

Міністерство освіти і науки України
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт
з дисципліни

“Якість інформаційних систем”

для студентів
напряму підготовки 6.050101
“Комп’ютерні науки”
всіх форм навчання

2016

Методичні вказівки до лабораторних робіт з дисципліни “Якість інформаційних систем” для студентів напряму підготовки 6.050101 “Комп’ютерні науки” всіх форм навчання / Г.В Табунщик, Т.І. Каплієнко, Н.О. Миронова. – Запоріжжя: ЗНТУ, 2016. – 62 с.

Автори: Г. В. Табунщик, к.т.н., проф.
Т.І. Каплієнко, к.т.н., доцент
Н. О. Миронова, к.т.н., доцент

Рецензент: Р.К.Кудерметов, к.т.н., доцент

Відповідальний
за випуск: Г. В. Табунщик, к.т.н., проф.

Затверджено
на засіданні кафедри
програмних засобів

Протокол №
від “ ” _____ 2016 р.

ЗМІСТ

Вступ.....	4
Лабораторна робота №1 Обчислення метрик розміру та складності програм.....	6
Лабораторна робота №2 Обчислення об'єктно-орієнтованих метрик.....	28
Лабораторна робота №3 Основи GIT	34
Лабораторна робота №4 Робота з віддаленим репозиторієм GIT	50

ВСТУП

Метою даного курсу є отримання теоретичних основ та практичних навичок з оцінки та забезпечення якості програмних продуктів. Дисципліна "Якість інформаційних систем" спрямована на поглиблення знань та практичних навичок студентів з сучасної технології створення складних програмних продуктів на базі ідей і принципів об'єктно-орієнтованого методу.

На основі вивчення дисципліни "Якість інформаційних систем" студент повинен знати:

- етапи життєвого циклу програмного забезпечення;
- метрики програмного забезпечення;
- методи керування якістю відносно до розробки проектуванню та експлуатації програмних продуктів;
- CASE-засоби для тестування програмного забезпечення.

Студент повинен вміти:

- оцінювати якість та ефективність програмного забезпечення при рішенні завдань розробки;
- перевіряти коректність;
- виконувати тестування;
- проводити вибір на вдосконалення програмного забезпечення.

Особливу увагу треба приділяти правильному оформленню звітів з практичних робіт. Звіт повинен містити: титульний аркуш (на ньому вказують назву міністерства, назву університету, назву кафедри, номер, вид і тему роботи, виконавця та особу, що приймає звіт, рік); мету, варіант і завдання роботи; лаконічний опис теоретичних відомостей; текст програми, що обов'язково містить коментарі; змістовний аналіз отриманих результатів та висновки.

Звіт виконують на білому папері формату А4 (210 x 297 мм). Текст розміщують тільки з однієї сторони листа. Поля сторінки з усіх боків – 20 мм. Аркуші скріплюють за допомогою канцелярських скріпок. Для набору тексту звіту використовують редактор MS Word: шрифт Times New Roman, 14 пунктів. Міжрядковий інтервал: полуторний – для тексту звіту, одинарний – для листингів програм і роздруківок даних.

Під час співбесіди студент повинний виявити знання про мету роботи, по теоретичному матеріалу, про методи виконання кожного етапу роботи, по змісту основних розділів розробленого звіту з демонстрацією результатів на конкретних прикладах. Студент повинний вміти правильно аналізувати отримані результати. Для

самоперевірки при підготовці до виконання і здачі роботи студент повинний відповісти на контрольні питання, приведені наприкінці опису відповідної роботи. Загальний залік студент одержує після виконання і здачі останньої роботи.

ЛАБОРАТОРНА РОБОТА №1

ОБЧИСЛЕННЯ МЕТРИК РОЗМІРУ ТА СКЛАДНОСТІ ПРОГРАМ

1.1 Мета роботи

Вивчити основні метрики коду програмних систем та реалізувати їх із застосуванням мови C++ та Qt.

1.2 Основні теоретичні відомості

Атрибути програмної системи, що характеризують її якість, вимірюються з використанням метрик якості.

Метрика – це комбінація конкретного методу вимірювання (способу отримання значень) атрибута сутності та шкали вимірювання (засобу, використовуваного для структурування одержуваних значень).

Метрика визначає *міру атрибуту* – змінну, якій присвоюється значення в результаті вимірювання. Наприклад, сутність – «звіт про виявлені дефекти», атрибут – «список дефектів», метод вимірювання – «підрахунок кількості дефектів у списку», шкала – «цілі числа більше нуля», міра атрибуту – «загальне число дефектів», ім'я метрики (зазвичай однойменне мірою) – «загальне число дефектів».

Міра атрибута може бути безпосередньою, якщо вона не залежить від заходів інших атрибутів, або непрямою, отриманої по заходам інших атрибутів.

За визначенням стандарту *ISO/IEC 9126-2* метрика якості програмної системи являє собою «модель вимірювання атрибута, який пов'язує з характеристикою якості ПЗ. Служить індикатором одного або багатьох атрибутів».

Метрика називається *базовою*, якщо в її основі лежить елементарний метод (примітив) вимірювання атрибута.

Для того щоб правильно користуватися результатами вимірювань, для кожної міри потрібно ідентифікувати шкалу вимірювання.

Стандарт *ISO/IEC 9126-2* рекомендує застосовувати 5 видів шкали вимірювання значень (впорядкованих від менш суворих до більш суворим):

- *номінальна шкала*. Це класифікаційна шкала, що виконує категоризацію властивостей об'єкта, що оцінюється. Категорії не

впорядковані. Наприклад, дефекти можуть класифікуватися на дефекти інтерфейсу, логіки, оголошення даних та ін.;

– **порядкова шкала.** Дозволяє упорядковувати характеристики за зростанням або убуття шляхом порівняння їх з базовими значеннями. Наприклад, для рівня серйозності наслідків події шкала може містити значення «низький», «середній», «високий», «критичний». Для рівнів СММ – {1, 2, 3, 4, 5}. Відстань між значеннями за шкалою не грає ролі. Характеристики, що мають номінальну або порядкову шкалу вимірювання, називаються якісними.

– **інтервальна шкала.** Відзначає істотні відмінності властивостей об'єкта, «дистанцію» між ними (наприклад, календарні дати або значення щільності дефектів. Використовується в арифметичних операціях і операціях порівняння. Нульове значення не пропустимо;

– **відносна шкала.** Значення за цією шкалою розрізняються по відношенню до вибраної одиниці (наприклад, часу, змінюваному від 0 до безкінечності, або вартості). Застосовуючи цю шкалу можна розрахувати, наприклад, час між відмовами, розмір програмного компонента в SLOC та ін. Вважається найбільш кращою шкалою вимірів. Дозволяє застосовувати широкий спектр інструментів вимірювання (гістограм, діаграм Парето та ін.);

– **абсолютна шкала.** Це спеціальний випадок відносної шкали. У цій шкалі вказується абсолютне значення величини. Наприклад: «розмір програми дорівнює 2К», «число виявлених помилок одно 20».

Виміряне значення метрики саме по собі не несе інформації про рівень задоволення вимог до якості. Для цих цілей шкала повинна бути розділена на області (ранги), які відповідають різним ступеням задоволення вимог. Приклади розподілу шкали:

– розподіл значень за двома категоріями – задовільні і незадовільні значення;

– поділ шкали за чотирма категоріями, обмеженим трьома рівнями значень – поточним, гіршим і плановим.

По мірі накопичення практики вимірювань і знань про вивчаємі атрибути шкал їх вимірювання можуть еволюціонувати від менш інформативних (номінальної і порядкової) до більш інформативним (відносної чи абсолютної).

Для розробки процедур збирання даних, інтерпретації заходів та їх нормалізації з метою порівняння, потрібно розрізняти такі типи міри, визначених метриками.

Міри розміру. Показують розмір ПЗ в різних одиницях виміру, наприклад:

- **функціональний розмір** – облік функціональних можливостей ПЗ;
- **розмір програми** – облік кількості рядків вихідного коду, кількості модулів, кількості операторів на мові програмування;
- **обсяг ресурсів**, що використовуються працюючою програмою – облік об'єму оперативної пам'яті, дискової пам'яті або мережевого трафіку, завантаженості процесора – кількості оброблюваних інструкцій у секунду і ін.;

Міри часу. Наводять періоди реального часу (в секундах, хвилинах або годинах), процесорного часу (в секундах, хвилинах або годинах роботи процесора) або календарного часу (в робочих годинах, календарних днях, місяцях, роках), наприклад:

- **час функціонування системи** з безперервною або дискретною роботою компонентів програмного забезпечення (облік минулого часу (при роботі програмного забезпечення в системі протягом встановленого періоду часу), облік часу з моменту включення (при безперервному використанні вбудованого програмного забезпечення системи реального часу), облік нормалізованого часу (при спільній роботі кількох комп'ютерів, що обмінюються даними);

- **час виконання** (час, необхідний працюючому програмному компоненту системи для завершення розв'язання певної задачі в певних умовах);

- **час використання** (час, що витрачається певним користувачем на вирішення завдань з допомогою ПЗ), наприклад: час сеансу роботи (від початку до завершення), час вирішення задачі і ін.;

Міри зусиль. Наводять корисний (продуктивний) час, пов'язаний з певним завданням проекту, наприклад:

- продуктивність праці (при індивідуальній роботі);
- трудомісткість (при колективній роботі);

Міри інтервалів між подіями. Це інтервали часу між настанням подій, що відбуваються в певний період спостереження, наприклад, час між послідовними відмовами. Замість цієї міри може використовуватися частота настання подій;

Лічильні міри. Являють собою статичні лічильники (для обліку певних елементів у робочих продуктах ПЗ (документах)) або кінетичні (динамічні) лічильники (для обліку подій або дій людини), наприклад:

- кількість виявлених помилок (облік помилок в ході інспекції, тестування, функціонування або супроводу ПЗ);
- структурна складність програми (кількість шляхів у програмі, цикломатична складність та ін.);
- кількість несумісних елементів (облік помилок узгодження (вимог, стандартів, форматів та ін.), облік відповідей типу «так» / «ні» у опитувальних);
- кількість змін – облік елементів конфігурації, в яких відбулися зміни;
- кількість виявлених відмов – облік відмов при тестуванні, функціонуванні чи супроводі ПЗ;
- кількість спроб – облік спроб корегування дефектів або помилок;
- ергономічні лічильники – число натиснень клавіш, клацань на кнопках та ін.
- лічильники-оцінки – облік результатів, зведених в опитувальних листах, контрольних листах та ін.)

При виконанні вимірювань базові міри розміру, часу і розхунки можуть використовуватися в різних комбінаціях. Вони служать основою для нормалізації та забезпечують можливість зіставлення метрик.

Для зручності застосування загальних прийомів вимірювань метрики зазвичай класифікують як:

- **об'єктивні/суб'єктивні**. Об'єктивні метрики містять підрахунки елементів, які можуть бути незалежно перевірені (кількість рядків коду, кількість помилок, складність та ін.) Вони знижують вплив особистої думки на обчислення й аналіз метрик. Суб'єктивні метрики ґрунтуються на індивідуальному чи колективному розумінні або перевазі певних характеристик або умов (рівень складності проблем, вартісні коефіцієнти та ін.);
- **примітивні/обчислювані**. Примітивні (базові) метрики можна безпосередньо спостерігати (розмір програми в KSLOC, кількість дефектів, знайдених при тестуванні та ін.) Обчислювані метрики не можуть безпосередньо спостерігатися, але можуть обчислюватися по примітивним метрик (число дефектів, що припадають на SLOC, трудомісткість та ін.);

– **динамічні/статичні.** Динамічним метрикам властивий компонент часу. Значення змінюються з плином часу, починаючи з моменту збирання даних (наприклад, кількість помилок в місяць). Статичні метрики інваріантні до часу (кількість виявлених дефектів, загальна трудомісткість робіт тощо);

– **прогнозуючі/пояснюючі.** Значення прогнозуючих метрик можуть бути отримані заздалегідь, наприклад, оцінювана інтенсивність відмов. Значення пояснюючих метрик з'являються пост-фактум, наприклад, реальна інтенсивність відмов.

По відношенню до виду об'єкта вимірювання (працююча програма або сукупність документів) міри та відповідні метрики підрозділяються на зовнішні, внутрішні і метрики використання ПЗ.

У загальному випадку застосування метрик дозволяє керівникам проєктів і підприємств вивчити складність розробленого або майбутнього проєкту, оцінити обсяг робіт, стилістику програми що розробляється і зусилля, витрачені кожним розробником для реалізації того чи іншого рішення. Однак метрики можуть служити лише рекомендаційними характеристиками, ними не можна повністю керуватися.

