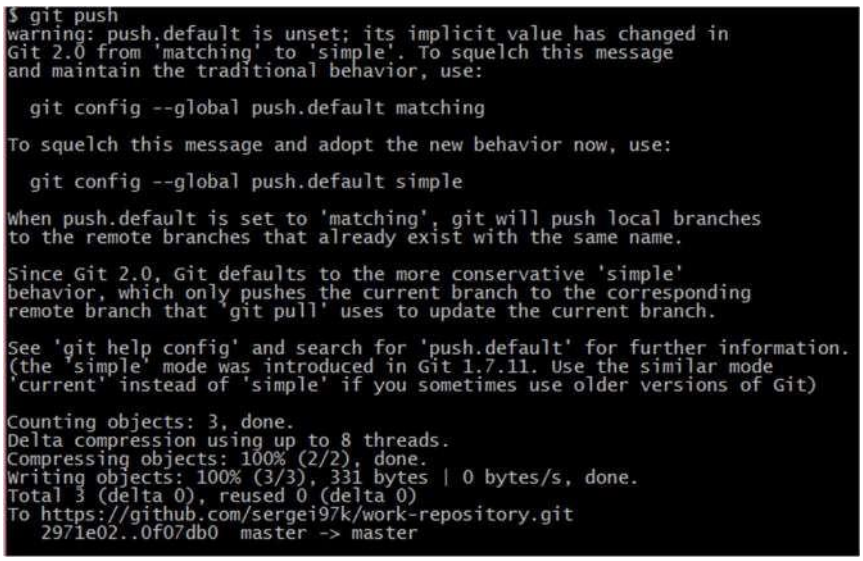


Рисунок 3.19 – Використання команди git push

– 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.

Рисунок 3.20 – Результат роботи по ігнорування файлу

IV. Контрольні питання

1. Як створити копію віддаленого репозиторія у локальній директорії?
2. Як перевірити наявність змін у файлі?
3. Яким чином відбувається відправлення змін у віддалений репозиторій?
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?

Лабораторна робота №4
Використання сучасних гнучких методологій розроблення інформаційних систем

I. Підготовка до лабораторної роботи

Ознайомитись з теоретичною та практичною частиною наведеною нижче для виконання даної лабораторної роботи

II. Теоретичні відомості

Scrum — одна з найпопулярніших гнучких методологій розробки програмного забезпечення з сімейства Agile. Легка й доступна у використанні, але складна в засвоєнні, якщо вірити офіційному опису. На практиці вся складність зводиться до того, щоб навчити розробників та інших фахівців дотримуватися цієї методології в роботі. Але все по порядку.

По-перше, методологія — це набір правил і практик, завдяки яким краще організувати роботу над проектом. Причому краще для всіх: самої команди, компанії-розробника, менеджерів і, звичайно ж, для вас як для замовника.

По-друге, Scrum — це не якась програма та не методичка, хоча ПЗ для управління проектами на основі скрам та відповідної літератури більш ніж достатньо. Це принцип, концепція-каркас та рекомендації, як менеджеру підвищити керованість, передбачуваність та ефективність роботи.

На офіційній сторінці The Scrum Guide можна почитати докладно, хто, як і навіщо придумав Скрам, а головне, що творці вкладають у це поняття.

Є багато методів проектного управління, і вашому проекту, хоч би яким він був, потрібно вибрати один з них. І як тільки ви вирішите, що використовуватимете методологію Scrum, ваш проектний менеджер адаптує всі ці принципи, правила та практики під конкретний проект, і почнеться робота.

Отже, 4 особливості Скрам як методології:

- робота над проектом розбивається на невеликі підзадачі;
- команда з 5-7 осіб виконує кожну з них у фіксований термін (1-4 тижні);
- протягом роботи над одним підзавданням проводиться 5 типів нарад;
- отриманий результат роботи над кожним підзавданням має цінність для замовника.

Рисунок 4.1 — Скрам

Тепер докладніше про кожну з особливостей:

Sprint

Спринт — головна фішка Скрам. Саме так називається кожне невелике підзавдання, з яких складається проект. Всі спринти повинні бути однаковими за тривалістю, та ви не повірите, але найчастіше довжина одного — два тижні, рідше за місяць. А скільки саме, залежить від особливостей вашого проекту. Зазвичай, чим складніше та незвичайніше завдання, тим спринт коротший, щоб швидше зрозуміти, скільки реально часу потрібно для досягнення більш масштабної мети, та не витрачати час розробці на те, що може не знадобитися.

Загалом спринт — це про конкретні задачі. Бракувало якоїсь функції? Додали. Щось не працювало? Полагодили. Завдяки йому зручно організовувати роботу та ще зручніше стежити за прогресом проекту загалом.

Артефакти

Красивим словом «Артефакти» в Scrum називають речі, що створюються під час розробки:

1. Беклог продукту. Product Backlog — зона відповідальності власника продукту. Це список завдань або, як його називає Вікіпедія, "журнал побажань до проекту". Беклог — це не щось, що затвердили раз і назавжди, а гнучкий перелік функцій, покращень, виправлень тощо. У ньому вказуються актуальні задачі для команди та зазначаються ті, що вже виконані.
2. Беклог спринту. Ще один беклог, але менший і конкретніший. Це список завдань на конкретний спринт, який формується на мітингу щодо його планування. Він теж може змінюватися, якщо команда зіткнулася зі складнощами, і потрібно зробити щось ще, крім того, що запланували. Але його мета, вона мета спринту, залишається незмінною.
3. Інкременти. Це саме той, отриманий результат роботи над кожним підзавданням, що має цінність для замовника. Інкрементом він називається тому, що його вже можна так чи інакше додати до решти проекту та подивитися, як він працює. Це не обов'язково має бути ціла нова функція, цілком можливо й удосконалення тієї, що вже є, або взагалі виправлення помилки. Але обов'язково те, що команда мала зробити протягом спринту. Загалом це очікуваний (найчастіше) результат, який показують власнику продукту, щоб він бачив, як йде робота над його проектом.

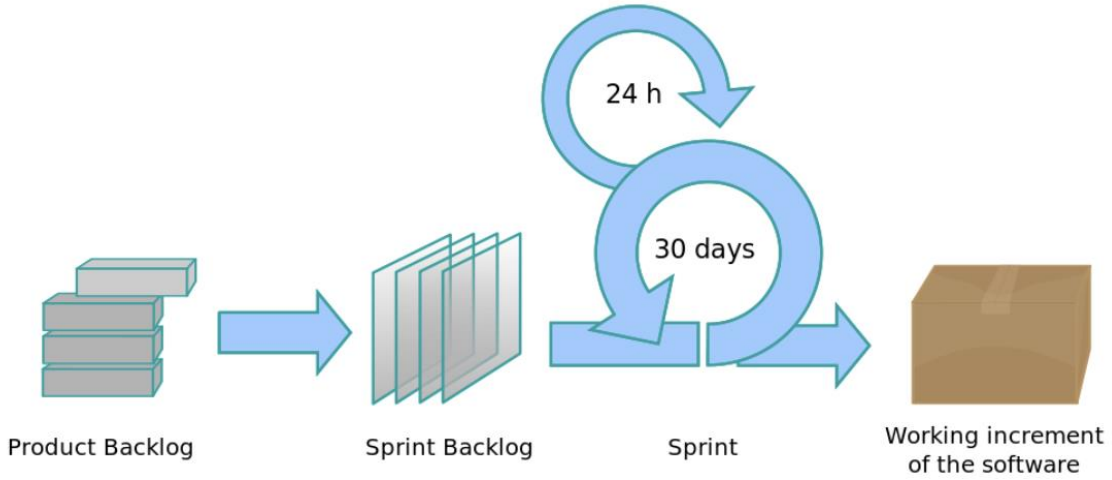


Рисунок 4.2 — Артефакти Скрам

Наради

Scrum — штука циклічна, і цей цикл складається з повторення різних нарад, вони ж мітинги:

– Project/Product backlog. Власник продукту приходить на першу таку нараду з найголовнішим артефактом проекту — підготовленим списком завдань, які потрібно вирішити для запуску. Беклог — це сучасна версія технічного завдання, в якій можна й потрібно міняти будь-що, якщо щось змінилося на ринку або в уподобаннях користувачів. У ньому зібрані та відсортовані за пріоритетом усі робочі завдання (Story, Bug, Task та інше) та в нього ж вноситься інформація про хід виконання робіт: зробили завдання — поставили галочку. На цьому мітингу всі просто ознайомлюються з тим, над чим вони мають працювати в цілому, а ось на наступному починають обговорювати.

– Sprint Planning. Планування самого спринту — обговорення найпріоритетнішого завдання командою та скрам-майстром. На цьому етапі вибираються історії з беклог, які потрібно зробити для

виконання мети спринту. Наприкінці цієї наради всі учасники команди повинні чітко розуміти, що їм слід робити.

