

Міністерство освіти і науки України

Львівський національний університет

імені Івана Франка

Кафедра Оптоелектроніки та інформаційних технологій

ЗБІРНИК ЛАБОРАТОРНИХ РОБІТ

З ДИСЦИПЛІНИ

**“ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. УПРАВЛІННЯ ІТ
ПРОЕКТАМИ ”**

**ДЛЯ СТУДЕНТІВ СПЕЦІАЛЬНОСТІ 121
«ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»**

Львів 2024

Збірник лабораторних робіт з дисципліни «Життєвий цикл ПЗ. Управління ІТ проектами» для студентів спеціальності 121 «Інженерія програмного забезпечення». – Львів: ЛьвівНТУ, 2024. – 61 с.

Укладач: Франів В.А., к.ф.м.н., доцент кафедри оптоелектроніки та інформаційних технологій

Відповідальний за випуск: О.С. Кушнір, завідувач кафедри оптоелектроніки та інформаційних технологій, доктор фізико-математичних наук, професор

Рецензенти: М.В. Лобур, д.т.н., професор.;

Затверджено науково-
методичною радою
університету
Протокол № ? від квітня 2024 р.

На правах рукопису

ЗМІСТ

ЛАБОРАТОРНА РОБОТА №1 «СТВОРЕННЯ ПРОЄКТ BASELINE ДЛЯ ІТ ПРОДУКТУ»	4
ЛАБОРАТОРНА РОБОТА №2 «ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ WATERFALL»	11
ЛАБОРАТОРНА РОБОТА №3 «ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ SCRUM»	15
ЛАБОРАТОРНА РОБОТА №4 «ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ KANBAN»	25
ЛАБОРАТОРНА РОБОТА №5 «ВЕРСІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ЛОКАЛЬНА СИСТЕМА КОНТРОЛЮ ВЕРСІЙ.»	34
ЛАБОРАТОРНА РОБОТА №6 «ВЕРСІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ДЕЦЕНТРАЛІЗОВАНА СИСТЕМА КОНТРОЛЮ ВЕРСІЙ.»	44
ЛАБОРАТОРНА РОБОТА №7 «НЕПЕРВНА ІНТЕГРАЦІЯ ТА НЕПЕРЕРВНА ДОСТАВКА (CI/CD). ВПРОВАДЖЕННЯ CI/CD ВИКОРИСТОВУЮЧИ ПРОГРАМНИЙ ПАКЕТ JENKINS»	54
ЛАБОРАТОРНА РОБОТА №8 «НЕПЕРВНА ІНТЕГРАЦІЯ ТА НЕПЕРЕРВНА ДОСТАВКА (CI/CD). ВПРОВАДЖЕННЯ CI/CD ВИКОРИСТОВУЮЧИ ВЕБ СЕРВІС GITHUB ACTIONS WORKFLOW»	58
ЛАБОРАТОРНА РОБОТА №9 «ВЕДЕННЯ ДОКУМЕНТАЦІЯ У ІТ ПРОЕКТІ »	60

ЛАБОРАТОРНА РОБОТА №1

«СТВОРЕННЯ PROJECT BASELINE ДЛЯ ІТ ПРОДУКТУ»

Мета роботи:

Отримати розуміння та практичних навичок у створенні та управлінні Project Baseline (PB) – вихідним документом для ІТ проекту. Оволодіти інструментами для ефективного планування та контролю за проектом, враховуючи ключові аспекти, такі як обсяг робіт, вимоги, ресурси та бюджет. Важливий акцент робиться на здатності аналізувати та адаптувати Project Baseline до змін в ході проекту, розглядаючи його як ключовий інструмент управління змінами та ризиками.

Методичні вказівки

Робота спрямована на розвиток навичок управління проектами та забезпечення стабільності ІТ продукту протягом його життєвого циклу.

Вимоги до результатів виконання лабораторної роботи:

1. Наявність сформованого PB для вибраного ІТ проекту.
2. Проведений аналіз усіх ключових частин PB.
3. Наявність висновку що містить рекомендації стосовно майбутнього розвитку PB.

Теоретичні відомості:

Project baseline – це основний документ що визначає стан проекту на початку виконання. Включає ключові параметри: обсяг робіт(scope) та вимоги, графік виконання(schedule), ресурси та бюджет (budget).

Обсяг робіт та вимоги. Це невідемна части PB яка складається з короткого опису ідеї ІТ проекту, переліку основних елементів розробки. Типово основні елементи розробки включають в себе:

- Розробка дизайну (UI/UX)
- Розробка поверхневої архітектури проекту.
- Розробка ключових компонентів проекту.
- Підготовка інфраструктури для постійного впровадження проекту(CI/CD).

Розробка дизайну (UI/UX). UI (User Interface) - Користувацький Інтерфейс, це те, як користувач взаємодіє з програмою або системою через графічний інтерфейс. Елементи UI включають кнопки, поля введення, меню та графічні об'єкти. Добре розроблений UI робить взаємодію з продуктом простішою та інтуїтивно зрозумілою, що покращує загальний досвід користувача. Команда що працює над розробкою UI зазвичай включає: дизайнера інтерфейсу (UI Designer), розробника інтерфейсу (UI Developer).

Приклад з реального життя: меню в мобільному додатку Uber - просте та зрозуміле для замовлення послуги. UX (User Experience) - Користувацький досвід, це враження та емоції користувача від взаємодії з продуктом або системою. Забезпечення позитивного UX забезпечує лояльність користувачів та покращує репутацію продукту. Технології що використовуються: Аналітика користувацької поведінки, A/B-тестування, опитування користувачів. Команда: Дослідник користувацького досвіду (UX Researcher), дизайнер взаємодії (Interaction Designer). Приклад з реального життя: Процес оформлення замовлення в онлайн-магазині Amazon - швидкий та інтуїтивно зрозумілий. Добре розроблений UI/UX дозволяє забезпечити ефективну та задовільну взаємодію з користувачем.

Короткий перелік технологій та інструментів які використовуються:

Завдання	Технології
Дизайн UI	Figma, Adobe XD, Sketch
Розробка UI	HTML, CSS, JavaScript, React, Vue.js.
Аналітика UX	Google Analytics, Hotjar, UsabilityHub

Приклади з Реального Життя:

☑ Apple: Стильний та ергономічний дизайн продуктів (UI) та простий процес взаємодії (UX).

☑ Spotify: Інтуїтивно зрозумілий інтерфейс (UI) та персоналізований вміст, що покращує взаємодію (UX).

☑ Google Maps: Простий та зручний UI, а також інформативний UX для навігації та пошуку місць.

Узгоджена розробка UI/UX є ключовою частиною Project Baseline, оскільки це визначає спосіб, яким користувачі будуть взаємодіяти з продуктом, і впливає на його успіх на ринку.

Розробка поверхневої архітектури проекту. На цьому етапі здійснюється створення архітектури проекту без деталізації. Визначаються основні компоненти системи, оцінюються потенційно затребувані апаратні ресурси (у більшості випадків на сьогоднішній день оцінюється хмарні сервіси для проекту).

Учасники та їх ролі: Архітектор системи (System Architect). Архітектор системи відповідає за визначення загальної архітектури проекту та вибір стратегії розробки. Визначення основних компонентів, взаємозв'язків та структури системи. Врахування аспектів масштабованості та продуктивності. Архітектор баз даних (Database Architect), його функція зазвичай полягає у визначенні структури бази даних та взаємозв'язків між різними сутностями. Розробка схеми бази даних, визначення типів даних, розробка стратегії забезпечення цілісності даних. Технічний лідер

(Technical Lead), відповідає за вибір технічних рішень та інструментів для реалізації архітектури. Вибір технологій, розробка стратегії інтеграції, оцінка та вибір зовнішніх сервісів. Часто заради економії ресурсів усю вищеперелічену роботу виконує тільки один спеціаліст, наприклад: System Architect.

Фінальним документом цього етапу зазвичай являється Component UML Diagram. Діаграма компонентів UML (Unified Modeling Language) - це вид діаграми, який використовується для візуалізації та опису структури високорівневих компонентів в системі програмного забезпечення. Ця діаграма вказує на те, як компоненти взаємодіють один з одним в рамках системи.

Основні елементи діаграми компонентів UML:

1. Компоненти:
 - Представляють фізичні чи логічні модулі програми.
 - Можуть бути бібліотеками, виконавчими файлами, пакетами коду, DLL тощо.
2. Інтерфейси:
 - Вказують, як інші компоненти можуть взаємодіяти з даним компонентом.
 - Подібно до контрактів чи API.
3. Залежності:
 - Показують, як один компонент використовує послуги іншого компонента.
4. Артефакти:
 - Представляють фізичні або логічні об'єкти, які створюють, обробляють чи зберігають інші компоненти.
5. Порти:
 - Вказують місце взаємодії між компонентами або інтерфейсами.

Часто вживаний інструмент для розробки Component diagram: Draw.io.

Розробка ключових компонентів проекту. На цьому етапі здійснюється детальне проектування кожної компоненти. Розробка програмного коду. Створення технічної документації. В продовж цього етапу основну роль відіграє команда розробників (development team). Команда розробників зазвичай налічує 7 осіб: інженери програмісти (software engineers, tech lead), інженери тестувальники програмного забезпечення (manual and automation quality assurance engineers) та інші члени команди в залежності від обраного підходу до розробки програмного забезпечення (product owner, business analytic, scrum master). Цей етап займає лівову частину часу закладену на розробку проекту. Якщо проект являється успішним і кожного разу знаходяться інвестиції то етап розробки

компонентів проекту повторюється циклічно, появляється потреба у новому функціоналі продукту (new features) а це у свою чергу вимагає створення нових компонент і подальшого написання програмного коду та його тесування.

Підготовка інфраструктури для постійного впровадження проекту(CI/CD). Цей етап є надзвичайно можливим оскільки він відповідає з розгортання проекту на серверах (deploymet) та забезпечення програмістів системою контролю версій, інструментами для автоматичної перевірки якості коду.

CI/CD в рамках життєвого циклу програмного забезпечення (ЖЦПЗ) відноситься до двох взаємопов'язаних практик: Continuous Integration (CI) і Continuous Delivery (CD). Обидві практики спрямовані на полегшення та автоматизацію процесів розробки, тестування та впровадження програмного забезпечення.

1. Continuous Integration (CI):

- Основна мета: Інтеграція коду розробників в основний репозиторій регулярно і автоматично.
- Як працює: Розробники регулярно зливають свій код в спільний репозиторій. Після кожного злиття проводяться автоматизовані тести, перевірки та збірка проекту для виявлення можливих конфліктів та помилок.
- Переваги: Зменшуємо ризик виникнення конфліктів між кодом різних розробників. Швидше виявляються та виправляються помилки.

2. Continuous Delivery (CD):

- Основна мета: Забезпечення того, що програмний продукт може бути автоматично розгорнутий у будь-якому середовищі (тестовому, випробувальному, виробничому) за необхідності.
- Як працює: Після успішного завершення CI і вирішення всіх проблем, програмний код готується до автоматичного впровадження. Це може включати в себе створення збірок, створення образів контейнерів та інші підготовчі етапи для автоматичного розгортання.
- Переваги: Забезпечуємо швидке та безпечне впровадження нового функціоналу або виправлень без ручного втручання.

Інтеграція Continuous Integration та Continuous Delivery дозволяє розробникам та командам забезпечувати швидке та надійне розгортання нового коду в будь-якому середовищі. Це сприяє автоматизації та зменшенню ризиків при розробці програмного забезпечення.

На сьогоднішній день існує великий перелік сервісів які забезпечують CI/CD процес ось деякі з них: GitHub, GitLab, Bitbucket. Варто також відмітити що GitHub окрім CI/CD полегшує процес написання коду за

рахунок впровадження технологій штучного інтелекту у їхній сервіс GitHub Copilot.

Графік виконання (schedule). У Project Baseline детально визначає послідовність та тривалість робіт, які потрібно виконати для успішного завершення проекту. Цей елемент включає у себе часові рамки для кожного завдання чи етапу (дивись **Обсяг робіт та вимоги**), що допомагає визначити, коли саме будуть виконуватися конкретні дії та яким чином розподілений час між різними фазами проекту.

Ключові аспекти графіка виконання:

1. Завдання та етапи:
 - Визначення конкретних завдань та етапів, які потрібно виконати в рамках проекту.
 - Кожне завдання пов'язане з конкретною роботою чи діяльністю.
2. Часові терміни:
 - Визначення тривалості кожного завдання чи етапу в днях, тижнях чи інших одиницях виміру.
 - Якщо є залежності між завданнями, це враховується для правильного визначення графіка.
3. Лінійний графік:
 - Створення лінійного графіка, який відображає послідовність виконання завдань та етапів протягом часу.
 - Візуальне представлення того, як різні елементи проекту взаємодіють та впливають один на одного.
4. Критичний шлях:
 - Визначення критичного шляху, який включає найбільш критичні за часом завдання, що визначають загальний тривалість проекту.
 - Ідентифікація та управління завданнями, які можуть впливати на завершення проекту у визначений термін.
5. Залежності між завданнями:
 - Визначення, які завдання повинні бути завершені перед початком інших.
 - Визначення послідовності та взаємозалежності завдань.

Ресурси та бюджет (budget). Ця частина визначає та управляє ресурсами, які будуть використовуватися під час проекту, а також бюджетом, що буде витрачений на реалізацію проекту. Цей розділ грає важливу роль у плануванні та виконанні проекту, забезпечуючи ефективне розподіл ресурсів та використання бюджетних коштів.

Елементи, які включаються до розділу "Ресурси та бюджет":

1. Людські ресурси (Human Resources):
 - Визначення кількості та типу спеціалістів, які будуть залучені до проекту.

- Розподіл ролей та відповідальностей серед членів команди.
 - Розрахунок часу, який кожен член команди відділить на проект.
2. Фінансові ресурси (Financial Resources):
 - Визначення бюджету, доступного для проекту.
 - Розподіл бюджету між різними витратами та етапами проекту.
 - Розрахунок вартості ресурсів, необхідних для виконання завдань.
 3. Матеріальні ресурси (Material Resources):
 - Визначення необхідних матеріалів та обладнання для виконання робіт.
 - Розрахунок витрат на закупівлю та використання матеріалів.
 4. Технічні ресурси (Technical Resources):
 - Визначення необхідного програмного забезпечення, апаратного забезпечення та інших технічних ресурсів.
 - Забезпечення доступності необхідних інструментів для реалізації проекту.
 5. Розподіл ресурсів в часі (Resource Allocation):
 - Визначення, як ресурси будуть розподілятися в часі в рамках графіка виконання проекту.
 - Розрахунок завантаженості ресурсів та уникнення перевантаження.
 6. Контроль за бюджетом (Budget Control):
 - Встановлення механізмів моніторингу та контролю за витратами та використанням бюджету.
 - Виявлення та вирішення відхилень від планованих бюджетних показників.
 7. Резерв ресурсів та бюджету (Resource and Budget Reserve):
 - Врахування можливих ризиків та встановлення резервів ресурсів та бюджету для управління неочікуваними ситуаціями чи змінами в проекті.

Project baseline не має єдиного шаблонного вигляду і може різнитися в залежності від практик різних компаній. У більшості випадків цей документ формують у вигляді презентації. Презентація повинна бути гарно оформлена оскільки її презентують потенційному замовнику, інвестору або клієнту. Project baseline являється базовим документом на основі якого в подальшому після переговорів формується комерційна пропозиція і контракт.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.

2. Створити Project Baseline для існуючого або потенційного проекту. Project Baseline повинен містити щонайменше 3 розділи: обсяг робіт(scope) та вимоги, графік виконання(schedule), ресурси та бюджет (budget).
3. Провести аналіз кожного розділу, з метою його максимальної оптимізації.
4. Project Baseline пропонується оформити у виді презентації яка буде являти собою звіт про виконання лабораторної роботи.
5. Здійснити презентацію створеного Project Baseline.

Зміст звіту:

1. Для даної лабораторної роботи в якості звіту приймається створений Project Baseline.

Література

1. Wysocki, Robert K. Effective Project Management: Traditional, Agile, Extreme. Wiley, 2019, ISBN: 978-1119562801.
2. Kogon, Kory; Covey, Suzanne; Wood, James. Project Management for the Unofficial Project Manager: A Franklin Covey Title. BenBella Books, 2015, ISBN: 978-1941631102.
3. Horine, Greg. Project Management Absolute Beginner's Guide. Que Publishing, 2017, ISBN: 978-0789759002.
4. Berkun, Scott. The Art of Project Management. O'Reilly Media, 2005, ISBN: 978-0596102344.