1.2.1 Кількісні метрики

Кількість рядків коду. Найпоширенішою метрикою вихідного коду, що відображає розмір програмного проєкту, є показник кількості рядків коду *SLOC* (*Source Lines Of Code*,). Не вдаючись в чергове перерахування недоліків даної метрики, розглянемо особливості її обчислення, а також існуючі різновиди та похідні.

Спочатку *SLOC* застосовувався в умовах, коли мови програмування були відносно прості за структурою і для них характерним було відповідність одного рядка коду одній команді мови. Згодом мови програмування еволюціонували, стали набагато гнучкіші, і ця відповідність для них більше не виконувалося – один фізичний рядок коду тепер міг містити кілька команд мови. З урахуванням цього виділяють і два основні різновиди показника *SLOC*:

– кількість фізичних *SLOC* (використовувані аббревіатури: *LOC*, *SLOC*, *KLOC*, *KSLOC*, *DSLOC*) – визначається як загальна кількість рядків коду, з урахуванням коментарів і порожніх рядків (при вимірюванні показника на кількість порожніх рядків, як правило, вводиться обмеження – враховується їх кількість, що не перевищує

25% загальної кількості рядків у вимірюваному блоці коду). Літера "D" (Delivered) означає "поставлений" код, тобто тільки той, що увійшов до закінченого програмного продукту. Дуже часто виникає необхідність враховувати не тільки поставлений код, але і допоміжний або проміжний, який не входить до закінченого продукту, але що потрібен для реалізації проекту і вимагає певних витрат праці;

– кількість логічних SLOC (використовувані абревіатури: LSI, DSI, KDSI, де SI – *Source Instructions*) – визначається як кількість команд і залежить від використовуваної мови програмування. В тому випадку, якщо мова не допускає розміщення на одному рядку кількох команд, кількість логічних SLOC буде відповідати числу "фізичних за винятком порожніх рядків та рядків коментарів. Якщо ж мова програмування підтримує розміщення кількох команд на одному рядку, то один «фізичний» рядок повинний бути врахований як декілька «логічних», якщо він містить більше однієї команди мови, що використовується.

Нерідко показник SLOC використовується на ранніх стадіях роботи над проектом з метою отримання попередніх оцінок його загального обсягу. І хоча завдання це досить складне, а про отримання точних оцінок не може бути й мови, для його вирішення є спеціальні підходи, які забезпечують прийнятний результат:

– оцінка показника SLOC за аналогією. Дана методика дозволяє отримувати оцінку показника для всього проекту, що розробляється, або окремих його складових за допомогою порівняння функціональних властивостей з існуючими аналогами. При оцінці показника за даною методикою підбирається близький за функціональністю замітник з відомим значенням SLOC, далі на його підставі визначається прогнозна оцінка з урахуванням відмінностей у функціональності, у мовах програмування, можливості оптимізації на основі коду аналога й інш.;

– оцінка показника SLOC «знизу-вгору» експертним методом. Цією методикою передбачено розподіл проекту на ієрархічну структуру завдань (*Work Breakdown Structure, WBS*), на підставі яких виконується експертне оцінювання показника, та підсумовується в результаті для всього проекту. Як правило, експертами надаються три оцінки (найбільш вірогідна, максимальна і мінімальна), а далі вони усереднюються за допомогою бета-розподілу (розраховується сума мінімальної, максимальної і чотириразової найбільш імовірної оцінки, яка потім ділиться на шість).

Для метрики SLOC є багато похідних метрик, що дозволяють оцінити окремі показники проекту. Основними серед яких є:

- кількість пустих рядків (*Blank Lines Of Code, BLOC*);
- кількість рядків, що містять коментарі (*Comment Lines Of Code, CLOC*);
- кількість рядків, що містять вихідний код і коментарі (*Lines with Both Code and Comments, C&SLOC*);
- кількість рядків, що містять декларативний вихідний код;
- кількість рядків, що містять імперативний вихідний код;
- відсоток коментарів (кількість рядків коментарів, помножене на 100 і поділена на кількість рядків коду);
- середня кількість рядків для функцій (методів);
- середня кількість рядків, що містять вихідний код для функцій (методів);
- середня кількість рядків для модулів;
- середня кількість рядків для класів.

Не зважаючи на всі недоліки цієї метрики, вона досі є ефективним інструментом вимірювання якості та є складовою багатьох інших метрик.

Стилістика програми. Іноді важливо не просто підрахувати кількість рядків коментарів, а визначити їх щільність, тобто визначити якість документованості коду. Для оцінки **стилістики програми** (F), використовують вираз

$$F = \sum_{i=1}^n F_i, \quad (1.1)$$

де $F_i = \text{SIGN}(\sqrt{N_{\text{комм},i}} / N_i - 0,1)$ – стилістика i -го фрагмента програми, де $N_{\text{комм},i}$ – кількість коментарів у i -м фрагменті, N_i – загальна кількість рядків коду в i -му фрагменті.

Метрики Холстеда. Моріс Холстед (Maurice H. Halstead) у 1977 запропонував групу метрик, побудованих на аналізі кількості рядків і синтаксичних елементів вихідного коду програми.

Оцінка складності програми (Halstead Difficulty, HDiff) оцінюється як:

$$Hdiff = (NUoprtr / 2) \times (NOprnd / NUoprnd), \quad (1.2)$$

де $NUOprtr$ – словник операторів (кількість унікальних операторів програми, також символи-роздільники, імена процедур і знаки операцій);

$NUOprnd$ – словник операндів (кількість унікальних операндів програми);

$Noprtr$ – загальна кількість операторів в програмі;

$Noprnd$ – загальна кількість операндів в програмі.

На основі (2.2) пропонується оцінювати зусилля програміста при розробці за допомогою показника $HEff$ (Halstead Effort):

$$HEff = HDiff \times HPVVol, \quad (1.3)$$

де $HPVoc = NUOprtr + NUOprnd$ – словник програми (Halstead Program Vocabulary, $HPVoc$).

Крім того, для оцінювання програми можливо використовувати:
– довжина програми (Halstead Program Length, $HPLen$):

$$HPLen = Noprtr + Noprnd;$$

– обсяг програми (Halstead Program Volume, $HPVVol$):

$$HPVVol = HPLen \times \log_2 HPVoc.$$

Незважаючи на теоретичну можливість отримання попередніх оцінок трудомісткості проекту з використанням показника $HEff$, метрики Холстеда, як правило, обчислюються вже на основі готового вихідного коду і застосовуються для отримання апостеріорних оцінок. Вони можуть бути використані для того, щоб запобігти надмірному ускладненню окремих структурних елементів програмного проекту і зростання пов'язаних із цим ризиків.

Метрики Джилба. Ці метрики показують складність програмного забезпечення на основі насиченості програми умовними операторами або операторами циклу. Дана метрика, не дивлячись на свою простоту, досить добре відображає складність написання і розуміння програми, а при додаванні такого показника, як максимальний рівень вкладеності умовних і циклічних операторів, ефективність цієї метрики значно зростає.

Абсолютна складність за Джилбом:

$$Cl = \sum N_{ifoprtr}, \quad (1.4)$$

де $N_{ifoprtr}$ – кількість умовних операторів.

Відносна складність за Джилбом:

$$Cl^* = \frac{CL}{N_{oprtr}}, \quad (1.5)$$

де N_{oprtr} – загальна кількість операторів в програмі.

1.2.2 Метрики складності потоку керування програми

Наступний великий клас метрик, заснований вже не на кількісних показниках, а на аналізі керуючого графа програми, називається метрики складності потоку керування програм.

Перед тим як безпосередньо описувати метрики, для кращого розуміння буде наведено визначення керуючого графу програми і спосіб його побудови.

Нехай представлена певна програма. Для даної програми будується орієнтований граф, що містить лише один вхід і один вихід, при цьому вершини графа співвідносять з тими ділянками коду програми, в яких є лише послідовні обчислення і відсутні оператори розгалуження і циклу, а дуги співвідносять з переходами від блоку до блоку і гілками виконання програми. Умова при побудові даного графа: кожна вершина досяжна з початкової, і кінцева вершина досяжна з будь-якої іншої вершини.

Цикломатична складність програми. Найпоширенішою оцінкою, заснованою на аналізі отриманого графа, є цикломатичне число Мак-Кейба, що визначається як:

$$V(G) = e - n + 2p, \quad (1.6)$$

де e – кількість дуг, n – кількість вершин, p – кількість компонентів зв'язності. Кількість компонентів зв'язності графа можна розглядати як кількість дуг, що необхідно додати для перетворення графа в сильно зв'язний. Сильно зв'язним називається граф, будь-які

дві вершини якого взаємно досяжні. Для графів коректних програм, тобто графів, що не мають недосяжних від точки входу ділянок і "висячих" точок входу і виходу, сильно зв'язний граф, як правило, виходить шляхом замикання дугою вершини, що означає кінець програми, на вершину, що позначає точку входу в цю програму. По суті $V(G)$ визначає кількість лінійно незалежних контурів в сильно доладному графі. Так що в коректно написаних програмах $p=1$, і тому формула (2.6) для розрахунку цикломатичної складності набуває вигляду:

$$V(G) = e - n + 2. \quad (1.7)$$

Як правило, при обчисленні цикломатичної складності логічні оператори не приймаються до уваги, допускається також спрощений підхід, згідно з яким власне побудова графа не проводиться, а показник визначається на підставі підрахунку кількості операторів керуючої логіки (*if*, *switch* і т. д.) і можливої кількості шляхів виконання програми. Метрика цикломатичної складності може бути розрахована для модуля, методу та інших структурних одиниць програми.

В процесі аналізу значень показника для окремих структурних елементів можна виявити елементи з високим значенням показника (наприклад, нормальне значення показника для методу – не вище 5-7), що свідчить про складність їх керуючої логіки і, відповідно, високих трудовитратах на розробку, тестування і супровід.

Обчислення метрики в ході реалізації проекту (а при детальному проектуванні воно можливо ще на цьому етапі, не чекаючи стадії кодування) дозволяє вчасно визначити найбільш складні, що супроводжуються високими ризиками, структурні одиниці та вжити заходів щодо усунення ризиків за рахунок внесення корективів.

Для метрики цикломатичної складності існує безліч варіацій, зокрема:

- "модифікована" цикломатична складність – розглядає не кожне розгалуження оператора множинного вибору (*switch*, *case*), а весь оператор як єдине ціле;

- "сувора" цикломатична складність – містить логічні оператори;

- "спрощена" цикломатична складність – передбачає обчислення на основі не графа, а підрахунку керуючих операторів;

– "актуальна" складність – визначається як кількість незалежних шляхів, що проходить програма при тестуванні;

– метрика складності глобальних даних – обчислюється як цикломатична складність модуля та збільшується на кількість взаємозв'язків з глобальними даними.

В цілому, метрики цикломатичної складності є досить гарним показником, що дозволяє своєчасно припинити подальше ускладнення окремих складових проекту і спростити їх, попередивши ймовірні проблеми з заплутаним і нестабільним кодом в майбутньому.

Метрики похідні від цикломатичної складності. Метрика цикломатичної складності стала основою для створення похідних і якісно нових метрик, таких як:

– інтервальна метрика Дж. Майерса (G. Mayers) – розраховується як різниця значень цикломатичної складності й кількості окремих умов плюс одиниця;

– метрика У. Хансена (W. Hansen) – визначається парюю "циклوماتична складність" і "кількість операторів";

– метрика Пивоварського – обчислюється на основі модифікованої цикломатичної складності і дозволяє враховувати структурованість програми, виявляючи код з поганою структурою, і багатьох інших, нерідко мають швидше за наукову, ніж практичну цінність:

$$N(G_i) = V^*(G_i) + \sum P_i, \quad (1.8)$$

де $V^*(G)$ – модифікована цикломатична складність, обчислена так як і $V(G)$, але з однією відмінністю: оператор CASE з n виходами розглядається як один логічний оператор, а не як $n - 1$ операторів;

P_i – глибина вкладеності i -ї предикатної вершини. Для підрахунку глибини вкладеності предикатних вершин використовується число "сфер впливу". Під глибиною вкладеності розуміється кількість всіх "сфер впливу" предикатів, які повністю утримуються у сфері даної вершини, або перетинаються з нею. Глибина вкладеності збільшується за рахунок вкладеності не самих предикатів, а "сфер впливу". Міра Пивоварського зростає при переході від послідовних програм до вкладених і далі до неструктурованих, що є її величезною перевагою перед багатьма іншими заходами даної групи.

Топологічна міра Чена. Ця метрика висловлює складність програми через кількість перетинів кордонів між областями, утвореними графом програми:

$$M(G_i) = (n(G_i), N, Q_0), \quad (1.9)$$

Цей підхід застосовується лише до структурованих програм, що допускає лише послідовне з'єднання керуючих конструкцій. Для неструктурованих програм міра Чена істотно залежить від умовних і безумовних переходів. У цьому випадку можна вказати верхню і нижню границі. Верхня – є $m+1$, де m – кількість логічних операторів при їх взаємній вкладеності. Нижня – дорівнює 2. Коли керуючий граф програми має тільки одну компоненту зв'язності, захід Чена збігається з цикломатичною мірою Мак-Кейба.