– Daily Standup Meeting. Щодня всі учасники команди збираються в один і той же вибраний час, щоб розповісти, як у них справи. Із задачами за проектом. Кожен у двох реченнях описує, що він зробив учора та що робитиме сьогодні, а якщо зіткнувся з труднощами, то ще й про них — як вирішив чи як збирається вирішувати. Назва стендап буквально означає, що, якщо справа відбувається в офісі, то розробникам рекомендується спілкуватися стоячи. Щоб нікому не хотілося балакати довше, ніж треба. В умовах віддаленої роботи щоденні мітинги теж не затягуються — докладні обговорення проблем або ідей, що виникли, виносяться на окремі дзвони між зацікавленими учасниками, а решта відзвітували й пішли займатися своїми справами. Тому що Скрам — це про ефективність!

– Sprint Review. Наприкінці спринту, коли все готово, інкремент показують власнику продукту, а також усім, кому це цікаво, якщо досвід може бути корисним колегам. Якщо все добре, то інкремент випускають на прод, а в беклог вносяться відповідні зміни. Дуже часто цей етап плавно перетікає до першого з наступного спринту.

– Sprint Retrospective. Зустріч команди для обговорення робочих і супутніх моментів. Скрам-майстер проводить аналітику спринту, всі діляться думкою про те, як він пройшов, а також про учасників команди — хто молодець, а хто не дуже, а потім обговорюють, як працювати краще.

Дійові особи

У класичному Scrum існує 3 базові ролі:

1. Product owner (PO). Це ви як власник продукту, а найчастіше хтось із ваших співробітників, кого ви зробите відповідальним за спілкування з командою розробки. Та людина, яка створюватиме беклог проекту та доповнюватиме його, слухатиме наприкінці спринту, що ж там ця команда розробки зробила, а що ні, і що робитиме далі. PO не обов'язково повинен розумітися на технологіях розробки, але мусить бути спеціалістом у своїй галузі. Його робота — точно знати, що має робити готовий проект і кожна його частина, а також вникати в те, як розробляється.

2. Scrum master (SM). Скрам-майстер — проджект-менедже. Його робота, з одного боку, допомагатиме продукт оунеру розібратися в нюансах роботи зі Скрам, а з іншого — організовувати роботу команди. Він відповідає і за пошук кадрів для команди, і за те, щоб у них були матеріально-технічні ресурси, і загалом за те, щоб усі товаришували та ефективно працювали. Планування та проведення всіх мітингів у спринті — теж робота SM.

3. Development team (DT). Команда розробників — +/- 5 спеціалістів, які займатимуться роботою над проектом. The Scrum Guide вимагає від них не тільки навичок для виконання завдань, але ще й бути спроможними самостійно організовувати робочі процеси, а також нести колективну відповідальність за досягнення мети спринту. Команд цих може бути будь-яка потрібна вашому проекту кількість, але вони повинні складатися з фахівців у певних технологіях і бути невеликими, щоб уникнути проблем із комунікацією. А комунікувати потрібно буде багато, інакше як навчитися самоорганізовуватися, працювати разом, брати відповідальність, аналізувати успіхи та невдачі та робити інші речі, що постулюються методологією Скрам? Загалом, для роботи в командах Scrum мало бути хорошим технічним фахівцем, потрібні ще й soft skills вищі за середні.

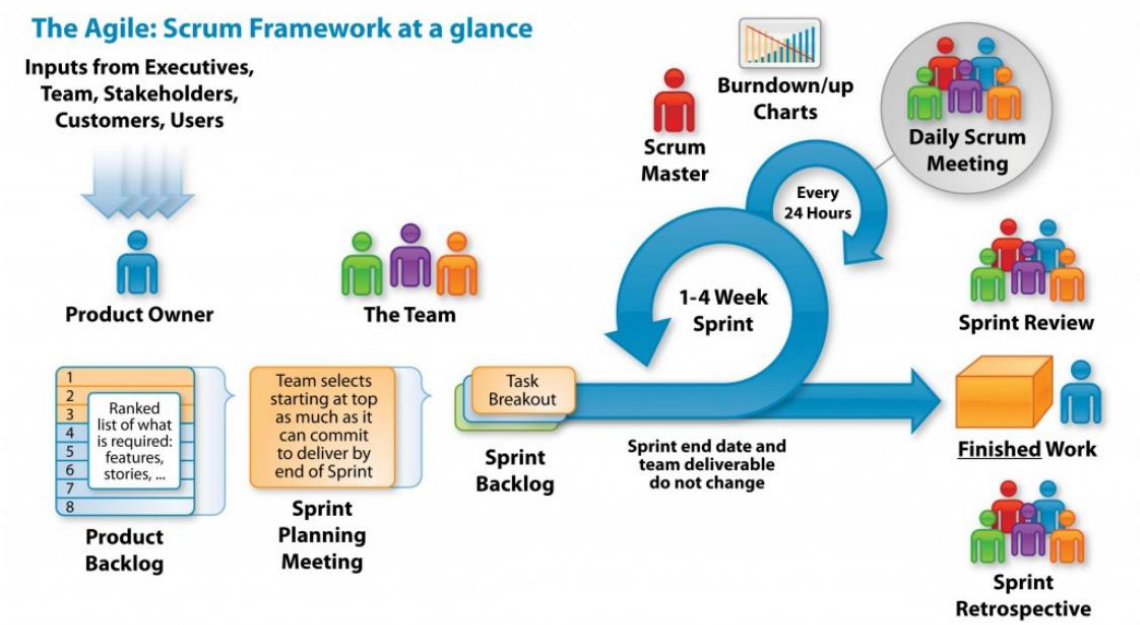


Рисунок 4.3 — Дійові особи Скрам

2 переваги та 2 недоліки Scrum

1. Гнучкість. Scrum — просто ідеальна система управління проектами, які ростуть і масштабуються, а це буквально будь-яка мобільна або веб-додаток, і навіть сайти. Сьогодні ви додали нову функцію, подивилися, як вона працює, і вже у наступному спринті можете почати її вдосконалювати, міняти чи прибрати! І це актуально не тільки для MVP, для яких кожна функція нова, але й для проектів, яким вже кілька років, і вони постійно тестують гіпотези, щоб стати кращими.

2. Видимі результати. Підсумок кожного спринту — щось реальне. Нова функція або виправлення помилки не так важливо, як можливість бачити, що робота йде, і йде успішно. Саме за це, окрім можливості міняти беклог, коли їм хочеться, і люблять Scrum власники продуктів. Учасникам команди це теж дуже важливо, оскільки умовно «закриває гештальт», дає змогу відчути задоволення від виконаної роботи.

3. Мотивація. Хотіти дотримуватися принципів Agile і робити це насправді — дві великі різниці. Знаєте, скільки людей здатні на самоорганізацію? 15%, у кращому випадку! А скільки готові погодитись на колективну відповідальність? А скільком подобаються «безкорисні» наради? Робота з усім цим і є та сама «складність в освоєнні». Тільки від скрам-майстра залежить, чи працюватиме Scrum для команди. Зробити так, щоб дорослі та розумні дяді та тьоті якщо не дружили, то поважали один одного, та розуміли важливість спільної роботи, може бути дуже складно. І хоча, з одного боку, вас як замовника не повинно хвилювати, як ваш підрядник свої справи вирішує. З іншого боку, рекомендую все-таки звернути на це увагу, якщо вам не хочеться одного дня залишитися без найважливішої для проекту команди. А, ну і будьте готові, що й вам доведеться готуватися до цих нарад раз на два тижні.

4. Невідповідність мети та інструменту. Scrum безумовно хороший для багатьох завдань, навіть не пов'язаних із розробкою. Але, при цьому, всі методології сімейства Agile об'єднує не просто терпимість, а любов до змін. Якщо вашому проекту заявлена гнучкість ні до чого, та ви впевнені, що точно знаєте, як має виглядати проект від початку й до кінця, краще вибрати класичне проектне управління, що буде значно ефективнішим.

Отже, Scrum — гнучка й неймовірно популярна методологія управління проектами. У ній великий проект розбивається на безліч маленьких підзадач-спринтів, кожна з яких виконується досвідченою та злагодженою командою в середньому за 2 тижні. Результати спринту — завжди щось цінне для проекту, що можна оцінити й протестувати в роботі. Для кожного спринту вибираються задачі зі списку-беклогу, який може вільно змінюватися відповідно до нової інформації про споживачів, ситуації на ринку та інших даних аналітики.

Головні принципи Scrum — ясність комунікації, прозорість і прагнення постійного вдосконалення.

III. Завдання на лабораторну роботу

1. Ознайомтеся з основними принципами методології Scrum.
2. Розробіть систему поділу ролей у проекту за принципами методології Scrum.
3. Визначте основні фази проекту за методологією Scrum.
4. Сформуйте журнал продукту, журнал спринту і графіка спринту.

IV. Контрольні питання

1. Що таке Scrum і як він працює?
2. Які ролі і відповідальності існують у Scrum?
3. Які засоби управління проектами використовуються в Scrum?
4. Які події відбуваються під час ітераційного процесу Scrum?
5. Які артефакти Scrum і яку роль вони відіграють у процесі розробки програмного забезпечення?
6. Як Scrum відрізняється від інших методологій розробки програмного забезпечення?

