МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.

«___» _____2021 р.

ДИПЛОМНИЙ ПРОЄКТ

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
"БАКАЛАВР"

Тема: Програмний модуль агрегації курсів валют з сайтів банків

Виконавець: Рябець А.В.

Керівник: Ткаченко В.Г.

Нормоконтролер: Тупота Є.В.

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління
Спеціальність 123 "Комп'ютерна інженерія"
(шифр, найменування)

Освітньо професійна програма «Системне програмування»
Форма навчання заочна

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____Литвиненко О. Є.

«_____»_____2020 р.

ЗАВДАННЯ
на виконання дипломного проєкту

Рябця Андрія Володимировича
(прізвище, ім'я, по батькові)

1.Тема роботи: "Програмний модуль агрегації курсів валют з сайтів банків"

затверджена наказом ректора від "21" грудня 2020 року № 2523 /ст.

2. Термін виконання роботи: з 11.01.2021 до 28.02.2021

3. Вихідні дані до роботи: 1) вимоги до модуля адміністратора сайту;
2) основні операції в управлінні контентом сайту торогового підприємства

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

1) аналіз підходів до моделювання бізнес-процесів;

2) методи побудови агрегаторів даних банків і валютних бірж;

3) проектування системи обробки даних банків і валютних бірж.

5. Перелік обов'язкового графічного матеріалу:

1) принципи роботи програмного модуля з біржами і банками;
2) принцип роботи багатопотокового серверу *Node.js*;
3) робочі вікна системи;
4) вікна додатку з аналізом валютних пар;
5) схема алгоритму роботи модуля підключення до сайтів бірж і банків;
6) схема алгоритму роботи модуля парсингу.

6. Календарний план

№ п/п	Етапи виконання дипломного проєкту	Термін виконання етапів	Примітка
1	Провести аналіз літератури за темою дипломного проєкту та аналіз існуючих систем	11.01.21 12.01.21	
2	Зробити вибір компонентів системи	13.01.21- 14.01.21	
3	Розробити структуру програмних засобів для реалізації системи торгових операцій онлайн	15.01.21- 16.01.21	
4	Розробити програмні засоби для впровадження модифікованих функцій адміністратора онлайн магазину	17.01.21- 28.01.21	
5	Провести налаштування програмних засобів на сервері	29.01.21- 31.01.21	
6	Написати пояснювальну записку	01.02.21- 12.02.21	
7	Підготувати презентацію і захистити роботу	13.02.21- 23.02.21	

7. Дата видачі завдання «_11_» січня 2021 р.

Керівник дипломного проєкту_____Ткаченко В.Г.
(підпис)

Завдання прийняв до виконання_____Рябець В.А.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломного проєкту “Програмний модуль агрегації курсів валют з сайтів банків”: 71 с., 25 рис., 21 літературне джерело, 1 додаток.

АГРЕГАТОР, ВАЛЮТА, ЕЛЕКТРОННІ ГРОШІ, ВАЛЮТНА БІРЖА, ВАЛЮТНИЙ ОБМІННИК, ЦИФРОВА ВАЛЮТА.

Мета дипломного проєкту – аналіз методів обробки даних криптовалютних бірж.

Об'єкт дослідження – агрегації даних з валютних бірж і банків.

Предмет дослідження – система обробки даних валютних бірж і банків.

Наукова значимість полягає у реалізації методів обробки даних валютних бірж і банків для їх агрегації і подальшої обробки.

Практична значимість полягає у розробці програмного забезпечення, що планується використовувати широкою аудиторією для швидкого доступу до відкритих джерел даних.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

BTC – загальноприйнята одиниця, що використовується для позначення одного біткоїну.

P2P – p*eer-to-peer*. Термін означає системи, що працюють як організована спільнота, надаючи можливість кожному учаснику безпосередньо взаємодіяти з іншими.

PAP – *Password Authentication Protocol* (протокол простої перевірки автентифікації)

PS – *Project system* (система проектів)

UML – *Unified Modeling Language*

ВСТУП

Сучасні віртуальні тенденції, які взяли свій початок в хайп-індустрії призводять до того, що дана область перетворюється в далеко не єдине місце заробітку. І одним з них як раз і є біржі криптовалюта. Хоча, багато експертів сходяться на думці, що це не що інше, як черговий «фінансовий міхур», який через якийсь час просто лопне. Однак, такий підхід зовсім хибний. Маючи достатній досвід, певні специфічні знання та володіючи інформацією, можна працювати так само ефективно, як і на фондовому ринку або на ринку Форекс.

На сьогоднішній день в Україні та в усьому світі програмування є основною технологією. Без програмування неможливо розвивати науку та техніку.

Однією з важливих частин програмування є зменшення зайвого коду та його оптимізація для створення ефективних та високопродуктивних програм. Якість вихідного коду залежить від багатьох факторів, наприклад, мови програмування, знань та досвіду програміста та багатьох інших. Аналіз вихідного коду показує, що основна частина його якості залежить від дотримання програмних показників.

Компілятори вихідного коду призначені для пошуку лише помилок, а частина, що відповідає метрикам, залежить лише від програміста.

Практика показує, що необхідне термінове вивчення метрик програмування, щоб пришвидшити процес підготовки висококваліфікованих фахівців. Ось чому проблема створення програмного забезпечення, що відповідає метрикам, є актуальною.

Підібравши прийнятний варіант, користувач здійснює перехід на сервіс, де грунтовніше аналізує пропоновані умови і реалізує заплановану конвертацію. На сайтах-агрегаторах обмінні операції не виконуються, а лише надається зведений набір ключової інформації для комфортного пошуку необхідних веб-проектів.

Це зразок тематичних баз даних, куди автоматично надходять інформаційні повідомлення про підтримуваних валютах, направлених обміну, нові сервіси

онлайн-конвертації криптовалюта, тощо. Користувачам залишається переглядати сортовані значення, а потім направлено йти на конкретний обмінник віртуальних валют.

Саме тому актуальність розробки програмного забезпечення для агрегації даних різних майданчиків в одній базі для подальшої обробки і аналізу отриманих даних є на сьогодні дуже актуальною задачею, яка вимагає побудови математичних моделей ціноутоверення і використання елементів системного аналізу для визначення тенденції в змінах на ринку.

РОЗДІЛ 1
АНАЛІЗ ПІДХОДІВ ДО МОДЕЛЮВАННЯ БІЗНЕС-ПРОЦЕСІВ

Покриття коду

Покриття кодує мірою, що використовується при тестуванні програмного забезпечення. Він описує ступінь, до якої був протестований вихідний код програми. Це форма тестування, яка безпосередньо перевіряє код, і тому є формою тестування білих ящиків. Охоплення коду було одним із перших методів, винайдених для систематичного тестування програмного забезпечення. Перше опубліковане посилання було зроблено Міллером та Мелоні у журналі Communications of ACM у 1963 році.

Покриття коду є одним із факторів, що враховують сертифікацію безпеки авіоніки. Стандарт, за яким сертифікується авіаційне спорядження Федеральною авіаційною адміністрацією (FAA), задокументований у DO-178B.

Згуртованість

У комп'ютерному програмуванні згуртованість є мірою того, наскільки сильно пов'язана функціональність, виражена вихідним кодом програмного модуля. Методи вимірювання згуртованості варіюються від якісних показників, що класифікують аналізований вихідний текст за допомогою рубрики з герменевтичним підходом, до кількісних вимірів, що вивчають текстові характеристики вихідного коду, щоб отримати числовий показник згуртованості. Згуртованість - це звичайний тип вимірювань, який, як правило, виражається як „висока згуртованість” або „низька зв’язність”. Модулі з високою згуртованістю, як правило, є кращими, оскільки висока згуртованість пов'язана з декількома бажаними рисами програмного забезпечення, включаючи надійність, надійність, багаторазовість та зрозумілість, тоді як низька згуртованість пов'язана з небажаними рисами, такими як важко підтримувати, важко перевіряти,

Зчеплення

Зв’язок, як правило, протиставляється згуртованості. Низька зв'язок часто корелює з високою згуртованістю, і навпаки. Метрики якості програмного забезпечення для зчеплення та згуртованості були винайдені Ларрі Константином, оригінальним розробником Структурного дизайну, який також був першим прихильником цих концепцій (див. Також SSADM). Низька зв'язок часто є ознакою добре структурованої комп'ютерної системи та гарного дизайну, а в поєднанні з високою згуртованістю підтримує загальні цілі високої читабельності та ремонтопридатності.

Цикломатична складність (Складність Маккейба)

Цикломатична складність(або умовна складність) - це програмна метрика (вимірювання). Він був розроблений Томасом Дж. Маккейбом-старшим у 1976 р. І використовується для позначення складності програми. Він безпосередньо вимірює кількість лінійно незалежних шляхів через вихідний код програми. Концепція, хоча і не метод, дещо схожа на концепцію загальної складності тексту, виміряну за допомогою Тесту на читаність Флеша-Кінкейда.

Цикломатична складність обчислюється за допомогою графіку управління потоком програми:

Вузли графіка відповідають неподільним групам команд рограми, а спрямоване ребро з'єднує два вузли, якщо друга команда може бути виконана відразу після першої команди. Цикломатична складність може також застосовуватися до окремих функцій, модулів, методів або класів в рамках програми.

Складність Холстеда

Заходи складності Холстеда - це програмні метрики, представлені

Моріс Говард Халстед в 1977 році як частина його трактату про створення емпіричної науки про розробку програмного забезпечення. Холстед робить зауваження, що метрики програмного забезпечення повинні відображати реалізацію або вираження алгоритмів різними мовами, але не залежати від їх

виконання на певній платформі. Тому ці показники обчислюються статично з коду.

Метою Халстеда було виявити вимірювані властивості програмного забезпечення та взаємозв'язок між ними. Це схоже на виявлення вимірюваних властивостей речовини (таких як об’єм, маса та тиск газу) та взаємозв’язків між ними (наприклад, рівняння газу). Таким чином, його метрики насправді є не просто метриками складності.

Кількість рядків коду

Багато корисних порівнянь включають лише порядок величин рядків коду в проекті. Проекти програмного забезпечення можуть варіюватися від 1 до 100 000 000 або більше рядків оди. Використання рядків коду для порівняння 10 000 рядкових проектів із 100 000 рядкових проектів набагато корисніше, ніж при порівнянні 20 000 рядкових проектів з 21 000 рядковими. Хоча дискусійно, як саме вимірювати рядки коду, розбіжності на порядок можуть бути чіткими показниками складності програмного забезпечення або людських годин.

Існує два основних типи вимірювань SLOC: фізичний SLOC (LOC) та логічний SLOC (LOC). Конкретні визначення цих двох показників різняться, але найпоширенішим визначенням фізичного SLOC є підрахунок рядків у тексті вихідного коду програми, включаючи рядки коментарів. Також містяться порожні рядки, якщо тільки рядки коду в розділі не містять більше 25% порожніх рядків. У цьому випадку порожні рядки, що перевищують 25%, не враховуються до рядків коду.