Метрики Харрісона і Мейджела. Ці метрики враховують рівень вкладеності і довжини програми.

Кожній вершині присвоюється своя складність у відповідності з оператором, що вона зображує. Ця початкова складність вершини може обчислюватися будь-яким способом, включаючи використання заходів Холстеда. Виділимо для кожної предикатної вершини підграф, породжений вершинами, які є кінцями вихідних з неї дуг, а також вершинами, досяжними з кожної такої вершини (нижня межа підграфа), і вершинами, що лежать на шляхах з предикатної вершини в яку-небудь нижню межу. Цей підграф називається сферою впливу предикатної вершини.

Приведеною складністю предикатної вершини називається сума початкових або наведених складнощів вершин, що входять в її сферу впливу, плюс первинна складність самої предикатної вершини.

Функціональна міра (*SCOPE*) програми:

$$f_i = \sum \chi_i, \quad (1.10)$$

де χ_i – це складність вершини керуючого графа.

Функціональним ставленням (*SCORT*) називається відношення числа вершин в керуючому графі до його функціональної складності, причому з числа вершин виключаються термінальні:

$$f_i^* = \frac{N\chi_i}{f_i}, \quad (1.11)$$

де N – це кількість вершин керуючого графа.

SCORT може приймати різні значення для графів з однаковим цикломатичним числом.

Метод граничних значень так само заснований на аналізі керуючого графа програми. Для визначення даного методу необхідно ввести кілька додаткових понять.

Нехай $G = (V, E)$ – керуючий граф програми з єдиною початковою і єдиною кінцевою вершинами.

Кількість вершин, що входять до дуг, називається негативним ступенем вершини, а кількість вихідних з вершини дуг – позитивним ступенем вершини. Тоді набір вершин графа можна розбити на дві групи: вершини, у яких позитивна ступінь ≤ 1 ; вершини, у яких позитивна ступінь ≥ 2 .

Вершини першої групи назвемо приймаючими вершинами, а вершини другої групи – вершинами відбору.

Кожна приймаюча вершина має наведену складність, рівну 1, крім кінцевої вершини, наведена складність якої дорівнює 0. Наведені складності всіх вершин графа G підсумовуються, утворюючи абсолютну граничну складність програми. Після цього визначається відносна гранична складність програми:

$$S_g = 1 - (v - 1) / S_a, \quad (1.12)$$

де S_g – відносна гранична складність програми, S_a – абсолютна гранична складність програми, v – загальна кількість вершин графа програми.

Існує **метрика Шнейдевінда**, що виражається через кількість можливих шляхів в керуючому графі.

Міра Вудворда. В основі цієї метрики кількість перетинів дуг керуючого графа. Так як в добре структурованій програмі таких ситуацій виникати не повинно, то дана метрика застосовується в основному в слабо структурованих мовах (Асемблер, Фортран). Точка перетину виникає при виході керування за межі двох вершин, які є послідовними операторами.

1.2.3 Метрики складності потоку керування даними

Наступний клас метрик – метрики складності потоку керування даних.

Метрика Чепина. Суть методу полягає в оцінці інформаційної міцності окремо взятого програмного модуля з допомогою аналізу характеру використання змінних зі списку введення-виведення.

Вся безліч змінних, що складають перелік введення-виведення, розбивається на 4 функціональні групи :

- змінні, що використовуються для розрахунків і для забезпечення виведення (P);
- модифіковані, або створювані всередині програми змінні (M);
- «керуючі» змінні – змінні, які беруть участь в керуванні роботою програмного модуля (C),
- "паразитні" змінні – змінні не використовуються в програмі, (T).

Оскільки кожна змінна може виконувати одночасно кілька функцій, необхідно враховувати її в кожній відповідній функціональній групі.

Отже метрика Чепина має вигляд:

$$Q = a_1 * P + a_2 * M + a_3 * C + a_4 * T, \quad (1.13)$$

де a_1, a_2, a_3, a_4 – вагові коефіцієнти, що відображають різний вплив на складність програми кожної функціональної групи. На думку автора метрики, найбільшу вагу, рівну 3, має функціональна група C , так як вона впливає на потік керування програми. Вагові коефіцієнти інших груп розподіляються наступним чином: $a_1=1, a_2=2, a_4=0.5$. Ваговий коефіцієнт групи T не дорівнює 0, оскільки "паразитні" змінні не збільшують складність потоку даних програми, але іноді ускладнюють її розуміння. З урахуванням вагових коефіцієнтів:

$$Q = P + 2M + 3C + 0.5T. \quad (1.14)$$

Метрика Спена. Ця метрика ґрунтується на локалізації звернень до даних усередині кожної програмної секції. **Спен** – це кількість тверджень, які містять цей ідентифікатор, між першою і останньою появою в тексті програми. Отже, ідентифікатор, що з'явився n раз, має спен, рівний $n-1$. При великому спені ускладнюється тестування та налагодження.

Метрика "модуль – глобальна змінна". Ця метрика, що зв'язує складність програм із зверненнями до глобальних змінних. Пара "модуль-глобальна змінна" позначається як (p, r) , де p – модуль, що має доступ до глобальної змінної r . У залежності від наявності у програмі реального звернення до змінної r формуються два типи пар "модуль – глобальна змінна": фактичні та можливі. Можливе звернення до r за допомогою p показує, що область існування r містить у собі p .

Дана характеристика позначається A_{up} і говорить про те, скільки разів модулі up дійсно отримували доступ до глобальних змінних, а число P_{up} – скільки разів вони могли б отримати доступ.

Відношення кількості фактичних звернень до можливих визначається як:

$$R_{up} = A_{up}/P_{up}. \quad (1.15)$$

Ця формула показує наближену ймовірність посилання довільного модуля на довільну глобальну змінну. Очевидно, що чим вище ця ймовірність, тим вище ймовірність "несанкціонованої" зміни будь-якої змінної, що може істотно ускладнити роботу, пов'язану з модифікацією програми.

Міра Кафура. Вона створена на основі концепції інформаційних потоків. Для використання цієї міри вводяться поняття локального та глобального потоку.

Локальний потік інформації з модуля A до модуля B існує, якщо:

- модуль A викликає модуль B (прямий локальний потік);
- модуль B викликає модуль A і модуль A повертає значення, що використовується у модулі B (непрямий локальний потік)
- модуль C викликає модулі A і B та передає результат виконання модуля A до модуля B .

Глобальний потік інформації з модуля A через глобальну структуру даних D існує, якщо модуль A розміщує інформацію в D , а модуль B використовує інформацію з D .

На основі цих понять вводиться величина I – **інформаційна складність процедури**:

$$I = \text{length}^*(fan_in * fan_out)^2, \quad (1.16)$$

де $length$ – складність тексту процедури (вимірюються через метрику обсягу, типу метрик Холстеда (2.2-2.3), Маккейба (2.6), LOC і т.п.); fan_in – кількість локальних потоків входять всередину процедури плюс число структур даних, з яких процедура бере інформацію fan_out – кількість локальних потоків вихідних з процедури плюс число структур даних, які оновлюються процедурою.

Можна визначити інформаційну складність модуля як суму інформаційних складнощів процедур, що входять до нього.

Далі розглянемо інформаційну складність модуля щодо певної структури даних. Інформаційна міра складності модуля щодо структури даних:

$$J = N_W * N_R + N_W * N_{RW} + N_{RW} * N_R + N_{RW} * (N_{RW} - 1), \quad (1.17)$$

де N_W – кількість процедур, що тільки оновлюють структуру даних; N_R – кількість процедур, що тільки читають інформацію з структури даних; N_{RW} – кількість процедур, що і читають, і оновлюють інформацію в структурі даних.

Міра Овієдо. Суть її полягає в тому, що програма розбивається на лінійні непересічні ділянки – промені операторів, які утворюють керуючий граф програми.

Автор метрики виходить з таких припущень: програміст може знайти відношення між визначеними та використаними входженнями змінної всередині променя більш легко, ніж між променями; число різних визначальних записів у кожному промені більш важливо, ніж загальне число використовуваних входжень змінних в кожному промені.

Позначимо через $R(i)$ безліч визначальних входжень змінних, які розташовані в радіусі дії променя i (визначальне входження змінної знаходиться в радіусі дії променя, якщо змінна або локальна в ньому і має визначальне входження, або для неї є визначальне входження в деякому попередньому промені, і немає локального визначення по шляху). Позначимо через $V(i)$ множину змінних, які використовують входження і вже є в промені i . Тоді міра складності i -го променя визначається як:

$$DF(i) = \sum (DEF(v_j), \quad (1.18)$$

де $DEF(v_j)$ – визначає кількість входжень змінної v_j з безлічі $R(i)$, $j = i...//V(i)//$, а $//V(i)//$ – потужність безлічі $V(i)$.

1.2.4 Метрики складності потоку керування і даних програми

Четвертим класом метрик є метрики, близькі як до класу кількісних метрик, класу метрик складності потоку керування програми, так і до класу метрик складності потоку керування даними (строго кажучи, цей клас метрик і клас метрик складності потоку керування програми є одним і тим же класом – топологічними метриками, але має сенс розділити їх у даному контексті для більшої ясності). Даний клас метрик встановлює складність структури програми як на основі кількісних підрахунків, так і на основі аналізу структур керування.

Тестуюча М-Міра. Тестуючою мірою M називається міра складності, що задовольняє наступним умовам:

1. Міра складності простого оператора дорівнює 1.
2. $M(\{F_1; F_2; \dots; F_N\}) = e_i^n M(F_p)$.
3. $M(\text{if } p \text{ then } f1 \text{ else } f2) = 2 \max(M(F1), M(F2))$.
4. $M(\text{while } p \text{ do } F) = 2M(F)$.

Міра зростає з глибиною вкладеності й враховує протяжність програми. До тестуючої міри близько примикає міра на основі регулярних вкладень. Ідея цієї міри складності програм полягає в підрахунку сумарної кількості символів (операндів, операторів, дужок) у регулярному виразі з мінімально необхідною кількістю дужок, що описує керуючий граф програми. Всі міри цієї групи чутливі до вкладеності керуючих конструкцій і до протяжності програми. Проте зростає рівень трудомісткості обчислень.

Зв'язаність модулів програми. Якщо модулі сильно пов'язані, то програма стає важкомодифікуємою і важкою в розумінні. Дана міра не виражається кількісно.

Види зв'язаності модулів:

- зв'язаність за даними – якщо модулі взаємодіють через передавання параметрів і при цьому кожен параметр є елементарним інформаційним об'єктом. Це найбільш бажаний тип зв'язаності (зчеплення);

- зв'язаність за структурою даних – якщо один модуль посилає іншому складовою інформаційний об'єкт (структуру) для обміну даними;

- зв'язаність за керуванням – якщо один модуль посилає іншому інформаційний об'єкт – флаг, призначений для керування його внутрішньою логікою;

- зв'язаність за загальною областю, якщо модулі посилаються на одну і ту ж область глобальних даних. Зв'язаність (зчеплення) за загальною областю є небажаним, оскільки, по-перше, помилка в модулі, що використовує глобальну область, може несподівано проявитися в будь-якому іншому модулі; по-друге, такі програми важкі для розуміння, так як програмісту важко визначити, які саме дані використовуються конкретним модулем;

- зв'язаність за вмістом – якщо один з модулів посилається всередину іншого. Це неправильний тип зчеплення, оскільки повністю суперечить принципу модульності, тобто подання модуля у вигляді чорного ящика. Зовнішня зв'язаність – два модуля використовують зовнішні дані, наприклад комунікаційний протокол.

- зв'язаність за допомогою повідомлень – найбільш вільний вигляд зв'язаності, модулі безпосередньо не пов'язані один з одним, і взаємодіють через повідомлення, що не мають параметрів.

- підкласова зв'язаність – відношення між класом-батьком і класом-нащадком, причому нащадок пов'язаний з батьком, а батько з нащадком – ні.

- зв'язаність за часом – два дії згруповані в одному модулі лише тому, що через обставини вони відбуваються в один час.

Відсутність зв'язаності – модулі не взаємодіють між собою.

Міра Колофелло. Ще одна міра, що стосується стабільності модуля $h(G)$. вона може бути визначена як кількість змін, які потрібно зробити в модулях, відмінних від модуля, стабільність якого перевіряється, при цьому ці зміни повинні стосуватися модуля, що перевіряється.

Метрика Мак-Клур. Виділяються три етапи обчислення цієї метрики:

1. Для кожної змінної обчислюється значення її функції складності $C(i)$ за формулою:

$$C(i) = (D(i) * J(i)) / n, \quad (1.19)$$

де $D(i)$ – величина, що стосується сфери дії змінної i . $J(i)$ – міра складності взаємодії модулів через змінну i , n – кількість окремих модулів у схемі розподілу.