ЛАБОРАТОРНА РОБОТА №2

«ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ WATER-FALL»

Мета роботи:

Вивчити та зрозуміти основні принципи та етапи методології розробки Waterfall. Оволодіти ключовими концепціями цього підходу, включаючи послідовність етапів, взаємодію між ними та роль кожного етапу в життєвому циклі розробки програмного забезпечення. Метою є також забезпечення студента навичками використання методології Waterfall для ефективного управління проектами та розробки ІТ-продуктів в умовах фіксованих вимог і чітко визначених етапів виконання робіт.

Методичні вказівки

Робота спрямована на розвиток навичок успішної розробки програмного забезпечення при використанні підходу до розробки програмного забезпечення Waterfall.

Вимоги до результатів виконання лабораторної роботи:

1. Розробити проект використовуючи підхід Waterfall.
2. Описати ключові стадії розробки проекту.
3. Проаналізувати плюси та мінуси підходу до розробки Waterfall базуючись на отриманому досвіді. При наявності ідей для покращення роботи підходу навести ці ідеї

Теоретичні відомості:

Waterfall (Каскад) - це традиційна методологія розробки програмного забезпечення, яка передбачає послідовний перехід від одного етапу до іншого без можливості повернення до попереднього етапу після його завершення.

Методологію Waterfall (водоспад, або каскад) вперше описав інженер у галузі програмного забезпечення і вчений Уїнстон Ройс (Winston W. Royce) у своєму документі під назвою "Managing the Development of Large Software Systems" ("Управління розробкою великих програмних систем"), який був опублікований у журналі "Proceedings of IEEE" в 1970 році.

Варто відмітити, що Ройс в своєму оригінальному документі вказав на важливість виконання певних етапів у процесі розробки програмного забезпечення, але також висловив обурення стосовно того, що ця модель може привести до недоліків та складнощів. Він зауважував, що цей підхід може працювати лише тоді, коли всі етапи визначаються ідеально та вичерпно перед початком проекту, що, за його словами, є малоімовірним.

Історія Waterfall вказує на те, що Ройс сам не підтримував цю методологію як ідеальний підхід і вказав на її обмеження. Тим не менш, модель Waterfall стала основою для подальшого розвитку методологій управління проектами та розробки програмного забезпечення. Її переваги та недоліки розглядалися і критикувалися, що сприяло виникненню інших, більш гнучких підходів, таких як Agile та Scrum.

Етапи розробки за методологією Waterfall:

1. Оцінка та Планування (Requirements):
 - Вимоги: Збір і деталізація вимог до програмного забезпечення.
 - Планування: Визначення обсягу робіт, визначення ресурсів та часових рамок.
2. Проектування (Design):
 - Архітектура: Розробка загальної архітектури системи.
 - Дизайн інтерфейсу: Визначення вигляду та взаємодії користувача з програмою.
3. Реалізація (Implementation):
 - Вибір технологій: Вибір інструментів та технологій для реалізації проекту.
 - Програмування: Створення коду та реалізація функціональності.
4. Тестування (Testing): Проведення різновидів тестування для перевірки якості та стабільності програмного забезпечення.
5. Впровадження (Deployment): Реліз та впровадження програмного продукту в робоче середовище.

Як і кожна методологія розробки програмного забезпечення Waterfall має свої плюси та мінус. До плюсів відносять: простота управління - чітко визначені етапи полегшують управління проектом, фіксовані вимоги - вимоги фіксуються на ранніх етапах, що зменшує ризики змін під час розробки. До недоліків: зміни вимог після початку реалізації можуть бути важкими та дорогими, помітна затримка у випуску готового продукту через послідовний характер робіт.

Waterfall був доволі популярною методикою розробки не тільки проектів пов'язаних з ІТ, цей підхід широко використовувався у різних галузях, наприклад:

- NASA's Pioneer Program (1958-1978): Перші космічні місії, такі як Pioneer 1, використовували Waterfall для чіткої організації та керування проектом.
- Система банківського обліку та оплати:
 - Роки використання: 1990-2000.
 - Відгук: Проект був успішно впроваджений за розкладом та бюджетом, завдяки заздалегідь визначеним вимогам.

Не зважаючи на свою глибоку історію підхід до розробки програмного забезпечення Waterfall не втратив свою актуальність і широко використовується у сьогоденні.

Призначення методології Waterfall:

- Проекти з фіксованими вимогами: Де чітко визначені та стабільні вимоги, наприклад, створення програмного забезпечення для банківської системи.
- Проекти з низьким ризиком змін: Де ймовірність змін вимог протягом розробки невелика.
- Проекти з мінімальними технічними ризиками: Де технічні аспекти проекту стабільні та не вимагають експериментів.

Відповідальними за належне впровадження підходу waterfall підчас виконання проекту є проект менеджер (Project manager) або Engineering manager. Існує широке різноманіття інструментів для ефективного керування та контролю над проектом. Ось деякі типові інструменти, які вони можуть використовувати при методології waterfall:

1. **Microsoft Project:** Це один з найпоширеніших інструментів для керування проектами. Він дозволяє створювати графіки, визначати завдання, розподіляти ресурси та відстежувати прогрес проекту.
2. **Jira:** Цей інструмент широко використовується для керування завданнями, відстеження прогресу, організації спринтів та управління версіями.
3. **Asana:** Це інструмент для управління завданнями та проектами. Він дозволяє створювати завдання, встановлювати терміни та спостерігати за прогресом.
4. **Trello:** Це онлайн-інструмент для керування проектами за допомогою дошок та карток. Він часто використовується для візуалізації завдань та етапів проекту.
5. **Smartsheet:** Це інструмент для керування проектами, який об'єднує аркуші Excel з функціоналом керування завданнями та графіками.
6. **Basecamp:** Це веб-сервіс для спільної роботи над проектами, який включає в себе завдання, графіки, обговорення та інше.
7. **Confluence:** Це інструмент для спільної роботи та документування проектів, часто використовується в поєднанні з Jira.

Методологія Waterfall показує себе ефективно в проектах, де можливо заздалегідь фіксувати вимоги та мінімізувати ризики їх змін. Проекти, які передбачають стабільність у вимогах та низький ризик змін, такі як розробка банківської системи, можуть здобути від Waterfall підходу. Однак, варто розглядати більш гнучкі методології для проектів, які вимагають великої адаптабельності та можливості швидко реагувати на зміни.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Розробити ІТ проект використовуючи підхід до розробки програмного забезпечення Waterfall.
3. Здійснити детальний аналіз мінімум трьох етапів: планування та вимоги, розробка, тестування.
4. Для кожного з етапів підготувати належну документацію.
5. Провести аналіз сильних та слабких сторін підходу до розробки програмного забезпечення waterfall спираючись на отриманий досвід.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (опис проекту який створювався використовуючи підхід до розробки програмного забезпечення waterfall).
4. Висновок (у висновку обов'язково зазначити Ваші думки стосовно недоліків та сильних сторін даного підходу до розробки програмного забезпечення).
5. Список використаної літератури.

Література

1. Royce, Winston W. Managing the Development of Large Software Systems. Technical Papers of Western Electronic Show and Convention. 1970. p 1-9.
2. Rubin, Kenneth S. "Agile Project Management with Scrum" : O'Reilly Media, 2001. 192p.
Waterfall vs. Agile: Which is the Right Development Methodology for Your Project? URL: <https://www.seguetech.com/waterfall-vs-agile-methodology/>
3. Critical Success Factors in Software Projects. URL: <https://www.zar-tis.com/7-critical-success-factors-for-software-development/>

ЛАБОРАТОРНА РОБОТА №3

«ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ SCRUM»

Мета роботи:

Ознайомлення з підходом до розробки програмного забезпечення Scrum, вивчення його основних принципів та етапів. Набуття ключових концепцій Agile в імplementації Scrum, включаючи ітераційність, гнучкість та співпрацю, з метою здобуття навичок ефективного використання цього підходу у проектній діяльності.

Методичні вказівки

Ця робота спрямована на формування у студента розуміння гнучких методологій розробки програмного забезпечення та їхнього впливу на успішність проектів.

Вимоги до результатів виконання лабораторної роботи:

1. Розробити проект використовуючи підхід AGILE (SCRUM).
2. Описати ключові стадії розробки проекту.
3. Проаналізувати плюси та мінуси підходу до розробки програмного забезпечення базуючись на отриманому досвіді. При наявності ідей для покращення роботи підходу навести ці ідеї.

Теоретичні відомості:

Agile виник як відповідь на проблеми традиційних методологій, таких як Waterfall, які виявилися неефективними в умовах швидкозмінюваних вимог та невизначених умов проекту. Основні ідеї Agile виникли в середині 20-го століття, але справжній прогрес стався в 2001 році, коли було створено "Маніфест Agile". Цей документ визначив основні принципи гнучких методологій розробки та визначив чотири цінності Agile: індивідуальність та взаємодія, робочий продукт, співпраця з клієнтом та реагування на зміни (див. таблиця ЛЗ.1).

Таблиця ЛЗ.1 Основні принципи Agile (Agile Manifesto)

Індивідуальність та взаємодія (Individuals and Interactions)	Цінність підкреслює важливість людей у процесі розробки програмного забезпечення. Відповідно до принципів Agile, комунікація між членами команди та взаємодія між ними мають вищий пріоритет, ніж інструменти чи процеси. Успішні проекти будуються на сильних міжособистісних відносинах, взаєморозумінні та взаємодопомозі у команді. Колективність та ефективність команди мають визначати успіх.
--	--

Робочий продукт (Working Software)	Цінність акцентує на створенні функціонального та конструктивного програмного забезпечення як головного критерію успіху. Замість концентрації над процесами або документами, Agile покликаний акцентувати увагу на результаті роботи - робочому програмному продукті. Сутність цієї цінності полягає в тому, щоб швидко та регулярно випускати робочі версії продукту, отримуючи зворотний зв'язок від користувачів та пристосовуючи продукт до їхніх потреб (customer feedback).
Співпраця з клієнтом (Customer Collaboration)	Agile визнає важливість активної співпраці з клієнтом на протязі всього процесу розробки. Замість традиційного підходу, коли вимоги фіксуються наперед, Agile підтримує постійний обмін інформацією та зворотний зв'язок (customer feedback) з клієнтом. Співпраця замість контракту означає, що розробка може адаптуватися до змін у вимогах та забезпечити виробництво того, що справді необхідно клієнту.
Реагування на зміни (Responding to Change)	Цінність ставить у центр уваги гнучкість та здатність до адаптації. Agile визнає, що умови проекту та вимоги можуть змінюватися з часом, і успішна команда повинна бути готовою до реагування на ці зміни. Замість строгого дотримання плану, Agile підтримує ітераційний та інкрементальний підхід, що дозволяє командам швидко адаптуватися до нових умов та вимог.

Скрам - це гнучка методологія управління проектами, яка допомагає командам структурувати та управляти своєю роботою за допомогою набору цінностей, принципів та практик. Схоже на команду з регбі (звідси його назва), яка готується до важливого матчу, скрам сприяє вивченню командами на основі власних досвідів, самоорганізації при роботі над проблемою та рефлексії над своїми перемогами та поразками для постійного вдосконалення.

Хоча скрам, про який я говорю, найчастіше використовується командами розробників програмного забезпечення, його принципи та уроки

можна застосовувати до всіх видів командної роботи. Це одна з причин популярності скраму. Часто розглядається як гнучка методологія управління проектами, скрам описує набір зустрічей, інструментів та ролей, які спільно допомагають командам структурувати та управляти своєю роботою.

Люди часто вважають, що Scrum і Agile - це одне й те саме, оскільки Scrum спрямований на постійне удосконалення, що є основним принципом Agile. Однак Scrum - це фреймворк для виконання роботи, тоді як Agile - це філософія. Філософія Agile спрямована на постійне інкрементальне удосконалення через малі та часті релізи. Ви не можете дійсно "перейти на Agile", оскільки це вимагає відданості всієї команди змінювати свій підхід до того, як вони думають про постачання значущості вашим клієнтам. Але ви можете використовувати фреймворк, такий як Scrum, щоб допомогти вам почати думати в такий спосіб та впроваджувати принципи Agile у вашу щоденну комунікацію та роботу.

Відмінність між Agile та визначенням Scrum можна знайти в Scrum Guide та Agile Manifesto . Agile Manifesto визначає чотири цінності (див. таблиця ЛЗ.1)

Визначення Scrum базується на емпіризмі та принципах Lean. Емпіризм стверджує, що знання приходить з досвіду, і рішення приймаються на основі спостережень. Lean thinking зменшує витрати і фокусується на основних аспектах. Фреймворк Scrum базується на постійному навчанні та адаптації до змінливих факторів. Він визнає, що команда не знає всього на початку проекту і буде розвиватися через досвід. Scrum структурований так, щоб допомогти командам природно адаптуватися до змінних умов та вимог користувачів, з вбудованою можливістю перепріоритизації у процесі та короткими циклами випуску, щоб ваша команда могла постійно вчитися та удосконалюватися (рис ЛЗ.1).

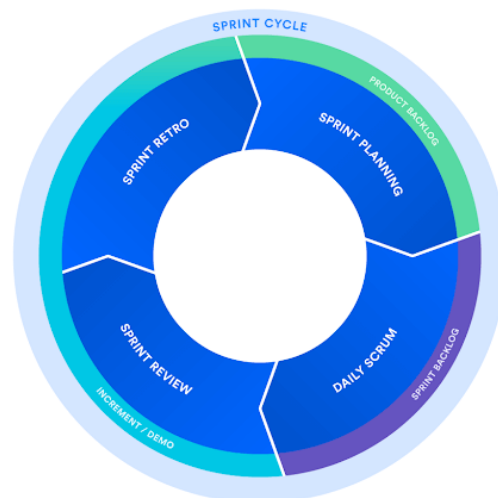


Рис ЛЗ.1 Spring cycle.

Хоча Scrum має структуровану форму, він не є абсолютно жорстким. Його виконання може бути адаптоване до потреб будь-якої організації. Існують багато теорій про те, як саме повинні працювати команди Scrum для досягнення успіху. Однак після більш ніж десятирічного досвіду допомоги командам Agile в Atlassian вивчили, що чітка комунікація, прозорість та відданість постійному вдосконаленню повинні завжди залишатися в центрі будь-якої обраної вами структури. А решта залежить від вас.

Фреймворк Scrum визначає набір цінностей, принципів та практик, які слідує команда Scrum для постачання продукту чи послуги. Він деталізує члени команди Scrum та їх відповідальності, "артефакти", які визначають продукт та працюють над його створенням, і церемонії Scrum, які керують командою Scrum під час роботи.

Команда Scrum - це невелика та гнучка група, присвячена постачанню зобов'язаних інкрементів продукту. Розмір команди Scrum зазвичай невеликий, приблизно 10 осіб, але великий достатньо для виконання значної кількості роботи протягом спринта. Команда Scrum потребує трьох конкретних ролей: власника продукту(product owner), майстра Scrum (scrum master) та команди розробників(development team). І оскільки команди Scrum є крос-функціональними, до розробників додаються тестувальники, дизайнери, фахівці з користувацького досвіду та інженери операцій.

Власник продукту Scrum (Product Owner).

Власники продукту є чемпіонами свого продукту. Вони фокусуються на розумінні бізнесу, вимог клієнтів та ринку, а потім пріоритизують роботу, яку слід виконати командою розробників відповідно. Ефективні власники продукту:

1. Створюють та управляють списком продукту (backlog).
2. Тісно співпрацюють з бізнесом та командою, щоб усі розуміли елементи роботи у списку продукту.
3. Надають команді чіткі вказівки щодо того, які функції поставити вперед.
4. Вирішують, коли випустити продукт, з підходом до частіших випусків.

Власник продукту не завжди є менеджером продукту. Власники продукту зосереджуються на тому, щоб команда розробників максимально вносила користь бізнесу. Також важливо, щоб власник продукту був індивідуальною особою, оскільки жодна команда розробників не хоче отримувати змішані вказівки від кількох власників продукту.

Майстри Scrum (Scrum Master)

Майстри Scrum є чемпіонами Scrum у своїх командах. Вони тренують команди, власників продукту та бізнес щодо процесу Scrum та шукають способи його вдосконалення.