Лабораторна робота №5
Управління ризиками

I. Підготовка до лабораторної роботи

Ознайомитись з теоретичною та практичною частиною наведеною нижче для виконання даної лабораторної роботи

II. Теоретичні відомості

Ризик-менеджмент – це організований процес, який дозволяє ідентифікувати, виміряти невизначеності; обирати, планувати та впроваджувати припустимі заходи пом'якшення ризиків (див. рис. 5.1).

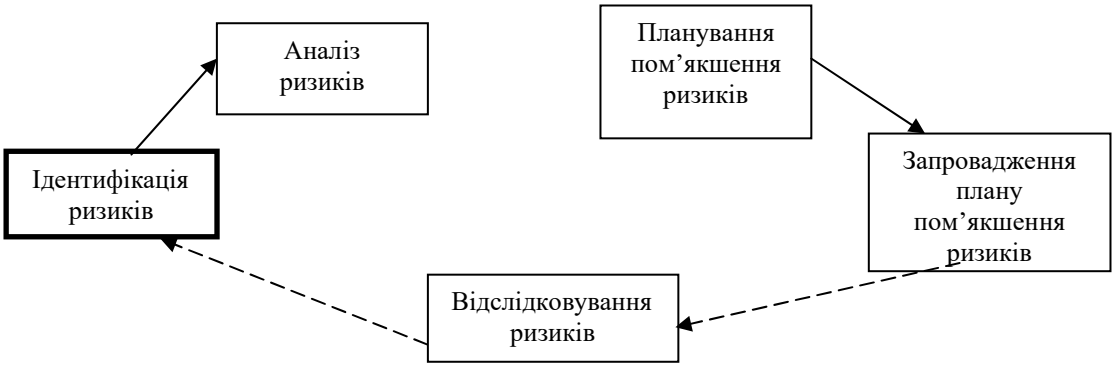


Рисунок 5.1 — Етапи процесу управління ризиками

Ідентифікація ризиків дозволяє відповісти на питання, що може спричинити проблеми за допомогою розгляду наявного і пропонованого процесу управління персоналом, створення дизайну, постачання, управління ресурсами, моніторингу результатів тестування, особливо невдач, дослідження можливих нестач відносно очікувань, аналізу негативних тенденцій. На цьому етапі досліджується кожен елемент процесу розробки, щоб визначити асоційовані джерела ризиків, розпочати їх документування і створити базу для успішного ризик-менеджменту.

Ідентифікація ризиків відбувається шляхом декомпозиції різних фаз процесу розробки: вимог, процесу, функціональної області, технічних особливостей чи фаз впровадження. Для ідентифікації ризиків та їх основних причин, необхідно поділити проект до того рівня деталізації, коли можна визначити ризики на самому нижньому рівні структури декомпозиції робіт. Під час декомпозиції ризики можуть бути ідентифіковані з використанням попереднього досвіду, мозкового штурму, аналізу результатів виконання подібних проектів, методик, розроблених інститутом управління проектами (Project Management Institute).

Щоб визначити головні причини ризиків та створити їх перелік необхідно:

- створити систему поділу робіт (Work Breakdown Structure);
- розглянути кожен елемент структури з точки зору джерел ризиків;
- визначити, що може спричинити проблеми;
- декілька разів відповісти на питання "чому?", поки не визначиться джерело ризику.

Типові джерела ризиків в проекті з розробки програмного забезпечення включають: процес розробки, вимоги, технічні можливості, тестування, моделювання та симуляцію, технологію, логістику, продуктивність/функціональність, витрати, управління, графік, зовнішні фактори, бюджет та ін.

Аналіз ризиків – дії спрямовані на дослідження кожного ідентифікованого ризику, щоб в результаті деталізувати опис ризику, виділити причину, визначити ефекти, допомогти в ранжуванні ризиків для подальшого їх пом'якшення.

Для того, щоб проаналізувати ризик необхідно розробити шкали ймовірності та наслідків, використовуючи методологію декомпозиції робіт WBS чи іншу систему декомпозиції проекту; визначити ймовірність виникнення кожного з ризиків; визначити наслідки як функцію від продуктивності, тривалості та вартості; занести отримані результати в базу даних проекту.

Пріоритети ризиків змінюються в залежності від етапу життєвого циклу проекту, тому необхідно виконувати аналіз ризиків декілька разів, щоб підтримувати базу даних в актуальному стані. Процес аналізу передбачає можливість визначення, наскільки великим є ризик. Для цього необхідно визначити ймовірність виникнення ризику. Рівень ймовірності приймає значення від 1 до 5, рівню 1 відповідає ризику, який не схоже, що виникне; відповідно рівень 5 відповідає ризику, який з'явиться майже напевно. Рівні ймовірності представлено в таблиці 5.1.

Таблиця 5.1 — Рівні ймовірності

Рівень	**Характеристика**	**Ймовірність виникнення**
1	Not Likely	10%
2	Low Likelihood	30%
3	Likely	50%
4	Highly Likely	70%
5	Near Certainly	90%

Можливі наслідки можна ідентифікувати з точки зору продуктивності, дотримання графіка і вартості, для цього пропонується таблиця 5.2, де наслідки розподілено на рівні в залежності від втрат, які спричинить ризик, якщо матеріалізується в проблему.

Таблиця 5.2 — Рівні та типи наслідків

Рівень	**Технічна продуктивність**	**Графік**	**Витрати**
1	Мінімальний вплив чи відсутність впливу на продуктивність	Мінімальній чи немає	Мінімальний чи немає
2	Незначне зменшення в технічній продуктивності чи можливості підтримки, яке можна подолати з невеликим впливом на систему чи без нього	Можливо дотримуватися ключових дат	Бюджет проекту зростає на 1%
3	Помірковане зменшення технічної продуктивності чи можливості підтримки з лімітованим впливом на цілі	Незначний зсув графіка. Проект виконується згідно з ключовими датами	Бюджет проекту зростає на 5%
4	Значні погіршення в технічній ефективності чи можливості підтримки, може поставити під загрозу успіх проекту	Є вплив на критичний шлях проекту.	Бюджет проекту зростає на 10%
5	Значне погіршення в технічнійпродуктивності; проект не виконується до ключових дат, поставить під загрозу успіх проекту	Проект не може бути виконано згідно з ключовими датами календарного плану.	Перевищує максимально припустиме збільшення бюджету на 10% і більше.

Виходячи з того, який рівень ймовірності та наслідків має ризик, можна визначити його

пріоритет, використовуючи Матрицю ідентифікації ризиків. Ризик може мати низький (L), середній (M) чи високий (H) пріоритет. В залежності від того, який пріоритет має ризик, слід приймати відповідні дії щодо управління цим ризиком. Так, ризик з низьким пріоритетом можна прийняти, а ризики з середнім чи високим пріоритетами необхідно пом'якшувати.

5	L	M	H	H	H
4	L	M	M	H	H
3	L	L	M	M	H
2	L	L	L	M	M
1	L	L	L	L	M
	1	2	3	4	5

Рівні наслідків

Рисунок 5.2 — Матриця ідентифікації ризиків

В процесі пом'якшення ризики можна уникати шляхом запобігання причини їх виникнення, контролювати причини й наслідки ризиків, делегувати певні функції ризик-менеджменту, зменшувати рівень ризику і продовжувати дотримуватись плану виконання проекту.

Планування пом'якшення ризиків – це дії, за допомогою яких визначають, оцінюють, та обирають можливості утримувати ризики на припустимому рівні згідно обмежень проекту та його цілей.

В процесі планування пом'якшення ризиків необхідно визначити тип пом'якшення і його особливості для кожного ризику, занести визначені можливості пом'якшення ризиків в план пом'якшення чи створити план пом'якшення ризиків для кожного ризику. В такий план звичайно включають назву ризику, дату в плані, особу, відповідальну за визначене джерело ризику, короткий опис ризику, причину виникнення та існування ризику, можливості пом'якшення ризику, визначення подій та дій, які можуть знизити ризик, критерій успіху для кожної події в плані робіт, статус ризику, можливі резервні шляхи виконання проекту, рекомендації з управління, ресурси, необхідні для пом'якшення ризику.

Дуже важливо пом'якшувати ризики на відповідному рівні структури декомпозиції проекту, щоб не дозволити їм перейти на вищий рівень.

На етапі впровадження плану пом'якшення ризиків відбувається впровадження дій, які можуть вплинути на ймовірність настання ризику та його наслідки. На цьому етапі необхідно визначити, які зміни в календарному плані, бюджеті, вимогах та контрактах потрібно внести, запроваджується зв'язок між керівництвом проекту, замовником та виконавцями, визначається, які дії необхідно виконати команді, щоб пом'якшити ризик, визначити, який вплив мало пом'якшення ризиків на план виконання робіт, задокументувати зміни.