2. Для всіх модулів, що входять у сферу розбиття, визначається значення їх функцій складності $M(P)$ за формулою:

$$M(P) = f_p * X(P) + g_p * Y(P), \quad (1.20)$$

де f_p і g_p – відповідно, кількість модулів, безпосередньо передують і безпосередньо наступних за модулем P , $X(P)$ – складність звернення до модулю P , $Y(P)$ – складність керування викликом з модулю P інших модулів.

3. Загальна складність MP ієрархічної схеми розподілу програми на модулі задається формулою:

$$MP = \sum M(P) \quad (1.21)$$

Для всіх можливих значень P – модулів програми. Дана метрика орієнтована на добре структуровані програми, складені з ієрархічних модулів, які задають функціональну специфікацію і структуру керування. Також мається на увазі, що в кожному модулі одна точка входу і одна точка виходу, модуль виконує рівно одну функцію, а виклик модулів здійснюється у відповідності з ієрархічною системою керування, яка задає ставлення виклику на безлічі модулів програми.

Міра Берлінгера. Метрика, заснована на інформаційній концепції. Міра складності обчислюється як:

$$M = \sum f_i * \log_2 p_i, \quad (1.22)$$

де f_i – частота появи i -го символу, p_i – імовірність його появи.

Недоліком цієї метрики є те, що програма, що містить багато унікальних символів, але в малій кількості, буде мати таку ж складність як програма, що містить невелику кількість унікальних символів, але у великій кількості.

Для створення системи автоматичного мережевого розподілення інформації було обрано платформу QT (Quasar

Technologies) – крос-платформний інструментарій розробки ПЗ мовою програмування C++, бібліотека класів. Говорячи більш конкретно, на чільне місце поставлено дві речі: перше – бібліотеки створені для мови C++; друге – основою є інтерфейс користувача.

1.2.5 Бібліотека QT

Бібліотека QT замислювалася і починалася як крос-платформний інструментарій (toolkit) для швидкої розробки графічних інтерфейсів (GUI) програм на мові C++, з метою спростити життя програмістів, які пишуть на C++ крос-платформні GUI-програми, що переносяться та повинні працювати і в середовищі Windows, і в середовищі Unix / Linux, і на комп'ютерах Macintosh.

В даний час QT значно переросла рамки інструментарію для розробки графічних інтерфейсів програм. Вона надає програмісту цілісний каркас (framework), що дозволяє при написанні більшої частини програми використовувати тільки класи QT і практично повністю відмовитися від написання системно-залежного коду, використання системних викликів (будь то Win32 API виклики або Unix система) або від винаходу власних крос-платформних обгортки. Класи QT покривають майже всі потреби програміста. QT дозволяє запускати написане за його допомогою ПЗ в більшості сучасних операційних систем шляхом простої компіляції програми для кожної ОС без зміни початкового коду. Включає в себе всі основні класи, які можуть знадобитися при розробці прикладного програмного забезпечення, починаючи від елементів графічного інтерфейсу і закінчуючи класами для роботи з рядками, для роботи з файлами, мережею, базами даних, XML, для забезпечення багатопоточності в програмі, і багато-багато іншого. QT є повністю об'єктно-орієнтованим, легко розширюваним і підтримує техніку компонентного програмування. За своїми можливостями і багатством бібліотека QT порівнянна з NET Framework або з системою класів Java 2 EE.

На сьогоднішній день QT задовольняє всім сучасним вимогам в створенні програм, як невеликих додатків, так і вузькоспеціалізованих програмних комплексів.

До переваг QT відносять:

1. Платформно-незалежність. QT дозволяє запускати написане за його допомогою ПЗ в більшості сучасних операційних систем шляхом простої компіляції програми для кожної ОС без зміни початкового

коду і при роботі ми завжди отримаємо вірний результат і в середовищі Windows, і в середовищі Unix / Linux, і на комп'ютерах Macintosh;

2. Легкість в написанні та розумінні програми. QT надає цілісний каркас (framework), що дозволяє при написанні програми використовувати тільки класи QT і практично повністю відмовитися від написання системно-залежного коду, використання системних викликів та ін. Включає в себе всі основні класи, які можуть знадобитися при розробці прикладного програмного забезпечення, починаючи від елементів графічного інтерфейсу і закінчуючи класами для роботи з рядками, файлами, мережею, базами даних, XML, для забезпечення багатопоточності в програмі і багато іншого;

3. Зручний інтерфейс та легкість в використанні. Однією з основ бібліотек QT є інтерфейс користувача. Бібліотека QT замислювалася як крос-платформний інструментарій (toolkit) для швидкої розробки графічних інтерфейсів (GUI) програм на мові C++.

1.3 Завдання на лабораторну роботу

1.3.1 Ознайомитися з теоретичними відомостями щодо метрик програмних систем.

1.3.2 Реалізувати на мові C++ з використанням крос-платформного інструментарію Qt програму розрахунку метрик: кількість рядків коду; кількість пустих рядків; кількість рядків, що містять коментарі; кількість рядків, що містять вихідний код і коментарі; відсоток коментарів; словник операторів (кількість унікальних операторів програми, також символи-роздільники, імена процедур і знаки операцій); словник операндів (кількість унікальних операндів програми); загальна кількість операторів в програмі; загальна кількість операндів в програмі; довжина програми; обсяг програми; цикломатична складність; максимальний рівень вкладеності і довжина програми.

1.3.3 Виконати аналіз отриманих результатів.

1.3.4 Оформити звіт та відповісти на контрольні питання.

1.4 Зміст звіту

1.4.1 Тема та мета роботи.

1.4.2 Код реалізованої на мові C++ програми підрахунку метрик коду.

1.4.3 Аналіз отриманих результатів тестування.

1.4.5. Висновки, що містять відповіді на контрольні запитання, а також відображують результати виконання роботи та їх критичний аналіз.

1.5 Контрольні запитання

1.5.1 Чим відрізняється метрика та міра якості?

1.5.2 Якими характеристиками повинні володіти якісні метрики?

1.5.3 Які шкали використовуються для вимірювання значень метрик?

1.5.4 Наведіть класифікацію мір якості.

1.5.5 Наведіть класифікацію метрик якості.

1.5.6 Методологія створення метрик якості.

1.5.7 Метрики програмних проєктів. Метрики розміру.

1.5.8 Метрики програмних проєктів. Метрики складності.

1.5.9 На чому ґрунтується розробка внутрішніх та зовнішніх метрик якості?

1.5.10 Особливості розроблення програмних додатків за допомогою бібліотеки QT.

1.5.11 Переваги використання бібліотеки QT.

ЛАБОРАТОРНА РОБОТА №2 ОБЧИСЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ МЕТРИК

2.1 Мета роботи

Вивчити основні об'єктно-орієнтовані метрики програмних систем та реалізувати їх із застосуванням мови C++ та Qt.

2.2 Основні теоретичні відомості

У зв'язку з розвитком об'єктно-орієнтованих мов програмування з'явився новий клас метрик, також названий об'єктно-орієнтованими (ОО) метриками. У даній групі найбільш часто використовуваними є набори метрик Мартіна і набір метрик Чидамбера і Кемерера.

Метрики Мартіна. Для використання цих метрик необхідно ввести поняття категорії класів. У реальності клас може досить рідко бути використаний повторно ізольовано від інших класів. Практично кожен клас має групу класів, з якими він працює в кооперації, і від яких він не може бути легко відокремлений. Для повторного використання таких класів необхідно повторно використовувати всю групу класів. Така група класів сильно пов'язана і називається **категорією класів**. Для існування категорії класів існують наступні умови:

- класи в межах категорії класу закриті від будь-яких спроб змін всі разом. Це означає, що, якщо один клас повинен змінитися, всі класи в цій категорії з великою ймовірністю зміняться. Якщо будь-який з класів відкритий для деякого різновиду змін, вони всі відкриті для такого різновиду змін;

- класи у категорії повторно використовуються тільки разом. Вони настільки взаємозалежні і не можуть бути відокремлені один від одного. Таким чином, якщо робиться будь-яка спроба повторного використання одного класу в категорії, всі інші класи повинні повторно використовуватися з ним;

- класи у категорії поділяють деяку загальну функцію або досягають деякої спільної мети. Відповідальність, незалежність і стабільність категорії можуть бути виміряні шляхом підрахунку залежностей, які взаємодіють з цією категорією.

Для категорій класів можуть бути визначені три метрики :

1. Доцентрове зчеплення (Ca) – кількість класів поза певної категорії, що залежать від класів всередині цієї категорії.
2. Відцентрове зчеплення (Ce) – кількість класів всередині певної категорії, що залежать від класів поза цієї категорії.
3. Нестабільність:

$$St = Ce / (Ca + Ce). \quad (2.1)$$

Ця метрика має діапазон значень $[0,1]$. Якщо $St = 0$, це вказує максимально стабільну категорію, а якщо $St = 1$, це вказує максимально нестабільну категорію.

Абстрактність категорії визначається наступним чином:

$$A = N_A / N_{All},$$

де N_A – кількість абстрактних класів в категорії, N_{All} – спільна кількість класів в категорії.

Значення цієї метрики змінюються в діапазоні $[0,1]$, якщо $A=0$, то категорія цілком конкретна, а якщо $A=1$ – категорія повністю абстрактна.

Тепер на основі наведених метрик Мартіна можна побудувати графік, на якому відображено залежність між абстрактністю і нестабільністю. Якщо на ньому побудувати пряму, що задається формулою $A+St=1$, то на цій прямій будуть лежати категорії, що мають кращу збалансованість між абстрактністю і нестабільністю. Ця пряма називається головною послідовністю.

Далі можна ввести ще 2 метрики:

Відстань до головної послідовності:

$$D = (A + St - 1) / \sqrt{2}$$

Нормалізованої відстань до головної послідовності:

$$Dn = A + St - 2$$

Практично для будь-яких категорій вірно те, що чим ближче вони знаходяться до головної послідовності, тим краще.

Загальні об'єктно-орієнтовані метрики. До найбільш поширених *ОО* метрик відносяться:

– кількість викликів віддалених методів (*Number Of Remote Methods, NORM*). При її обчисленні проглядаються всі конструктори і методи класу і підраховується кількість викликів віддалених методів (не визначених у класі та його батьків);

– відгук на клас (*Response For Class, RFC*) – кількість методів, які можуть викликатися примірниками класу, обчислюється як сума локальних і віддалених методів;

– зважена насиченість класу 1 (*Weighted Methods Per Class 1, WMPC1*) – відображає відносну міру складності класу на основі цикломатичної складності кожного його методу. Клас з більш складними методами і великою їх кількістю вважається більш складним (при обчисленні метрики батьківські класи не враховуються);

– зважена насиченість класу 2 (*WMPC2*) – міра складності класу, заснована на тому, що клас з більшою кількістю методів і метод з більшою кількістю параметрів є більш складними (при обчисленні метрики батьківські класи не враховуються);

– недолік зв'язності методів 1 (*Lack Of Зчеплення Methods Of 1, LOCOM1*) – відображає міру взаємозалежності методів (зв'язність характеризує ступінь взаємодії між елементами окремого модуля і його насиченість). Чим вище значення даної метрики, тим більше взаємозалежними і, відповідно, складними вважаються методи;

– недолік зв'язності методів 2 (*LOCOM2*) – відображає міру взаємозалежності класів. Обчислюється як процентне відношення методів, що не мають доступу до специфічних атрибутів класу, до всіх атрибутів даного класу. Клас, що може викликати значно більше методів, ніж рівні йому за рівнем, є більш складним;

– недолік зв'язності методів 3 (*LOCOM3*) – вимірює ступінь розходження методів у класі за атрибутами. Низьке значення свідчить про хорошу декомпозицію в класі, що виражається в його простоті, зрозумілості та готовності до повторного використання. Високе значення – нестачу зв'язності, що збільшує складність і підвищує ймовірність помилок в процесі розроблення.

Метрики Чидамбера і Кемерера. Ці метрики засновані на аналізі методів класу, дерева успадкування і т.д.

Метрика *WMC (Weighted methods per class)*, сумарна складність всіх методів класу:

$$WMC = \sum_{i=1}^n c_i,$$

де c_i – складність i -го методу, обчислена за будь-якою з метрик (Холстеда і т.д. в залежності від потрібного критерію), якщо у всіх методів складність однакова, то WMC дорівнює n .

Метрика *DIT (Depth of Inheritance tree)* – глибина дерева спадкування (найбільший шлях по ієрархії класів до цього класу від класу-предка), чим більше, тим краще, так як при більшій глибині збільшується абстракція даних, зменшується насиченість класу методами, однак при досить великій глибині сильно зростає складність розуміння і написання програми.

Метрика *NOC (Number of children)* – кількість нащадків (безпосередніх), чим більше, тим вище абстракція даних.