Ефективний майстер Scrum глибоко розуміє роботу, яку виконує команда, та може допомагати їй оптимізувати прозорість та потік постачання. Як головний фасилітатор, він/вона планує необхідні ресурси (як людські, так і логістичні) для планування ітерацій, стендапів, оглядів і ретроспектив ітерацій.

Команда розробників (Development team)

Команда розробників виконує роботу (пише програмний код, проводить тестування). Вони є чемпіонами сталих практик розробки. Найефективніші команди Scrum є тіснозв'язані, розташованими в одному місці та зазвичай складаються з п'яти до семи осіб. Один із способів визначити розмір команди - це використовувати відоме "правило двох піц" вигадане Джеффом Безосом, генеральним директором Amazon (команда повинна бути досить малою, щоб поділити дві піци, мова йде про піцу 60 см в діаметрі).

Члени команди мають різні навички і взаємно тренують один одного, щоб жодна особа не стала проблемою для постачання роботи(усі члени команди в ідеалі повинні бути взаємозамінні). Сильні команди Scrum самоорганізуються і підходять до своїх проектів з чітким "ми" настроєм. Усі члени команди допомагають один одному для успішного завершення ітерації.

Команда Scrum керує планом для кожної ітерації (sprint). Вони передбачають, скільки роботи вони вважають, що можуть завершити протягом ітерації, використовуючи свою історичну швидкість як директорію. Фіксована довжина ітерації дає команді розробників важливий зворотний зв'язок щодо їхнього оцінювання та процесу постачання, що в свою чергу робить їхні прогнози все більш точними з часом.

Артефакти Scrum - це важлива інформація, якою користується команда Scrum для визначення продукту та роботи, яку слід виконати для створення продукту. У Scrum існують три артефакти: список продукту (Product Backlog), список ітерації (Sprint Backlog) та інкремент із визначенням "готово". Це три постійні речі, на які команда Scrum повинна зосереджуватися під час ітерацій та з часом.

1. Список продукту (Product Backlog): Це основний перелік робіт, який потрібно виконати, і його утримує власник продукту або менеджер продукту. Це динамічний список функцій, вимог, вдосконалень та виправлень, який виступає в якості входу для списку ітерації. Це, по суті, "Список завдань" команди. Список продукту постійно переглядається, перепріорітезується та утримується власником продукту, оскільки, навчаючись більше або внаслідок змін на ринку, деякі елементи можуть втратити актуальність або проблеми можуть бути вирішені іншими способами.

2. Список ітерації (Sprint Backlog): Це список завдань (Task), історій користувачів (User story) чи виправлень помилок (Bug), обраних командою розробників для реалізації в поточному циклі ітерацій. Перед кожною ітерацією, на засіданні планування ітерації (про яке ми розговоримо пізніше) команда обирає, над якими завданнями вона буде працювати протягом ітерації зі списку продукту. Список ітерації може бути гнучким і може еволюціонувати протягом ітерації. Проте фундаментальна ціль ітерації - те, що команда хоче досягти протягом поточної ітерації - не може бути порушеною.

3. Інкремент (або Ціль ітерації): Це працюючий кінцевий продукт після завершення ітерації. В Atlassian зазвичай демонструють "інкремент" під час демонстрації в кінці ітерації, де команда показує, що було завершено протягом ітерації. Можливо, ви не почуєте слово "інкремент" у світі, оскільки його часто називають визначенням команди "Готово" (Done), важливою подією, ціллю ітерації або навіть повною версією чи випущеним епіком. Все залежить від того, як ваша команда визначає "Готово" та як ви визначаєте ваші цілі ітерації. Наприклад, деякі команди вирішують випускати щось для своїх клієнтів в кінці кожної ітерації. Таким чином, їхнє визначення 'Готово' буде 'випущено'. Однак це може бути не реалістичним для інших типів команд. Скажімо, ви працюєте над продуктом на основі сервера, який може відправляти продукт клієнтам лише щокварталу. Ви все ще можете вирішити працювати за двотижневими ітераціями, але ваше визначення "Готово" може бути завершенням частини більшої версії, яку ви плануєте випустити разом. Але, звичайно, чим більше часу займає випуск програмного забезпечення, тим вище ризик того, що програмне забезпечення не відповідає очікуванням. Як видно, є багато варіацій, навіть серед артефактів, якими ваша команда може вибрати визначення. Тому важливо залишатися відкритим для еволюції того, як ви утримуєте навіть ваші артефакти. Можливо, ваше визначення "Готово" створює непотрібний стрес для вашої команди, і вам потрібно повернутися і вибрати нове визначення.

Фреймворк Scrum включає в себе певні практики, церемонії та зустрічі, які команди Scrum проводять на регулярній основі. Церемонії Agile - це місце, де ми спостерігаємо за найбільшим числом варіацій серед команд. Наприклад, деякі команди вважають, що виконання всіх цих церемоній надто обтяжливе та повторюване, тоді як інші використовують їх як необхідну перевірку. Спробуйте використовувати всі церемонії протягом двох ітерацій і подивіться, як вам це видається. Потім можете провести швидку ретроспективу та подивитися, де можливо вам слід внести зміни.

Нижче наведений перелік всіх ключових церемоній, в яких може брати участь команда Scrum:

1. Організація списку продукту (Backlog Grooming or Refinement): Іноді відома як організація списку чи "грумінг", це подія, за відповідальність якої несе власник продукту. Основні завдання власника продукту - це направляти продукт до його візії та мати постійний пульс ринку та клієнта. Таким чином, він/вона утримує цей список, використовуючи відгуки від користувачів та команди розробників для допомоги у пріоритетизації та утриманні списку завдань, готових до виконання у будь-який момент.

2. Планування ітерації (Sprint Planning): Роботу, яку слід виконати (обсяг) протягом поточної ітерації, планує вся команда розробників на цьому засіданні. Цю зустріч очолює майстер Scrum і на ній команда визначає ціль ітерації. Конкретні історії користувачів потім додаються до ітерації зі списку продукту. Ці історії завжди відповідають цілі ітерації та також узгоджуються командою Scrum щодо того, що вони є реально виконаними протягом ітерації.

3. Ітерація (Sprint): Ітерація - це фактичний період часу, коли команда Scrum спільно працює над створенням інкременту. Два тижні - це досить типова тривалість ітерації, хоча деякі команди вважають, що тиждень легше планувати, або місяць - простіше для постачання цінного інкременту. Дейв Вест, з Scrum.org, радить, що чим складніша робота і більше невизначеностей, тим коротший повинен бути термін ітерації. Але це дійсно залежить від вашої команди, і ви не повинні боятися змінювати його, якщо це не працює! Протягом цього періоду обсяг може переглядатися між власником продукту та командою розробників за необхідності. Це є основою емпіричного характеру Scrum.

4. Щоденний Scrum або стендап (Daily Scrum/Stand-up): Це щоденна коротка зустріч, яка відбувається у той самий час (зазвичай ранок) та на одному місці для збереження простоти. Багато команд намагаються завершити зустріч за 15 хвилин, але це лише рекомендація. Цю зустріч також називають 'щоденний стендап', підкреслюючи, що вона повинна бути швидкою. Мета щоденного Scrum - щоб кожен у команді був на одній сторінці, відповідав цілям ітерації та створював план на наступні 24 години. Стендап - це час висловити всі ваші заняття щодо досягнення цілей ітерації або будь-яких перешкод. Розповідати про те, що ви робили вчора, про те, що ви плануєте робити сьогодні та про ті проблеми, з якими ви стикаєтесь.

5. Огляд ітерації (Sprint Review): В кінці ітерації команда збирається для неформальної сесії перегляду або огляду інкременту. Команда розробників демонструє елементи списку продукту, які тепер є 'Готові', зацікавленим стейкхолдерам та командам для отримання відгуку. Власник продукту може вирішити, чи випустити інкремент, хоча в більшості випадків інкремент випускається. На цьому оглядовому засіданні власник продукту переглядає список продукту на основі поточної ітерації, що може вплинути на наступне засідання

6. Ретроспектива ітерації (Sprint Retrospective):** Ретроспектива - це час, коли команда збирається, щоб задокументувати і обговорити, що працювало і що не працювало протягом ітерації, проекту, людей чи взаємин, інструментів або навіть певних церемоній. Ідея полягає в тому, щоб створити місце, де команда може сфокусуватися на тому, що пройшло добре, і на тому, що потрібно покращити наступного разу, і менше про те, що пішло не так.

У 2016 році до Scrum Guide було додано п'ять цінностей Scrum. Ці цінності вказують напрямок роботи, дій та поведінки команди Scrum і вважаються ключовими для успіху такої команди.

1. Зобов'язаність (Commitment): Оскільки команди Scrum є невеликими та гнучкими, кожен член команди відіграє важливу роль у її успіху. Тому кожен член команди повинен згоджуватися взяти на себе зобов'язання виконати завдання, яке він може завершити, і не завершувати більше, ніж може взяти на себе. Повинна бути часта комунікація щодо прогресу роботи, часто на стендапах.

2. Сміливість (Courage): Для команди Scrum сміливість - це просто відвага питати "чому" або будь-що, що заважає їй досягти успіху. Члени команди Scrum повинні мати відвагу та відчувати себе достатньо безпечно, щоб випробовувати нові речі. Команда Scrum повинна мати відвагу та відчувати себе безпечною для того, щоб бути прозорою стосовно труднощів, прогресу проекту, затримок і так далі.

3. Фокус (Focus): У центрі роботи команд Scrum знаходиться спринт - спрямований і визначений період часу, протягом якого команда завершує визначену кількість роботи. Спринт надає структуру, але також фокус для завершення запланованого обсягу роботи.

4. Відкритість (Openness): Щоденний стендап сприяє відкритості, яка дозволяє командам відкрито обговорювати роботу в процесі та перешкоди. У компанії Atlassian ми часто робимо так, щоб наші команди Scrum відповідали на такі питання:

- Над чим я працював вчора?
- Над чим я працюю сьогодні?
- Які проблеми мене турбують?

Це допомагає підкреслити прогрес та виявляти перешкоди. Це також допомагає зміцнити команду, коли всі діляться своїм прогресом.

5. Повага (Respect): Сила агільної команди полягає в її співпраці та усвідомленні, що кожен член команди вносить вклад у роботу протягом спринта. Вони вітають досягнення один одного та ввічливі один до одного, власника продукту, зацікавлених сторін і майстра Scrum.

Сам фреймворк Scrum є простим. Правила, артефакти, події та ролі легко зрозуміти. Його напів-прескриптивний підхід фактично допомагає

усунути неоднозначності в процесі розробки, надаючи при цьому достатньо місця компаніям для внесення власного стилю. Організація складних завдань управління користувацькими історіями робить його ідеальним для складних проектів. Крім того, чітке визначення ролей та запланованих подій забезпечує прозорість та колективну відповідальність протягом усього циклу розробки. Швидкі релізи підтримують високий рівень мотивації команди, а користувачі залишаються задоволеними, бачачи прогрес за короткий період часу.

Однак для повного розуміння scrum може знадобитися час, особливо якщо команда розробників привикла до типової моделі "водоспаду"(Waterfall). Концепції менших ітерацій, щоденних зустрічей Scrum, оглядів спринтів та визначення Scrum Master можуть бути викликом для культурної зміни для нової команди. Проте довгострокові переваги виявляються далеко кращими, ніж початкова крива навчання. Успішне використання Scrum у розробці складних апаратних та програмних продуктів у різних галузях та секторах робить його переконливим фреймворком для впровадження у вашій організації.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Розробити ІТ проект використовуючи підхід до розробки програмного забезпечення Scrum.
3. Використовуючи будь-який (на кшталт www.atlassian.com) відкритий сервіс для реалізації Scrum, створити product backlog.
4. Наповнити product backlog. 1 epic, 3 user story, 2 tasks для кожної user story, 2 sub tasks для кожної task.
5. Додати до product backlog 3 bugs.
6. Створи Scrum board з етапами які будуть найбільш релевантними до вашого проекту.
7. Запустити спринт, та виконати його.
8. Підвести підсумки.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (опис проекту який створювався використовуючи підхід до розробки програмного забезпечення scrum).
4. Висновок (у висновку обов'язково зазначити Ваші думки стосовно недоліків та сильних сторін даного підходу до розробки програмного забезпечення).
5. Список використаної літератури.

Література:

1. Sutherland, J. Scrum: The Art of Doing Twice the Work in Half the Time. Random House Business, 2022. 256p.
2. Cohn, M. Agile Estimating and Planning. Prentice Hall, 2019. 368p.
3. Rasmusson, J. The Agile Samurai: How Agile Masters Deliver Great Software. Pragmatic Programmers, LLC, 2017. 268p.
4. Schwaber, K., Sutherland, J. Scrum Guide. URL: <https://scrumguides.org/>
5. Schwaber, K., Beedle, M. Agile Software Development with Scrum. Prentice Hall, 2001. 158p.

ЛАБОРАТОРНА РОБОТА №4

«ОЗНАЙОМЛЕННЯ З ПІДХОДОМ ДО РОЗРОБКИ KANBAN»

Мета роботи:

Ознайомлення з підходом до розробки програмного забезпечення Agile Kanban, вивчення його основних принципів та етапів. Освоєння сильних та слабких сторін підходу до розробки програмного забезпечення Kanban. Зіставлення двох дуже близьких підходів Scrum і Kanban

Методичні вказівки

Ця робота спрямована на закріплення студентом розуміння гнучких методологій розробки програмного забезпечення та їхнього впливу на успішність проектів.

Вимоги до результатів виконання лабораторної роботи:

1. Розробити проект використовуючи підхід Kanban. Можна використати проект з лабораторної роботи №3.
2. Описати ключові стадії розробки проекту.
3. Проаналізувати плюси та мінуси підходу до розробки програмного забезпечення базуючись на отриманому досвіді. При наявності ідей для покращення роботи підходу навести ці ідеї.

Теоретичні відомості:

Що таке канбан? Канбан - це популярний фреймворк, який використовується для впровадження гнучких методологій у розробці програмного забезпечення. Він передбачає реальну комунікацію в середині команди та повну прозорість роботи. Робочі елементи візуально представлені на дошці канбан, що дозволяє членам команди в будь-який момент бачити стан кожної роботи.

Канбан користується великою популярністю серед сучасних команд у сфері розробки програмного забезпечення, які працюють за принципами гнучких методологій. Проте методологія канбан справжньою своєю суттю сягає більше ніж 50 років тому. У кінці 1940-х років Toyota почала оптимізацію своїх інженерних процесів на основі того ж самого підходу який супермаркети використовували для управління запасами на своїх полицях.

Супермаркети тримають на полицях рівно стільки товарів, скільки потрібно для задоволення потреб споживачів, що оптимізує потік між супермаркетом і споживачем. Тому що рівень запасів відповідає патернам споживання, супермаркет досягає значної ефективності у керуванні запасами, зменшуючи кількість надмірних запасів, які він повинен утримувати у будь-який момент часу. Тим часом супермаркет все ще може гарантувати наявність потрібного товару для споживача.

Коли Toyota застосувала цю саму систему на своїх заводських площах, цілю було краще вирівнювати масштабні рівні запасів з фактичним споживанням матеріалів. Для обміну інформацією про рівні потужностей в реальному часі на заводській дільниці (і для постачальників), робітники передавали картку або "канбан" між командами. Коли бак з матеріалами, які використовуються на лінії виробництва, порожній, "канбан" передавалася на склад із зазначенням, який матеріал потрібен, точної кількості цього матеріалу і так далі. На складі чекав новий бак із цим матеріалом, який вони відправляли на заводську дільницю і, в свою чергу, відправляли власний "канбан" постачальнику. Постачальник також мав бак з цим конкретним матеріалом, який він відправляв на склад. Хоча технологія сигналізації цього процесу змінилася з 1940-х років, цей самий процес виробництва "просто вчасно" (або JIT) все ще є його основою.

Команди розробників програмного забезпечення сьогодні можуть використовувати ті ж самі принципи JIT, відповідно відповідаючи кількість роботи в процесі (WIP) потужності команди. Це надає командам більше гнучких варіантів планування, швидший вивід, чітку фіксованість і прозорість на протязі всього циклу розробки.