Логічний SLOC намагається виміряти кількість виконуваних "операторів", але їх конкретні визначення прив'язані до конкретних комп'ютерних мов (одним простим логічним показником SLOC для C-подібних мов програмування є число крапок з комою, що закінчують оператори). Набагато простіше створити інструменти, які вимірюють фізичний SLOC, а фізичні визначення SLOC простіше пояснити. Однак фізичні заходи SLOC чутливі до логічно нерелевантного форматування та стилів, тоді як логічний SLOC менш чутливий до форматування та стилів. Однак заходи SLOC часто викладаються, не даючи їх

визначення, і логічний SLOC часто може суттєво відрізнятися від фізичного SLOC.

Час виконання програми

В інформатиці тривалість виконання, час виконання, час виконання або

час виконанняможуть мати різне значення залежно від контексту. З а

Технічна перспектива може стосуватися часу, протягом якого працює програма

(виконуючи). Це контрастує з іншими фазами програми, такими як час компіляції, посилання

час, час завантаження і т. д. Тісно пов'язаний з цим найосновнішим сенсом є стан

вся система, на якій працює програма, що впливає на програму

виконання, яке варіюється в різних випадках. Такі як ліцензування програмного забезпечення

контекст, "час роботи" відноситься до ширшої ідеї встановлення даного

програмне забезпечення або комп’ютерна програма на комп’ютері чи сервері, незалежно від того, запущено чи ні.

Помилка під час виконання виявляється після або під час виконання програми,

тоді як компілятор виявляє помилку під час компіляції раніше, ніж програма

коли-небудь страчували. Перевірка типу, розподіл пам’яті і навіть генерація коду та

Оптимізація коду може бути виконана під час компіляції або під час виконання, залежно

на мові та компіляторі.

Розмір програми (двійковий)

Двійковий файл - це комп'ютерний файл, який може містити будь-який тип даних,

закодовані у двійковій формі для комп'ютерного зберігання та обробки; для

Наприклад, файли комп'ютерних документів, що містять відформатований текст. Багато двійкових файлів

формати містять частини, які можна інтерпретувати як текст; двійкові файли, що містять

лише викликаються текстові дані - наприклад, без будь-якої інформації про форматування

текстові файли. У багатьох випадках звичайні текстові файли вважаються такими, що відрізняються від

двійкові файли, оскільки двійкові файли складаються з не лише простого тексту. Коли

часто завантажується також повністю функціональна програма без будь-якого інсталятора

називається програмним двійковим файлом, або двійковим файлом.

1.1.1 Прийняття та громадська думка

Деякі фахівці з розробки програмного забезпечення вказують на це спрощення

вимірювання може завдати більше шкоди, ніж користі. Інші відзначали, що метрики

стали невід’ємною частиною процесу розробки програмного забезпечення. Вплив

вимірювання на програмістах психології викликало побоювання щодо шкідливого

наслідки для продуктивності через стрес, тривожність та спроби обдурити

показники, тоді як інші вважають, що це позитивно впливає на цінність розробників

до власної роботи та запобігати їх недооцінці. Деякі стверджують це

визначення багатьох методологій вимірювання є неточними, і

отже, часто незрозуміло, як приходять засоби для їх обчислення

конкретний результат, тоді як інші стверджують, що недосконала кількісна оцінка краща за

none ("Ви не можете контролювати те, що не можете виміряти."). Це свідчать дані

програмні показники широко використовуються урядовими установами США

військові, NASA, IT-консультанти, академічні установи та комерційні та

програмне забезпечення для оцінки академічного розвитку.

Оскільки розробка програмного забезпечення є складним процесом, з великою різницею в обох

методологій та цілей, складно визначити або виміряти програмне забезпечення

якості та кількості та визначити дійсне та одночасне вимірювання

метрична, особливо коли робиться такий прогноз до проектування деталей.

Інше джерело труднощів і дискусій полягає у визначенні того, які показники важливі,

і що вони означають. Таким чином, практична корисність програмних вимірювань

обмежувався вузькими доменами, де вони включають:

- Розклад
- Розмір / Складність
- Вартість
- Якість

Загальна мета вимірювання може бути націлена на одну або декілька з перерахованих вище

аспекти, або баланс між ними як показник мотивації команди або

виконання проекту.

1.1 Порівняльний аналіз існуючих інструментів, що автоматизують вимірювання метричних характеристик вихідного коду

1.2.1 Інструменти аналізу коду з відкритим кодом чи некомерційним кодом

1.1.1.1 Багатомовний

Лось

Лосьце безкоштовна і відкрита платформа для програмного забезпечення та даних

аналіз, побудований у Фаро. Moose пропонує безліч послуг, починаючи від імпорту

аналіз даних, моделювання, вимірювання, запити, видобуток та побудова

інструменти інтерактивного та візуального аналізу. Лось народився в дослідницькому контексті і

в даний час його підтримують кілька дослідницьких груп у всьому світі. це є

дедалі більше застосовується у промисловості.

Філософія Лося полягає в тому, щоб дати можливість аналітику створювати нове

спеціальні інструменти аналізу та налаштування потоку аналізу. Поки Лось є

в основному використовується в програмному аналізі, він побудований для роботи з будь-якими даними.

Для досягнення цього він пропонує безліч механізмів та рамок:

 Імпорт та мета-мета-моделювання досягається за допомогою загального метаописаного механізму. Будь-яка мета-модель описується з точки зору самоописаної мета-мета-моделі, і на основі цього опису імпорт / експорт забезпечується через Формат файлу MSE. За допомогою цього формату файлу Moose може обмінюватися даними із зовнішніми інструментами.

 Для синтаксичного аналізу Moose пропонує нову структуру, яка використовує декілька технологій синтаксичного аналізу (наприклад, синтаксичний аналіз граматики виразів) і забезпечує вільний інтерфейс для зручності побудови.

 Аналіз програмного забезпечення спеціально підтримується сімейством метамоделей FAMIX. Ядро FAMIX - це мовна незалежна мета-модель, яка схожа

на UML, але орієнтована на аналіз. Крім того, він забезпечує розширений інтерфейс для запитів моделей.

 Візуалізація підтримується двома різними механізмами: один для вираження візуалізації графіків та інший для вираження діаграм. Вони обидва забезпечують вільний інтерфейс для зручності побудови.

 Перегляд - це важливий принцип у Moose, і він також підтримується різними способами. Загальний інтерфейс дозволяє аналітику переглядати будь-яку модель. Щоб мати можливість вказати конкретні браузери, Moose пропонує загальний механізм, який полегшує специфікацію за допомогою певного вільного інтерфейсу.

Детектор копіювання / вставки

PMD- це статичний аналізатор вихідного коду Java на основі набору правил, який ідентифікує

потенційні проблеми, такі як:

 Можливі помилки- Порожні блоки спроб / зловити / нарешті / переключити.

 Мертвий код- невикористані локальні змінні, параметри та приватні методи.

 Порожні оператори if / while.

 Надскладні вирази- Непотрібні оператори if для циклів, які можуть бути циклами while.

 Субоптимальний код- Використання марнотратного рядка / StringBuffer.

 Класи з високими вимірами цикломатичної складності.

 Дублікат коду- Скопійований / вставлений код може означати скопійовані / вставлені помилки та зменшує ремонтопридатність.

Хоча офіційно PMD ні за що не виступає, у нього є кілька неофіційних

імена, найбільш підходящим, мабуть, є помилка програмування

ДетекторЯк правило, помилки PMD - це не справжні помилки, а досить неефективний код,

тобто програма все одно може функціонувати належним чином, навіть якщо вони не були виправлені.

Гідролокатор

Гідролокаторєплатформа якості з відкритим вихідним кодом - -. Використання ехолота

різноманітніінструменти статичного аналізу коду, такі як Checkstyle, PMD, FindBugs та

Конюшина для вилучення програмних показників, які потім можна використовувати для вдосконалення програмного забезпечення

якість.

 Пропонує звіти про дубльований код, стандарти кодування, модульні тести, охоплення коду, складний код, потенційні помилки, коментарі та дизайн та архітектуру.

 Основною підтримуваною мовою є Java. Інші мови підтримуються розширеннями. Сьогодні кілька відкритих комерційних та комерційних розширень можуть охоплювати мови C, PHP, Flex, Groovy, JavaScript, PL / SQL, COBOL та Visual Basic 6.

 Він інтегрується з Maven, Ant та інструментами безперервної інтеграції (Atlassian Bamboo, Jenkins та Hudson).

 Розширюється за допомогою плагінів.

 Метрики проектування та архітектури.

 Впроваджує методологію SQALE.

Яська

Яскаєпрограма з відкритим кодом, яка шукає уразливості системи безпеки,

якість коду, продуктивність та відповідність найкращим практикам у джерелі програм

код. Він використовує зовнішніпрограми з відкритим кодом, такі

якFindBugs, PMD, JLint, JavaScript Lint, PHPLint, Cppcheck, ClamAV, Pixy,

iRATS для сканування певних типів файлів, а також містить багато спеціальних сканерів

розроблений для Яски. Це інструмент командного рядка, який генерує звіти в HTML,

CSV, XML,MySQL, SQLite та інші формати. Він вказаний як інструмент на

відомі Проект OWASPsecurity, а також у державному програмному забезпеченні

огляд інструментів на веб-сайті національної безпеки.

1.1.1.2 .Net

FxCop

FxCop це безкоштовний інструмент статичного аналізу коду від Microsoft, який

перевіряє збірки коду, керованих .NET, на відповідність Microsoft .NET

керівні принципи проектування. На відміну від інструмента програмування ворсу для C

мовою програмування, FxCop аналізує скомпільований об'єктний код, а не

оригінальний вихідний код. Для перевірки використовується аналіз CIL та аналіз callgraph

збірки для більш ніж 200 різних можливих порушень стандартів кодування в

у наступних областях:

- Правильність
- Дизайн бібліотеки
- Інтернаціоналізація та локалізація
- Конвенції про іменування
- Продуктивність
- Безпека

FxCop включає як графічний інтерфейс, так і версії командного рядка інструменту.

Microsoft Visual Studio 2005 і Visual Studio 2008 Team System

Обидві версії розробників містять функцію "Аналіз коду" на основі FxCop.

Для Visual Studio 2010 відповідний і трохи вдосконалений статичний код функції аналізу включені у видання Premium і Ultimate.

StyleCop

StyleCopє інструментом аналізу статичного коду з відкритим кодом від Microsoft, який

перевіряє код C # на відповідність рекомендованим стилям кодування StyleCop та

підмножина вказівок Microsoft .NET Framework Design Guidelines. Аналіз StyleCop

вихідний код, що дозволяє йому застосовувати інший набір правил від FxCop.