Метрика *CBO (Coupling between object classes)* – зчеплення між класами, показує кількість класів, з якими пов'язаний вихідний клас. Для даної метрики справедливі всі твердження, введені раніше для зв'язаності модулів, тобто при високому *CBO* зменшується абстракція даних і погіршується повторне використання класу.

Метрика *RFC (Response for a class)* – $RFC = /RS/$, де RS – відповідна множина класу, тобто безліч методів, які можуть бути потенційно викликані методом класу у відповідь на дані, отримані об'єктом класу. Тобто $RS = (\{M\}(\{R_i\}), i=1...n$, де M – всі можливі методи класу, R_i – всі можливі методи, які можуть бути викликані i -м класом. Тоді *RFC* буде потужністю цієї множини. Чим більше *RFC*, тим складніше тестування та налагодження.

Метрика *LCOM (Lack of cohesion in Methods)* – недолік зчеплення методів. Для визначення цього параметра розглянемо клас C з n методами $M_1, M_2, \dots, M_n$, тоді $\{I_1\}, \{I_2\}, \dots, \{I_n\}$ – безлічі змінних, що використовуються в цих методах. Тепер визначимо P – безліч пар методів, що не мають спільних змінних; Q – безліч пар методів, що мають загальні змінні.

Тоді

$$LCOM = /P/ - /Q/.$$

Кількість пар методів розраховується як

$$C_m^2 = \frac{m!}{2(m-2)!},$$

де m – кількість методів класу.

Недолік зчеплення може бути сигналом того, що клас можна розбити на кілька інших класів або підкласів, так що для підвищення інкапсуляції даних і зменшення складності класів і методів краще підвищувати зчеплення.

2.3 Завдання на лабораторну роботу

2.3.1 Ознайомитися з теоретичними відомостями щодо об'єктно-орієнтованих метрик програмних систем.

2.3.2 Реалізувати на мові C++ програму розрахунку об'єктно-орієнтованих метрик.

2.3.3 Виконати аналіз отриманих результатів.

2.3.4 Оформити звіт та відповісти на контрольні питання.

2.4 Зміст звіту

2.4.1 Тема та мета роботи.

2.4.2 Код реалізованої на мові C++ програми підрахунку об'єктно-орієнтованих метрик: середня кількість рядків для функцій (методів); середня кількість рядків, що містять вихідний код для функцій (методів); середня кількість рядків для модулів; середня кількість рядків для класів; кількість викликів віддалених методів; відгук на клас; зважена насиченість класу; недолік зв'язності методів; метрики *WMC*, *DIT*, *NOC*, *RFC*, *LCOM*.

2.4.3 Аналіз отриманих результатів тестування.

2.4.5. Висновки, що містять відповіді на контрольні запитання, а також відображують результати виконання роботи та їх критичний аналіз.

2.5 Контрольні запитання

2.5.1 Об'єктно-орієнтовані метрики.

2.5.2 Для розрахунку яких метрик необхідно зконструювати керуючий граф програми?

2.5.3 Для яких метрик використовується поняття «категорії класів»?

2.5.4 Що полягає в основі метрик Чидамбера та Кемерера?

2.5.5 Що аналізують об'єктно-орієнтовані метрики?

ЛАБОРАТОРНА РОБОТА №3

ОСНОВИ GIT

3.1 Мета роботи

Вивчити основні команди по роботі з локальним репозиторієм GIT.

3.2 Основні теоретичні відомості

ОСНОВНІ ХАРАКТЕРИСТИКИ GIT

Переваги:

1. Надійна система порівняння ревізій та перевірки коректності даних, заснована на алгоритмі хеширування SHA1 (Secure Hash Algorithm 1).
2. Гнучка система розгалуження проектів і злиття гілок між собою.
3. Наявність локального репозиторію,
4. Висока продуктивність і швидкість роботи.
5. Зручний і інтуїтивно зрозумілий набір команд.
6. Безліч графічних оболонок, що дозволяють швидко і якісно вести роботи з Git'ом.
7. Можливість робити контрольні точки.
8. Широка розповсюдженість, легка доступність і якісна документація.
9. Гнучкість системи.
10. Універсальний мережевий доступ з використанням протоколів http, ftp, rsync, ssh та ін.

Недоліки

1. Unix - орієнтованість.
2. Можливі (але надзвичайно низькі) збіги хеш - коду відмінних за змістом ревізій.
3. Не відстежуються зміни окремих файлів
4. При початковому (першому) створенні сховища та синхронізації його з іншими розробниками, буде потрібно досить тривалий час для скачування даних, особливо, якщо проект великий, так як потрібно скопіювати на локальний комп'ютер весь репозиторій.

АРХІТЕКТУРА GIT

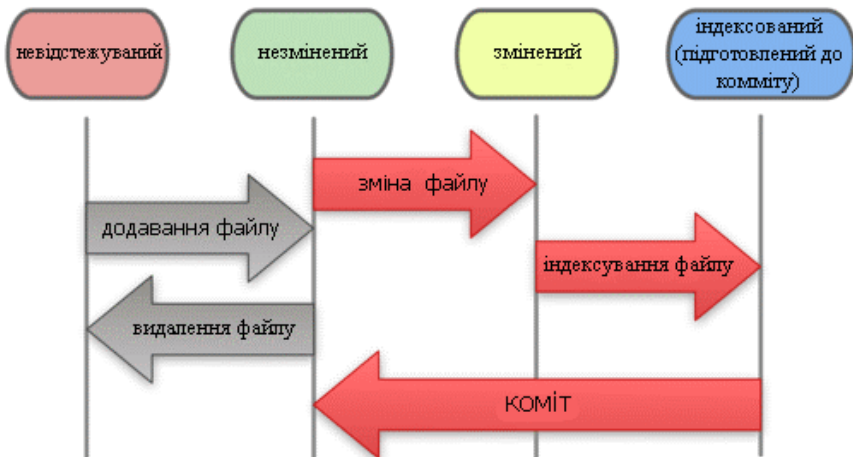
- SHA1 (Secure Hash Algorithm 1)

- Об'єкти Git
 - Blob (Binary Large Object)
 - Дерево (Tree)
 - Reference
 - Гілка (Head, Branch)
 - Тег (tag) /мітка
 - light tag
 - annotated tag

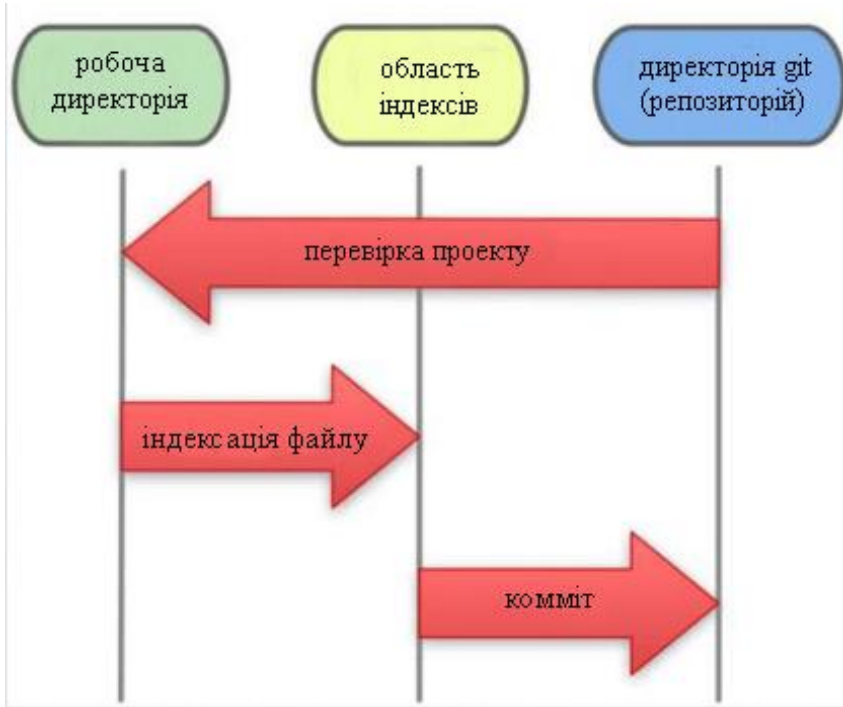
ТРИ СТАНИ ФАЙЛУ

- зафіксований
- змінений
- підготовлений

ЖИТТЕВИЙ ЦИКЛ ФАЙЛУ



GIT ПРОЕКТ



РОЗГАЛУЖЕННЯ

- git branch
- git branch <ім'я_гілки>
 - -d <ім'я_гілки>
 - -D <ім'я_гілки>
 - -m <ім'я_гілки>
 - --contains <ссылка_на_комміт>
- git checkout <им'я_ветви>
 - -f <ім'я_гілки>
 - -b <ім'я_гілки>
- git merge <гілка>
- git merge <ім'я_віддаленого_репозиторія>/<гілка_віддаленого_репозиторі

я>

3.3 Завдання на лабораторну роботу

3.3.1 Ознайомитися з теоретичними відомостями щодо основних команд GIT.

3.3.2 Виконати свій варіант завдання, аналогічно нульовому варіанту:

Варіант 0 (див. приклад)

Ініціалізувати каталог з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~).

Керівник проекту вносить свої дані в git, створює файл в якому прописує структуру майбутнього проекту, з коментарями щодо завдань іншим членам команди. Виконує комміт для фіксації внесених змін.

Перший член команди проекту вносить свої дані в git і створює свою гілку. Відкриває файл проекту, виконує завдання прописане керівником команди. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Другий член команди проекту вносить свої дані в git і створює свою гілку, відкриває файл проекту, виконує завдання прописане керівником команди. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Керівник проекту зливає гілки і перевіряє результат.

Відобразити всі результати роботи за допомогою gitk.

Варіант 1

Ініціалізувати каталог з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~). Користувач вводить свої дані, створює папку проекту, створює в ній файл з 2 завданнями. Виконує комміт для фіксації внесених змін. Створює нову гілку, відкриває файл і виконує першу задачу, виконує комміт. Створює ще одну підгілку, відкриває файл і виконує друге завдання, виконує комміт. Повертається в гілку master. Переглядає історію змін, отримує хеш всіх змін і по хеш відкочується до першого комміту. Відобразити всі результати роботи за допомогою gitk.

Варіант 2

Ініціалізувати каталог з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~). Користувач вводить свої дані, створює папку проекту, створює в ній файл із завданнями. Виконує комміт для фіксації внесених змін. Створює

нову гілку, відкриває файл і виконує завдання, виконує комміт. Повертається в гілку майстер, додає аліаси для branch, checkout, status, history. За допомогою аліасів переглядає всі гілки проекту, його поточний статус та історію коммітів.

Варіант 3

Ініціалізувати папку з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~). Користувач вводить свої дані, створює папку проекту, створює в ній файл із завданнями. Виконує комміт для фіксації внесених змін. Створює нову гілку, відкриває файл і виконує завдання, виконує комміт. Повертається в гілку master. Переглядає історію коммітів використовуючи одностроковий формат. Вносить зміни у файл, виконує комміт. Переглядає історію, використовуючи різні формати контролю відображення записів.

Варіант 4

Ініціалізувати папку з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~).

Керівник проекту вносить свої дані в git, створює кілька файлів в яких прописує структуру майбутнього проекту, з коментарями щодо завдань іншим членам команди. Виконує комміт для фіксації внесених змін.

Перший член команди проекту вносить свої дані в git і створює свою гілку. Відкриває проекти і виконує завдання (у декількох файлах) прописане керівником. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Другий член команди проекту вносить свої дані в git і створює свою гілку, відкриває проект проекту, переглядає його і видаляє один з файлів. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Керівник проекту зливає гілки, створює новий файл у проекті, вносить до нього зміни і виконує Ком. Переглядає історію коммітів і виконує останні зміни. Відобразити всі результати роботи за допомогою gitk.

3.3.3 Виконати аналіз отриманих результатів.

3.3.4 Оформити звіт та відповісти на контрольні питання.

2.3 Зміст звіту

2.4.1 Тема та мета роботи.

2.4.2 Результати виконання свого варіанту завдання, аналогічно нульовому варіанту

Виконання роботи (варіант 0)

Ініціалізувати каталог з майбутнім проектом в git. Створити файл .gitignore з описом ігнорованих файлів (*. [oa], * ~).

Керівник проекту вносить свої дані в git, створює файл в якому прописує структуру майбутнього проекту, з коментарями щодо завдань іншим членам команди. Виконує комміт для фіксації внесених змін.

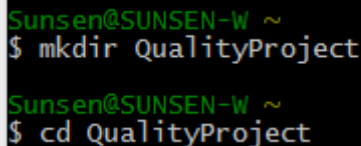
Перший член команди проекту вносить свої дані в git і створює свою гілку. Відкриває файл проекту, виконує завдання прописане керівником команди. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Другий член команди проекту вносить свої дані в git і створює свою гілку, відкриває файл проекту, виконує завдання прописане керівником команди. Виконує комміт для фіксації внесених змін і повертається в гілку master.