Хоча основні принципи цього фреймворку є вічними і можуть бути застосовані практично в будь-якій галузі, команди розробки програмного забезпечення досягли особливого успіху за допомогою гнучкої методології. Це, зокрема, через те, що команди розробників (development team, у подальшому контексті мається на увазі команда розробників програмного забезпечення) можуть почати впроваджувати практику з мінімальними

витратами часу та ресурсів, як тільки вони зрозуміють основні принципи. На відміну від впровадження канбану на заводській дільниці, що передбачало б зміни в фізичних процесах та додавання суттєвих матеріалів, єдиними фізичними речами, які потрібні командам розробників, є дошка та картки, і навіть це може бути віртуальним.

Дошки канбану (Kanban board). Робота всіх команд, що працюють за методологією канбан, обертається навколо дошки канбан, інструменту, який використовується для візуалізації роботи та оптимізації потоку роботи серед команди. Хоча фізичні дошки популярні серед деяких команд, віртуальні дошки є важливою функцією в будь-якому засобі розробки програмного забезпечення за їх відстежуваність, зручнішою співпрацею та доступністю з різних місць.

Незалежно від того, чи фізична, чи цифрова дошка у команди, її функція полягає в тому, щоб забезпечити візуалізацію роботи команди, стандартизацію їх робочого процесу та негайне виявлення та вирішення всіх блокувань і залежностей. У базовій версії дошка канбан має постійний робочий процес: "Зробити", "В процесі" та "Готово". Однак в залежності від розміру, структури та цілей команди, робочий процес може бути адаптований для відповіді унікальному процесу кожної конкретної команди (рис Л.3.1).

Методологія канбан ґрунтується на повній прозорості роботи та миттєвій комунікації потужностей. Таким чином, дошка канбан слід розглядати як єдине джерело правди для роботи команди.

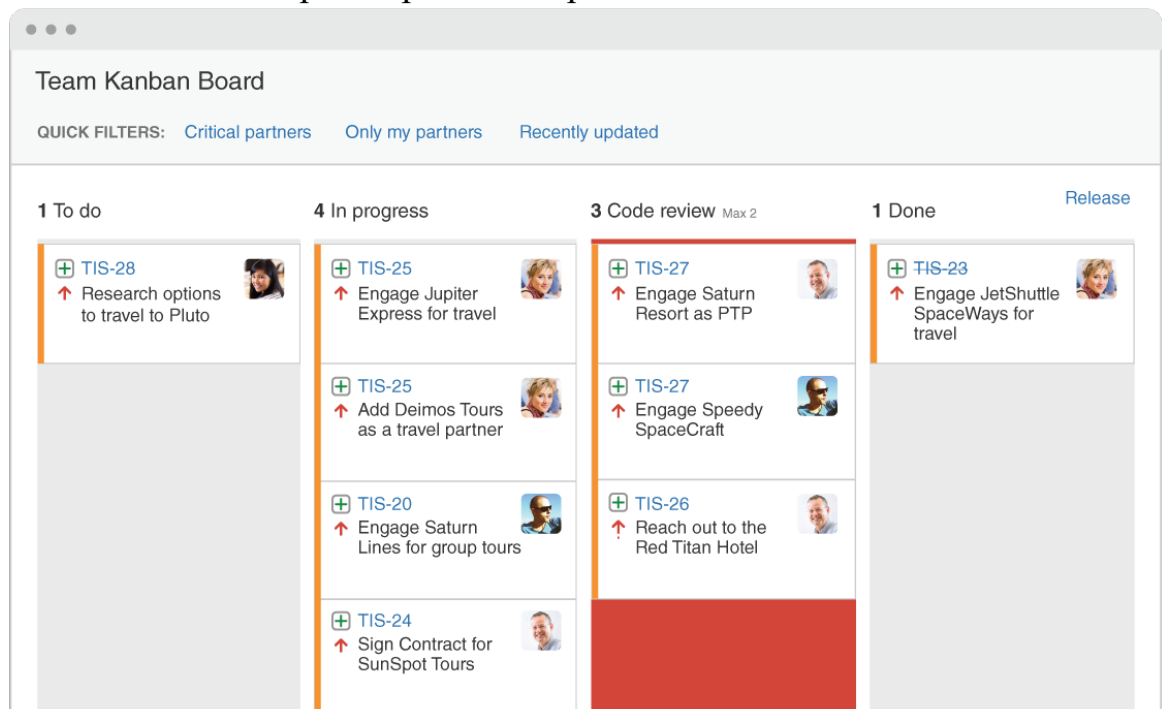


Рис Л4.1 Дошка Kanban з <http://atlasian.com>

У японській мові "канбан" буквально перекладається як "візуальний сигнал". Для команд, що працюють за методологією канбан, кожен робочий елемент представлений у вигляді окремої картки на дошці.

Головна мета відображення роботи у вигляді картки на дошці канбан полягає в тому, щоб дозволити членам команди відстежувати прогрес роботи через її робочий процес в дуже візуальний спосіб. Канбан-картки містять важливу інформацію про конкретний робочий елемент, що надає всій команді повний огляд того, хто відповідає за цей елемент роботи, короткий опис виконуваної роботи, тривалість цього робочого елемента та інше. Картки на віртуальних дошках канбан часто також містять знімки екрану та інші технічні деталі, які є важливими для виконавця. Можливість для членів команди бачити стан кожного робочого елемента у будь-який момент часу, а також всі пов'язані деталі, забезпечує підвищену увагу, повну відстежуваність та швидке виявлення блокаторів і залежностей.

Канбан є однією з найпопулярніших методологій розробки програмного забезпечення, що прийнята командами, які працюють у гнучкому режимі. Канбан пропонує кілька додаткових переваг для планування завдань та продуктивності команд будь-якого розміру:

- Гнучке планування

Команда канбан фокусується тільки на роботі, яка фактично знаходиться в розробці. Після завершення робочого елемента команда бере на обробку наступний елемент з вершини списку завдань. Власник продукту може вільно перепланувати завдання у списку завдань, не турбуючи команду, оскільки будь-які зміни поза поточними робочими елементами не впливають на роботу команди. Завдяки тому, що власник продукту тримає найважливіші завдання у верхівці списку завдань, команді гарантовано, що вона надає максимальну користь бізнесу. Таким чином, немає потреби в фіксованих ітераціях, які є характерними для Scrum.

Досвідчені власники продукту завжди взаємодіють з розробницькою командою, розглядаючи зміни у списку завдань. Наприклад, якщо користувальницькі історії (user story) з 1 по 6 є у списку завдань, оцінка користувальницької історії 6 може базуватися на завершенні користувальницьких історій 1-5. Завжди добре практикувати підтвердження змін з командою розробників для уникнення непередбачуваних ситуацій.

- Скорочення часових циклів

Часовий цикл є ключовим показником для команд канбан. Циклічний час - це час, який потрібно для одиниці роботи пройти через робочий процес команди - від моменту початку роботи до моменту відправки. Оптимізуючи часовий цикл, команда може з упевненістю прогнозувати виконання майбутньої роботи.

Якщо усі члени команди розробників мають однаковий досвід(володіють однаковою кількістю навичок, однаково володіють знаннями про продукт який розробляють) це призводить до скорочення часових циклів. Коли знаннями володіє лише одна людина, ця особа стає незамінною у робочому процесі. Таким чином, команди використовують базові практики, такі як код-рев'ю та менторинг, щоб розповсюджувати знання. Спільні навички означають, що члени команди можуть виконувати гетерогенну роботу, що додатково оптимізує часовий цикл. Це також означає, що якщо є затор роботи (bottle neck), вся команда може взятися за це, щоб відновити плавний рух процесу. Наприклад, тестування не робиться лише інженерами QA. Розробники також беруть участь (проте дуже рідко відбувається навпаки ☺).

У канбан-фреймворку всій команді доручається відповідальність за забезпечення плавності роботи в процесі.

- Менше заторів

Багатозадачність шкодить ефективності. Чим більше робочих елементів одночасно, тим більше перемикання контексту, що заважає їх завершенню. Тому ключовим принципом канбан є обмеження кількості роботи в процесі (WIP). Обмеження роботи в процесі вказує на затори і затримки в робочому процесі команди через відсутність фокусу, людей чи навичок.

Наприклад, типова команда розробників може мати чотири стани робочого процесу: "To Do" (Зробити), "In Progress" (В процесі), "Code Review" (Перегляд коду) та "Done" (Готово). Вони можуть вирішити встановити обмеження WIP на рівні 2 для стану перегляду коду. Це може здатися низьким обмеженням, але є важлива причина для цього: розробники часто віддають перевагу написанню нового коду, аніж витратити час на перегляд роботи інших. Низьке обмеження підтримує команду у приділенні особливої уваги проблемам у стані перегляду та у перегляді роботи інших до підняття їхніх власних переглядів коду. Це в кінцевому підсумку зменшує загальний часовий цикл.

- Візуальні метри

Одним із ключових принципів є сильний акцент на постійному покращенні ефективності та ефективності команди з кожним ітераційним робочим процесом. Діаграми надають візуальний механізм для команд, щоб переконатися, що вони продовжують вдосконалюватися. Коли команда може бачити дані, легше виявити затори у процесі (і виправити їх). Два поширені звіти, які використовують команди канбан, - це діаграми контролю (control chart) та кумулятивні діаграми потоку(cumulative flow diagrams).

Діаграма контролю показує часовий цикл для кожного завдання, а також ковзний середній для команди (рис. Л4.2).

Метою команди є зменшення часу, який потрібно для проходження завдання через весь процес. Перегляд середнього часового циклу в діаграмі контролю є показником успіху.

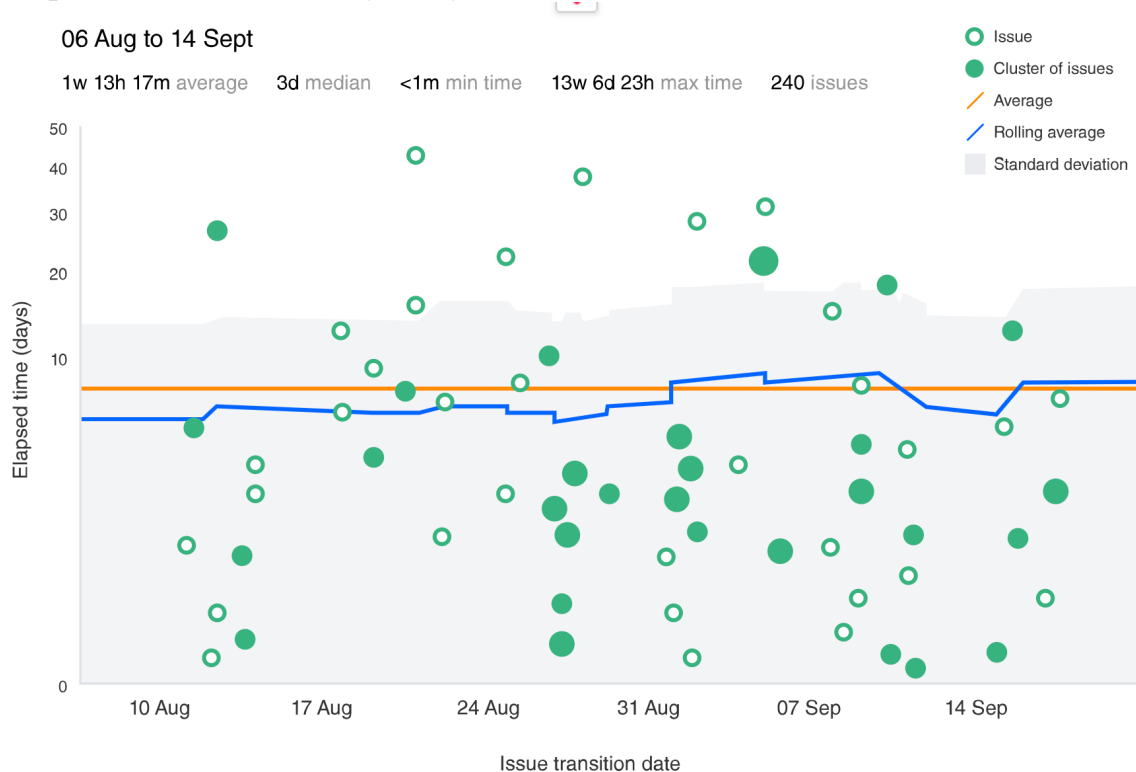


Рис Л.4.2 Діаграма контролю (<https://www.atlassian.com/agile/kanban>)

Кумулятивна діаграма потоку показує кількість завдань у кожному стані (рис Л. 4.3). Команда може легко виявити блокування, бачачи, як збільшується кількість завдань у будь-якому визначеному стані. Завдання у проміжних станах, таких як "В процесі" або "Підготовка до огляду", ще не були відправлені клієнтам, і блокування в цих станах може збільшити ймовірність виникнення масштабних конфліктів при інтеграції, коли робота фактично буде об'єднана в основний код.

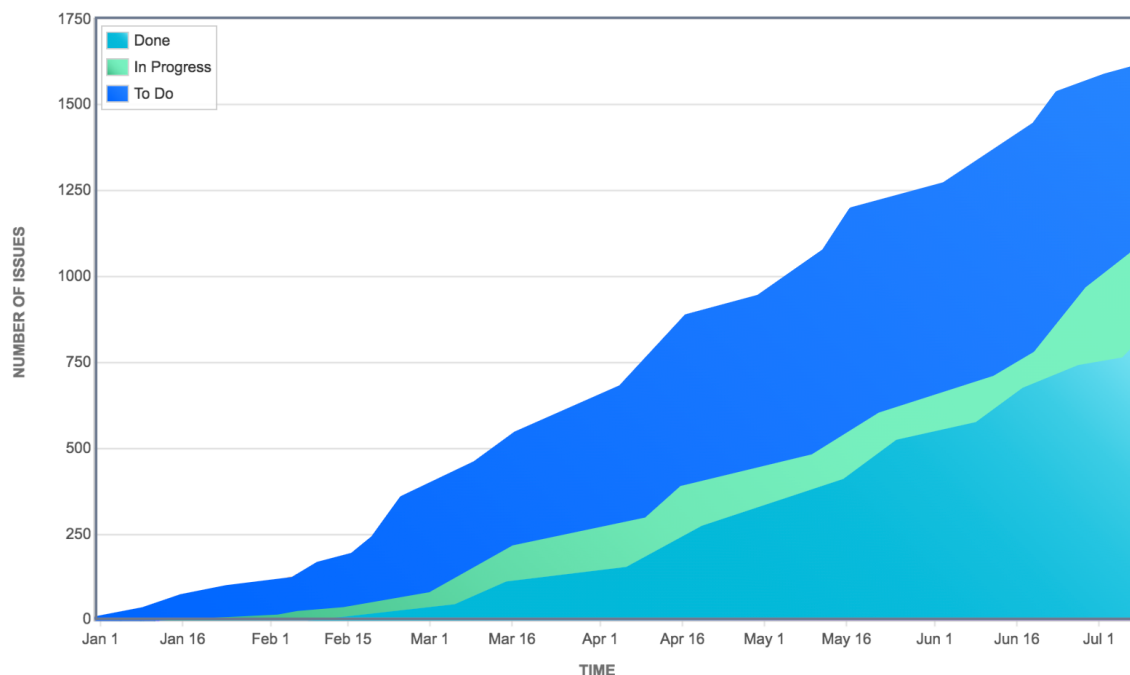


Рис Л.4.2 Кумулятивна діаграма потоку
(<https://www.atlassian.com/agile/kanban>)

Kanban та Scrum мають спільні концепції, але мають дуже різний підхід. Вони не повинні плутатися один з одним. У таблиці (табл Л4.1) наведені основні відмінності.

Таблиця Л4.1 Основні відмінності між Scrum та Kanban

	Scrum	Kanban
Часовий ритм	Регулярні фіксовані спринти (наприклад, 2 тижні)	Постійний потік
Методологія випуску	В кінці кожного спринта, якщо затверджено власником продукту. (Не обов'язково, можлива також постійна доставка)	Постійна доставка або за розсудом команди
Ролі	Власник продукту, Scrum майстер, команда розробників	Немає встановлених ролей. Деякі команди залучають коуча (agile coach)
Основні метрики	Швидкість (Velocity)	Час циклу (Cycle time)
Філософія змін	Команди повинні прагнути не вносити змін до спринта під час спринта. Такі зміни компрометують ціль спринта.	Зміни можуть відбуватися в будь-який момент.