правила класифікуються на такі категорії:

- Документація
- Макет
- Ремонтопридатність
- Найменування
- Замовлення
- Читаність
- Інтервал

StyleCop включає як графічний інтерфейс, так і версії командного рядка інструменту. це є

можна створити нові правила для використання.

1.1.1.3 C / C ++

ВЗРИВ

Інструмент перевірки програмного забезпечення для лінивої абстракції Берклі (BLAST) є

азасіб перевірки моделей програмного забезпечення для програм C. Завдання, яке вирішує BLAST

полягає у необхідності перевірити, чи програмне забезпечення відповідає поведінковим вимогам його

пов'язані інтерфейси. БЛАСТ працюєлічильник на прикладі автоматичного

уточнення абстракції для побудови абстрактної моделі, яку потім перевіряють

для безпечних властивостей. Абстракція побудованана льоту, і лише до

на запитточність.

Кланг

Кланг є інтерфейсом компілятора для C, C ++, Objective-C та Objective-

Мови програмування на C ++. Він використовує віртуальну машину низького рівня (LLVM) як

її задній кінець, і тому Clang є частиною випусків LLVM, починаючи з LLVM 2.6.

Його мета - запропонувати заміну колекції GNU Compiler Collection (GCC).

Розробку фінансує Apple. Clang доступний під вільним програмним забезпеченням

ліцензія.

Проект Clang включає передній кінець Clang та статичний Clang

аналізатор серед інших.

Кокцинель

Кокцинель(Французьке слово сонечко) - це інструмент, що відповідає

і перетворити вихідний код програм, написаних у програмуванні

мова C. Coccinelle спочатку використовувалася для сприяння еволюції Linux; з

підтримка змін таких інтерфейсів програмування бібліотеки (API), як

як перейменування функції, додавання аргументу функції, значення якого якимось чином

контекстно-залежні та реорганізація структури даних. Інструмент знаходиться у вільному доступі

під відкритим джерелом ліцензії.

Вихідний код, який потрібно зіставити та / або замінити, визначається за допомогою шаблону

що дуже схоже на C (семантична мова патчів).

Frama-C

Frama-Cрозшифровується як Framework for Modular Analysis ofПрограми C.

Frama-C - це набір взаємодіючиханалізатори програм для програм C. Frama-C

був розробленийCommissariat à l'Énergie Atomique et aux Énergies

Альтернативи та Індія. Frama-C дозволяє аналізувати програми C без

виконуючи їх.

Frama-C можна використовувати для наступних цілей:

 щоб зрозуміти код C, який ви не писали. Зокрема, Frama-C дозволяє: спостерігати за набором значень, розбивати програму на коротші програми та орієнтуватися в програмі.

 для доведення формальних властивостей коду. Використання специфікацій, написаних мовою специфікації ANSI / ISO C, дозволяє забезпечити властивості коду для будь-якої можливої поведінки. Frama-C обробляє числа з плаваючою комою

 застосовувати стандарти кодування або конвенції коду вихідного коду C за допомогою спеціальних плагінів.

 інструментувати код C проти деяких недоліків безпеки.

Ворсинка

Ворсинка- це ім’я, спочатку дане конкретній програмі, яка позначена

дещо підозрілі та непереносні конструкції (ймовірно, це помилки) в C

вихідний код мови. Зараз цей термін узагальнено застосовується до інструментів, які позначають

підозріле використання програмного забезпечення, написаного будь-якимкомп'ютерна мова. Термін lint-

як поведінкаіноді застосовується до процесу позначення підозрілого

використання мови. Волосяні інструменти, як правило, працюютьстатичний аналіз вихідного коду.

До підозрілих конструкцій належать: змінні, що використовуються перед встановленням,

умови, які є постійними, і розрахунки, результат яких, ймовірно, буде поза

діапазон значень, який можна представити у використаному типі.

Багато форм аналізу, що виконуються інструментами, подібними до ворсинок, також є

виконується за рахунок оптимізації компіляторів, головна мотивація яких - генерувати

швидший код. Сучасні компілятори часто можуть виявити багато конструкцій

традиційно попереджається ворсом.

Автори інструментів, схожих на ворсинки, продовжували вдосконалювати асортимент

підозрілі конструкції, які вони виявляють. Сучасні інструменти виконують форми аналізу

що багато оптимізаторських компіляторів зазвичай не роблять, наприклад, крос-модуль

перевірка узгодженості, перевірка того, що код буде переносним для інших компіляторів,

та підтримка анотацій, що визначають передбачувану поведінку або властивості коду.

Розріджений

Розріджений- це інструмент, призначений для пошуку можливих помилок кодування в Linux

ядро. Цей інструмент статичного аналізу відрізнявся від інших таких інструментів тим, що був

спочатку розроблені для позначення конструкцій, які лише могли зацікавити

для розробників ядра, наприклад змішування покажчиків до адресного простору користувача та покажчиків на

адресний простір ядра.

Sparse містить вбудовані перевірки на наявність відомих проблемних і набір

анотації, призначені для передачі семантичної інформації про типи, наприклад, про що

вказівники на простір адреси вказують на те, на що блокує функція, яку набуває, або

Деякі перевірки, що виконуються Sparse, вимагають анотування джерела

код із використанням розширення __attribute__ GCC або Sparse

специфічний специфікатор __context__. Розріджений визначає наступний перелік атрибутів:

- адресу_простір (число)
- побітове
- сили
- контекст (вираз, in_context, out_context)

Шина

Шина, скорочене від Secure Programming Lint, - це програмування

інструмент для статичної перевірки програм C на наявність уразливостей та кодування

помилки. Раніше він називався LCLint, це сучасна версія інструменту Unix Lint.

Splint має можливість інтерпретувати спеціальні анотації до вихідного коду, який

дає йому сильнішу перевірку, ніж це можливо, просто подивившись на джерело

поодинці. Splint - це безкоштовне програмне забезпечення, випущене на умовах GNU General

Публічна ліцензія.

Статичний аналізатор Clang

Статичний аналізатор Clang - це інструмент аналізу вихідного коду, який виявляє помилки в програмах C та Objective-C.

В даний час його можна запустити як самостійний інструмент, так і в рамках Xcode.

Автономний інструмент викликається з командного рядка і призначений для запуску в тандемі зі збіркою кодової бази.

Аналізатор є 100% відкритим кодом і є частиною проекту Clang. Як і решта Clang, аналізатор реалізований як бібліотека C ++, яка може використовуватися іншими інструментами та програмами.

Cppcheck

Cppcheckєаналізатор статичного коду з відкритим кодом - -

інструмент для мов програмування C / C ++. Це універсальний інструмент, який може перевірити не-

стандартний код.

Cppcheck підтримує широкий спектр статичних перевірок, які можуть не охоплюватися

самим компілятором. Ці перевірки - це перевірки статичного аналізу, які можуть бути

виконується на рівні вихідного коду. Програма спрямована на статичний аналіз

перевірки. Деякі перевірки, які підтримуються, включають:

- Автоматична перевірка змінних Перевірка меж на перевищення масиву
- Перевірка занять. (наприклад, невикористані функції, ініціалізація змінних та дублювання пам'яті).
- Використання застарілих або замінених функцій відповідно до http://www.opengroup.org
- Перевірка безпеки на виняток, наприклад, використання виділення пам'яті та перевірка деструктора
- Витік пам'яті, наприклад через втрату обсягу без вивільнення
- Витоки ресурсів, наприклад через забуття закрити обробник файлів.
- Недійсне використання функцій та ідіом стандартної бібліотеки шаблонів
- Різні стилістичні та виконавські помилки

Контрольний стиль

Контрольний стильєзасіб статичного аналізу коду, що використовується при розробці програмного забезпечення для

перевірка, якщо Вихідний код Java відповідає правилам кодування.

стиль програмування, прийнятий проектом розробки програмного забезпечення може

допомагають виконувати належні практики програмування, що покращують код

якість, читабельність, повторне використання та зменшення витрат на розробку.

виконані перевірки в основному обмежуються презентацією та не аналізують

вмісту та не підтверджують правильність чи повноту програми. В

На практиці, це може бути вибагливим, щоб дотримуватися всіх стильових обмежень, деякі з них

що може зашкодити динаміці етапів програмування; так, може бути

корисно визначити, який рівень перевірки необхідний для певного типу програма.

Контрольний стильвизначає набір доступних модулів, кожен з яких забезпечує

перевірка правил із настроюваним рівнем суворості (обов'язковий, необов'язковий ...).

Кожне правило може викликати сповіщення, попередження та помилки.

Це дозволяє перевірити, наприклад:

- Коментарі Javadoc для класів, атрибутів та методів;
- Правила іменування атрибутів та методів;
- Обмеження кількості параметрів функції, довжини рядків;
- Наявність обов’язкових заголовків;
- Використання пакетів імпорту, класів, модифікаторів обсягу та блоків інструкцій;
- Пробіли між деякими символами;

- Передові практики побудови класів;
- Повторені розділи коду;
- Кілька вимірювань складності, серед яких вирази.

Контрольний стильвбудований у файл JAR, який може працювати у віртуальній машині Java або як завдання Apache Ant. Він також може інтегруватися в IDE або інші інструменти.

Плагін Checkstyle може надати нові функції, такі як:

- перевантажити забарвлення синтаксису або прикраси в редакторі коду;
- прикрасити дослідник проекту, щоб виділити проблемні ресурси;
- додати виходи попереджень та помилок до виходів.

Таким чином, розробник може безпосередньо отримати доступ до частин коду, виділених

Контрольний стиль.

1.1.1.4 Java

FindBugs

FindBugsєпрограма з відкритим кодом, створена Біллом П'ю та Девідом

Hovemeyer, який шукає помилок вКод Java. Для ідентифікації використовується статичний аналіз

сотні різних можливих типів помилок у Програми Java. FindBugs

діє на Байт-код Java, а не вихідний код. Програмне забезпечення поширюється

як самотній Графічний інтерфейс.

Хаммурапі

Хаммурапі- це інструмент статичного аналізу для виявлення потенційних проблем у

вихідний код та збір метрик. Архітектура інструменту дозволяє аналізувати

вихідні файли, написані різними мовами програмування. Станом на версію 5.6.0

підтримується лише Java. Є 88 нестандартних інспекторів Java.

Інструмент був названий на честь Хаммурапі, батька писаних законів.

Хаммурапі використовує шаблон відвідувача та механізм прямого ланцюгового висновку -

“Правила Хаммурапі” - для аналізу коду та повідомлення про потенційні проблеми та

метрики.

PMD

PMD- це статичний набір правилАналізатор вихідного коду Javaщо ідентифікує

потенційні проблеми, такі як:

 Можливі помилки- Порожні блоки спроб / зловити / нарешті / переключити.