Керівник проекту зливає гілки і перевіряє результат.

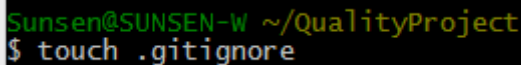
Відобразити всі результати роботи за допомогою gitk.

1. Створюємо каталог з майбутнім проектом та переходимо у нього

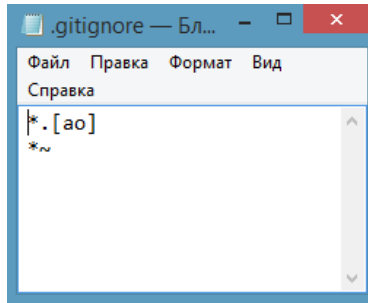


```
Sunsen@SUNSEN-W ~  
$ mkdir QualityProject  
Sunsen@SUNSEN-W ~  
$ cd QualityProject
```

2. Створюємо файл .gitignore з описом ігнорованих файлів



```
Sunsen@SUNSEN-W ~/QualityProject  
$ touch .gitignore
```



3. Створюємо репозиторій для проекту

```
Sunsen@SUNSEN-W ~/QualityProject
$ git init
Initialized empty Git repository in c:/Users/Sunsen/QualityProject/.git/
```

4. Додаємо у нього файли

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git add .
```

5. Робимо коміт

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git commit -m "First commit"
[master (root-commit) 56c1c0f] First commit
1 file changed, 2 insertions(+)
create mode 100644 .gitignore
```

6. Перший користувач(керівник проекту) виконує наступні дії:

- 6.1 Вносить свої дані (ім'я и email)

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.name "teamLeader"

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.email "teamleader.en@gmail.com"
```

- 6.2 Створює файл з описом структури проекту

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ touch quality.cpp
```

```

#include "mainwindow.h"
#include <cmath>
#include "ui_mainwindow.h"

int n1, n2, m, N1, N2, N, R;

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

//задания для partyOne
//написать методы для подсчета количества строк кода и пустых строк

//задания для partyTwo
//написать методы для подсчета количества комментариев и строк кода(общее кол-во строк -
комментарии - пустые строки)

```

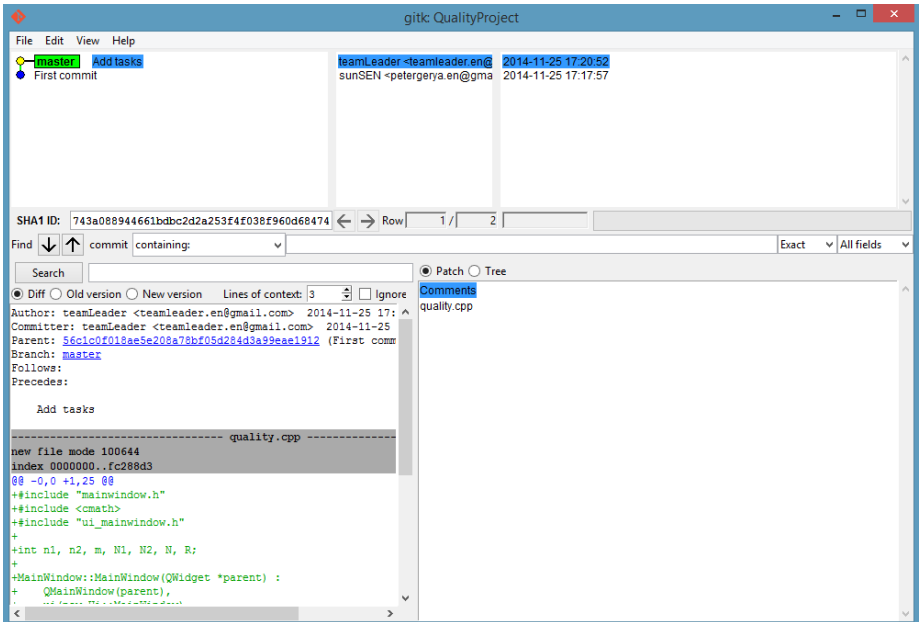
6.3 Додає усі файли до репозиторію

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git add .
```

6.4 Виконує коміт

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git commit -m "Add tasks"
[master 743a088] Add tasks
1 file changed, 25 insertions(+)
create mode 100644 quality.cpp
```

При виконанні команди gitk отримуємо:



7. Перший член проекту виконує наступні дії:

7.1 Вносить свої дані (ім'я та email)

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.name "crewmanOne"

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.email "crewman.one@gmail.com"
```

7.2 Створює свою гілку та переходить у неї

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git branch crewmanOne

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git checkout crewmanOne
Switched to branch 'crewmanOne'
```

7.3 Виконує своє завдання, яке прописано керівником команди

```

18
19 //задания для crewmanOne
20 //написать методы для подсчета количества строк кода и пустых строк
21
22 //количество строк кода
23 void MainWindow::on_SLOC_clicked()
24 {
25     QString str=ui->plainTextEdit->toPlainText();
26     int n=str.count("\n");
27     ui->label->setText(QString::number(n+1));
28 }
29
30 // количество пустых строк
31 void MainWindow::on_EmptyLines_clicked()
32 {
33     QString str=ui->plainTextEdit->toPlainText();
34     int n=str.count("\n\n");
35     ui->label_2->setText(QString::number(n));
36 }

```

7.4 Додає усі файли до репозиторію

```

Sunsen@SUNSEN-W ~/QualityProject (crewmanOne)
$ git add .

```

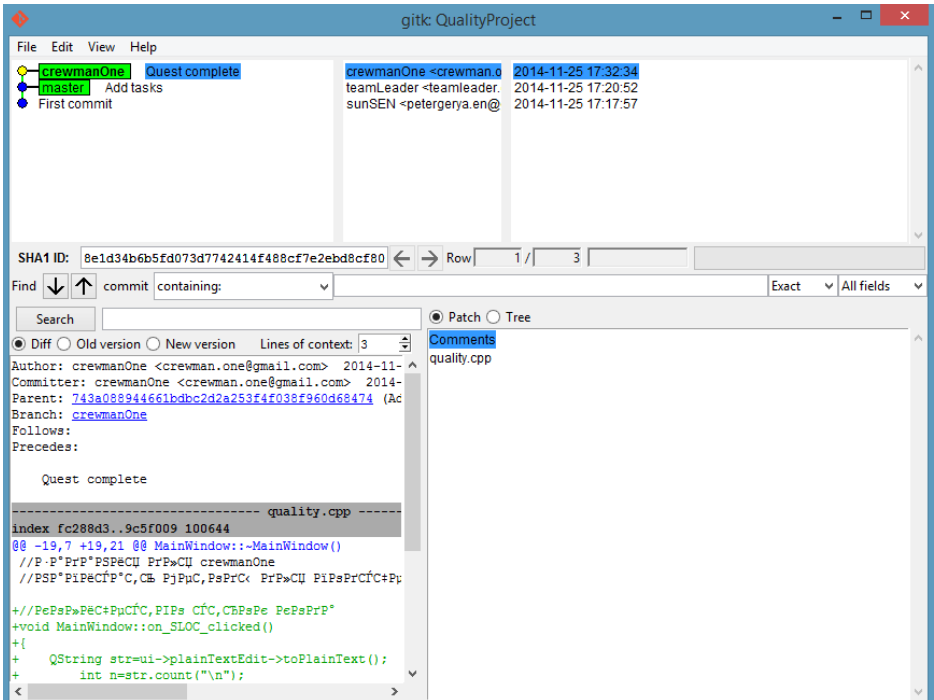
7.5 Робить коміт

```

Sunsen@SUNSEN-W ~/QualityProject (crewmanOne)
$ git commit -m "Quest complete"
[crewmanOne 8e1d34b] Quest complete
1 file changed, 14 insertions(+)

```

При виконанні команди gitk отримуємо:



7.6 Повертається у гілку майстра

```
Sunsen@SUNSEN-W ~/QualityProject (crewmanOne)
$ git checkout master
Switched to branch 'master'
```

8. Другий член проекту виконує наступні дії:

8.1 Вносить свої дані (ім'я та email)

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.name "crewmanTwo"

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git config --global user.email "crewman.two@gmail.com"
```

8.2 Дивиться назви усіх існуючих гілок, щоб визначитись з назвою своєї гілки

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git branch
crewmanOne
* master
```

8.3 Створює свою гілку та переходить у неї

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git branch crewmanTwo

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git checkout crewmanTwo
Switched to branch 'crewmanTwo'
```

8.4 Виконує своє завдання прописане керівником проекту

```
quality.cpp  <Виберіть символ>  # Строка: 1, Столбец: 1
23
24 //задания для crewmanTwo
25 //написать методы для подсчета количества комментариев и строк кода(общее кол-во строк - коммент
26
27 //количество комментариев
28 void MainWindow::on_Comments_clicked()
29 {
30     QString str=ui->plainTextEdit->toPlainText();
31     int n=str.count("\n/");
32     ui->label_3->setText(QString::number(n));
33     int k=str.count("\n");
34     ui->label_5->setText(QString::number(((float)n)/(k+1)*100)+"%");
35 }
36
37 //количество строк кода(общее кол-во строк - комментарии - пустые строки)
38
39 void MainWindow::on_OnlyCode_clicked()
40 {
41     QString str=ui->plainTextEdit->toPlainText();
42     int n1=str.count("\n/");
43     int n2=str.count("\n\n");
44     int n=str.count("\n");
45     ui->label_4->setText(QString::number(n-n1-n2+1));
46 }
47
48
```

8.5 Додає усі файли до репозиторію

```
Sunsen@SUNSEN-W ~/QualityProject (crewmanTwo)
$ git add .
```

8.6 Виконує коміт

```
Sunsen@SUNSEN-W ~/QualityProject (crewmanTwo)
$ git commit -m "Quest complete"
[crewmanTwo 4de08e1] Quest complete
1 file changed, 22 insertions(+)
```

При виконанні команди gitk отримуємо:

gitk: QualityProject

File Edit View Help

crewmanTwo Quest complete
 master Add tasks
 First commit

SHA1 ID	Author	Committer	Date
4de08e11d18cbcf149d9a0a7181f78fe4c71ab5	crewmanTwo <crewman.two@gmail.com>	crewmanTwo <crewman.two@gmail.com>	2014-11-25 18:07:18
	teamLeader <teamleader.en@gmail.com>	teamLeader <teamleader.en@gmail.com>	2014-11-25 17:20:52
	sunSEN <petergerya.en@gmail.com>	sunSEN <petergerya.en@gmail.com>	2014-11-25 17:17:57

SHA1 ID: 4de08e11d18cbcf149d9a0a7181f78fe4c71ab5 Row 1 / 3

Find commit containing:

Search

Diff Old version New version Lines of context: 3 Ignore space

Author: crewmanTwo <crewman.two@gmail.com> 2014-11-25 18:07:18
 Committer: crewmanTwo <crewman.two@gmail.com> 2014-11-25 18:07:18
 Parent: 743a088944661bdbc2d2a253f4f038f960d68474 (Add tasks)
 Branch: crewmanTwo
 Follows:
 Precedes:

Quest complete

----- quality.cpp -----
 index fc288d3..30c9a69 100644
 @@ -23,3 +23,25 @@ MainWindow::~MainWindow()
 //P*P*PrP*FSPeCj PrP*CU crewmanTwo
 //PSP*PiPeCfP*C, Cb PjPuC, PsPrC< PrP*CU PiPsPrCfC+PuC, P* PePsP*Pe
 +
 +//PePsP*PeC+PuCfC, PIPs PePsPjPjPuPSC, P*CBPePuPI
 +void MainWindow::on_Comments_clicked()
 +{
 + OString stravi=OPlainTextEdit->OPlainText();

Comments
 quality.cpp

8.7 Повертається у гілку майстра

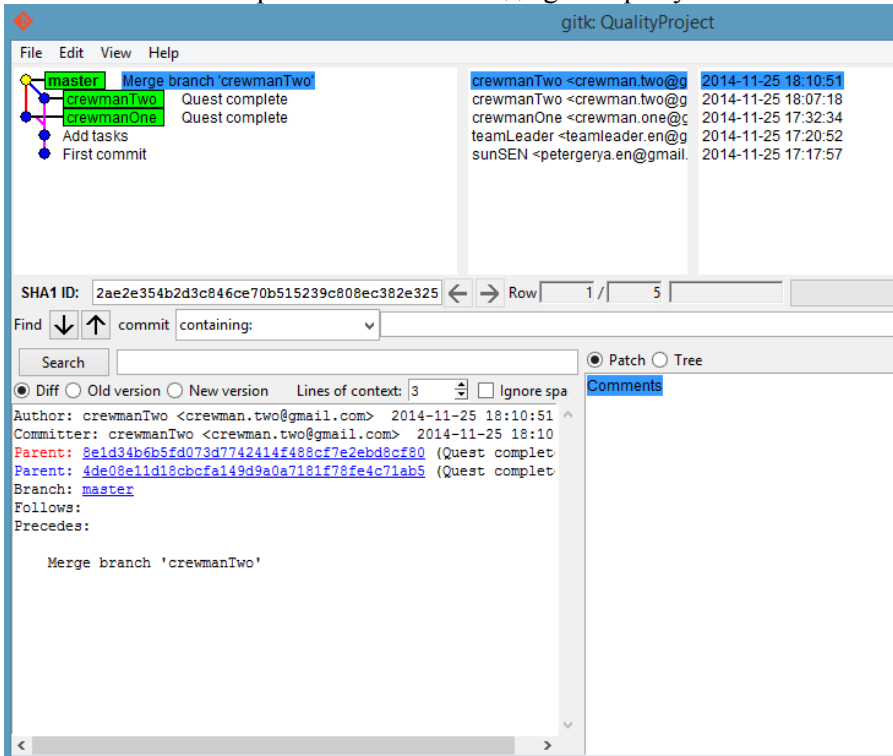
```
Sunsen@SUNSEN-W ~/QualityProject (crewmanTwo)
$ git checkout master
Switched to branch 'master'
```

9. Керівник проекту зливає усі гілки разом