Деякі команди поєднують ідеї Kanban та Scrum у "Scrumban". Вони беруть фіксовані спринти та ролі від Scrum і фокусуються на обмеженнях роботи в процесі та часі циклу з Kanban. Однак для команд, що тільки починають працювати за методологією Agile, рекомендують вибрати одну методологію і працювати з нею протягом певного часу. Ви завжди можете спробувати щось нове пізніше.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Розробити ІТ проект використовуючи підхід до розробки програмного забезпечення Kanban.
3. Використовуючи будь-який (на кшталт www.atlassian.com) відкритий сервіс для реалізації Kanban, створити product backlog.
4. Наповнити product backlog. 1 epic, 3 user story, 2 tasks для кожної user story, 2 sub tasks для кожної task (Можна використати уже існуючі з ЛР №3).
5. Додати до product backlog 3 bugs (Можна використати уже існуючі з ЛР №3).
6. Створи Kanban board з етапами які будуть найбільш релевантними до вашого проекту.
7. Проаналізувати на практиці основні відмінності між підходом Scrum і Kanban
8. Підвести підсумки.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (опис проекту який створювався використовуючи підхід до розробки програмного забезпечення scrum).
4. Висновок (у висновку обов'язково зазначити Ваші думки стосовно недоліків та сильних сторін даного підходу до розробки програмного забезпечення).
5. Список використаної літератури.

Література:

1. David J. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010. 368 p.
2. Jim Benson and Tonia DeMaria Barry, Personal Kanban: Mapping Work | Navigating Life, Modus Cooperandi Press, 2011. 208 p.
3. Mary Poppendieck and Tom Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003. 240 p.
4. Corey Ladas, Scrumban: Essays on Kanban Systems for Lean Software Development, Modus Cooperandi Press, 2008. 150 p.

5. Mike Cohn, Agile Estimating and Planning, Prentice Hall, 2005. 368 p.
6. Marcus Hammarberg and Joakim Sunden, Kanban in Action, Manning Publications, 2014. 360p.

ЛАБОРАТОРНА РОБОТА №5

«ВЕРСІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ЛОКАЛЬНА СИСТЕМА КОНТРОЛЮ ВЕРСІЙ.»

Мета роботи:

Мета лабораторної роботи полягає у вивченні концепції версіонування програмного забезпечення, розумінні важливості та переваг використання систем контролю версій, а також у освоєнні практичних навичок створення локальних репозиторіїв та ефективного користування ними базуючись на технології Git.

Методичні вказівки:

Ця робота спрямована на вивченні сутності версіонування програмного забезпечення, розумінні його необхідності та переваг, а також у практичному опануванні студентами системи контролю версій Git для створення та управління репозиторіями програмного коду.

Вимоги до результатів виконання лабораторної роботи:

1. Створити локальний репозиторій з проектом з попередніх лабораторних робіт.
2. Описати ключові стадії версіонування проекту.
3. Проаналізувати плюси та мінуси використання системи контролю версій GIT .

Теоретичні відомості:

Система контролю версій

Система контролю версій (СКВ) - це інструмент, який фіксує зміни у файлі або групі файлів протягом певного періоду часу, дозволяючи повертатися до конкретних версій у майбутньому. Наприклад, у роботі над проектами, що піддаються версійному контролю, СКВ використовується для збереження програмного коду, але насправді його можна використовувати для будь-яких типів файлів.

Для графічних або веб-дизайнерів, які бажають зберігати кожен версію зображення або макету, СКВ надзвичайно корисна. Вона дозволяє відновити файли до попередніх станів, переглядати всі зміни, визначати, хто і коли вніс певні зміни, а також виправляти помилки або втрати з легкістю. Використання СКВ також економить час та зусилля, оскільки дозволяє вам керувати проектами ефективно та знижує ризик втрати даних.

Локальні системи контролю версій

Багато людей використовують копіювання файлів до окремої директорії як один із методів контролю версій (можливо, навіть з

додаванням відмітки за часом, якщо вони досить винахідливі). Цей підхід досить поширений через його простоту, але він також дуже схильний до помилок. Легко забути, в якій директорії ви знаходитесь, і випадково змінити неправильний файл або скопіювати не ті файли, які вам потрібні.

Для вирішення цієї проблеми програмісти давно розробили локальні системи контролю версій, що мають просту базу даних для зберігання всіх змін у файлах, які перебувають під контролем версій.

Одним з найбільш поширених інструментів СКВ є система під назвою RCS, яка досі широко використовується багатьма комп'ютерами сьогодні. RCS зберігає набори латок (тобто, відмінності між файлами) у спеціальному форматі на диску; він може відтворити будь-який файл у будь-який момент, додавши всі латки до оригінального файлу (рис. Л5.1).

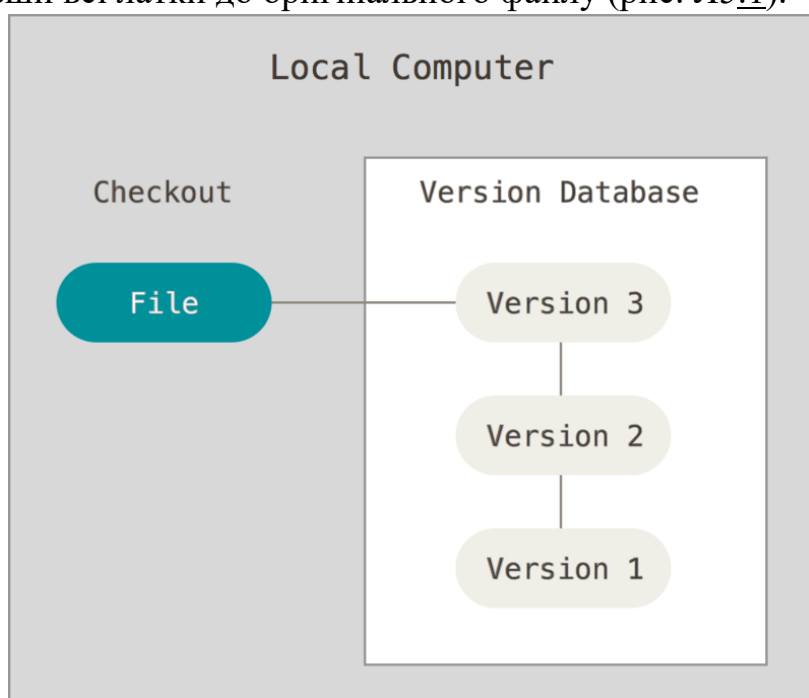


Рис Л5.1 Локальні системи контролю версій.

Централізовані системи контролю версій

Ще однією важливою проблемою, з якою зіштовхуються люди, є необхідність співпрацювати з іншими розробниками. Для вирішення цього завдання були створені централізовані системи контролю версій (ЦСКВ, рис. Л5.2). Такі системи, як CVS, Subversion і Perforce, мають єдиний сервер, на якому зберігаються всі версії файлів, та ряд клієнтів, які отримують файли з центрального місця. Протягом багатьох років цей підхід був стандартом для систем контролю версій.

Такий підхід має безліч переваг, особливо порівняно з локальними СКВ. Наприклад, кожен учасник проекту знає, що роблять інші учасники. Адміністратори мають повний контроль над тим, хто і що може робити. Також адмініструвати ЦСКВ набагато легше, ніж мати справу з локальними базами даних для кожного клієнта.

Проте цей підхід також має серйозні недоліки. Найочевиднішим з них є наявність єдиної точки відмови (single point of failure (SPOF)), якою є централізований сервер. Якщо сервер вийде з ладу протягом години, то протягом цього часу ніхто не зможе співпрацювати або зберігати зміни під версійним контролем. Якщо жорсткий диск центральної бази даних на сервері пошкоджено, а резервні копії не зроблені вчасно, можливі втрати всіх даних, крім окремих знімків проекту, що зберіглися на локальних машинах користувачів. Такі ж проблеми спостерігаються і у локальних СКВ, коли всю історію проекту зберігають в одному місці, що може призвести до повної втрати даних.

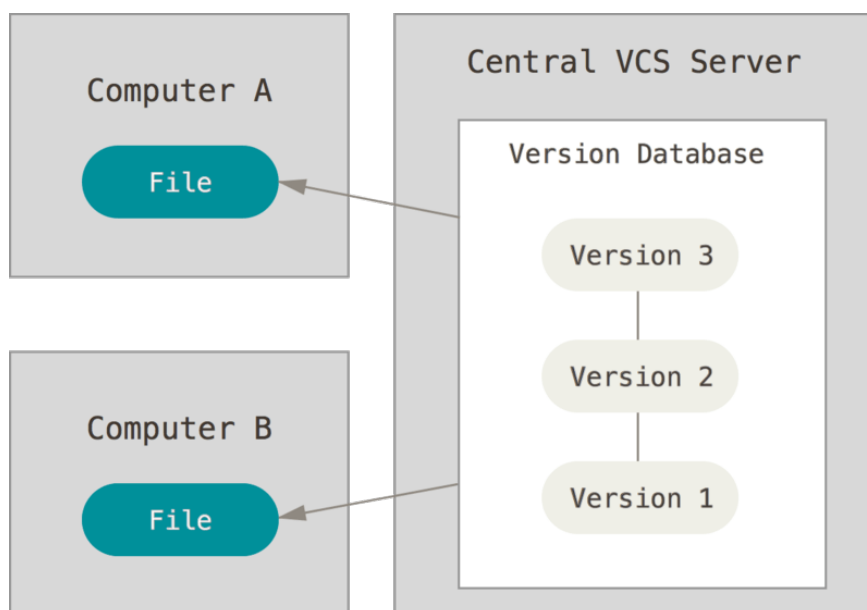


Рис Л5.2. Централізовані системи контролю версій.

Децентралізовані системи контролю версій

Приєднуються до гри децентралізовані системи контролю версій (рис Л5.3) (ДСКВ). У таких системах, як Git, Mercurial, Bazaar або Darcs, клієнти не просто отримують останній знімок файлів з репозиторія; натомість вони мають повну копію сховища з усією його історією. Таким чином, у разі виходу з ладу сервера, через який взаємодіють розробники, будь-який з клієнтських репозиторіїв може бути скопійований назад до сервера для відновлення. Кожна копія, фактично, є повною резервною копією всіх даних.

Крім того, багато з цих систем відмінно взаємодіють з декількома віддаленими репозиторіями, що дозволяє співпрацювати з різними групами людей та використовувати різні підходи в межах одного проекту одночасно. Це дозволяє налаштувати декілька типів робочих процесів, таких як ієрархічні моделі, які неможливі в централізованих системах.

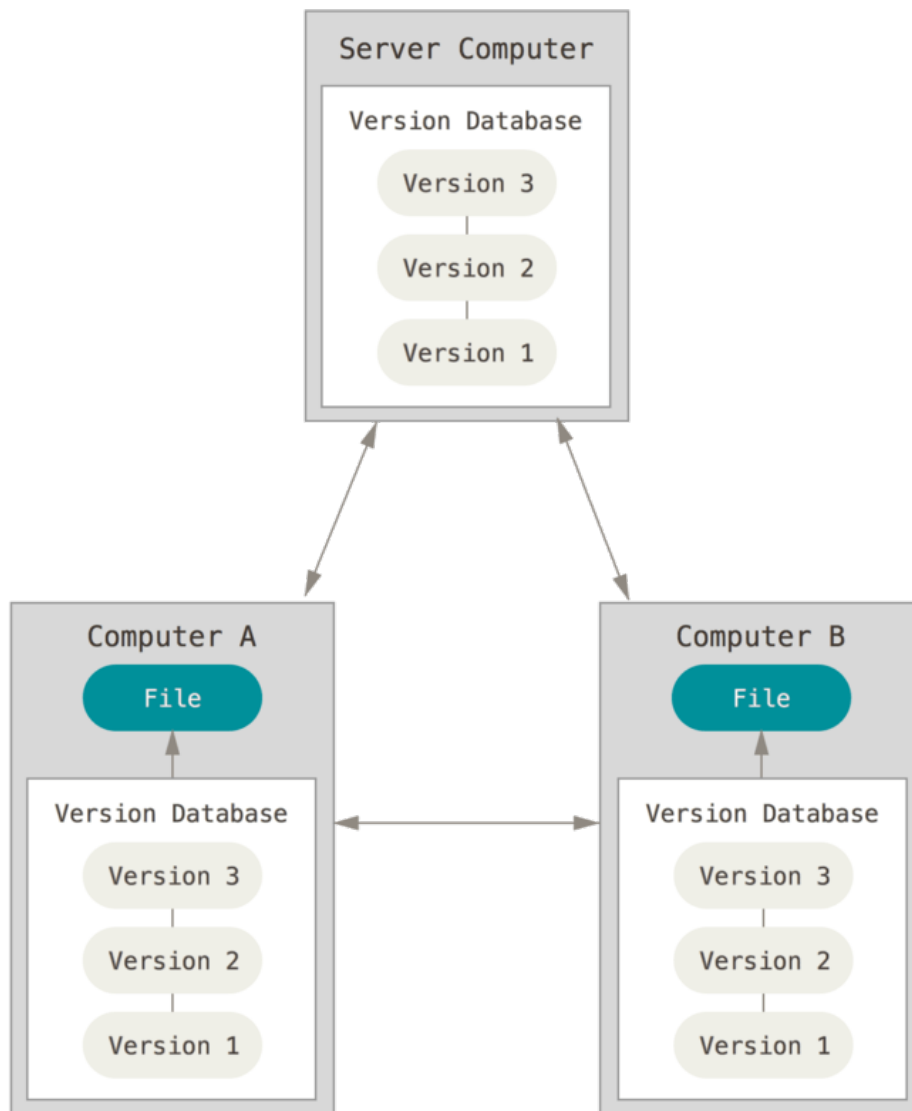


Рис Л.5.3. Децентралізовані системи контролю версій.

Так, як і багато великих проектів, Git виник з невеликого хаосу та жвавих дискусій. Ядро Linux - це значний відкритий проект. Протягом більшої частини свого існування (1991-2002), підтримка ядра Linux здійснювалася через патчі та архіви. У 2002 році проект ядра Linux перейшов на закриту ДСКВ BitKeeper.

У 2005 році відносини між спільнотою розробників ядра Linux і комерційною компанією, що стояла за BitKeeper, почали погіршуватись, і безкоштовне використання цим продуктом було скасовано. Це заохотило розробників Linux, включаючи Лінуса Торвальдса, автора Linux, створити свою власну систему, використовуючи деякі уроки, які вони вивчили під час роботи з BitKeeper. Деякі з цілей нової системи були:

- швидкість
- проста архітектура

- сильна підтримка для нелінійного розвитку (тисячі паралельних гілок)
- децентралізація
- можливість ефективно управляти великими проектами, такими як ядро Linux (швидкість і розмір даних)

З моменту свого народження в 2005 році, Git розвинувся і дозрів, щоб бути простим у використанні і в той же час зберегти свої первинні властивості. Git дивовижно швидкий, та дуже ефективний для великих проектів, і має неймовірну систему галуження для нелінійного розвитку.

Стани у системі GIT

Git має три основних стани, в яких можуть перебувати ваші файли: **збережений у коміті** (committed), **змінений** (modified) та **індексований** (staged) (рис Л.5.4):

- Збережений у коміті означає, що дані безпечно збережено в локальній базі даних.
- Змінений означає, що у файл внесено редагування, які ще не збережено в базі даних.
- Індексований стан виникає тоді, коли ви позначаєте змінений файл у поточній версії, щоб ці зміни ввійшли до наступного знімку коміту.

З цього випливають три основні частини проекту під управлінням Git: директорія .git, робоче дерево та індекс.