 Мертвий код- Невикористанийлокальні змінні, параметри та приватні методи.

 Порожні оператори if / while.

 Надскладні вирази- Непотрібні оператори if для циклів, які можуть бути циклами while.

 Субоптимальний код- Використання марнотратного рядка / StringBuffer.

 Заняття з високимЦикломатична складністьвимірювання.

 Дублікат коду- Скопійований / вставлений код може означати скопійовані / вставлені помилки та зменшує ремонтопридатність.

Хоча офіційно PMD ні за що не виступає, у нього є кілька неофіційних

імена, найбільш підходящим, мабуть, є Програмувальний детектор помилок.

Сажа

Сажа- це система управління мовами та оптимізації, що складається

проміжних мов для мови програмування Java. Це було

розроблена дослідницькою групою Соболя з Університету Макгілла, відомого

його SableVM, віртуальна машина Java та компілятор AspectBench, відкритий

компілятор досліджень для AspectJ.

Сажа пропонує ряд проміжних зображень для використання обох

через свій API для інших програм аналізу для доступу та вдосконалення. Це

включають Jimple, спрощену версію вихідного коду Java, що містить максимум

три компоненти на оператор і Baf, майже представлення байт-коду.

Поточний випуск програмного забезпечення Soot також містить детальний аналіз, який може

використовуватися нестандартно, наприклад, контекстно-залежні потоки, нечутливі до потоку

аналізи, аналізи графіків та аналізи домінування (відповідаючи на запитання

Msgstr "Повинна відбутися подія b?").

Soot - це безкоштовне програмне забезпечення, доступне в рамках GNU Lesser General Public

Ліцензія (LGPL).

Сквер

Squale (покращення якості програмного забезпечення)- це платформа з відкритим кодом, яка

допомагає контролювати якість програмного забезпечення для багатомовних програм. Це в даний час

підтримує Java "з коробки", а також може аналізувати C / C ++ та код Cobol за допомогою

адаптер до інструменту McCabe. Squale поширюється на умовахLGPL v3

ліцензія.

Щоб допомогти командам розробників розібратися з якістю вашого програмного забезпечення

розробки, проект з підвищення якості програмного забезпечення з відкритим кодом (він же

Squale) зосереджується на двох основних аспектах:

- Працює на моделях підвищеної якості
- натхненний існуючими стандартами (ISO / IEC 9126) та підходами (GQM, McCall),
- підтверджено та вдосконалено відомими дослідниками, які є частиною команди Squale,
- врахування як технічних, так і економічних аспектів якості,
- Розробка програми з відкритим кодом, яка допомагає оцінити якість програмного забезпечення та вдосконалити його з часом
- на основі сторонніх технологій (комерційних або з відкритим кодом), які виробляють необроблену якісну інформацію (наприклад, метрики),
- використання моделей якості для агрегування цієї необробленої інформації у високоякісні фактори якості,
- Все це націлено на різні мови, включаючи Java, C / C ++, .NET, PHP, Cobol.

1.1.1.5 JavaScript

Компіляція закриття

Компілятор закриття - це інструмент для завантаження та запуску JavaScript

швидше. Це справжній компілятор для JavaScript. Замість компіляції з джерела

мова до машинного коду, він компілюється з JavaScript на кращий JavaScript. Це

аналізує ваш JavaScript, аналізує його, видаляє мертвий код і переписує і мінімізує те, що залишилось.

Він також перевіряє синтаксис, посилання на змінні та типи, і попереджає про

загальні підводні камені JavaScript.

Ви можете використовувати компілятор закриття як:

- Додаток Java з відкритим кодом, який можна запустити з командного рядка.
- Простий веб-додаток.
- API RESTful.

JSLint

JSLintєзасіб статичного аналізу коду, що використовується при розробці програмного забезпечення для

перевірка, якщоВихідний код JavaScript відповідає правилам кодування. Вона розроблена

відДуглас Крокфорд. Він надається в основному як Інтернет-інструмент, але є

також адаптації командного рядка.

РОЗДІЛ 2
МЕТОДИ ПОБУДОВИ АГРЕГАТОРІВ ДАНИХ БАНКІВ І ВАЛЮТНИХ БІРЖ

2.1.Засоби аналізу комерційного коду

2.1.2.1 Багатомовна

Люкс "Axivion Bauhaus"

Набір інструментів Bauhaus (або просто "інструмент Bauhaus") містить статику інструмент аналізу коду для коду C, C ++, C #, Java та Ada. Він включає різні такі аналізи, як перевірка архітектури, аналіз інтерфейсу та виявлення клонів.

Баухаус спочатку був похідним від старшої зворотної техніки Ріджі навколишнього середовища, яке було розширено Баугаузом через обмеження Рігі. це є серед найбільш помітних інструментів візуалізації в цій галузі.

Набір інструментів Bauhaus допомагає аналізувати вихідний код створення абстракцій (подань) коду на проміжній мові, як а також через графік потоку ресурсів (RFG). RFG - це ієрархічний графік з набраними вузлами та ребрами, які структуровані за різними поданнями.

ПЗ Black Duck

Програмне забезпечення Black Duck є приватною компанією в штаті Массачусетс, США. Компанія Duck Software стала піонером автоматизації змішаних програмних компонентів управління повторним використанням. Товари та послуги компанії дозволяють організаціям для аналізу складу вихідного коду програмного забезпечення та двійкових файлів, пошук багаторазовий код, керуваннясхвалення коду з відкритим кодом та сторонніми кодами, пов'язані з кодом змішаного походження, та контролювати пов'язану безпеку

CAST Application Intelligence Platform

CAST Application Intelligence Platform (AIP) є автоматизованою система вимірювання якості та розміру бізнес-додатків. Це зроблено CAST Inc., що базується у місті Медон, Франція. AIP перевіряє вихідний код, визначає та відстежує проблеми якості, а також надає дані для моніторингу продуктивність розвитку. Основне завдання рішення полягає в аналізі декількох рівнів і безліч технологій бізнес-додатків та вимірювання якості та дотримання архітектурних стандартів та стандартів кодування, одночасно забезпечуючи візуальність специфікаційні моделі. Менеджери отримують доступ до цієї інформації в режимі реального часу через веб-інтерфейс (панель керування програмами). Менеджери та розробники може виявити проблеми із заявкою до того, як заявка буде запущена у виробництво.

Покриття

Статичний аналіз покриття- це інструмент статичного аналізу коду для вихідних кодів C, C ++, C # та Java. Це комерційний продукт, який виникла як Stanford Checker, яка використовувала абстрактну інтерпретацію виявити дефекти у вихідному коді.

Найбільш помітним використанням Prevent є під керівництвом Департаменту США

Договір про національну безпеку, в якому він використовується для вивчення понад 150 відкритих вихідні програми для помилок. 6 березня 2007 року було оголошено, що понад 6000 виправлено помилки 53 проектів, виявлених скануванням.

DevPartner

DevPartner- це набір розроблених інструментів для розробки та тестування програмного забезпечення компанією Nu-Mega Technologies, придбаною корпорацією Compuware у 1997 р., яка 1 червня 2009 року продала його компанії Micro Focus International. Є два версії: одна для власних і .NET-додатків Windows, а інша для Java додатків.

DevPartner Studio Professional- це набір інструментів, що дозволяє розробнику

проаналізувати власний (некерований) та .NET (керований) код для:

- Якість і складність коду
- Виявлення витоків пам'яті
- Оптимізація пам'яті
- Аналіз ефективності (терміни)
- Експерт продуктивності (використання процесора, диска та мережі)
- Аналіз охоплення коду
- Моделювання несправностей (як .NET, так і середовище)
- Виявлення помилок та моніторинг взаємодії C / C ++ за допомогою технології BoundsChecker.

Кожен аналіз може бути налаштований на відображення деталей на рівні методу або лінії.

DevPartner Studio інтегрується з Visual Studio 6, 2003, 2005, 2008 та

2010, що забезпечує кнопки панелі інструментів та параметри меню для доступу до всіх інструментів. Всі

з інструментів також можна запускати з командного рядка, що дозволяє автоматизувати та

Постійні інтегровані процеси тестування, які слід налаштувати.

DevPartner Java Edition (DPJ)інтегрує набір функціональних можливостей

розробники для аналізу коду Java

- Якість і складність коду
- Виявлення витоків пам'яті
- Профілювання та оптимізація пам'яті.
- Профілювання та оптимізація продуктивності.
- Аналіз потоків та виявлення блокування.
- Аналіз охоплення коду.

DPJ може показувати графік викликів під час усунення несправностей, а також може викопувати

детально на рівні методу та рядка. DPJ інтегрується з Eclipses 3.2 / 3.3,

OptimalJ, JBuilder та IBM RAD 6.0 забезпечують меню та інструменти для доступу до всіх

його функціональність. Усі інструменти також можна запускати з командного рядка,

що забезпечує можливість автоматизації та безперервної інтеграції.

Набір інструментів для реінжинірингу програмного забезпечення DMS

Інструментарій реінжинірингу програмного забезпечення DMS - це власний набір

засобів перетворення програм, доступних для автоматизації користувацького джерела

аналіз програм, модифікація, переклад або генерація програмних систем для

довільні суміші мов-джерел для великомасштабних програмних систем.

DMS був використаний для реалізації широкого спектру практичних інструментів,

включати мови домену (наприклад, генерацію коду для заводського контролю),

тестувати засоби охоплення та профілювання, виявлення клонів, засоби міграції мови та

Реінжиніринг компонентів C ++.

Набір інструментів забезпечує засоби для визначення мовних граматик та волі

створюють парсери, які автоматично будують абстрактні дерева синтаксису (AST),

і симпатичні принтери для перетворення оригінальних або модифікованих AST назад на компілятивні

вихідний текст. Дерева синтаксичного аналізу захоплюють, і симпатичні друкарі відновлюються, завершують

подробиці про програму оригіналу, включаючи позицію джерела, коментарі,

radix та формат цифр тощо, щоб переконатись, що відтворений вихідний текст виглядає як

впізнаваний програмістом як оригінальний текст за модулем будь-якого застосованого

перетворення.

GrammaTech

GrammaTech- постачальник засобів розробки програмного забезпечення, що базується вІтака,

Нью-Йорк. Запропонована технологія базується на дослідженнях, проведених в Корнеллі

Університет та Університет Вісконсіна.

CodeSonar- це інструмент аналізу вихідного коду, який виконує цілу програму,

міжпроцедурний аналіз на C, C ++ та виявляє помилки програмування під час компіляції

час. CodeSonar використовується FDA, Центром приладів та радіології

Health, яке використовує його для виявлення дефектів медичних виробів, що виводяться на полю.