Впровадження пом'якшення ризику повинно відбуватися згідно категорій ризику. Дуже важливо, щоб дії з пом'якшення ризиків були зрозумілі та прийнятні як для замовника, так для керівництва проекту і виконавців.

Відслідковування ризиків дозволяє відповісти на питання "Яким чином відбувається розробка проекту?». Цей процес включає в себе дії з систематичної перевірки та оцінювання ефективності заходів з пом'якшення ризиків. Головним засобом відслідковування ризиків є встановлення індикаторів на кожному етапі розробки проекту, які повинні забезпечити систему раннього попередження, коли ймовірність виникнення чи наслідки ризику перевищують попередньо задані межі таким чином, щоб можна було прийняти заходи з пом'якшення ризиків.

Заходи з пом'якшення ризиків включають: обговорення ризиків з усіма зацікавленими особами, моніторинг планів пом'якшення ризиків, дослідження наявних ризиків за допомогою матриці ідентифікації ризиків, повідомлення всіх зацікавлених осіб щодо того, коли плани пом'якшення ризиків повинні бути запроваджені та перевірені.

Для невеликих проектів процес управління ризиками відрізняється від управління ризиками в великому проекті. Малими вважаються проекти, які тривають менше 1 року, кількість виконавців для них менше 12 осіб, трудовитрати не перевищують 120 людино/місяців, бюджет менше $ 2 млн., менеджер в такому проекті виконує як керівницькі функції, так і роль розробника, в такому проекті може бути не більше одного підрядника.

Особливості малого проекту впливають на процес ризик-менеджменту, роблять його більш напруженим. Вплив характеристик невеликого проекту на процес ризик- менеджменту зображено в таблиці 5.3.

Таблиця 5.3 — Особливості малого проекту

Характеристики проекту	**Вплив на процес ризик-менеджменту**
Короткий графік	Невеликий час для пом'якшувальних дій Короткострокова перспектива залишає менше можливостей ризикувати
Лімітовані ресурси	Обмежена кількість розробників, які можуть керувати ризиками Незначні резерви бюджету, які можна використати для найняття додаткового персоналу
Маленька команда розробників	Ризик, пов'язаний з персоналом – критичний для кожного проекту
Менше зовнішніх зв'язків	Менше ризиків, через те що менше некерованих залежностей.

Для невеликих проектів визначають наступні джерела ризиків: незрозумілі вимоги, нереалістичні часові характеристики проекту, неадекватна кваліфікація виконавців, проблеми з фінансуванням.

Ранжування ризиків в малих проектах відбувається за допомогою матриці ідентифікації ризиків, данні в якій визначаються за допомогою матриці імовірності ризиків та матриці наслідків матеріалізації ризиків. Після ідентифікації ризиків, складають план пом'якшення ризиків та запроваджують його в життя.

Для дослідження процесу управління ризиками в маленькому проекті візьмемо проект з розробки Internet-порталу. Такий проект обмежений у часі, має невелику кількість робіт та виконавців, його бюджет в середньому не перевищує $ 1000. Як правило, компанія одночасно займається реалізацією декількох таких проектів, тому ризики, що ідентифіковані і пом'якшені для одного проекту, можуть істотно допомогти при розробці інших.

Розглянемо невеликий проект з наступною структурою декомпозиції робіт:

1. Формування вимог
1.1. Вивчення вимог клієнта
1.2. Обробка вимог фахівцями
1.3. Затвердження вимог клієнтом
2. Розробка графічного дизайну
2.1. Розробка корпоративного стилю
2.2. Розробка макета
3. Верстка індексної сторінки і підсторінок
3.1. Логічний поділ макету
3.2. "Порізка" макету
3.3. Верстка згідно зі стандартами W3C
3.4. Валідація сайту
4. Кросбраузерне тестування
5. Розробка структури сайту
5.1. Розробка первинної ієрархії

5.2. Розробка зв’язків між сторінками
5.3. Реалізація структури
6. Наповнення сайту графічними елементами
7. Розробка логіки обробки даних
7.1. Розробка структури бази даних
7.2. Розробка і реалізація запитів
7.3. Дослідження можливості роботи на сервері
7.4. Розробка механізму представлення даних
7.5. Розробка механізму захисту даних і доступу
7.6. Наповнення бази даних необхідними даними
7.7. Локальне тестування бази даних
8. Тестування сайту
8.1. Тестування дизайну
8.2. Тестування структури сайту
8.3. Тестування логіки
8.4. Тестування бази даних
9. Інсталяція
9.1. Пошук хостінгу
9.2. Пошук домена, аналіз домена
9.3. Встановлення зв’язку хостінгу з доменом
9.4. Поставка сайту замовнику
10. Підтримка.

Дослідимо проект за методам критичного шляху і визначимо довжину критичного терміну виконання проекту та роботи, які знаходяться на критичному шляху. Дані графіку наведено у таблиці 5.4.

Таблиця 5.4 — Дослідження календарного плану за методом критичного шляху

Номер роботи	Тривалість Роботи (діб)	Безпосередньо попередня робота	Безпосередньо наступна робота	Ранній час початку	Ранній час закінч.	Пізній час початку	Пізній час закінч.	Повний резерв часу
1.1	1	-	1.2	0	1	0	1	0
1.2	1	1.1	1.3	1	2	1	2	0
1.3	1	1.2	2.1, 5.1, 7.1	2	3	2	3	0
2.1	2	1.3	2.2	3	5	7	9	4
2.2	2	2.1	3.1	5	7	9	11	4
3.1	0.5	2.2	3.2	7	7.5	11	11.5	4
3.2	0.5	3.1	3.3	7.5	8	11.5	12	4
3.3	1	3.2	3.4	8	9	11	12	3
3.4	0.5	3.3	4	9	9.5	12	12.5	3
4	0.5	3.4	8.1	9.5	10	12.5	13	3
5.1	1	1.3	5.2	3	4	8	9	5
5.2	1	5.1	5.3	4	5	9	10	5
5.3	2	5.2	6	5	7	10	12	5
6	1	5.3	8.2	7	8	12	13	5
7.1	1	1.3	7.2	3	4	3	4	0
7.2	2	7.1	7.3	4	6	4	6	0
7.3	0.5	7.2	7.4	6	6.5	6	6.5	0
7.4	1	7.3	7.5	6.5	7.5	6.5	7.5	0
7.5	2	7.4	7.6	7.5	9.5	7.5	9.5	0
7.6	1	7.5	7.7	9.5	10.5	9.5	10.5	0
7.7	1	7.6	8.4, 8.3, 9.1	10.5	11.5	10.5	11.5	0
8.1	0.5	4	9.4	10.5	10.5	13	13.5	2.5
8.2	0.5	6	9.4	8.5	10.5	13	13.5	4.5
8.3	0.5	7.7	9.4	11.5	12	13	13.5	1.5

8.4	0.5	7.7	9.4	11.5	12	13	13.5	1.5
9.1	0.5	7.7	9.2	11.5	12	11.5	12	0
9.2	0.5	9.1	9.3	12	12.5	12	12.5	0
9.3	1	9.2	9.4	12.5	13.5	12.5	13.5	0
9.4	0.5	8.1, 8.2, 8.3, 8.4, 9.3	-	13.5	14	13.5	14	0

Критичний шлях даного проекту становить 14 робочих днів. 14 з 29 робіт лежить на критичному шляху, отже це значною мірою ускладнює процес управління ризиками. Найбільш простий спосіб пом'якшення ризику – це збільшити час виконання робіт, які можуть спричинити його виникнення. Якщо ці роботи лежать на критичному шляху, то збільшення часу на їх виконання може дуже вплинути на вартість проекту, на загальний час його виконання та навіть на успіх проекту в цілому.

За допомогою експертної оцінки та на базі попередніх проектів було сформульовано основні ризики, які можуть виникнути в процесі розробки проекту.

Далі необхідно визначити пріоритет ризику, який, як було зазначено раніше, визначається як функція від ймовірності виникнення ризику та розміру його впливу на проект. За допомогою матриць рівнів імовірності ризиків та рівнів та типів наслідків, заповнюють матрицю ідентифікації ризиків (див. таблицю 5.5). Після цього виконаємо ранжування ризиків та розробимо план пом'якшення ризиків, який наведено у таблиці 5.6.