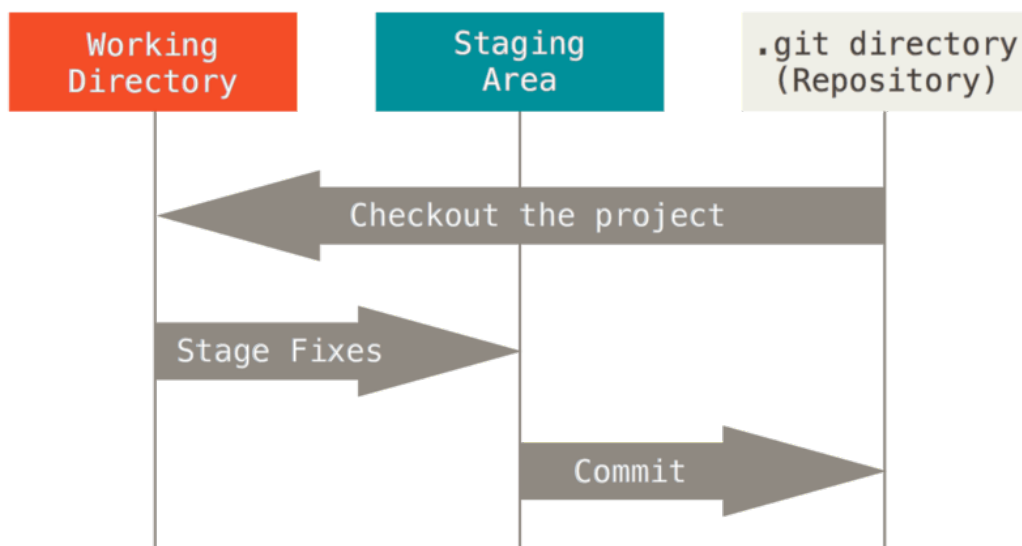


Рисунок Л.5.4. Робоча директорія, індекс та директорія Git.

У каталозі .git система Git зберігає метадані та базу даних об'єктів вашого проекту. Це найважливіша частина Git, яка копіюється при клонуванні сховища з іншого комп'ютера.

Робоче дерево - це конкретна версія проекту, витягнута із сховища. Ці файли вилучаються з бази даних у теці Git та зберігаються на диску для подальшого використання та редагування.

Індекс - це файл, що зазвичай розташовується у каталозі `.git` і містить інформацію про те, що буде збережено у наступному коміті. Його також називають "областю додавання" (staging area), але ми переважно будемо використовувати технічний термін Git "індекс".

Найпростіший процес взаємодії з Git виглядає наступним чином:

1. Ви редагуєте файли у своєму робочому каталозі (project folder).
2. Вибірково надсилаєте до індексу лише ті зміни, які ви хочете зберегти в наступному коміті, і лише ці зміни будуть збережені в індексі.
3. Створюєте коміт: знімок з індексу остаточно зберігається у каталозі `.git`.

Якщо окрема версія файлу вже присутня у каталозі `.git`, цей файл вважається збереженим у коміті. Якщо він зазнав змін та перебуває в індексі, то він індексований. Якщо ж його стан відрізняється від того, який був у коміті, і файл не знаходиться в індексі, то він вважається зміненим.

Створення Git-репозиторія

Зазвичай Git репозиторій отримують одним з двох способів:

1. Беруть локальну директорію, що наразі не під контролем версій, та перетворюють її на сховище Git, або
2. Звідкілясь **клонують** існуючий Git репозиторій.

У будь-якому разі ви отримуєте на локальній машині готове до роботи Git сховище.

Ініціалізація репозиторія в існуючому каталозі

Якщо у вас вже є тека з проектом, яка на даний момент не знаходиться під контролем версій, і ви хочете почати використовувати Git з цим проектом, спочатку потрібно перейти до теки цього проекту.

Потім виконайте команду (не збудьте встановити GIT на ваш комп'ютер, знак \$ вводити не потрібно ☺):

```
$ git init
```

Це створить новий підкаталог `.git`, який містить всі необхідні файли вашого репозиторія - основу Git-репозиторія. На цей момент у вашому проекті ще нічого не відстежується.

Якщо ви бажаєте додати існуючі файли під версійний контроль (на відміну від порожнього каталогу), вам, ймовірно, слід проіндексувати ці файли та зробити перший коміт. Ви можете це зробити за допомогою декількох команд `git add`, що визначають файли, які ви хочете відстежувати, після чого виконаєте команду `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Перша версія проекту'
```

В результаті у вас є Git-репозиторій з декількома файлами та першим комітом.

Клонування існуючого репозиторія

Якщо ви бажаєте отримати копію існуючого Git репозиторія - наприклад, проекту, в якому ви хочете взяти участь - вам потрібно використати команду `git clone`. Ця команда відрізняється від команди "checkout", яку ви могли б побачити у інших системах контролю версій, таких як Subversion. Важливий момент полягає в тому, що Git отримує повну копію майже всіх даних, які є на сервері, а не лише робочу копію. Кожна версія кожного файлу в історії проекту автоматично витягується при виконанні команди `git clone`. Фактично, у випадку неполадок з сервером, ви зазвичай зможете використати будь-який з клонів на будь-якому клієнті, щоб відновити сервер до стану на момент клонування.

Для клонування репозиторію потрібно використати команду `git clone <url>`. Наприклад, якщо ви хочете зробити клон бібліотеки Git `libgit2`, ви можете виконати таку команду:

```
$ git clone https://github.com/lib3/lib3
```

Це створить каталог з назвою `lib3`, ініціалізує каталог `.git`, завантажить всі дані з репозиторія та оновить каталог до останньої версії. Після переходу до цього каталогу `libgit2` ви побачите, що всі файли проекту там наявні і готові до використання.

Git підтримує кілька різних протоколів передачі даних, які можна використовувати. У попередньому прикладі використовується протокол `https://`, але ви також можете побачити `git://` або `user@server:шлях/до/репозиторія.git`, що використовує SSH протокол.

Тепер у вас на локальній машині має бути справжній Git репозиторій та робоча тека з усіма файлами цього проекту. Зазвичай, ви хочете зробити деякі зміни та записати їх у вашому репозиторії кожного разу, коли ваш проект набуває стану, що ви бажаєте зберегти.

Пам'ятайте, що кожен файл вашої робочої директорії може бути в одному з двох станів: **контрольований** (tracked) чи **неконтрольований** (untracked). Контрольовані файли — це файли, що були в останньому знімку. Вони можуть бути не зміненими, зміненими або індексованими. Якщо стисло, контрольовані файли — це файли, про які Git щось знає.

Неконтрольовані файли — це все інше, будь-які файли у вашій робочій директорії, що не були у вашому останньому знімку та не існують у вашому індексі. Якщо ви щойно зробили клон репозиторія, усі ваші файли контрольовані та не змінені, адже Git щойно їх отримав, а ви нічого не редагували.

По мірі редагування файлів, Git бачить, що вони змінені, адже ви їх змінили після останнього коміту. Впродовж роботи ви вибірково індексуєте ці змінені файли та потім зберігаєте всі індексовані зміни, та цей цикл повторюється (рис Л5.5).

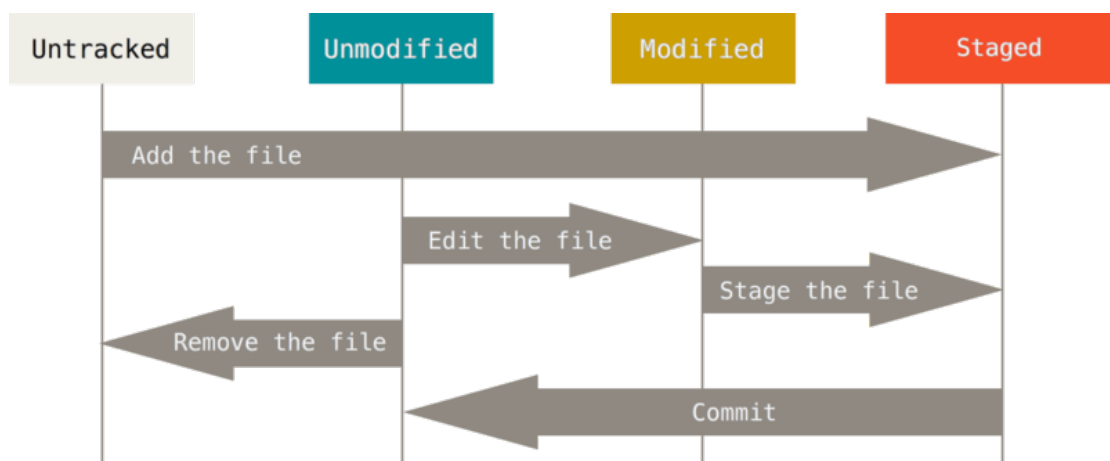


Рис. Л5.5. Цикл життя статусу ваших файлів.

Перевірка статусу ваших файлів

Щоб дізнатись, в якому стані ваші файли, варто використати команду `git status`. Якщо ви виконаєте цю команду відразу після клонування, ви побачите наступне:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Це означає, що ваша робоча директорія чиста — іншими словами, жоден з контрольованих файлів не змінено. Git також не бачить неконтрольованих файлів, інакше він би їх тут вказав. Крім того, ця команда показує вам, в якій гілці ви знаходитесь, та інформує вас про те, що вона не відрізняється від такої ж гілки на сервері. Наразі ця гілка завжди буде "master", так як вона створюється автоматично.

Припустимо, ви додали новий файл до вашого проекту, наприклад, простий файл README. Якщо файл раніше не існував, і ви виконаєте `git status`, ви побачите такий вихід:

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

README

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

Тут ви можете побачити, що ваш новий файл README є неконтрольованим, оскільки він під заголовком "Untracked files" у статусі. "Неконтрольований" означає, що Git бачить файл, якого нема у попередньому знімку (коміті). Git не буде включати його до ваших комітів, поки ви явно не скажете йому це зробити. Такий підхід уникне випадкового включення генерованих бінарних файлів чи інших файлів, які ви не маєте наміру включати. Якщо ж ви все ж хочете розпочати включення README, ви можете почати контролювати цей файл за допомогою команди `git add..`

Збереження ваших змін у комітах

Припустімо, що ваш індекс знаходиться в потрібному вам стані і ви готові зробити коміт зі своїх змін. Варто пам'ятати, що будь-які неіндексовані зміни — це файли, які ви створили чи змінили, але не додали до індексу за допомогою команди `git add` — не включатимуться до цього коміту. Вони лишатимуться незбереженими на вашому диску. У цьому випадку, якщо ви вже перевірили статус за допомогою `git status` і побачили, що всі зміни індексовані, ви готові до збереження змін. Найпростіший спосіб зробити коміт — використати просто `git commit`:

```
$ git commit -m "Мій перший коміт"
```

Тепер ваш перший коміт створено! Ви можете бачити, що команда `commit` надала вам інформацію про коміт: до якої гілки ви зберегли зміни. Варто пам'ятати, що кожен коміт створює знімок вашого проекту, який ви можете повернутися або порівняти з пізнішими змінами.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Встановити систему контролю версій Git.

3. Використовуючи наявний проект з попередньої лабораторної роботи ініціалізувати для цього проекту локальний git репозиторій.
4. Створити декілька нових файлів та додати їх до вашого git репозиторію.
5. Створити перший коміт для добавлених файлів з повідомленням: This is my first commit.
6. Здійснити зміни у файлах та зробити другий коміт.
7. Об'єднати два попередні коміти в один.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (описати ваш git репозиторій, навести послідовність виконаних команд).
4. Висновок. Зробіть акцент на доцільності використання системи контролю версій Git
5. Список використаної літератури.

Література:

1. Scott Chacon, Ben Straub. Pro Git. Apress. 2014. 456 p.
2. Jon Loeliger, Matthew McCullough. Version Control with Git. O'Reilly Media. 2012. 456 p.
3. Ryan Hodson. Git Essentials. Packt Publishing. 2015. 194 p.
4. Roger Dudler. Git Pocket Guide. O'Reilly Media. 2013. 234 p.
5. Seth Robertson, Steve Oualline. Practical Git: For Everyday Use. No Starch Press. 2014. 300 p.

ЛАБОРАТОРНА РОБОТА №6

«ВЕРСІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ДЕЦЕНТРАЛІЗОВАНА СИСТЕМА КОНТРОЛЮ ВЕРСІЙ.»

Мета роботи:

Полягає в глибшому розумінні концепції децентралізованих систем контролю версій та їх роллю в процесі розробки програмного забезпечення. Ознайомитись з основними принципами та перевагами використання децентралізованих систем контролю версій порівняно з централізованими аналогами. Створити власний децентралізований репозиторій використовуючи сервіс GitHub.

Методичні вказівки:

ЛАБОРАТОРНА робота спрямована на практичне освоєння роботи з децентралізованими системами контролю версій базуючись на досвіді створення децентралізованого репозиторія.

Вимоги до результатів виконання лабораторної роботи:

1. Створити локальний репозиторій з проектом з попередніх лабораторних робіт, створити віддалений репозиторій використовуючи сервіс GitHub.
2. Описати ключові стадії версіонування проекту з використання децентралізованого репозиторію.
3. Проаналізувати плюси та мінуси використання децентралізованих систем контролю версій.

Теоретичні відомості:

Децентралізовані системи контролю версій

У попередній лабораторній роботі ми уже коротко згадували про децентралізовані системи контролю версій (рис Л5.3) (ДСКВ). У таких системах, як Git, Mercurial, Bazaar або Darcs, клієнти не просто отримують останній знімок файлів з репозиторія; натомість вони мають повну копію сховища з усією його історією. Таким чином, у разі виходу з ладу сервера, через який взаємодіють розробники, будь-який з клієнтських репозиторіїв може бути скопійований назад до сервера для відновлення. Кожна копія, фактично, є повною резервною копією всіх даних.

Крім того, багато з цих систем відмінно взаємодіють з декількома віддаленими репозиторіями, що дозволяє співпрацювати з різними групами людей та використовувати різні підходи в межах одного проекту одночасно. Це дозволяє налаштувати декілька типів робочих процесів, таких як ієрархічні моделі, які неможливі в централізованих системах.

GitHub – ознайомлення, та основні налаштування.

GitHub - це найбільший хостинг для сховищ Git, який використовується мільйонами розробників та проектів для співпраці та управління кодом. Багато відкритих проектів активно використовують GitHub для зберігання, спільного редагування та відстеження змін у вихідному коді. Хоча GitHub не є необхідною частиною Git, він став невід'ємною частиною професійного використання Git, і ви, мабуть, знатимете про нього більше під час взаємодії з Git у вашій роботі.

Спочатку вам потрібно створити безкоштовний обліковий запис. Просто перейдіть на веб-сайт <https://github.com>, виберіть унікальне ім'я користувача, яке ще не використовується, введіть адресу електронної пошти та пароль, і натисніть на велику зелену кнопку "Sign up for Github" (Зареєструватись на Github). Ми не будемо зупинятися на процесі реєстрації оскільки він є доволі типовим для усіх інтернет ресурсів, зверніть увагу що вам потрібно використовувати public repositories (оскільки вони є безкоштовними).

Підказка: На момент виходу цієї роботи студенти які зареєструються під корпоративною поштою університету (доменне ім'я повинне закінчуватися edu.ua, наприклад djonsenyk.borys@lnu.edu.ua ☺) можуть безкоштовно користуватися деякими платними сервісами від GitHub (наприклад GitHub Copilot) для цього перейдіть на сторінку GitHub campus <https://education.github.com/students>

Внесок до проекту

Тепер, коли ваш обліковий запис налаштовано, розглянемо деякі подробиці, що можуть стати вам у нагоді при роботі над вже існуючим проектом.

Якщо ви бажаєте зробити внесок до існуючого проекту, проте у вас немає права на викладання до нього змін, ви можете створити "форк" (fork) проекту. Коли ви створюєте "форк" проекту, GitHub створює копію проекту, що є повністю вашою. Вона існує в просторі імен вашого користувача, ви можете викладати до неї зміни.

Таким чином, проектам не доводиться піклуватися про надання користувачам права на викладання змін. Кожен може створити форк проекту, викласти до нього зміни, та додати свої зміни до оригінального сховища за допомогою "Запита на Забирання Змін" (Pull Request), про який ми поговоримо далі. Цей запит відкриває нитку дискусії з можливістю переглядати код, власник та автор змін можуть спілкуватися про зміни доки власник не стане ними задоволений, після чого власник може злити зміни до свого сховища.

Щоб зробити форк проекту, зайдіть на сторінку проекту та натисніть на кнопку "Fork" зверху праворуч.

Через декілька секунд, ви опинитесь на сторінці вашого нового проекту, з вашою власною копією коду, яку ви можете змінювати.

Здебільшого працюючи над комерційними або власними проектами вам надають повний доступ до репозиторію, в такому випадку немає змісту робити операцію fork.

GitHub був спроектований навколо конкретного методу співпраці, в центрі якого Запит на Пул. Цей метод працює і в разі співпраці у маленькій команді в одному спільному сховищі, і в глобальній розподіленій компанії, чи мережі незнайомців, які допомагають проекту через десятки форків. Він базується на понятті гілок у Git. Гілка являє собою копію репозиторія в яку розробник вносить свої зміни. В подальшому гілки можуть об'єднуватися у одну.

Ось як це в цілому працює робота з децентралізованою системою контролю версій:

1. Створіть форк проекту.
2. Створіть гілку на основі master, в ній ви будете робити всі свої зміни.
3. Зробіть якісь коміти, що поліпшують проект.
4. Викладайте цю гілку до свого проекту GitHub.
5. Відкрийте Запит на Пул за допомогою GitHub.
6. Обговоріть зміни, можливо зробіть ще декілька комітів.
7. Власник проекту заливає до проекту Запит на Пул, або закриває його.

Детально розглянемо приклад того, як запропонувати зміни до проекту з відкритим кодом, що зберігає код на GitHub, за допомогою цієї схеми.


Реальний приклад

Студент шукає код, який можна запустити на його мікроконтролері на платі Arduino, та знайшов чудовий програмний файл на GitHub за адресою <https://github.com/schacon/blink> (Рис Лб.1).

Єдине, що нас турбує, - це висока частота миготіння. Ми вважаємо, що краще зменшити інтервал між змінами стану з 1 мілісекунди до 3 секунд. Тому ми вдосконалюємо програму та пропонуємо її зміни до проекту.

Спочатку натискаємо кнопку "Fork", як ми раніше розглядали, щоб отримати свою власну копію проекту. Наше ім'я користувача тут - "tonychacon", отже, адреса нашої копії проекту буде <https://github.com/tonychacon/blink>, і саме за цією адресою ми можемо вносити зміни. Ми робимо локальний клон, створюємо гілку для наших змін, вносимо зміни в код і нарешті публікуємо ці зміни назад на GitHub.

branch: master blink / blink.ino

 schacon on Jun 12 my arduino blinking code (from arduino.cc)

1 contributor

25 lines (20 sloc) 0.71 kb

Raw Blame History