CodeSurferє інструментом для розуміння програм. Конструкції програми—

включаючи директиви препроцесора, макроси та шаблони C ++ - аналізуються.

CodeSurfer обчислює різноманітні подання, які можна дослідити

графічний інтерфейс користувача або доступ до нього за допомогою додаткового програмування

інтерфейс.

Ада-ЗАСТЕРЕЖЕНОта Ada-Utilities - це інструменти, що забезпечують послідовне кодування

стиль, запобігати синтаксичним помилкам та надавати функції для написання та

перегляд коду Ада. Їх можна використовувати з будь-яким компілятором Ada.

Imagix 4D

Imagix 4D- це інструмент аналізу вихідного коду від Imagix Corporation, що використовується в першу чергу для розуміння, документування та розвитку існуючих C, C ++ та Програмне забезпечення Java.

Застосовані технології включають повний семантичний аналіз джерел. Програмне забезпечення

візуалізація підтримує розуміння програми. На основі аналізу статичного потоку даних

перевірки виявляють проблеми у використанні змінних, взаємодії завдань

і паралельність. Показники програмного забезпечення вимірюють якість дизайну та визначають потенціал

питання тестування та обслуговування.

JustCode

JustCodeєрефакторинг та плагін продуктивності аналізу коду для Microsoft Visual Studio .NET 2005, 2008 та 2010.

JustCode використовується для повного аналізу коду та помилок на рівні рішення перевірка, навігація кодом та рефакторинг, а також модульне тестування для Структури тестування MSTest, XUnit, NUnit 2.5 та MbUnit 2.4. JustCode

опори C #, Visual Basic.NET, ASP.NET, XAML, JavaScript, HTML і

багатомовні рішення.

Klocwork

Klocwork Insight- це інструмент статичного аналізу коду, який використовується для ідентифікації

проблеми якості та безпеки для C, C ++, Java та C #. Продукт включає

численні настільні плагіни для розробників, інструмент аналізу архітектури та

метрики та звітування. Klocwork Insight Pro був представлений у листопаді 2009 року

і включає функції Klocwork Insight, а також інструмент для рецензування коду

andrefactoring особливості для розробників C / C ++.

LDRA Тестовий стенд

Ліверпульські дослідники даних (LDRA)є постачальником програмного забезпечення

інструменти аналізу, тестування та відстеження вимог для державного та приватного секторів

секторів та новатор у статичному та динамічному аналізі програмного забезпечення.

LDRA Testbed - власний інструмент аналізу програмного забезпечення, що забезпечує статику

аналіз коду, а також забезпечує аналіз покриття коду, код, якість та

огляди дизайну. Це комерційне впровадження тестування програмного забезпечення

створений Хеннелом в рамках його університетських досліджень. Це був перший рекламний ролик

продукт, що включає підтримку програмного забезпечення Linear Code Sequence і Jump

метод аналізу, який був результатом того самого дослідження. Він використовується в першу чергу

де програмне забезпечення повинно бути надійним, надійним і максимально без помилок,

такі як аерокосмічна електроніка (або авіаційна техніка), що має важливе значення для безпеки. Це також було

використовується для виявлення та усунення вразливих місць безпеки. LDRA - тестовий стенд

частина набору інструментів від LDRA, що включає:

- TBrun - автоматизований засіб модульного тестування
- TBreq - інструмент відстеження вимог
- TBmanager - розширює TBreq
- TBevolve - підтримує програмне управління базовим сценарієм
- TBsafe - для сертифікації DO-178B

- TBpublish - для публікації індексів HTML
- TBaudit - для звітів Microsoft Word
- DO-178B Tool Qualification

MALPAS

MALPAS- це набір програмних засобів, що забезпечує засоби дослідження та

доведення правильності програмного забезпечення шляхом застосування суворої форми статики

аналіз програм. Інструмент використовує спрямовані графіки та регулярну алгебру для представлення

програма, що аналізується. Використовуючи автоматизовані інструменти в MALPAS, аналітик

може описати структуру програми, класифікувати використання даних та

забезпечують інформаційні взаємозв'язки між вхідними та вихідними даними. Він також

підтримує офіційний доказ того, що код відповідає своїм специфікаціям.

MALPAS був використаний для підтвердження правильності безпеки

критично важливі програми в ядерній, аерокосмічній та оборонній промисловості. Це має

також використовувався для забезпечення перевірки компілятора в ядерній промисловості на Sizewell

B. Проаналізовані мови включають: Ada, C, PLM та Intel

Монтажник.

Унційні лабораторії

Унційні лабораторії(компанія IBM) - цеУолтем, штат Массачусетс

постачальник програмного забезпечення для захисту -. Компанія була заснована в2002 та передбачає а

продукт, що аналізує вихідний код програмного забезпечення для виявлення вразливостей системи безпеки в

Продукт призначений для допомоги розробникам, забезпечення якості та

аналітики безпеки виявляють та усувають уразливості програмного забезпечення.

сканування шукає низку вразливостей, які залишають програму відкритою для атаки.

Поліпростір

Поліпростірєінструмент статичного аналізу коду, натхненний трагедією

першого польоту Ariane 5, де помилка часу роботи призвела до знищення

ракета-носій. Це перший приклад широкомасштабного статичного аналізу коду

відабстрактне тлумачення для виявлення та доведення відсутності певного часу виконання

помилки ввихідний код для мов програмування C, C ++ та Ada.

Polyspace також перевіряє дотримання вихідного кодуMISRA C та інші пов’язані

стандарти коду.

Polyspace коментує вихідний код кольоровою схемою для позначення

статус кожного елемента в коді.

- Зелений означає надійний код
- Червоний вказує на несправний код, що спричиняє помилку під час виконання
- Сірий вказує на мертвий або недосяжний код
- Помаранчевий колір позначає недоведений код

Polyspace використовує для перевірки статичний аналіз коду на основі офіційних методів

виконання програми на мовному рівні. Інструмент перевіряє кожну інструкцію коду

беручи до уваги всі можливі значення кожної змінної в кожній точці

код. Цей процес перевірки забезпечує офіційну діагностику кожної операції

в коді. Polyspace перевіряє код при звичайному та ненормальному використанні

умови.

Рисунок 1.1 Результати поліпростору, коментовані у вихідному коді

ResourceMiner

ResourceMinerє комерційним засобом аналізу статичного коду для програмного забезпечення

архітектори та розробники. Він графічно візуалізує статичну структуру синглу

або декілька інтегрованих програм, написаних старше 30 років та сучасними мовами

і всі основні бази даних до деталей.

Структура коду відображається як залежність TopDown і BottomUp

дерева на різних рівнях абстракції; Система, Комп'ютер, Рівень, Пакет, Об'єкт,

Запис, заява та дані.

UI ResourceMiner дозволяє виконувати та документувати будь-які

аналітичні завдання на рівнях абстракції та сферах інтересів. Часто та / або

складні завдання аналізу можна автоматизувати за допомогою його API. Багато метрик

можливості. Може використовуватися для створення нового джерела з моделей у

База даних ResourceMiner.

Інспектор SOfCheck

Інспектор SofCheck - цеінструмент статичного аналізу для Java та Ada. Це

статично визначає та документує пере- та післяумови

зМетоди Java або підпрограми Ada і використовує цю інформацію для ідентифікації логіки

вади,умови перегонів та надлишковий код в окремому класі Java або Ada

пакет, підсистема або повна програма. Інспектор SofCheck є

випускається SofCheck, Inc., компанією, що займається програмним продуктом уБерлінгтон, Массачусетс.

Сотоаре / Сотограф

Сотоаркє комерційнимінструмент статичного аналізу коду для архітекторів програмного забезпечення.

Він графічно візуалізує статичну структуру написаних програмних систем

вJava, C # або в коді C ++. Структура коду відображається у вигляді ієрархій (дерев)

модулів, пакетів та файлів. До того ж користувач може описувати графічно

означає зазначену архітектуру програмного забезпечення програмної системи. Роблячи це

інструмент відразу порівнює цю задуману архітектуру з реалізованою

структуру коду та висвітлює всі порушення архітектури.

Syhunt Sancat

Сихунтє всесвітньою веб-мережею, що займається програмним забезпеченням для забезпечення безпеки

штаб-квартира в Ріо-де-Жанейро, Бразилія. Syhunt був заснований в серпні 2003 року

Феліпе Арагон, фахівець з мережевої безпеки. Діяльність компанії є

в даний час зосереджена на розробці програмного забезпечення, що стосується оцінки

веб-сервери та веб-програми.

У 2003 році Syhunt випустив програмне забезпечення для оцінки безпеки веб-додатків

відомий як Sandcat, який зосереджується на безпеці відкритих веб-додатків

Проект (OWASP) та вразливості Інституту SANS. Syhunt також

випустив ряд програмних засобів захисту, включаючи засоби для видалення хробаків

(під час спалахів хробаків), інструменти зміцнення сервера та журналу аналізу.

Зрозумійте

Зрозумійтеє комерційним програмним засобом для аналізу статичного коду від SciTools. Він в основному використовується для зворотного проектування, автоматичного документування та

розрахувати метрики коду для проектів з великими кодовими базами.

Розуміння робіт за допомогою інтегрованого середовища розробки (IDE) розроблений, щоб допомогти підтримувати та розуміти старий та новий код, використовуючи детальну перехресну інформацію

посилання та різноманітні графічні подання.

Зрозумійте синтаксичний аналіз вихідного коду мов програмування Ada (83,

95), K&R C, ANSI C і C ++, Delphi, Fortran (77, 90,

95), Java, JOVIAL, JavaScript, PHP, HTML, мова таблиць стилів CSS та

мова опису обладнання VHDL. Надано версії Understand

для операційних систем Windows і Unix, варіанти HP-UX,

SGI IRIX, Linux, Mac OS X та Solaris.

2.1.2.2 Чистий

CodeRush

CodeRushце плагін рефакторингу та продуктивності від DevExpress розширює нативну функціональність Microsoft Visual Studio .NET 2003, 2005,

2008 та 2010 роки.

CodeRush виконує статичний аналіз коду на рівні рішення (виявлення помилок увімкнено

муха, без необхідності компіляції), надає додаткові можливості для помилок виправлення, заповнення коду, навігація, пошук, виділення синтаксису,

форматування, генерація та оптимізація коду, здійснює понад 180

автоматизовані рефакторинги та раціоналізаторська одиниця

використання NUnit, XUnit, MbUnit та MSTest, серед інших функцій. Візуальна

Basic 2010 (VB 10.0) Остання версія, 2010.2, вийшла 01 грудня 2010 року.

Він підтримує C #, VB10, ASP.NET, HTML, JavaScript, XML, XAML та C ++.