Таблиця 5.5 — Ідентифікація та аналіз ризиків

№	Опис ризику	Рівень ймовірність виникнення	Рівень наслідків	Пріоритет
1.	Складна структура сайту	2	3	L
2.	Грубі помилки, виявлені при тестуванні та валідації	3	3	M
3.	Конфлікти в графіку	2	4	M
4.	Низька кваліфікація розробників	3	5	H
5.	Складний дизайн	3	2	L
6.	Зміни вимог клієнтом	2	5	M
7.	Нестача матеріальних ресурсів	2	4	M
8.	Помилки в процесі тестування	2	3	L
9.	Некоректна робота серверів, комп'ютерів та програмного забезпечення	2	5	M
10.	Вимоги не відповідають реальності	2	3	L
11.	Затримка при встановленні зв'язку між доменом та хостінгом	2	4	M
12.	Звільнення виконавця	2	4	M
13.	Складність структури бази даних	2	4	M
14.	Складність запровадження алгоритму захисту даних	2	4	M

15.	Обрані сервери не підтримують необхідну для розробки СКБД	1	2	L

Таблиця 5.6 — Ранжування та пом’якшення ризиків

№	Пріоритет	Роботи чи етапи, на яких виникають ризики.	План пом’якшення ризиків
4.	H	Протягом проекту	Покращити якість відбору виконавців проектів, організувати курси підвищення кваліфікації виконавців, додаткові тренінги
2.	M	3.4, 8.1, 8.2, 8.3, 8.4	Покращити якість процесу тестування, проводити збори, присвячені обговоренню складностей і проблем в процесі розробки. Роботи знаходяться не на критичному шляху, отже можна виділити час на виправлення помилок та перетестування програмного продукту
3.	M	Протягом проекту	Переглянути календарний план, знайти вузькі місця в плані, об’єднати деякі роботи, викинути надлишкові пункти, перемістити трудові ресурси таким чином, щоб максимально знизити наслідки від змін в календарному плані.
6.	M	Протягом проекту,1.1, 1.2, 1.3	Покращити рівень взаємодії з клієнтом, роботи знаходяться на критичному шляху, отже ми не можемо збільшити час обробки вимог. Виділити кошти на залучення додаткових фахівців.
7.	M	Протягом проекту,7.3, 9.3	Виділити резервні кошти на закупівлю необхідних ресурсів, дослідити наявність та потребу ресурсів до початку розробки
9.	M	Протягом проекту	Виділити кошти на обладнання додаткових робочих місць, дослідити наявність необхідного програмного забезпечення та ліцензій на користування ним
11.	M	9.3	Робота знаходиться на критичному шляху, слід врахувати в бюджет витрати, які можуть виникнути при затримці, чи розпочати процес зв’язування заздалегідь
12.	M	Протягом проекту	Врахувати витрати на підготовкудодаткового персоналу, покращити якість робот з персоналом

13.	M	7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7	Застосувати додаткові трудові ресурси на дослідження вимог та розробку структури бази. Виконувати розробку запитів в декілька ітерацій, на кожній ітерації нарощувати складність логіки
14.	M	7.5	Врахувати в бюджет витрати на обрання альтернативного алгоритму, надодатковий час на дослідження існуючого
1.	L	5.1, 5.2, 5.3	Розробляти структуру сайту в декілька ітерацій, починаючи від простого до складного. Врахувати додаткові витрати на запрошення експерта
5.	L	2.1, 2.2	Домовитись з клієнтом про спрощення дизайну, замінити складні елементи більш простими, які не зменшать істотно функціональність
8.	L	3.4, 7.7, 8.1, 8.2, 8.3, 8.4	Роботи не знаходяться на критичному шляху, отже можна виділити додатковий час на тестування та на розроблення додаткових тестових наборів даних та процедур тестування
10.	L	Протягом проекту	Дослідити вимоги до затвердження вимог клієнтом, якщо немає можливості переконати клієнта, що вимоги не відповідають реальності і проект зазнає невдачі – передбачити можливості швидко згорнути проект та перенести наявні ресурси на виконання інших проектів
15.	L	7.3, 7.5	Врахувати витрати на пошук інших серверів, які підтримують необхідну СКБД, дослідити можливість виконання проекту без застосування СКБД

Після дослідження ризиків можна прийняти рішення: ризики не ігнорувати та не приймати, а пом’якшувати їх. З таблиці 5.6 бачимо, що більшість планів пом’якшення ризиків передбачає збільшення витрат часу чи збільшення бюджету. В залежності від реальних умов розробки можна прийняти рішення: ризики з низьким пріоритетом прийняти, а ризики з високим та середнім пріоритетами пом’якшувати.

Після виконання проекту, ідентифіковані ризики, ризики, які не перетворились на проблеми, ризики, які спричинили збитки, слід занести в базу даних та використовувати під час розробки подібних проектів. Слід зауважити, що в проектах з Web-розробок така база – дуже ефективне рішення, бо всі проекти дуже схожі, в процесі розробки виконуються майже однакові процедури. Ідентифікація спільних ризиків не буде викликати проблем, буде залишатися більше часу і ресурсів на дослідження унікальних ризиків та розробку планів їх пом’якшення.

Управління ризиками – складний та важливий процес. Його слід провадити на кожному етапі життєвого циклу програмного забезпечення. В залежності від етапу розробки пріоритет ризику змінюється. Процес ідентифікації ризиків та їх оцінювання – достатньо суб’єктивний. Він включає в

себе досвід з попередніх проектів, експертну оцінку та інтуїцію менеджера. Дуже складно ідентифікувати ризики, коли розробляється перший подібний проект, коли експерти висловлюють протилежні думки, які складно привести до якоїсь функції розподілу, коли команда розробників має культуру розробки, яка ігнорує наявність ризиків. В таких умовах ризики легко перетворюються в проблеми і викликають збитки та поразки проектів. Методи оцінювання ризиків довели свою ефективність завдяки рокам їх успішного використання. Вони розроблялись для управління великими проектами і пристосовані до життєвого циклу проекту зі створення складного програмного забезпечення.

III. Завдання на лабораторну роботу

Виконайте ризик-менеджмент свого проєкту.

IV. Контрольні питання

1. Що таке ризик в контексті розробки програмного забезпечення?
2. Які інструменти використовують для ідентифікації ризиків?
3. Які критерії можна використовувати для оцінки ризиків?
4. Які стратегії управління ризиками можуть бути використані при розробці?
5. Які кроки потрібно вжити для мінімізації впливу ризиків на розробку програмного забезпечення?
6. Що таке план управління ризиками і як він пов'язаний з розробкою програмного забезпечення?
7. Які можуть бути наслідки невдачі управління ризиками?

Лабораторна робота №6
Тестування програмного забезпечення

I. Підготовка до лабораторної роботи

Ознайомитись з теоретичною та практичною частиною наведеною нижче для виконання даної лабораторної роботи

II. Теоретичні відомості

1. Вступ

Життєвий цикл програмного забезпечення (SDLC – Software Development Life Cycle) – період часу, який починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації. Цей цикл – процес побудови і розвитку програмного забезпечення.

Етап 1 – Планування (Planning). На цій фазі клієнт пояснює основні деталі і концепції проекту, оговорюється необхідний ресурс, час і бюджет, що необхідний для розробки.

Етап 2 – Аналіз вимог (Requirements analysis). Ця фаза розрахована для підготовки набору вимог. Потім йде етап узгодження вимог. Як результат ми маємо отримати узгоджений документ з вимогами.

Етап 3 – Дизайн і розробка (Design & Development). На цій фазі визначаються основні концепції дизайну програмного забезпечення. Після узгодження дизайну починається безпосередньо розроблення продукту.

Етап 4 – Впровадження (Implementation). Включає в себе програмування і отримання кінцевого продукту (бібліотеки, білди, документація).

Етап 5 – Тестування (Testing). На цій фазі проводиться перевірка на відповідність вимогам і підтвердження того, що продукт розроблений згідно з ними.

Етап 6 – Оцінка (Evaluation). На фазі оцінки (або пререлізу) продукт оцінюється замовником і вносяться останні уточнення.

Етап 7 – Реліз (Release). Заключна фаза розробки, враховуються уточнення, що зроблені замовником на фазі оцінки. Підготовка продукту в «коробці».

Етап 8 – Підтримка (Support). Фаза технічної підтримки продукту.

Тестування ПЗ (Software testing) – перевірка відповідності між реальною і очікуваною поведінкою програми.

Тестування – це процес дослідження ПЗ з метою виявлення помилок і перевірки якості.

У більш широкому сенсі: Тестування – це одна з технік контролю якості, що включає в себе активності з планування робіт (Test Management), проектування тестів (Test Design), виконання тестування (Test Execution) та аналізу отриманих результатів (Test Analysis).

Тестування так само можна описати як процес верифікації та валідації того чи іншого програмного продукту, щоб дізнатися на скільки точно він задовольняє всім встановленим вимогам.

Верифікація (Verification – узгодження) – це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу (чи виконуються наші цілі, терміни, завдання, по розробці проекту, визначені на початку поточної фази.)

Валідація (Validation – затвердження) – це визначення відповідності ПЗ очікуванням і потребам користувача, вимогам до системи.

Приклад:

Замовник хоче щось. Узгоджується ТЗ, з якого ви розумієте що повинні зробити “синій автомобіль”. Ви починаєте процес виробництва, і в процесі перевіряєте те, що виходить: Чи автомобіль це? Чи він синій? Це – верифікація.

А потім ви приходите до замовника і говорите – ось дивіться, вийшов синій автомобіль, приймаєте роботу? А він каже: «хлопці, ви не так зрозуміли, я хотів рожевий трактор, переробити

швидко». Або навпаки каже: «так, це саме те що я хотів, швидше підписуйте акт здачі-приймання». Це – валідація.