```
Sunsen@SUNSEN-W ~/QualityProject (master)
$ git merge crewmanOne
Updating 743a088..8e1d34b
Fast-forward
 quality.cpp | 14 ++++++++
 1 file changed, 14 insertions(+)

Sunsen@SUNSEN-W ~/QualityProject (master)
$ git merge crewmanTwo
Auto-merging quality.cpp
Merge made by the 'recursive' strategy.
 quality.cpp | 22 ++++++++
 1 file changed, 22 insertions(+)
```

10. Перевіряє результат
При виконанні команди gitk отримуємо:



Файл quality.cpp має вигляд:

```

MainWidget::~MainWidget()
{
    delete ui;
}

//задания для sgewmanOne
//написать методы для подсчета количества строк кода и пустых строк

//количество строк кода
void MainWidget::on_SLOC_clicked()
{
    QString str=ui->plainTextEdit->toPlainText();
    int n=str.count("\n");
    ui->label->setText(QString::number(n+1));
}

// количество пустых строк
void MainWidget::on_EmptyLines_clicked()
{
    QString str=ui->plainTextEdit->toPlainText();
    int n=str.count("\n\n");
    ui->label_2->setText(QString::number(n));
}

//задания для sgewmanTwo
//написать методы для подсчета количества комментариев и строк кода(общее кол-во строк - комментарии - пустые строки)

//количество комментариев
void MainWidget::on_Comments_clicked()
{
    QString str=ui->plainTextEdit->toPlainText();
    int n=str.count("\n//");
    ui->label_3->setText(QString::number(n));
    int k=str.count("\n");
    ui->label_5->setText(QString::number(((float)n)/(k+1)*100)+"%");
}

//количество строк кода(общее кол-во строк - комментарии - пустые строки)
void MainWidget::on_OnlyCode_clicked()
{
    QString str=ui->plainTextEdit->toPlainText();
    int n1=str.count("\n//");
    int n2=str.count("\n\n");
    int n=str.count("\n");
    ui->label_4->setText(QString::number(n-n1-n2+1));
}

```

3.4.3. Аналіз отриманих результатів.

3.4.5. Висновки, що містять відповіді на контрольні запитання, а також відображують результати виконання роботи та їх критичний аналіз.

3.5 Контрольні запитання

- 3.5.1. Як створити Git-репозиторій?
- 3.5.2. Як додати під версійний контроль усі файли?
- 3.5.3. У скількох станах можуть знаходитися файли в репозиторії Git?
- 3.5.4. Яка команда використовується для визначення стану файлів в репозиторії?
- 3.5.5. Для чого використовується файл .gitignore?

- 3.5.6. Якою командою переглянути не проіндексовані зміни, зроблені після останнього коміту?
- 3.5.7. Якою командою виконується фіксація змін?
- 3.5.8. Як видалити файл з Git-репозиторію?
- 3.5.9. Як перемістити файл у Git-репозиторії?
- 3.5.10. Як переглянути історію комітів?
- 3.5.11. Як змінити останній коміт?
- 3.5.12. Яка команда створить нову гілку в Git-репозиторії?
- 3.5.13. Як об'єднати гілку А та Б?

ЛАБОРАТОРНА РОБОТА №4

РОБОТА З ВІДДАЛЕНИМ РЕПОЗИТОРІЄМ GIT

4.1 Мета роботи

Вивчити основні команди по роботі з віддаленим репозиторієм GIT.

4.2 Основні теоретичні відомості

Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (push) и получение (pull) данных в и из них тогда, когда нужно обмениваться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как отслеживаемые (tracked) или нет и прочее. Данный раздел охватывает все перечисленные навыки по управлению удалёнными репозиториями.

Отображение удалённых репозиторияв

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду `git remote`. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы клонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, `origin` — это имя по умолчанию, которое Git присваивает серверу, с которого вы клонировали:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
```

```
origin
```

Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию `-v`:

```
$ git remote -v
origin git://github.com/schacon/ticgit.git (fetch)
origin git://github.com/schacon/ticgit.git (push)
```

Если у вас больше одного удалённого репозитория, команда покажет их все. Например, мой репозиторий Grit выглядит следующим образом.

```
$ cd grit
$ git remote -v
bakdoor git://github.com/bakdoor/grit.git
cho45 git://github.com/cho45/grit.git
defunkt git://github.com/defunkt/grit.git
koke git://github.com/koke/grit.git
origin git@github.com:mojombo/grit.git
```

Это означает, что мы легко можем получить изменения от любого из этих пользователей. Но, заметьте, что `origin` — это единственный удалённый сервер прописанный как SSH-ссылка, поэтому он единственный, в который я могу помещать свои изменения.

Добавление удалённых репозитория

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозитория, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните `git remote add [сокращение] [url]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

Теперь вы можете использовать в командной строке имя `pb` вместо полного URL. Например, если вы хотите извлечь (fetch) всю

информацию, которая есть в репозитории Павла, но нет в вашем, вы можете выполнить `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]    master    -> pb/master
* [new branch]    ticgit    -> pb/ticgit
```

Ветка `master` Павла теперь доступна локально как `pb/master`. Вы можете слить (`merge`) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [имя удал. сервера]
```

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты. (В главе 3 мы перейдём к более детальному рассмотрению, что такое ветки и как их использовать.)

Когда вы клонируете репозиторий, команда `clone` автоматически добавляет этот удалённый репозиторий под именем `origin`. Таким образом, `git fetch origin` извлекает все наработки, отправленные (`push`) на этот сервер после того, как вы скопировали его (или получили изменения с помощью `fetch`). Важно отметить, что команда `fetch` забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду `git pull`. Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным. К тому же по умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка `master`). Выполнение `git pull`, как

правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая:

```
git push [удал. сервер] [ветка].
```

Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён. Вам придётся сначала вытянуть (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

Инспекция удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториях, вы можете использовать команду `git remote show [удал. сервер]`. Если вы выполните эту команду с некоторым именем, например, origin, вы получите что-то подобное:

```
$ git remote show origin
```

```
* remote origin
```

```
URL: git://github.com/schacon/ticgit.git
```

```
Remote branch merged with 'git pull' while on branch master
```

```
master
```

```
Tracked remote branches
```

```
master
```

```
ticgit
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке master, выполните `git pull`, ветка master с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере. И для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториев

Для переименования ссылок в новых версиях Git'a можно выполнить `git remote rename`, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать `pb` в `paul`, вы можете сделать это следующим образом:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также меняет для вас имена удалённых веток. То, к чему вы обращались как `pb/master`, стало `paul/master`.

Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определённое зеркало, или, возможно, контрибьютор перестал быть активным), вы можете использовать `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

4.3 Завдания на лабораторну роботу

1. Зарегистрируйтесь на Bitbucket (<https://bitbucket.org>) (рис.1)

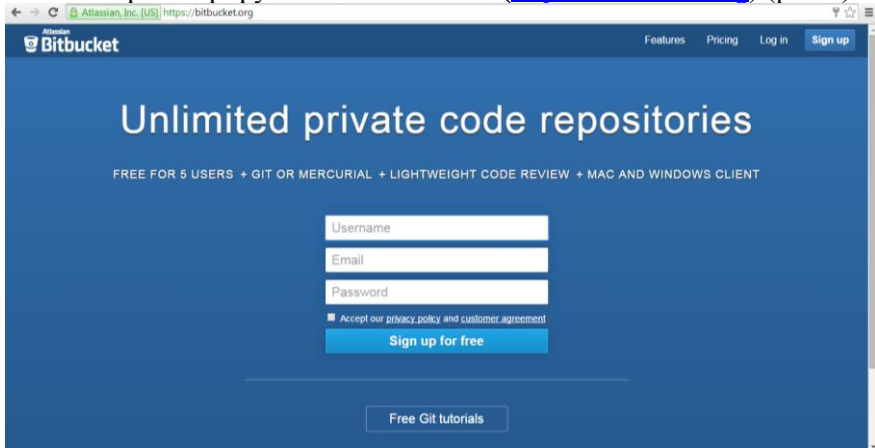


Рисунок 1. Регистрация на Bitbucket

2. Установите и настройте Git-клиент (для Windows – `msysgit` (скачать можно по адресу <http://msysgit.github.io/>)).

3. После установки, запускаем "Git Bash" и вводим следующие команды (для вставки используйте клавиши Shift+Insert):

```
- задаём глобальное имя и e-mail:
git config --global user.email "my_email@mail.com"
git config --global user.name "my_nickname"
- устанавливаем метод по-умолчанию для push в simple.
git config --global push.default simple
Краткое описание методов push:
```

nothing - "git push" не будет работать без явно указанных имени и ветки репозитория

current - обновляется удалённая ветка, имя которой совпадает с текущей веткой

upstream - обновляется upstream ветка, т.е. ветка, из которой выполняется "pull"

simple - то же, что и "upstream", только дополнительно выполняется проверка на совпадение имён веток (самый безопасный метод)

matching - обновляет все ветки по имени (метод по-умолчанию в Git < 2.0)

Если не устанавливать метод, то при "git push" будут появляться warning сообщения.

4. Копирование проекта из удаленного репозитория

Создайте fork указанного проекта и скачайте проект на локальную машину двумя способами в разные папки - с помощью командной строки и с помощью qt creator.

Внимание! Для доступа к удаленному репозиторию через proxy-server пропишите следующую команду:

```
git config --global http.proxy
http://proxyuser:proxypass@10.0.2.1:8080
```

4.1. Через командную строку

Запускаем "Git Bash" и переходим в директорию для будущего проекта:

```
cd /c/Projects/My_project (или cd C:\\Projects\\My_project)
```

Затем, в корневой папке проекта, выполняем следующие команды:

- Инициализируем пустой локальный репозиторий.