Далі наводиться частковий перелік функцій і груп функцій CodeRush:

- 180 + рефакторинг коду
- 70+ постачальників коду
- 120+ випусків коду
- Найшвидший тест-бігун на ринку
- Підтримка декількох модульних модулів тестування NUnit, XUnit, MbUnit та MSTest
- Підтримка розширюваного модульного тестування для підтримки будь-якої основи тестування
- Інструменти візуалізації
- Додаткові засоби вибору
- Інструменти буфера обміну
- Інструменти навігації
- Шаблони коду
- Розширюваність CodeRush - велика спільнота плагінів

NDepend

NDependєінструмент статичного аналізу для керованого коду .NET. Цей інструмент

підтримує велику кількість метрик коду, дозволяє візуалізувати

використання залежностейспрямовані графіки та матриця залежностей. Інструменти також

виконує порівняння базових знімків коду та перевірку архітектурних та

правила якості. Правила, визначені користувачем, можуть бути написані спеціальною мовою

називається "Мова кодових запитів" (CQL). Ця мова дуже схожа наSQL,

отже, користувачі можуть запитувати основу коду так само, як і запит

ареляційна база даних. Інструмент також постачається з великою кількістю попередньо визначених

правила якості коду. Правила можна перевіряти автоматично протягомбезперервний

інтеграція.

Основними особливостями NDepend є:

 Візуалізація залежностей (за допомогою графіків залежностей та матриці залежностей)

 Програмні метрики (NDepend в даний час підтримує 82 метрики коду: Цикломатична складність; Аферентне та Ефективне зчеплення; RelationalCohesion; Рейтинг сторінок Google типів .NET; Відсоток коду, охопленого тестами тощо)

 Декларативний запит коду за допомогою CQL

 Інтеграція з CruiseControl та TeamCity

 Необов'язкові обмеження коду у вихідному коді з використанням атрибутів .NET

 Порівняння версій двох версій однієї і тієї ж збірки

ReSharper

ReSharper- це рефакторинг та розширення продуктивності JetBrains, що

розширює нативну функціональність Microsoft Visual Studio 2003, 2005, 2008 та 2010 рік.

2.1.2.4 C / C ++

Астрі

Астрі- це статичний аналізатор, заснований на абстрактній інтерпретації. Він аналізує

програми, написані на підмножині мови програмування C і виводять an

вичерпний перелік можливих помилок виконання та порушень тверджень.

Інструмент призначений для вбудованого коду, що відповідає безпеці: джерело

передбачається, що програми не містять динамічного розподілу (malloc); конкретні

методи аналізу використовуються для загальних конструкцій теорії управління (фільтри, швидкість

обмежувачі) та числа з плаваючою комою.

Astrée була розроблена в групі п-ра Патріка Кузо в École Normale

Supérieure, спільна група з CNRS, і продається компанією Absint

GmbH. Авіаційна та імітаційна техніка Airbus є однією з основних галузей промисловості

користувачів.

PC-Lint

ПК-ворсинкає комерційним програмним засобом для аналізу статичного коду, виробленим

Програмне забезпечення Gimpel для мов C / C ++.

PC-lint - це інструмент командного рядка для розробників, який вказує на підозрілі або

прості неправильні проблеми у вихідному коді. PC-lint (або його багатоплатформна версія,

FlexeLint) може бути інтегрована в IDE як зовнішній інструмент, а також формат

попереджувальні повідомлення можуть бути адаптовані до форми, яку IDE здатна розпізнавати та

процес.

PC-lint можна використовувати для забезпечення якості вихідного коду і перевірки

код відповідності керівним принципам кодування, таким як MISRA C або MISRA

C ++.

PVS-Studio

PVS-Studioє комерційним засобом статичного аналізу коду для C \ C ++ \ C ++ 0x

розроблена системою верифікації програм. PVS-Studio розроблений на

основа бібліотеки VivaCore з відкритим кодом, яка сама базується на

бібліотека OpenC ++.

Основною галуззю роботи аналізатора є міграція 32-розрядної Windows

додатки до 64-розрядної Windows. PVS-Studio включає наступну діагностику

модулі:

- Viva64 - діагностика 64-розрядних помилок.
- VivaMP - діагностика помилок, пов’язаних з OpenMP.
- Модуль загальної діагностики помилок.

PVS-Studio інтегрований в IDE Microsoft Visual Studio або може використовуватися

самостійно через інтерфейс командного рядка.

QA-C

QA-Cє комерційним програмним засобом для аналізу статичного коду, виробленим

Дослідження програмування для мови C, створене в 1986 році.

Це інструмент, який був використаний для вимірювання вихідного коду C.

подано в книзі "Безпечніше C" Лес Хаттона:

- Хаттон, Л., "Безпечніше C: Розробка програмного забезпечення в системах, що відповідають високій цілісності та критиці безпеки", McGraw-Hill (1995), ISBN 0-07-707640-0

QA-C може бути використаний для забезпечення якості вихідного коду C та перевірки

код відповідності керівним принципам кодування, таким як MISRA C. Інше

функціональність включає можливість обчислення метрик коду для проектів з великими

кодових баз.

Інструменти працюють через IDE, призначену для підтримки та підтримки

зрозуміти старий і новий код, використовуючи докладні перехресні посилання та різні

графічні види. Інструменти також можна використовувати в командному рядку. Графічна IDE може

викликати для відображення результату.

2.1.2.5 Java

Jtest

Jtestє автоматизованимПродукт для тестування Java та статичного аналізу коду

зроблено Parasoft. Він спрямований на підвищення надійності, функціональності, коду Java

безпека, продуктивність та ремонтопридатність. Базова функціональність включаєОдиниця

генерація тестових кейсів, статичний аналіз, регресійне тестування, виявлення помилок під час виконання,

іогляд коду. Jtest використовується такими компаніями, як Cisco

Systems, TransCore, AIG United Guaranty та Wipro Technologies.

SemmleCode

SemmleCodeє інструментом підвищення якості комп'ютерного програмного забезпечення. Це

можна використовувати для пошукупрограмування шаблонів помилок, для обчислення метрик програмного забезпечення та

примуситиконвенції кодування. Всі ці завдання можна сформулювати як запити в об’єктно-орієнтованиймова запитів з назвою .QL.

СонарJ

СонарJє комерційним інструментом для статичного аналізу коду написаного програмного забезпечення на Java. Аналізуючи скомпільовані класи та вихідний код, він створює файл залежність пам'яті та метрична модель аналізованого коду. Модель Потім залежності можна візуалізувати графічно, щоб користувач міг розуміти структуру системи. Крім того, інструмент дозволяє визначити моделі логічної архітектури (передбачувана структура програмного забезпечення) та відобразити її на код. Порівнюючи логічну модель із реальною

структурою залежностей SonarJ виявляє та перераховує всі порушення архітектури (відхилення від передбачуваного структура).

2.4. Висновки до розділу

Агрегатори валютних онлайн-обмінників собою являють зручний інструмент, істотно мінімізує часові витрати і сили користувачів. Вони збирають найактуальніші дані з працюючих сервісів конвертації валюта і надають цю важливу, оптимально сгруппированную інформацію відвідувачам.

Люди швидко використовують фільтри, щоб підшукати прийнятні варіанти обміну криптомонет. Майданчики також корисні трейдерам цифрових монет, а також користувачам, які заробляють на арбітражі валюта.

РОЗДІЛ 3
ПРОЕКТУВАННЯ СИСТЕМИ ОБРОБКИ ДАНИХ БАНКІВ І ВАЛЮТНИХ БІРЖ

Необхідність оцінки

За допомогою експерименту можна перевірити гіпотези про появу різних відходів у процесі розробки програмного забезпечення.

Такий експеримент можна проводити як in vitro під час контрольованого експерименту або як тестовий випадок у компанії, яка займається розробкою програмного забезпечення.

На додаток до своєї основної функції, такі експерименти можуть мати ще одну перевагу. Вони можуть покращити розуміння причин, що призводять до появи відходів, і дати змогу дослідникам зробити кількісну оцінку таких втрат.

Плани деяких таких експериментів представлені нижче. Ці експерименти були розроблені, щоб зосередити увагу на одному виді відходів розробки програмного забезпечення. Їх описи також були розроблені як самодостатня наукова робота.

Перемикання завдань

Вступ

Нерідка ситуація, коли компанія, що розробляє програмне забезпечення, має кілька активних проектів.

Наприклад, перше видання програми може бути на стадії її обслуговування, тоді як друге видання цієї ж програми розробляється на етапі написання коду тією ж компанією-розробником.

Це явище може бути широко поширеним серед компаній, що використовують гнучкі методології. Бажання використовувати солідну команду розробників на всіх етапах розробки, що характерно для даної методології, може призвести до того, що одні і ті ж люди будуть залучені до кількох проектів.

Існує два можливі способи організації роботи, якщо команда розробників працює одночасно над кількома проектами:

Реалізація всіх проектів одночасно, перемикання між ними через задані проміжки часу.

Реалізація проектів послідовно, один за одним.

Організація команди розробників програмного забезпечення, яка використовує перемикання завдань, критикується різними дослідниками[7] [18].

Зокрема, стверджується, що така організація роботи призводить до збільшення загального часу розробки проектів у порівнянні з випадком, коли проекти розробляються послідовно.

Іншими словами, стверджується, що сам процес переходу від одного проекту до іншого призводить до втрати часу та ресурсів.

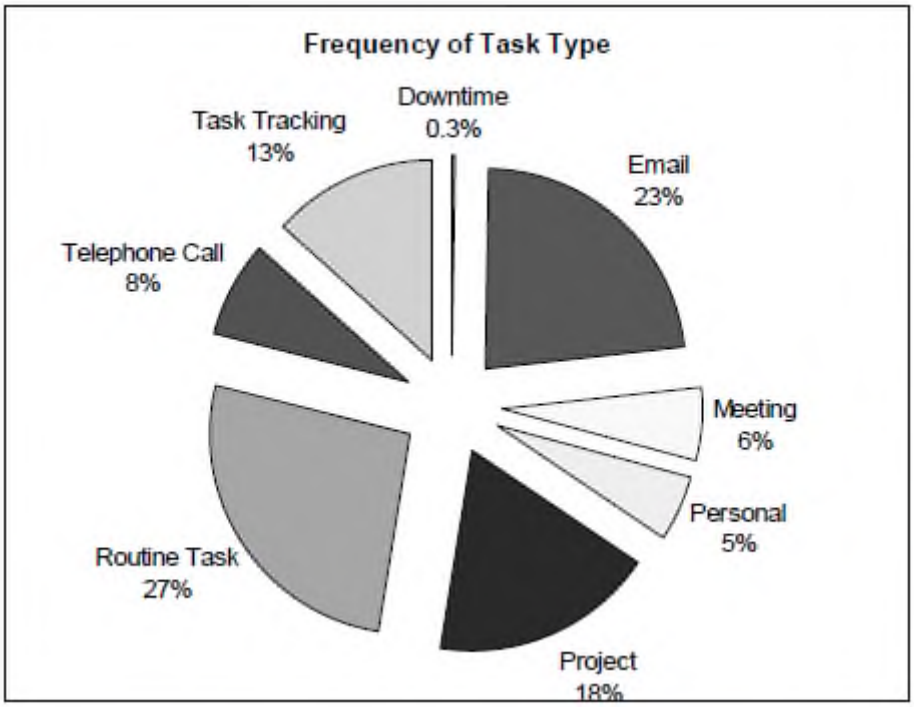
Передумови та відповідні роботи

Не було жодних доказів існування академічних дослідницьких робіт, що оцінювали б ефект переключення завдань у процесі розробки програмного забезпечення; однак подібні дослідження існують і в інших областях.