Цілі і завдання процесу тестування

Основною метою процесу тестування – є доказ того, що результат розробки відповідає пред'явленим до нього вимогам.

Основна завданням тестування ПЗ: отримання інформації про статус готовності заявленої функціональності системи або програми.

Необхідні і достатні умови для проведення тестування

Необхідними умовами істинності твердження А називаються умови, без дотримання яких А не може бути істинним.

Достатніми називаються такі умови, за наявності (виконанні, дотриманні) яких твердження А є істинним.

Необхідні умови:

• Наявність об'єкта тестування, доступного для проведення випробувань.

• Наявність виконавця (ів) (людина або машина, або комбінація людина + машина)

Достатні умови:

• Наявність об'єкта тестування, доступного для проведення випробувань

• Наявність виконавця (ів) (людина або машина, або комбінація людина + машина)

• Наявність плану тестування

• Наявність тест кейсів / тестів

• Наявність звіту, що підтверджує виконання завдань і досягнення цілей, з тестування об'єкта

План Тестування (Test Plan) – це документ, що описує весь обсяг робіт з тестування, починаючи з опису об'єкта, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх дозволу .

Тестовий випадок (Test Case) – це артефакт, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації функції, що тестується або її частини.

Основні атрибути Test Case:

1) ID (номер),

2) Name (ім'я),

3) Preconditions (умови і параметри),

4) Steps (кроки до відтворення),

5) Expected result (очікуваний результат),

6) Actual result (фактичний результат),

7) Postconditions (постумови)

Тест дизайн (Test Design) – це етап процесу тестування ПЗ, на якому проектуються і створюються тестові випадки (тест кейси), відповідно з визначеними раніше критеріями якості і цілями тестування.

Баг/Дефект Репорт (Bug Report) – це документ, що описує ситуацію або послідовність дій, що призвела до некоректної роботи об'єкта тестування, із зазначенням причин і очікуваного результату.

Тестове Покриття (Test Coverage) – це одна з метрик оцінки якості тестування, що представляє із себе щільність покриття тестами.

Специфікація Тест Кейсів (Test Case Specification) – це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття

Час Проходження Тест Кейса (Test Case Pass Time) – це час від початку проходження кроків тест кейса до отримання результату тесту.

Проблемні тестові випадки:

– Які залежні один від одного (наприклад, очікується, що частина кроків виконана в попередньому test case, є посилання на інші test cases)

– З нечітким формулюванням кроків

– З нечітким формулюванням ідеї або очікуваного результату

Test cases можуть бути:

– Позитивні – використовуються тільки коректні дані і перевіряють, чи правильно додаток виконує функцію;

– Негативні – використовуються як коректні, так і некоректні дані (мінімум 1 некоректний параметр) і ставить за мету перевірку виняткових ситуацій (спрацьовування валідаторів), а також перевіряє, що функція не виконується при спрацьовування валідатора.

Основні стани тест кейсу:

- Created (створений)
- Modified (змінений)
- Retired (більше не дійсний)

Основні стани проходження тест кейсу:

- No run (не запущено)
- Passed (пройдено)
- Failed (помилка)
- Blocked (заблоковано)
- Not Completed (не завершено)

2. Особливості вимог програмного забезпечення

Вимоги (Requirements) до програмного забезпечення – сукупність тверджень щодо атрибутів, властивостей, або якостей програмної системи, що підлягає реалізації:

- Одиничність – Вимога описує одну і тільки одну річ.
- Завершеність – Вимога повністю визначена в одному місці і вся необхідна інформація присутня.
- Послідовність – Вимога не суперечить іншим вимогам і повністю відповідає зовнішній документації.
- Атомарність – Вимога не може бути розбита на ряд більш детальних вимог без втрати завершеності.
- Відстежування – Вимога повністю або частково відповідає діловим потребам як заявлено зацікавленими особами і задокументовано.
- Актуальність – Вимога не стала застарілою з часом.
- Здійснимість – Вимога може бути реалізовано в межах проекту.
- Недвозначність – Вимога коротко визначена без звернення до технічного жаргону та інших прихованих формулювань. Вона виражає об’єктивні факти, можлива одна і тільки одна інтерпретація. Визначення не містить нечітких фраз. Використання негативних тверджень заборонено.
- Обов’язковість – Вимога представляє певну характеристику, відсутність якої призведе до неповноцінності рішення, яка не може бути проігнорована.
- Верифікованість – Реалізованість вимоги може бути визначена через один з чотирьох можливих методів: огляд, демонстрація, тест чи аналіз..

3. Методи та фази тестування

Методи тестування:

1. Білий ящик (WhiteBox) – цей метод заснований на тому, що розробник тесту має доступ до коду програм і може писати код, який пов’язаний з бібліотеками ПЗ, що тестується.
2. Чорний ящик (Black Box) – цей метод заснований на тому, що тестувальник має доступ до ПЗ тільки через ті ж інтерфейси, що і замовник або користувач, або зовнішні інтерфейси, що дозволяють іншому комп’ютеру або іншому процесу підключитися до системи для тестування.
3. Сірий ящик (Grey Box) – поєднує елементи двох попередніх підходів.

Фази тестування:

1. Створення тестового набору (Test Suit) для конкретного середовища тестування (Testing Environment)
2. Прогон програми на тестах з отриманням протоколу результатів тестування (Test Log)

3. Оцінка результатів виконання програми на наборі тестів з метою прийняття рішення про продовження або зупинку тестування.

4. Класи еквівалентності (Equivalence class)

Підхід полягає в наступному: вхідні/вихідні дані розбиваються на класи еквівалентності, за принципом, що програма веде себе однаково з кожним представником окремого класу. Таким чином, немає необхідності тестувати всі можливі вхідні дані, необхідно перевірити по окремо взятому представнику класу.

Клас еквівалентності – це набір значень змінної, який вважається еквівалентним.

Тестові сценарії еквівалентні, якщо:

- Вони тестують одне і те ж;
- Якщо один з них знаходить помилку, то й інші виявлять її;
- Якщо один з них не знаходить помилку, то й інші не виявлять її.

Еквівалентне розбиття: Розробка тестів методом чорного ящика, в якому тестові сценарії створюються для перевірки елементів еквівалентної області. Як правило, тестові сценарії розробляються для покриття кожній області як мінімум один раз.

Приклад:

Припустимо, ми тестуємо Інтернет-магазин, який продає олівці. У замовленні необхідно вказати кількість олівців (максимум для замовлення – 1000 штук). Залежно від замовленої кількості олівців змінюється вартість:

1 – 100 – 10 грн. за олівець;

101 – 200 – 9 грн. за олівець;

201 – 300 – 8 грн. за олівець і т.д.

З кожною новою сотнею, ціна зменшується на гривню.

Якщо тестувати «в лоб», то, щоб перевірити всі можливі варіанти обробки замовленої кількості олівців, потрібно написати дуже багато тестів (згадуємо, що можна замовити аж 1000 штук), а потім ще все це і протестувати. Спробуємо застосувати розбиття на класи еквівалентності. Очевидно, що наші вхідні дані можна розділити на наступні класи еквівалентності:

Невалідне значення:> 1000 штук;

Невалідне значення: <= 0;

Валідне значення: від 1 до 100;

Валідне значення: від 101 до 200;

Валідне значення: від 201 до 300;

Валідне значення: від 301 до 400;

Валідне значення: від 401 до 500;

Валідне значення: від 501 до 600;
Валідне значення: від 601 до 700;
Валідне значення: від 701 до 800;
Валідне значення: від 801 до 900;
Валідне значення: від 901 до 1000.

На основі цих класів ми і складемо тестові сценарії. Отже, якщо взяти по одному представнику з кожного класу, то отримуємо 12 тестів.

5. Багтрекінгові системи

Система відстеження помилок (багтрекінгова система, bugtracking system) – програма, розроблена з метою допомоги розробникам ПЗ враховувати і контролювати помилки (баги), знайдені в програмах, побажаннях користувачів, а також стежити за процесом усунення цих помилок і виконання або невиконання побажань.