```
git init
```

- Копируем файлы в локальный репозиторий, предварительно сделав fork из репозитория <https://bitbucket.org/tata007/zntulabs> (рис. 2)

```
git clone https://usernamebitbucket@bitbucket.org/tata007/zntulabsfork.git
```

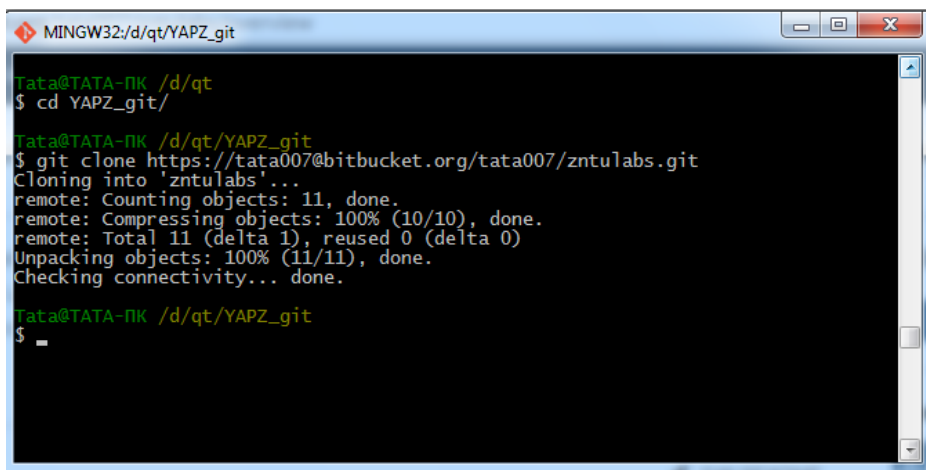


Рисунок 2. Копирование проекта из удаленного репозитория через командную строку

4.2. С использованием qt creator

При создании проекта выбираете вариант "Импортировать проект / Git Repository Clone" (рис. 3).

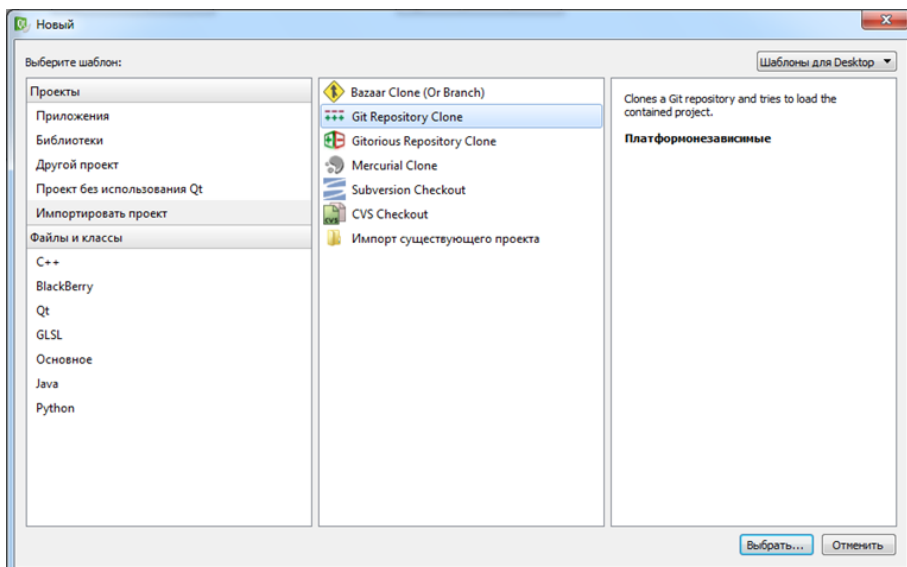


Рисунок 3. Выбор шаблона при копировании проекта из удаленного репозитория в qt creator

Следующий шаг – выбор репозитория (клонировать из созданного fork'a <https://bitbucket.org/tata007/zntulabs>) (рис. 4).

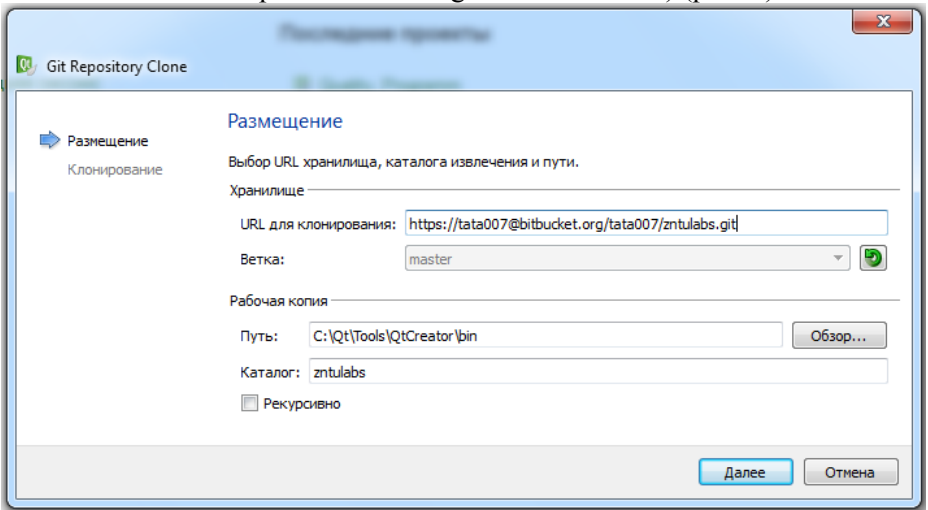


Рисунок 4. Выбор хранилища при копировании проекта из удаленного репозитория в qt creator

5. Для каждой версии проекта создать свою ветку, внести изменения в обе версии и зафиксировать их commit'ом

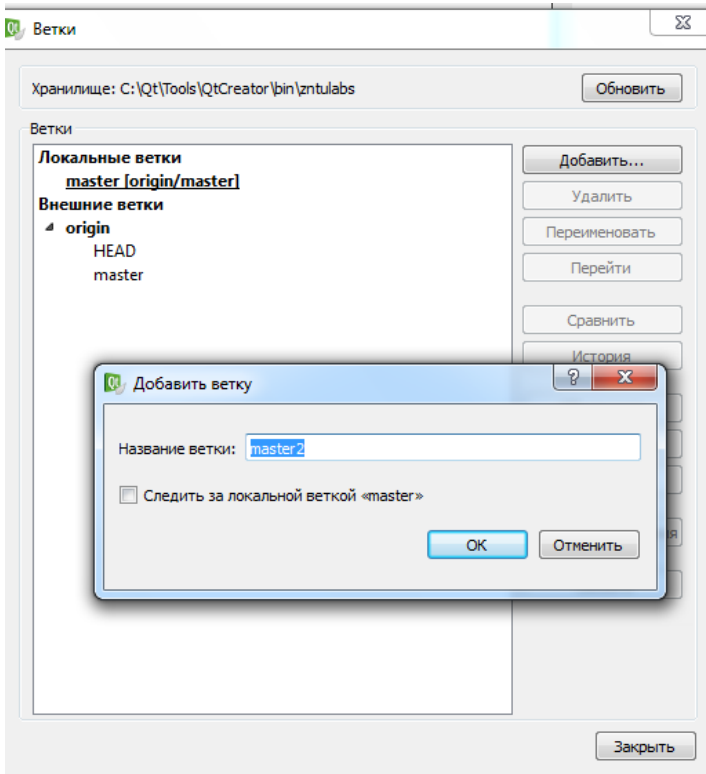


Рисунок 5. Создание ветки в qt creator

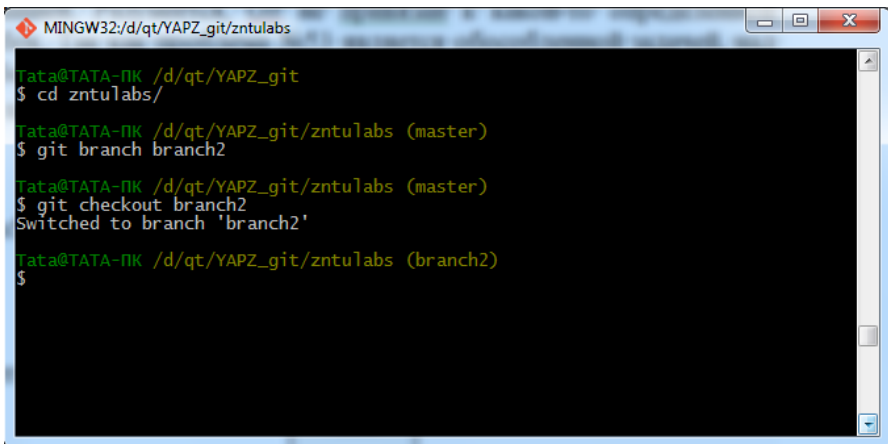


Рисунок 6. Создание ветки с помощью командной строки

6. Изучить команды git, которые предоставляет qt creator (рис. 7)

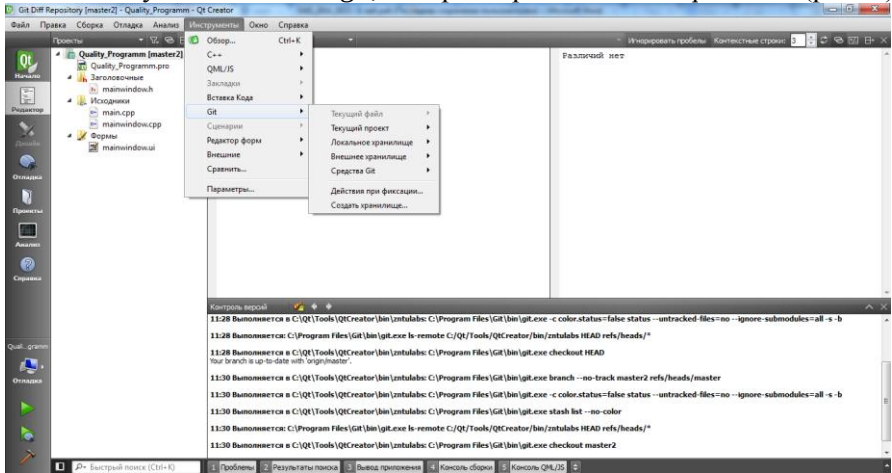


Рисунок 7. Инструментарий для работы с git в qt creator

7. Залить созданные ветки на fork <https://bitbucket.org/tata007/zntulabs> в ветки, названные "фамилия+qt" и "фамилия+bash" (рис. 8 и 9)

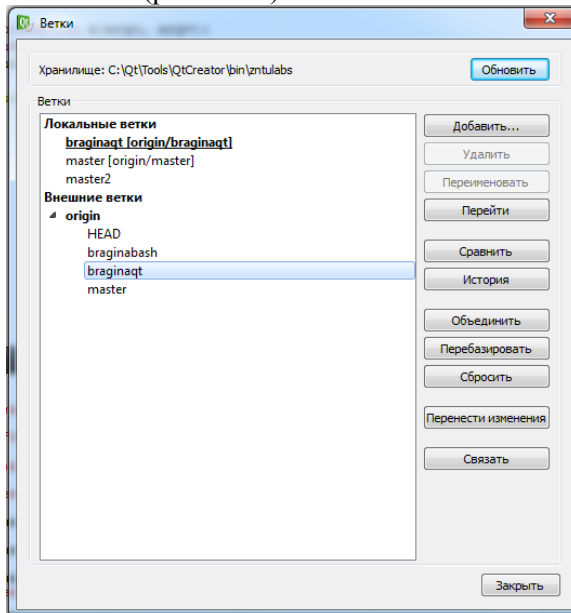


Рисунок 8. Создание удаленной ветки в qt creator

```

MINGW32/d/qt/YAPZ_git/zntulabs
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 304 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://tata007@bitbucket.org/tata007/zntulabs.git
 * [new branch]      branch2 -> branch2

Tata@TATA-ПК /d/qt/YAPZ_git/zntulabs (branch2)
$ git checkout --track -b braginabash origin/braginabash
Branch braginabash set up to track remote branch braginabash from origin.
Switched to a new branch 'braginabash'

Tata@TATA-ПК /d/qt/YAPZ_git/zntulabs (braginabash)
$ git branch
* braginabash
  branch2
  master

Tata@TATA-ПК /d/qt/YAPZ_git/zntulabs (braginabash)
$ git commit -a -m "commit in braginabash"
[braginabash c38cab2] commit in braginabash
1 file changed, 1 insertion(+), 1 deletion(-)

Tata@TATA-ПК /d/qt/YAPZ_git/zntulabs (braginabash)
$ git push
Password for 'https://tata007@bitbucket.org':
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 314 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://tata007@bitbucket.org/tata007/zntulabs.git
 * [new branch]      braginabash -> braginabash

Tata@TATA-ПК /d/qt/YAPZ_git/zntulabs (braginabash)
$

```

Рисунок 9. Создание удаленной ветки в командной строке

8. Сделайте pull request в репозиторий <https://bitbucket.org/tata007/zntulabs>

2.5 Контрольні запитання

4.5.1 Як копіювати віддалений Git-репозиторій?

4.5.2 Як відправити зроблені в локальній репозиторії зміни до віддаленого Git-репозиторію?

ЛІТЕРАТУРА

1. Bevan N. Specifying and Measuring Quality in Use // Proceedings of the 22nd international conference on Software engineering, 2000. – P. 322–331.
2. Halpern M. Addressing Quality Risk in Collaborative Automotive Design: Commentary, – Gartner Group, 2001. – P. 262–293.
3. Hassenzahl M., Platz A., Burmester M., Lehner K. Hedonic and Ergonomic Quality Aspects Determine a Software's Appeal // Proceedings of the CHI 2000 conference on Human factors in computing systems, 2000. – P. 193–201.
4. Hutcheson M. L. Software Testing Fundamentals: Methods and Metrics. John Wiley & Sons. – 2003. – 408 p.
5. IEEE Standard for a Software Quality Metrics Methodology Copyright ©, Institute of Electrical and Electronics Engineers, Inc. 2005.
6. Kan H.N. Metrics and Models in Software Quality Engineering, Second Edition, 2002. – 560 p.
7. Pandian C. R. Software Metrics: A Guide to Planning, Analysis, and Application Auerbach Publications, 2004. – 312 p.
8. Хорстман К., Корнелл Г. Java 2. Библиотека профессионала, том 1. Основы, 7-е издание. – М: Издательский дом "Вильямс", 2006. – 896с.
9. Хорстман К., Корнелл Г. Java 2. Библиотека профессионала, том 2. Тонкости программирования, 7-е издание. – М: Издательский дом "Вильямс", 2006. – 1168с.