Різні дослідження в галузі когнітивних наук, такі як Перемикання завдань Стівена Монселла зосереджуються на нейропсихологічних аспектах перемикання між завданнями як окремої людини. Хоча такі дослідження можуть підтвердити існування змінних витрат[19], існує необхідність оцінити такі витрати в області розробки програмного забезпечення.

У щоденниковому дослідженні перемикання завдань та переривань, проведеному в Microsoft Research у формі контрольованого експерименту, явище перемикання завдань було досліджено протягом типового робочого дня офісного працівника [20].

Суб'єктам експерименту було запропоновано відмітити в електронній таблиці кожне нове завдання, на яке було переведено виконання. Їх також попросили оцінити складність такого перемиканнязавдання. Прикладами завдань були: ділові зустрічі, телефонні дзвінки, читання електронної пошти.



Малюнок 3.1 Частота перемикань завдань у дослідженні переключення завдань Microsoft Research [19].

Дослідження показало, що пересічний офісний працівник переходить від завдання до завдання приблизно 50 разів на тиждень.

Однак таке перемикання слід відрізняти від переключення проектів при розробці програмного забезпечення, оскільки в першому випадку працівники, можливо, щойно змінили робочий інструмент, яким користувались, тоді як у останньому розробники програмного забезпечення перемикаються з одного набору інформації на інший, іноді не переходячи на інший інструмент.

Дослідження не досліджувало час, витрачений на сам процес переключення завдань.

Хоча дослідження виявило сильну позитивну кореляцію між складністю проблеми та оцінкою працівником складності перемикання між такими завданнями.

Цілі експерименту

Метою цього експерименту є порівняння процесу послідовної роботи над кількома проектами та роботи з використанням перемикання проектів з метою оцінки

Відсутність або наявність втрат часу, викликаних перемиканням

Зв'язок кількості втраченого часу та складності проектів, що перемикаються між собою

в контексті етапу написання коду під час розробки програмного забезпечення.

Гіпотези

Розробляється програмний продукт характеризується значним обсягом інформації, як формальної, так і неформальної. Це включає код, специфікацію, рішення, прийняті під час розробки, план подальшого розвитку та багато іншого.

При переході від одного проекту до іншого частина такої інформації може бути втрачена, і програмісту доведеться витратити час на її відновлення.

Це становить гіпотезу H1: перехід від проекту до проекту вимагає часу, і, отже, розробка всіх проектів, використовуючи перемикання завдань, за інших рівних умов, закінчиться пізніше порівняно з розробкою тих самих проектів у послідовній справі.

Нульова гіпотеза H1 0 у цьому випадку полягає у твердженні, що перехід від проекту до іншого не матиме значного впливу на загальний час розробки проекту.

Обсяг робіт, необхідних для реалізації проекту, залежить від його складності [13], отже, можливо, існує взаємозв'язок між втратами від перемикання проектів та складністю проектів, що перемикаються між собою.

Гіпотеза H2: Існує позитивна кореляція між складністю завдань, що перемикаються, і втратами часу, що виникають внаслідок переключення між цими завданнями.

Гіпотеза H2 залежить від гіпотези H1, H1 повинна бути істинною, щоб оцінити гіпотезу H2.

Нульова гіпотеза H20: Не існує позитивної кореляції між складністю переключених завдань та втратами часу, що виникають внаслідок такого переключення.

Предмети

Студенти старших курсів, знайомі з мовою програмування, яка використовується в експерименті, можуть грати роль суб'єктів у цьому експерименті.

Рівень досвіду випробовуваних повинен бути визначений заздалегідь і повинен бути приблизно рівним, оскільки наявність значної різниці у досвіді випробовуваних та рівнях навичок є однією із загроз для обгрунтованості експерименту.

Бажано, щоб кількість учасників була кратною 2.

Дизайн експерименту

Схема експерименту - це двофакторний експеримент та кілька (принаймні трьох) процедур.

Факторами є:

Організація роботи, що використовується: послідовне виконання або перемикання між проектами.

Складність проектів.

План експерименту

Перед початком експерименту експериментатор має завдання підготовки завдань для випробовуваних.

Складність доручень буде різною, але вона повинна входити в окрему кількість категорій. Вони також не повинні сильно відрізнятися за складністю в межах категорії.

Першим кроком є визначення кількості категорій складності. Мінімальна кількість категорій - 3.

Наступний опис експерименту буде продовжено з використанням цієї кількості категорій складності завдання.

Ми позначаємо категорії таким чином: прості, помірні та складні.

Наприклад, проста категорія може мати завдання, виконання яких предметом має зайняти близько 1-3 годин. Прикладом такого завдання є написання тіла програмної функції.

Помірнийкатегорія отримає більш вимогливі завдання, такі як реалізація окремих функціональних можливостей програми. Для таких завдань може знадобитися півдня - день, який повинен закінчити учасник.

Завдання з жорсткої категорії повинні виконуватися приблизно протягом декількох днів і можуть складатися із завдань, що вимагають написання невеликої заявки.

Приклади завдань та категорії:

Важко: написання повнофункціонального текстового редактора ("Блокнот") з можливістю збереження тексту у файл та кількома іншими функціями.

Помірний: реалізація діалогового вікна «Зберегти файл» із вибором місця розташування та кодування файлу.

Простий: реалізація алгоритму, який зберігає рядок у файл.

Оцінка складності завдань виконується експертом, може виконуватися експериментатором.

Кількість завдань = кількість категорій * 2.

Експериментатор оцінює рівень підготовки предметів.

Рівень підготовки можна оцінити простим тестуванням, що оцінює знання мови програмування та знання технологій, що використовуються в експерименті.

Випробовуваних вказують про план експерименту.

Випробовувані об’єднуються в пари.

Пари призначаються зі складністю завдання.

Розподіл на пари та категорії складності завдань повинен ґрунтуватися на навичках учасників.

Це має призвести до рівного рівня підготовки учасників, як у парі, так і серед членів категорії складності завдання. Тоді предмети слід розподіляти випадковим чином.

Кожен предмет отримує два завдання (проекти) його категорії складності завдання.

Одна людина з пари виконує завдання в послідовній матерії, а інша перемикається між завданнями відповідно до заданого інтервалу.

Інтервал перемикання вибирається експериментатором на основі загальної тривалості виконання завдання. Бажано, щоб кількість таких перемикань було однаковою для кожної категорії складності.

Вибираючи тривалість інтервалу, слід будувати оцінку, виходячи з тривалості завдання із легкої категорії. Залежно від тривалості запропонованих простих завдань (наприклад, 1 година для роботи та 2 години для двох завдань), інтервал перемикання може становити півгодини.

Випробовувані виконують завдання.

Виконуючи завдання, вони повинні вести облік чистого часу, витраченого на виконання кожного завдання. Час, витрачений на перемикання між завданнями, не слід виключати з оцінки.

Після виконання завдань експериментатор оцінює готове завдання.

Оцінка якості завдань може бути виконана за допомогою офіційних тестів прийняття, які використовують метод чорної скриньки, або простою оцінкою експериментатора.

Завдання, які не відповідають певному мінімальному рівню якості, відкидаються.

Змінні та формальні гіпотези

Дві незалежні змінні експерименту: організація використаної роботи та складність завдання.

Залежною змінною є час доставки товару суб'єктом.

Час T (n) - чистий час, витрачений одним учасником пари на послідовне виконання завдань, де n - ступінь складності завдань.

Час T '(n) є Uчистий час, витрачений іншим членом пари на виконання тих самих завдань, але з використанням підходу перемикання між завданнями.

Таблиця 3.1

Змінні часу, витраченого на роботу над завданнями.

	Послідовна манера	Використання перемикання завдань
Простий	T (1)	T '(1)
Помірний	T (2)	T '(2)
Важко	T (3)	T '(3)

Формальна гіпотеза H1:

$$T'(n) > T(n).$$

Нульова гіпотеза H10:

$$T(n) \geq T'(n)$$

Якщо ця гіпотеза правильна, то переключення між завданнями вимагає часу. Різниця T '(n) - T (n) дозволяє кількісно визначити відставання, викликане перемиканням між завданнями.

Кількість втраченого часу на переключення порівняно із загальним часом розробки можна обчислити як:

$$LOSS(n) = \frac{T'(n) - T(n)}{T(n)} 100$$

ВТРАТА (n) вказує відсоток часу, витраченого на перемикання, порівняно із загальним часом виконання завдання.

Формальна гіпотеза H2:

$$LOSS(3) > LOSS(2) > LOSS(1)$$

Тобто, зі збільшенням складності завдання, що перемикається, час, необхідний для виконання такого перемикання, збільшується.

Нульова гіпотеза H20:

$$LOSS(3) \leq LOSS(2)$$

або

$$LOSS(2) \leq LOSS(1)$$

Щоб визначити, яка функція описує взаємозв'язок між складністю завдання, що перемикається, і втратами часу перемикання, результати можуть бути представлені у вигляді діаграми.

Залежна змінна на осі Y може означати ВТРАТУ (n) кожної пари учасників. Незалежна змінна - це складність завдань.

У цьому випадку складність завдань можна сміливо виразити як середню кількість рядків коду поставленого товару членами пари.

Використання рядків кодової метрики для вираження складності програми має свої недоліки, але в цьому випадку використання такої метрики виправдане.

Фредерік Брукс стверджує, що обсяг роботи (в людино-місяцях) збільшується як функція потужності до розміру програми (у рядках машинного коду) з показником, рівним 1,5 [13]

Можливо, взаємозв'язок між втратами часу перемикання та складністю проблем також описується степеневою функцією, оскільки причини, що лежать в основі явищ, подібні. В основі втрат часу лежать проблеми, пов'язані з обміном інформацією.

Загрози дійсності

Невідповідність процесу

Як і в будь-якому емпіричному дослідженні, важливо, щоб випробувані точно слідували плану експерименту та виконували його вимоги.

У цьому конкретному випадку випробовувані повинні зробити правильний розрахунок та відзначити час, витрачений під час виконання завдань.