Головний компонент такої системи – база даних, що містить відомості про виявлені дефекти:

- Номер (ідентифікатор об'єкта)
- Дата і час, коли був виявлений дефект
- Хто повідомив про дефект
- Хто відповідальний за усунення дефекту
- Короткий опис проблеми (обов'язкове поле)
- Критичність дефекту (обов'язкове поле) і пріоритет рішення
- Опис кроків для виявлення дефекту (відтворення неправильної поведінки програми) (обов'язкове поле)
- Очікуваний результат (обов'язкове поле)
- Фактичний результат (обов'язкове поле)
- Обговорення можливих рішень і їх наслідків
- Статус дефекту
- Версія продукту, в якій дефект був знайдений
- Версія продукту, в якій дефект виправлений
- Скріншот

Серйозність (Severity) – це атрибут, що характеризує вплив дефекту на працездатність програми:

1. Блокуюча (Blocker) – розробка або використання продукту неможливо. Необхідно негайне виправлення проблеми.
2. Критична (Critical) – серйозні проблеми (не працює критичний блок функціоналу, спостерігаються серйозні помилки, пов'язані з втратою даних тощо).
3. Значна (Major) – проблема, пов'язана з важливим функціоналом продукту.
4. Незначна (Minor) – проблема пов'язана з другорядним функціоналом продукту або є легкий обхідний шлях для цієї проблеми.
5. Тривіальна (Trivial) – проблема косметичного рівня («недопрацьований» інтерфейс: друкарські помилки, різнобій з кольорами)

Пріоритет (Priority) – це атрибут, який вказує на черговість виконання завдання або усунення дефекту. Чим вище пріоритет, тим швидше потрібно виправити дефект.

1. Високий (High) – помилка повинна бути виправлена якомога швидше, тому що її наявність є критичною для проекту.
2. Середній (Medium) – помилка повинна бути виправлена, її наявність не є критичною, але вимагає обов'язкового рішення.
3. Низький (Low) – помилка повинна бути виправлена. Її наявність не є критичною, і не вимагає термінового вирішення.

Порядок виправлення помилок з їх пріоритетами: High -> Medium -> Low

Життєвий цикл дефекту

Система відслідковування помилок використовує один з варіантів «життєвого циклу» помилки, стадія якого визначається поточним станом, або статусом, в якому знаходиться помилка.

Типовий цикл дефекту (Defect cycle):

1. Новий (New) – дефект зареєстрований тестувальником.
2. Призначено (Open) – призначений відповідальний за виправлення дефекту.
3. Вирішений (Fixed) – дефект переходить назад в сферу відповідальності тестувальника. Як правило супроводжується резолюцією, наприклад:
    a. Виправлено (виправлення включені у версію таку-то),
    b. Дубль (повторює дефект, вже знаходиться в роботі),
    c. НЕ виправлено (працює відповідно до специфікації, має занадто низький пріоритет, виправлення відкладено до наступної версії і т.п.)
4. Відхилений (Rejected) – запит додаткової інформації про умови, в яких дефект проявляється, або не згоду з тим, що виявлена поведінка системи дійсно являється дефектом.
5. Відкрито повторно (Reopen) – дефект знову знайдений в іншій версії.
6. Закритий (Closed) – використовується системою для закриття вирішених завдань.

Принцип: «Де? Що? Коли?»

Де?: У якому місці інтерфейсу користувача або архітектури програмного продукту знаходиться проблема.

Що?: Що відбувається або не відбувається згідно специфікації або вашому уявленню про нормальну роботу програмного продукту. При цьому вказуйте на наявність або відсутність об'єкта проблеми, а не на його утримання (його вказують в описі).

Коли?: У який момент роботи програмного продукту, по настанню якої події або за яких умов проблема проявляється.

6. Функціональне тестування (Functional Testing). Тестування безпеки (Security and Access Control Testing). Тестування взаємодії (Interoperability Testing)

Всі види тестування програмного забезпечення, залежно від переслідуваних цілей, можна умовно розділити на наступні групи:

• Функціональні (Functional testing)
• Нефункціональні (Non-functional testing)
• Пов'язані зі змінами (Regression testing)

Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: компонентному або модульному (Component / Unit testing), інтеграційному (Integration testing), системному (System testing) і приймальному (Acceptance testing).

Функціональні види тестування розглядають зовнішню поведінку системи. Далі перераховані одні з найпоширеніших видів функціональних тестів:

- Функціональне тестування (Functional testing)
- Тестування безпеки (Security and Access Control Testing)
- Тестування взаємодії (Interoperability Testing)

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. В цілому, це тестування того, "Як" система працює. Далі перераховані основні види нефункціональних тестів:

- Всі види тестування продуктивності:
    - тестування навантаження (Performance and Load Testing)
    - стресове тестування (Stress Testing)
    - тестування стабільності або надійності (Stability / Reliability Testing)
    - об'ємне тестування (Volume Testing)
- Тестування установки (Installation testing)
- Тестування зручності користування (Usability Testing)
- Тестування на відмову і відновлення (Failover and Recovery Testing)
- Конфігураційне тестування (Configuration Testing)
- Тестування, пов'язане зі змінами

Після проведення необхідних змін, таких як виправлення бага / дефекту, програмне забезпечення повинне бути перетестоване для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

- Димове тестування (Smoke Testing)
- Регресійне тестування (Regression Testing)
- Тестування збірки (Build Verification Test)
- Санітарне тестування або перевірка узгодженості / справності (Sanity Testing)

Тестування функціональності може проводиться у двох аспектах:

• вимоги

• бізнес-процеси

Тестування в аспекті «вимоги» використовує специфікацію функціональних вимог до системи як основу для дизайну тестових випадків (Test Cases). У цьому випадку необхідно зробити список того, що буде тестуватися, а що ні, пріоритезувати вимоги на основі ризиків (якщо це не зроблено в документі з вимогами), а на основі цього пріоритезувати тестові сценарії (test cases). Це дозволить сфокусуватися і не упустити при тестуванні найбільш важливий функціонал.

Тестування в сенсі «бізнес-процеси» використовує знання цих самих бізнес-процесів, які описують сценарії щоденного використання системи. У цьому випадку тестові сценарії (test scripts), як правило, ґрунтуються на випадках використання системи (use cases).

Переваги функціонального тестування:

- імітує фактичне використання системи;

Недоліки функціонального тестування:

- можливість упущення логічних помилок у програмному забезпеченні;
- ймовірність надмірного тестування.

Досить поширеною є автоматизація функціонального тестування.

Тестування безпеки (Security and Access Control Testing) – це стратегія тестування, що використовується для перевірки безпеки системи, а також для аналізу ризиків, пов’язаних із забезпеченням цілісного підходу до захисту додатків, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних.

Загальна стратегія безпеки ґрунтується на трьох основних принципах: конфіденційність, цілісність, доступність.

Конфіденційність – це приховування певних ресурсів або інформації. Під конфіденційністю можна розуміти обмеження доступу до ресурсу деякої категорії користувачів, або іншими словами, за яких умов користувач авторизований отримати доступ до даного ресурсу.

Цілісність – Існує два основних критерії при визначенні поняття цілісності:

Довіра – Очікується, що ресурс буде змінений тільки відповідним способом певною групою користувачів.

Пошкодження і відновлення – у разі коли дані пошкоджуються або неправильно змінюються авторизованим або авторизованим користувачем, потрібновизначити наскільки важливою є процедура відновлення даних.

Доступність являє собою вимоги про те, що ресурси повинні бути доступні авторизованому користувачеві, внутрішньому об’єкту або пристрою. Як правило, чим більш критичний ресурс тим вище рівень доступності.

Тестування взаємодії (Interoperability Testing) – це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності (compatibility testing) і інтеграційне тестування (integration testing).

7. Нефункціональне тестування

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. В цілому, це тестування того, “Як” система працює. Далі перераховані основні види нефункціональних тестів:

- Всі види тестування продуктивності (Performance testing):
  - тестування навантаження (Load Testing)
  - стресове тестування (Stress Testing)
  - тестування стабільності або надійності (Stability / Reliability Testing) об'ємне тестування (Volume Testing)
- Тестування установки (Installation testing)
- Тестування зручності користування (Usability Testing)
- Тестування на відмову і відновлення (Failover and Recovery Testing)
- Конфігураційне тестування (Configuration Testing)

Тестування навантаження або тестування продуктивності – це автоматизоване тестування, що імітує роботу певної кількості бізнес користувачів на ресурсі.

Завданням тестування продуктивності є визначення масштабованості програми під навантаженням, при цьому відбувається:

- вимір часу виконання обраних операцій при певних інтенсивностях виконання цих операцій
- визначення кількості користувачів, що одночасно працюють з додатком
- визначення меж прийнятної продуктивності при збільшенні навантаження (при збільшенні інтенсивності виконання цих операцій)
- дослідження продуктивності на високих, граничних, стресових навантаженнях

Стресове тестування дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу і також оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення впливу стресу. Стресом в даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійна зміна конфігурації сервера.

Завданням тестування стабільності (надійності) є перевірка працездатності програми при тривалому (багатогодинному) тестуванні з середнім рівнем навантаження. Часи виконання операцій можуть грати в даному вигляді тестування другорядну роль. При цьому на перше місце виходить відсутність втрат пам'яті, перезапусків серверів під навантаженням та інших аспектів, що впливають саме на стабільність роботи.

Завданням об'ємного тестування є отримання оцінки продуктивності при збільшенні обсягів даних в базі даних програми, при цьому відбувається:

- вимір часу виконання обраних операцій при певних интенсивностях виконання цих операцій;
- може проводитися визначення кількості користувачів, що одночасно працюють з додатком.

Тестування Установки (Installation Testing) направлено на перевірку успішної інсталяції і настройки, а також оновлення або видалення програмного забезпечення.