```

1  /*
2   * Blink
3   * Turns on an LED on for one second, then off for one second, repeatedly.
4   *
5   * This example code is in the public domain.
6   */
7
8   // Pin 13 has an LED connected on most Arduino boards.
9   // give it a name:
10  int led = 13;
11
12  // the setup routine runs once when you press reset:
13  void setup() {
14    // initialize the digital pin as an output.
15    pinMode(led, OUTPUT);
16  }
17
18  // the loop routine runs over and over again forever:
19  void loop() {
20    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
21    delay(1000);             // wait for a second
22    digitalWrite(led, LOW);  // turn the LED off by making the voltage LOW
23    delay(1000);             // wait for a second
24  }

```

Рисунок Л6.1 Проект в якому ми хочемо зробити внесок (приклад взято з сайту <https://git-scm.com/>)

```
$ git clone https://github.com/tonychacon/blink (1)
Cloning into 'blink'...
```

```
$ cd blink
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'
```

```
$ sed -i '' 's/1000/3000/' blink.ino (macOS) (3)
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino (3)
```

```
$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage
level)
    [-delay(1000);-]{+delay(3000);+} // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage
LOW

```

```
    [-delay(1000);-]{+delay(3000);+}           // wait for a second
}
```

```
$ git commit -a -m 'three seconds is better' (5)
[slow-blink 5ca509d] three seconds is better
1 file changed, 2 insertions(+), 2 deletions(-)
```

```
$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

1. Ми починаємо з клонування нашого форка проекту локально.
2. Потім ми створюємо окрему гілку з чіткою назвою.
3. Після цього ми вносимо зміни в код.
4. Перевіряємо, щоб бути впевненими, що зміни є правильними.
5. Потім ми робимо коміт наших змін у свою гілку.
6. Наступний крок - викладання нашої нової гілки у наш форк на GitHub.

Тепер, коли ми повертаємося до нашого форка на GitHub, ми бачимо, що GitHub помітив нашу нову гілку і відображає велику зелену кнопку (Compare & pull request), щоб переглянути наші зміни та відкрити запит на злиття у вихідний проект.

Також ви можете перейти на сторінку "Branches" (гілки) за адресою <https://github.com/<user>/<project>/branches>, щоб знайти вашу гілку та відкрити новий запит на злиття звідти.

Якщо ми натиснемо на цю зелену кнопку, ми перейдемо на сторінку, де буде запропоновано надати назву та опис нашому запиту на злиття. Зазвичай варто відвести деякий час на це, оскільки якісний опис допомагає власнику оригінального проекту зрозуміти ваші наміри та перевірити, чи пропоновані зміни коректні, і чи вони дійсно покращують проект. Також ми бачимо список комітів вашої гілки, які перевищують гілку master (у цьому випадку може бути лише один коміт), а також об'єднані різниці (unified diff) для всіх змін, які будуть застосовані, якщо ця гілка буде злита у вихідний проект. Коли ви натиснете кнопку Create pull request (створити запит на пул) на цій сторінці, власник проекту, від якого ви створили форк, отримає повідомлення про те, що хтось пропонує зміни з посиланням на сторінку, що містить усю інформацію про ці зміни.

Тепер власник проекту має можливість переглянути пропоновані зміни та злити їх з основним проектом, відхилити їх або залишити коментарі. Нехай припустимо, що власнику сподобалася ваша ідея, але він хоче, щоб світлодіод був увімкненим та вимкненим трохи довше. На GitHub вся комунікація відбувається прямо на сайті. Власник проекту може переглянути об'єднану різницю (unified diff) та залишити коментарі, натискаючи на будь-який рядок (Рис Л.6.2).

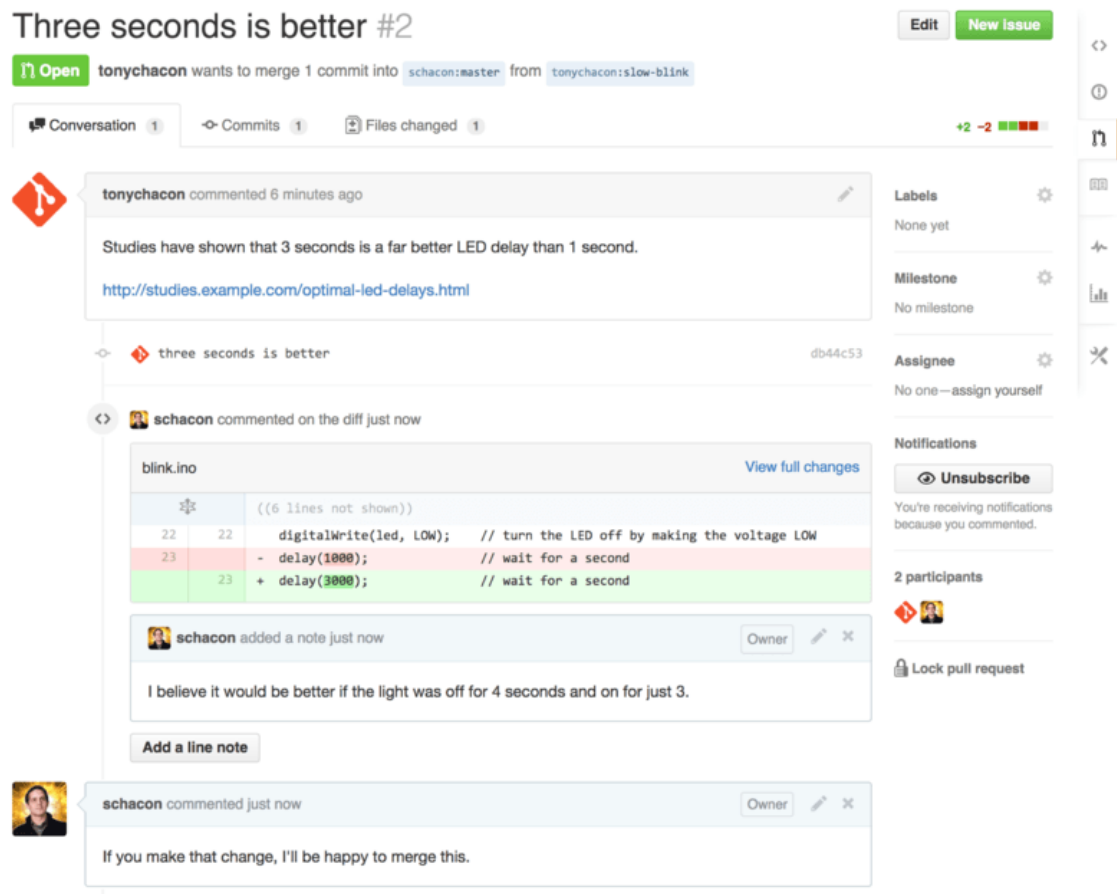


Рисунок Л.6.2 Приклад роботи з пул ріквестом (з сайту <https://git-scm.com/>)

Коли супроводжувач (maintainer) зробить цей коментар, людина, що відкрила Запит на Пул (та насправді будь-хто, хто слідкує за сховищем) отримає повідомлення (зазвичай на електронну пошту). Хто завгодно може залишати загальні коментарі до Запиту на Пул. Ви можете побачити, що коментарі до коду також є частиною обговорення. Тепер автори змін можуть бачити, що їм треба зробити, щоб їх зміни прийняли. На щастя це дуже просто зробити. Якби ви використовували пошту для спілкування, вам довелося б переробити ваші зміни та знову відправляти їх до поштової розсилки (mailing list), а за допомогою GitHub ви можете просто знову зробити коміт до своєї гілки та викласти зміни до GitHub, що автоматично оновить Запит на злиття. На Фінальний Запит на Пул ви також можете бачити, що коментарі до старого коду сховано в оновленому запиті на злиття, оскільки вони були зроблені до рядків, які відтоді змінилися.

Додавання комітів до існуючого запиту на злиття не спричиняє повідомлень, отже щойно студент надіслав свої виправлення, він вирішує залишити коментар, щоб проінформувати власників проекту, що він зробив потрібну зміну.

Детальніше про пул ріквест.

Ми розглянули основи того, як робити внесок до проекту на GitHub, розглянемо декілька цікавих порад та хитрощів про Запити на Пул, щоб ви могли користуватись ними ефективніше. Запити на Пул як Патчі, важливо розуміти, що багато проектів не розглядають Запити на Пул як чергу ідеальних патчів, які мають чисто накладатись саме в цьому порядку, як більшість основаних на поштових списках проектів розглядає послідовність патчів. Більшість проектів GitHub розглядають гілки Запитів на Пул як інтерактивну бесіду про запропоновані зміни, їх цікавить тільки результуючі зміни, які додаються до проекту за допомогою злиття. Це важлива відмінність, адже зазвичай зміни пропонуються до того, як код доводять до ідеалу, що є дуже рідкісним у проектах, які використовують поштові списки з послідовностями патчів для співпраці. Це дозволяє обговоренню з супроводжувачем проекту розпочатись раніше, тому вірне рішення досягається спільними зусиллями. Коли код пропонується за допомогою Запита на Пул, та супроводжувачі або спільнота пропонує зміну, послідовність патчів зазвичай не створюється заново, а замість цього зміни просто викладаються як новий коміт до гілки, що просуває обговорення далі без втрати контексту попередньої праці. Наприклад, якщо ви повернетесь та знову подивитесь на Фінальний Запит на Пул, ви можете помітити, що автор змін не робив перебазування свого коміту та не відправляв ще один Запит на Пул. Натомість він додав нові коміти та виклав їх до існуючої гілки. Таким чином, якщо ви повернетесь до цього Запиту на Пул у майбутньому, ви легко знайдете весь контекст всіх прийнятих рішень. При натисканні на кнопку “Merge” (злити) на сайті, GitHub навмисно створює коміт злиття з посиланням на Запит на Пул, щоб було легко повернутись та досліджувати обговорення при необхідності.

Якщо ваш Запит на Пул стає застарілим, чи не може бути чисто злитим через щось інше, ви забажаєте виправити його, щоб супроводжувач проекту міг легко злити ваші зміни. GitHub перевіряє це та повідомляє вам наприкінці кожного Запиту на Пул чи може він бути злитим без конфліктів..

Якщо ви бачите щось схоже на Запит на Пул не може бути злитим чисто, ви захочете виправити свою гілку щоб зображення стало зеленим та супроводжувачу не довелося виконувати зайвої праці. У вас є два основних варіанти, як це зробити. Ви можете перебазувати вашу гілку поверх цільової гілки (зазвичай це гілка master сховища, від якого ви зробили форк), або ви можете злити цільову гілку до вашої гілки. Більшість розробників на GitHub обирають другий варіант через вже зазначені раніше причини. Важлива

історія та фінальне злиття, а перебазування дає нам тільки трохи чистішу історію натомість воно набагато складніше та в цьому варіанті легше набриднути помилок. Якщо ви бажаєте злити цільову гілку до вашої, щоб з'явилася можливість чисто злити ваш Запит на Пул, вам слід додати оригінальне сховище як нове віддалене сховище, зробити з нього фетч, злити головну гілку цього сховища до вашої гілки, виправити будь-які проблеми та нарешті залити це до гілки, на яку відкритий Запит на Пул.

Створення нового репозиторію

Створимо нове сховище, до якого ми додамо код нашого проекту. Спочатку натиснемо кнопку “New repository” (нове сховище) праворуч панелі керування, чи за допомогою кнопки + у верхній панелі інструментів біля вашого імені користувача, як можна побачити в “New repository”.

Вам потрібно лише вказати ім'я проекту. Усі інші поля абсолютно необов'язкові (вибирайте публічний репозиторій public). Просто натисніть кнопку "Створити сховище", і ось воно - ваше нове сховище на GitHub, під назвою <ім'я користувача>/<назва проекту>. Так як у вашому проекті наразі немає коду, GitHub покаже вам інструкції щодо створення абсолютно нового сховища Git або приєднання існуючого проекту Git.

Тепер у вас є проект на GitHub, і ви можете поділитися URL з будь-ким, з ким бажаєте поділитися своїм проектом. Кожен проект на GitHub доступний через HTTPS за адресою `https://github.com/<user>/<project_name>` та через SSH за адресою `git@github.com:<user>/<project_name>`. Git може отримувати та викладати зміни, використовуючи обидва URL, проте вони мають контроль доступу, що базується на ім'я користувача/паролі.

Додавання друзів до репозиторію та інші нюанси

Якщо ви працюєте з іншими людьми, та бажаєте надати їм право робити коміти, ви маєте додати їх до “співпрацівників” (collaborators). Якщо усі члени вашої команди мають облікові записи на GitHub, та ви бажаєте надати їм доступ на запис до вашого сховища, ви можете додати їх до свого проекту. Це надасть їм можливість робити “push”, тобто вони матимуть доступ і на читання, і на запис до проекту та сховища Git.

Крім особистих облікових записів, на GitHub також існують організації. Як і в особистих облікових записах, у облікових записах організацій є свій простір імен, де розташовані всі їхні проекти, але є декілька відмінностей. Організаційні облікові записи представляють групу людей з спільним правом власності проектів, і для керування підгрупами цих людей є багато інструментів. Зазвичай організації використовуються для груп з відкритим кодом (таких як "perl" або "rails") або компаній (таких як "google" або "twitter").

Створення організації досить просте: просто натисніть на іконку "+" у верхньому правому куті на будь-якій сторінці GitHub і виберіть у меню "Нова організація".

Спочатку вам потрібно назвати вашу організацію та вказати поштову адресу, яка буде головним контактом групи. Потім ви можете запросити інших користувачів бути співвласниками облікового запису, якщо бажаєте. Виконайте ці кроки, і невдовзі ви станете власником нової організації. Як і в особистих облікових записах, організації є безкоштовними, якщо усе, що ви зберігаєте в них, є відкритим кодом. Як власник організації, коли ви робите форк сховища, у вас є вибір: робити форк у вашому власному просторі імен або у просторі імен організації. Коли ви створюєте нові сховища, ви можете створити їх під особистим обліковим записом або під будь-якою організацією, яка належить вам. Також ви автоматично станете "підписником" всіх сховищ, які ви створили для цих організацій.

Так само, як і для вашого аватара, ви можете завантажити аватар для вашої організації, щоб трохи надати їй особливого стилю. Також, як і з особистими обліковими записами, у вас є головна сторінка організації, де перелічені всі ваші сховища, і її можуть бачити інші користувачі.