Для того щоб додаток був популярним, йому мало бути функціональним – він має бути ще й зручним. Тестування зручності користування (Usability Testing) – це метод тестування, спрямований на встановлення ступеня зручності використання, зрозумілості та привабливості для користувачів розроблюваного продукту в контексті заданих умов.

Тестування зручності користування дає оцінку рівня зручності використання програми за наступними пунктами:

- продуктивність, ефективність (efficiency) – скільки часу і кроків знадобиться користувачеві для завершення основних завдань програми, наприклад, розміщення новини, реєстрації, покупка і т.д. (менше – краще);
- правильність (accuracy) – скільки помилок зробив користувач під час роботи з додатком (менше – краще);
- активізація в пам'яті (recall) – як багато користувач пам'ятає про роботу програми після призупинення роботи з ним на тривалий період часу (повторне виконання операцій після перерви повинно проходити швидше ніж у нового користувача);
- емоційна реакція (emotional response) – як користувач себе почуває після завершення завдання – розгублений, випробував стрес? Порекомендує користувач систему своїм друзям? (позитивна реакція – краще).

Тестування на відмову і відновлення (Failover and Recovery Testing) перевіряє тестований продукт з точки зору здатності протистояти і успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмовами обладнання або проблемами зв'язку (наприклад, відмова мережі). Метою даного виду тестування є перевірка систем відновлення (або дублюючих основний функціонал систем), які, у разі виникнення збоїв, забезпечать збереження і цілісність даних тестованого продукту. Методика подібного тестування полягає в симулюванні різних умов збою і наступному вивченні та оцінці реакції захисних систем. У процесі подібних перевірок з'ясовується, чи була досягнута необхідна ступінь відновлення системи після виникнення збою.

Конфігураційне тестування (Configuration Testing) – називається тестування сумісності продукту, що випускається (програмне забезпечення) з різним апаратним і програмним засобами. Основні цілі – визначення оптимальної конфігурації і перевірка сумісності програми з необхідним оточенням (обладнанням, ОС і т.д.).

8. Види тестування, пов'язані зі змінами. Кросбраузерність

Після проведення необхідних змін, таких як виправлення бага / дефекту, програмне забезпечення повинне бути перетестоване для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

- Регресійне тестування (Regression Testing)
- Димове тестування (Smoke Testing)
- Санітарне тестування або перевірка узгодженості / справності (Sanity Testing)
- Тестування збірки (Build Verification Test)

Регресійне тестування (Regression Testing) – це вид тестування спрямований на перевірку змін, зроблених в додатку або середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб сервер або сервер додатки), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше.

Регресійними можуть бути як функціональні, так і нефункціональні тести.

Як правило, для регресійного тестування використовуються тест кейси, написані на ранніх стадіях розробки і тестування. Це дає гарантію того, що зміни в новій версії програми не пошкодили вже існуючу функціональність.

Рекомендується робити автоматизацію регресійних тестів, для прискорення подальшого процесу тестування і виявлення дефектів на ранніх стадіях розробки програмного забезпечення. Сам по собі термін "Регресійне тестування", залежно від контексту використання може мати різний зміст. Сем Канер, наприклад, описав 3 основних типи регресійного тестування:

- Регресія багів (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена.
- Регресія старих багів (Old bugs regression) – спроба довести, що недавня зміна коду чи даних зламала виправлення старих помилок, тобто старі баги стали знову відтворюватися.
- Регресія побічного ефекту (Side effect regression) – спроба довести, що недавня зміна коду чи даних зламала інші частини продукту.

Димове тестування (Smoke Testing)

Поняття пішло з інженерної середовища: "При введенні в експлуатацію нового обладнання вважалося, що тестування пройшло вдало, якщо з установки не пішов дим." В області ж тестування програмного забезпечення, воно застосовується для поверхневої перевірки всіх модулів програми на предмет працездатності і наявності швидкого знаходження критичних і блокуючих дефектів.

Санітарне тестування (Sanity Testing)

Це вузьконаправлене тестування, достатнє для доказу того, що конкретна функція працює згідно заявленим в специфікації вимогам. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін вироблених в ній або навколишньому середовищі. Зазвичай виконується вручну.

Тестування збірки (Build Verification Test)

Тестування спрямоване на визначення відповідності, випущеної версії, критеріям якості для початку тестування. За своїми цілями є аналогом димового тестування, спрямованого на приймання нової версії в подальше тестування або експлуатацію. Вглиб воно може проникати далі, залежно від вимог до якості випущеної версії.

Відмінність санітарного тестування від димового (Sanity vs Smoke testing)

У деяких джерелах помилково вважають, що санітарне та димове тестування – це одне і теж. Ми ж вважаємо, що ці види тестування мають "вектори руху", що спрямовані в різні боки. На відміну від димового (Smoke testing), санітарне тестування (Sanity testing) направлено вглиб функції, що перевіряється, в той час як димове направлено вшир, для покриття тестами якомога більшого функціоналу в найкоротші терміни.

Кросбраузерність (Cross-browser) – властивість сайту відображатися і працювати у всіх популярних браузерах ідентично. Під ідентичністю розуміється відсутність недоліків верстки і здатність відображати матеріал з однаковим ступенем читабельності.

Тестування сайту на кроссбраузерность необхідно проводити:

У різних браузерах (сімейство Mozila, Internet Explorer, Opera, Safari, Мобільні браузери)

При різних розширеннях екрану (зазвичай 640 * 480, 800 * 600, 1024 * 768, 1280 * 800, 1280 * 1024, 1366 * 768)

В різних операційних системах (Mac OS, Linux, Win)

III. Завдання на лабораторну роботу

1. Зареєструйтеся у TestRail (або іншій аналогічній системі): https://www.gurock.com/testrail/
2. Напишіть 5 Test Case на п'ять будь-яких функцій для подальшої перевірки продукту.
3. Зареєструйся у Jira (або іншій аналогічній системі).
4. Заповніть баг репорти на підставі тест кейсів.

IV. Контрольні питання

1. Що таке тестування програмного забезпечення?
2. Які види тестування програмного забезпечення ви знаєте?
3. Які основні принципи тестування програмного забезпечення?
4. Що таке багтрекінгова система?
5. Які основні функції багтрекінгової системи?
6. Які типи багів можна відстежувати в багтрекінговій системі?
7. Які є переваги використання багтрекінгової системи для проектів розробки ПЗ?
8. Які принципи керування процесом відстеження багів в багтрекінговій системі ви знаєте?
9. Які види тестування програмного забезпечення ви знаєте?
10. Що таке модульне тестування і для чого воно використовується?
11. Які підходи використовуються для тестування взаємодії програмного забезпечення зі зовнішніми системами?
12. Які основні принципи проведення тестування регресії?
13. Які методики тестування використовуються для перевірки відповідності вимогам функціональності програмного забезпечення?

Перелік літератури

1. Pro GIT/Вступ/Про контроль версій. [Електронний ресурс]. Режим доступу: https://uk.wikibooks.org/wiki/Pro_GIT/%D0%92%D1%81%D1%82%D1%83%D0%BF/%D0%9F%D1%80%D0%BE_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8C_%D0%B2%D0%B5%D1%80%D1%81%D1%96%D0%B9.

2. Системи контролю версій на прикладі Git. [Електронний ресурс]. Режим доступу: https://dut.edu.ua/ua/news-1-626-9170-sistemi-kontrolyu-versiy-na-prikladi-git_kafedra-kompyuternih-nauk-ta-informaciynih-tehnologiy.

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

4. Скрам – що це таке та як тим користуватися. [Електронний ресурс]. Режим доступу: https://brander.ua/blog/skram-shcho-tse-take-ta-yak-tsym-korystuvatysya.

5. Дослідження процесу управління ризиками в невеликих проектах зі створення програмного забезпечення / Галенко Н. І. // Наукові праці [Чорноморського державного університету імені Петра Могили]. Сер. : Комп’ютерні технології. - 2008. - Т. 90, Вип. 77. - С. 69-79. - Режим доступу: http://nbuv.gov.ua/UJRN/Npchduct_2008_90_77_10.

6. Лекційні матеріали з теорії тестування програмного забезпечення. [Електронний ресурс]. Режим доступу: https://qalearning.com.ua/category/theory/lectures/material/.

7. Sutherland J. Scrum. The Art of Doing Twice the Work in Half the Time / Jeffrey Sutherland., 2015.

8. Rigby D. Doing Agile Right: Transformation Without Chaos / D. Rigby, S. Elk, S. Berez., 2020.

9. Schwaber K. The Scrum Guide / K. Schwaber, J. Sutherland., 2017.

10. Derby E. Agile Retrospectives: Making Good Teams Great 1st Edition / E. Derby, D. Larsen., 2006.

11. GIT Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://git-scm.com/doc.

12. How to Write a Software Requirements Specification (SRS Document) [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.perforce.com/blog/alm/how-write-software-requirements-specification-srs-document.