Команди

Організації пов'язані з окремими людьми через команди, які є простим групуванням окремих облікових записів і сховищ в організації, а також надають можливість керувати рівнем доступу цих людей до цих сховищ. Наприклад, припустимо, у вашій компанії є три сховища: frontend, backend та deployscripts. Ви бажаєте, щоб ваші розробники HTML/CSS/JavaScript мали доступ до frontend та, можливо, backend, а ваші співробітники з Операційного відділу мали доступ до backend та deployscripts. За допомогою команд цього легко досягти, необхідно керувати співробітниками для кожного окремого сховища.

На сторінці організації є проста панель інструментів зі всіма сховищами, користувачами та командами, що належать до цієї організації.

Щоб керувати вашими командами, ви можете натиснути на бічну панель "Команди" праворуч на сторінці сторінки організації. Тоді ви потрапите на сторінку, де ви зможете додавати користувачів до команди, додавати сховища до команди та керувати налаштуваннями та рівнем доступу команди. Кожна команда може мати доступ тільки на читання, доступ на читання та запис або доступ адміністрування до сховищ. Ви можете змінити рівень, натиснувши кнопку "Налаштування" на сторінці команди.

Коли ви запрошуєте когось до команди, вони отримають листа, що повідомить їм про запрошення.

Крім того, @згадки команди (такі як @acmecorp/frontend) працюють так само, як і для окремих користувачів, за винятком того, що всі

користувачі команди підписані на ці повідомлення. Це корисно, якщо ви хочете звернути увагу на щось комусь з команди, але не знаєте, кому саме.

Користувач може бути в декількох командах, тому не обмежуйте себе тільки командами для контролю рівня доступу. Команди спеціальних інтересів, такі як `ux`, `css` або `refactoring`, корисні для деяких типів питань, а інші команди, такі як `legal` або `colorblind`, для зовсім інших.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Створити профіль уGitHub.
3. Створіть публічний репозиторій.
4. Додайте проект з попередньої лабораторної роботи до віддаленого репозитору.
5. Створіть гілку “`branch-1`” на локальному репозиторії і зробіть декілька комітів.
6. Запускайте гілку “`branch-1`” до ремотного репозитору.
7. Створіть пул ріквест.
8. Зробіть ревью пул ріквесту та замірджіть його.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (описати ваш git репозиторій, навести послідовність виконаних команд).
4. Висновок. Зробіть акцент на перевагах використання децентралізованих систем контролю версій порівняно з централізованими системами версій.
5. Список використаної літератури.

Література:

6. Scott Chacon, Ben Straub. Pro Git. Apress. 2014. 456 p.
7. Jon Loeliger, Matthew McCullough. Version Control with Git. O'Reilly Media. 2012. 456 p.
8. Ryan Hodson. Git Essentials. Packt Publishing. 2015. 194 p.
9. Roger Dudler. Git Pocket Guide. O'Reilly Media. 2013. 234 p.
10. Seth Robertson, Steve Oualine. Practical Git: For Everyday Use. No Starch Press. 2014. 300 p.

ЛАБОРАТОРНА РОБОТА №7

«НЕПЕРВНА ІНТЕГРАЦІЯ ТА НЕПЕРЕРВНА ДОСТАВКА (CI/CD). ВПРОВАДЖЕННЯ CI/CD ВИКОРИСТОВУЮЧИ ПРОГРАМНИЙ ПАКЕТ JENKINS»

Мета роботи:

Мета лабораторної роботи полягає в тому, щоб студенти ознайомилися з процесом впровадження безперервної інтеграції та безперервної доставки (CI/CD) у реальному проекті за допомогою популярного інструменту Jenkins.

Методичні вказівки:

На основі лабораторної роботи студенти вивчать основні концепції CI/CD, налаштовують програмний пакет Jenkins для автоматизації процесу збирання, тестування, розгортання та відстеження програмного забезпечення. Особлива увага приділена побудові Jenkins Pipeline - центрального елементу для опису та автоматизації кожного кроку у процесі CI/CD. Навички, отримані під час цієї лабораторної роботи, допоможуть студентам розуміти та ефективно використовувати практики CI/CD у своїй подальшій професійній діяльності.

Вимоги до результатів виконання лабораторної роботи:

1. Продемонструвати налаштований програмний пакет Jenkins відповідно до обраного проекту.
2. Описати створений Jenkins Pipeline. Обґрунтувати кожен крок у Jenkins Pipeline
3. Проаналізувати плюси та мінуси використання програмного пакету Jenkins для впровадження CI/CD процесів.

Теоретичні відомості:

Неперервна інтеграція(CI)

Неперервна інтеграція - це практика розробки програмного забезпечення в рамках DevOps, при якій розробники регулярно об'єднують свої зміни коду в центральному репозиторії, після чого автоматично запускаються процеси збірки та тестування. Неperервна інтеграція найчастіше стосується етапу збірки або інтеграції процесу випуску програмного забезпечення і передбачає як автоматизований компонент (наприклад, сервіс CI або збірки), так і культурний компонент (наприклад, навчання інтеграції з високою частотою). Основні цілі неперервної інтеграції - це виявлення та виправлення помилок швидше, покращення якості програмного забезпечення та скорочення часу на перевірку та випуск нових оновлень програмного забезпечення.

У минулому розробники у команді могли працювати ізольовано протягом тривалого часу і об'єднувати свої зміни з гілки master лише після завершення своєї роботи. Це ускладнювало та забиало багато часу на об'єднання змін коду, а також призводило до накопичення помилок протягом тривалого часу без виправлення. Ці фактори ускладнювали швидку доставку оновлень клієнтам.

З неперервною інтеграцією розробники часто комітують свої зміни в спільний репозиторій, використовуючи систему контролю версій, таку як Git. Перед кожним комітом розробники можуть вирішити запустити локальні модульні тести на своєму коді як додатковий рівень перевірки перед інтеграцією. Сервіс неперервної інтеграції автоматично будує та запускає модульні тести на нових змінах коду, щоб відразу виявити будь-які помилки.

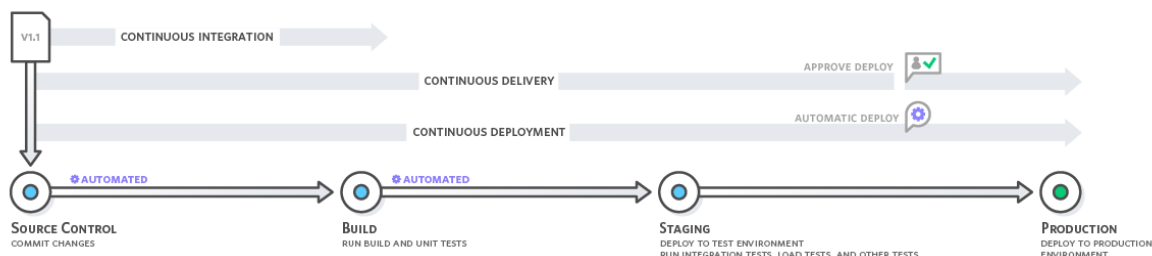


Рисунок Л7.1 Приклад впровадження CI компанією aws (<https://aws.amazon.com/>)

Неперервна інтеграція відноситься до етапів збірки та модульного тестування процесу випуску програмного забезпечення. Кожна ревізія, що комітиться, спричиняє автоматичну збірку та тестування.

Неперервна доставка передбачає автоматичну збірку, тестування та підготовку кодових змін до випуску в продукцію. Неперервна доставка розширює можливості неперервної інтеграції шляхом розгортання всіх змін коду в середовище тестування та/або в середовище продукції після етапу збірки.

Неперервна доставка(CD)

Неперервна доставка - це практика розробки програмного забезпечення, при якій зміни в коді автоматично підготовлюються для випуску в продукцію. Як один з основних принципів сучасного розвитку додатків, неперервна доставка розширює можливості неперервної інтеграції шляхом розгортання всіх змін коду в середовище тестування та/або в середовище продукції після етапу збірки. При правильній реалізації розробники завжди матимуть готовий до розгортання артефакт збірки, який пройшов стандартизований тестовий процес.

Неперервна доставка дозволяє розробникам автоматизувати тестування поза межами лише модульних тестів, щоб перевірити оновлення додатка в різних аспектах перед розгортанням для клієнтів. Ці тести можуть включати тестування користувацького інтерфейсу, навантаження, інтеграції, надійності API тощо. Це допомагає розробникам більш повноцінно перевіряти оновлення та передчасно виявляти проблеми. За допомогою хмарних технологій легко та вигідно автоматизувати створення та реплікацію кількох середовищ для тестування, що раніше було складно зробити на власних серверах.

З неперервною доставкою кожна зміна коду збирається, тестується, а потім відправляється до тестового або стейджингового середовища, яке не призначене для продуктивного використання. Існують можливість наявності кількох паралельних етапів тестування перед розгортанням в продукцію. Відмінністю між неперервною доставкою та неперервним розгортанням є наявність ручного підтвердження перед оновленням продукційного середовища. З неперервним розгортанням, розгортання в продукцію відбувається автоматично без явного підтвердження.

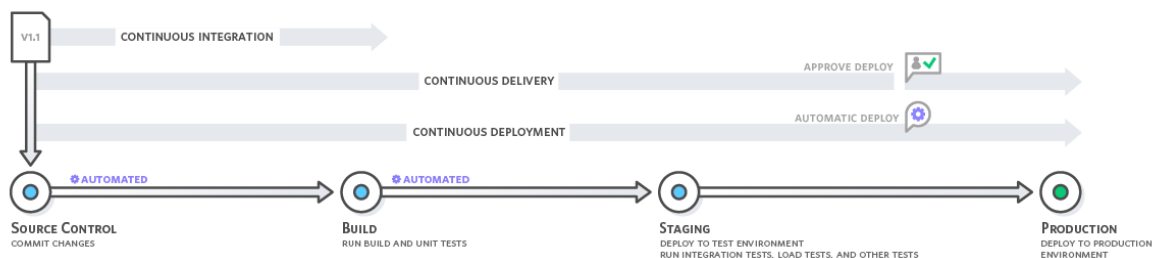


Рисунок Л7.2 Приклад впровадження CI/CD компанією aws (<https://aws.amazon.com/>)

Неперервна доставка автоматизує весь процес випуску програмного забезпечення. Кожна ревізія, яка комітиться, спричиняє автоматичний потік, що збирає, тестує та потім стадійно оновлює систему. Остаточне рішення про розгортання в живе виробниче середовище приймається розробником.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Встановіть програмний пакет Jenkins.
3. Створіть Jenkins pipeline з наступними етапами: збирання проекту(вихідний код знаходиться у GitHub репозиторії), запуск тестів, доставка складеного продукту (в рамках доставки потрібно помістити складений проект у папку `cargo` на робочому комп'ютері).
4. Успішно виконайте створений Jenkins pipeline

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (описати побудований Jenkins pipeline).
4. Висновок. Зробіть акцент на доцільності використання CI/CD процесу.
5. Список використаної літератури.

Література:

- 11.Scott Chacon, Ben Straub. Pro Git. Apress. 2014. 456 p.
- 12.Jon Loeliger, Matthew McCullough. Version Control with Git. O'Reilly Media. 2012. 456 p.
- 13.Ryan Hodson. Git Essentials. Packt Publishing. 2015. 194 p.
- 14.Roger Dudler. Git Pocket Guide. O'Reilly Media. 2013. 234 p.
- 15.Seth Robertson, Steve Oualline. Practical Git: For Everyday Use. No Starch Press. 2014. 300 p.

ЛАБОРАТОРНА РОБОТА №8

«НЕПЕРВНА ІНТЕГРАЦІЯ ТА НЕПЕРЕРВНА ДОСТАВКА (CI/CD). ВПРОВАДЖЕННЯ CI/CD ВИКОРИСТОВУЮЧИ ВЕБ СЕРВІС GITHUB ACTIONS WORKFLOW»

Мета роботи:

Мета лабораторної роботи полягає в тому, щоб студенти ознайомилися з процесом впровадження безперервної інтеграції та безперервної доставки (CI/CD) у реальному проекті за допомогою популярного веб ресурсу GitHub actions workflow.

Методичні вказівки:

На основі лабораторної роботи студенти глибше пізнають концепції CI/CD, налаштовують GitHub actions workflow для автоматизації процесу збирання, тестування, розгортання та відстеження програмного забезпечення. Особлива увага приділена ознайомленню з сервісом GitHub як альтернативній заміні програмного пакету Jenkins. Навички, отримані під час цієї лабораторної роботи, допоможуть студентам розуміти та ефективно використовувати практики CI/CD у своїй подальшій професійній діяльності.

Вимоги до результатів виконання лабораторної роботи:

4. Продемонструвати зконфігурований GitHub actions workflow для обраного проекту.
5. Описати створений GitHub actions workflow. Обґрунтувати кожен крок у Actions workflow
6. Проаналізувати плюси та мінуси використання програмного пакету Jenkins для впровадження CI/CD процесів.

Теоретичні відомості:

Неперервна інтеграція(CI)

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Ознайомитися з веб ресурсом GitHub (якщо Ви ще не зареєструвалися, зареєструйтесь використовуючи університетську емейл адресу).
3. Для існуючого репозиторію з проектом створіть GitHub action з наступними етапами: збирання проекту(вихідний код знаходиться у GitHub репозиторії), запуск тестів.
4. Успішно виконайте створений GitHub action.

Зміст звіту:

6. Мета роботи.
7. Введення.
8. Основна частина (описати побудований GitHub action).
9. Висновок. Зробіть порівняння між програмним пакетом Jenkins та веб ресурсом GitHub actions.
- 10.Список використаної літератури.

Література:

- 16.Scott Chacon, Ben Straub. Pro Git. Apress. 2014. 456 p.
- 17.Jon Loeliger, Matthew McCullough. Version Control with Git. O'Reilly Media. 2012. 456 p.
- 18.Ryan Hodson. Git Essentials. Packt Publishing. 2015. 194 p.
- 19.Roger Dudler. Git Pocket Guide. O'Reilly Media. 2013. 234 p.
- 20.Seth Robertson, Steve Oualline. Practical Git: For Everyday Use. No Starch Press. 2014. 300 p.

ЛАБОРАТОРНА РОБОТА №9

«ВЕДЕННЯ ДОКУМЕНТАЦІЯ У ІТ ПРОЕКТІ»

Мета роботи:

Мета лабораторної роботи полягає в створенні найбільш поширених документів які супроводжуються ІТ проект.

Методичні вказівки:

На основі лабораторної роботи студенти отримає практичні навички ведення документації при розробці програмного забезпечення в межах ІТ проектів.

Вимоги до результатів виконання лабораторної роботи:

1. Продемонструвати Atlasian Confluence workspace для обраного проекту.
2. Описати створений Atlasian Confluence workspace.
3. Проаналізувати плюси та мінуси використання Atlasian Confluence workspace як інструменту для ведення супроводжуючої документації.

Теоретичні відомості:

Неперервна інтеграція(CI)

.

Порядок виконання роботи:

1. Освоїти вище наведений теоретичний матеріал.
2. Ознайомитися з веб ресурсом Atlasian Confluence (якщо Ви ще не зареєструвалися, зареєструйтесь використовуючи університетську емейл адресу).
3. Для Вашого проекту створити Atlasian Confluence workspace та наповнити наступні сторінки: Project base line, загальна архітектура проекту (component diagram, відомості про репозиторії, CI/CD), основні бізнес процеси (sequence diagram), команда (відомості про команду).
4. Проаналізуйте інші інструменти для ведення супроводжуючої документації.

Зміст звіту:

1. Мета роботи.
2. Введення.
3. Основна частина (Короткий опис основних елементів вашого Atlasian Confluence workspace).
4. Висновок. Зробіть акцент на порівнянні різних інструментів для ведення супроводжуючої документації для ІТ проектів .
5. Список використаної літератури.

Література:

- 21.Scott Chacon, Ben Straub. Pro Git. Apress. 2014. 456 p.
- 22.Jon Loeliger, Matthew McCullough. Version Control with Git. O'Reilly Media. 2012. 456 p.
- 23.Ryan Hodson. Git Essentials. Packt Publishing. 2015. 194 p.
- 24.Roger Dudler. Git Pocket Guide. O'Reilly Media. 2013. 234 p.
- 25.Seth Robertson, Steve Oualline. Practical Git: For Everyday Use. No Starch Press. 2014. 300 p.