Суб'єкти повинні бути поінформовані про правильний спосіб оцінки чистого обсягу витраченої роботи. Їм також слід повідомити, чому важливо дотримуватися плану експерименту.

Завдання, що виконуються нереально, експериментатор повинен перевірити.

Для покращення якості оцінок часу, отриманих від учасників, експериментатор може попросити учасників оцінити помилку, яку вони допустили, роблячи такі оцінки (наприклад, плюс-мінус півгодини).

Учасники також повинні бути проінформовані про конкретний спосіб того, як вони повинні організовувати процес розвитку.

Наприклад, учасники, які використовують концепцію розробленого тестуванням, можуть отримати вищу якість за рахунок написання тестів, але це, ймовірно, вплине на загальний час виконання завдання.

Це може спотворити результати, якщо інший учасник пари взагалі не проводив тестів.

Різниця в навичках предметів

Важливо, щоб перед дослідженням експериментатор мав повну інформацію про навички та рівні досвіду кожного з учасників експерименту.

Якщо до однієї пари програміст-початківець приєднається до програміста, який має певний практичний досвід, швидше за все, досвідчений програміст скоротить час виконання, навіть якщо він витратив час на переключення із завдання на завдання.

Експериментатор повинен оцінити вміння учасників перед експериментом. Тестування є найкращим варіантом оцінювання, оскільки воно є більш об’єктивним показником, ніж рецензування та, іноді, академічні оцінки студента-учасника.

Учасники однієї пари повинні мати приблизно однаковий професійний рівень.

Крім того, повинен бути приблизно однаковий рівень підготовки учасників, які виконують завдання різної категорії складності. Інакше рівень підготовки стане однією з незалежних змінних і спотворить результати експерименту.

Зовнішня дійсність

Якщо результатом експерименту є така нульова гіпотеза H10, це правда, це не обов'язково означають, що переключення між завданнями не займає багато часу.

Ймовірно, завдання, які були використані в цьому експерименті, були недостатньо складними, щоб виявити статистично значущий вплив на загальне виконання завдань.

Однак це не виключає існування втрат часу від переключення завдань на більший масштаб у виробничому середовищі, яке має проекти зі значно більшими обсягами коду.

Однак відсутність підтвердження гіпотези H1 у цьому випадку також є результатом.

Непрямим доказом є те, що перемикання між завданнями не робить такого значного впливу на час розробки програмного забезпечення.

У цьому випадку експеримент повинен бути повторений із більш складними завданнями, якщо це можливо.

Іншою загрозою для зовнішньої валідності є питання про те, наскільки обґрунтованим є узагальнення результатів студентської роботи для професіоналів, що працюють у галузі розробки програмного забезпечення.

Використання учнів в експерименті завжди є загрозою для зовнішньої валідності результату.

Тим не менше, дослідження Хьоста [21] дати підстави вважати, що, принаймні для певних завдань, студентів можна використовувати як адекватну модель професійної індустрії розробки програмного забезпечення.

Однією з цілей цього експерименту є визначення взаємозв'язку між складністю завдань та витратами на перемикання між ними, таким чином пошук відносної різниці, а не абсолютної.

Не слід використовувати абсолютні значення зі змінних, зібраних під час експерименту. Наприклад, невелика модифікація експерименту може дозволити знайти середню кількість хвилин, витрачених на переключення одного завдання.

Однак такими цифрами слід користуватися з обережністю. Рунесон[22], наприклад, повідомили про значну різницю в абсолютних показниках студентів першого курсу та аспірантів.

У зв’язку з цим залишається відкритим питання про те, чи абсолютна ефективність студентів у цьому експерименті є дійсним поданням для комп’ютерної індустрії загалом.

Робочий рух

Вступ

Під час розробки програмний продукт може сприймати думки багатьох команд.

Розробка специфікації, проектування системи, програмування, тестування та обслуговування можуть виконуватися різними групами людей.

Щонайменше, на багатьох підприємствах є загальна практика - мати принаймні два етапи розвитку: програмування та тестування.

Том і Мері Поппендік[15] стверджують, що передача артефактів розробки (специфікацій, дизайну, коду) від однієї групи до іншої є "величезним джерелом втрат при розробці програмного забезпечення".

Щоб зменшити цей ефект, навіть рекомендується розміщувати кожне місце працівників сцени в одній кімнаті, щоб кожен мав доступ до представників клієнта, розробників та тестувальників.

Вважається, що відходи під час робочого руху виникають через відсутність обміну інформацією.

Документи, передані від однієї групи розробників іншій, можуть не містити всієї необхідної інформації, зокрема, їм може бракувати багато неформальної інформації, наприклад, передбачувані рішення, відомі приховані проблеми та рішення, відхилені з якихось причин.

Наприклад, припущення, зроблені командою архітекторів систем, можуть не погоджуватися з припущеннями команди програмістів. Команда програмістів може використовувати вбудовані кодові рішення, про які команда підтримки не знатиме.

Окрім усього цього, робочий рух створює додаткову бюрократію в процесі розробки та викликає затримки тим, що новій команді потрібен час для обробки нової інформації.

Аргументи опонентів цих тверджень полягають у тому, що навіть якщо передача роботи може певною мірою зашкодити ефективності процесу розробки, це дозволяє відокремити експертів, методи та процеси від різних стадій.

Це, в свою чергу, дозволяє організації краще керувати процесом розвитку.

Розглянемо поширену ситуацію в багатьох організаціях - наявність спеціального відділу підтримки, який отримує програмне забезпечення після його розробки. Серед іншого, відділ підтримки займається виправленням дефектів, виявлених під час використання програмного забезпечення кінцевим користувачем.

Цей експеримент не має на меті відповісти на запитання, чи необхідно існувати окрема команда підтримки. Натомість метою експерименту є виявлення та оцінка збитків, що виникають внаслідок передачі роботи.

Оцінка втрат проводиться на основі роботи, що передається від програмістів на етапах кодування-перевірки-інтеграції програмістам на етапі обслуговування.

Передумови та відповідні роботи

Не знайдено жодної робочої роботи, яка безпосередньо перевіряла б та оцінювала втрати від перенесення робіт між етапами розробки.

Проте існують інші різні непрямі докази.

Передача роботи - це, по суті, передача інформації. Є дослідження, які оцінюють важливість обміну інформацією у розробці програмного забезпечення.

Наприклад, важливість обміну інформацією показана в дослідженні Білла Кертіса [23], яке проводилось у формі співбесіди з кваліфікованими розробниками.

Стверджується, що організаційні бар'єри є важливою причиною погіршення обміну інформацією.

Такі бар'єри часто ігноруються, оскільки вважалося, що артефакти, вироблені однією групою, містять всю інформацію, необхідну для наступної групи, що відповідає заявам Тома та Мері Поппендік.

Дизайнери скаржилися на необхідність постійного усного спілкування між замовником, вимогами та інженерними групами. Організаційні структури, що

розділяють інженерні групи (апаратне забезпечення, програмне забезпечення та системи), часто гальмують своєчасне спілкування про функціональність додатків в одному напрямку та зворотний зв'язок про проблеми реалізації, що виникають внаслідок проектування системи в іншому напрямку.

Ще одне непряме підтвердження того, що передача роботи призводить до втрат, було проведено дослідженнями, що оцінюють гнучкі методології.

Покидання офіційного поділу етапів, як у класичних моделях розробки програмного забезпечення та використання інтегрованої команди, є звичайною практикою для спритних методологій.

Піккарайнен [24] у дослідженні того, як перехід до Agile впливає на обмін інформацією при розробці програмного забезпечення, стверджується, що гнучка практика покращує обмін як офіційною, так і неформальною інформацією в команді.

Цей експеримент спрямований на тестування та оцінку збитків, які виникають внаслідок перенесення роботи між етапами розробки програмного забезпечення. А саме в цьому конкретному експерименті втрати, які виникають внаслідок переведення роботи на стадію технічного обслуговування.

Для досягнення цього пропонується організувати експеримент таким чином, щоб частина учасників працювала, використовуючи передачу роботи, а інша частина - ні.

Цілі експерименту

Метою експерименту є порівняння двох практик розробки програмного забезпечення, однієї, яка передбачає рух роботи, та іншої, яка цього не робить

з метою оцінки:

втрата часу, спричинена перенесенням роботи

погіршення якості продукції, спричинене передачею роботи

в контексті етапу обслуговування програмного продукту.

Гіпотези

Передбачається, що ефект від передачі роботи проявиться двояко.

Перш за все, час розробки товару повинен збільшуватися, якщо робота передається. Це гіпотеза H1.

Нульова гіпотеза H10 стверджує, що час розробки не повинен збільшуватися, навіть якщо робота перенесена.

Друга гіпотеза H2 полягає в наступному: якість продукції погіршується в процесі розробки, що передбачає перенесення роботи.

Предмети

Студенти старших курсів, знайомі з мовою програмування, яка використовується в експерименті, можуть грати роль суб'єктів у цьому експерименті.

Дизайн експерименту

В експерименті використовується стандартна схема експерименту з одним фактором.

Кількість рівнів два, і вони відповідають двом способам організації розробки програмного забезпечення, що використовуються в експерименті: одному з перенесенням роботи (експериментальна група), а іншому без нього (контрольна група).

Рівень підготовки учасників у кожній групі повинен бути приблизно однаковим.

План експерименту

План полягає в оцінці ефекту перенесення роботи на межі двох етапів: імплантація продукту (написання коду) та технічне обслуговування (підтримка).

Запропоновано змоделювати процес передачі роботи, який відбувається в деяких організаціях, що мають окремий відділ підтримки продукції.

Перед початком експерименту експериментатор створює специфікації продукту для суб'єктів для реалізації.

Було б достатньо одного завдання для всіх учасників. Рекомендується, щоб передбачувана тривалість виконання завдання становила 1-2 дні.

Рекомендується формувати специфікацію у вигляді дискретних заяв про товар, так званих історій користувачів. Наприклад, "Я, як користувач, хочу мати можливість зберегти текст у файл" тощо.

Така чітка форма вираження вимог до товару є важливою, оскільки ці твердження будуть використовуватися для оцінки якості товару на пізніх етапах експерименту.

Окрім основних специфікацій, експериментатор повинен розробити ряд невеликих змін до специфікації. Ці зміни повинні стосуватися конкретних функціональних можливостей програми та бути менш трудомісткими для впровадження (до 1 дня для поточної схеми).

Також експериментатор повинен розробити приймальні тести на основі історій користувачів з основних та додаткових специфікацій. Ці тести будуть використовуватися експериментатором для перевірки якості кінцевого продукту, тому підготовка таких тестів є ключовою частиною всього експерименту.

Умови тестування суб’єктам експерименту не розкриваються, тому приймальні тести працюють як чорна скринька.

Експериментатор оцінює рівень підготовки випробовуваних.

Рівень підготовки можна оцінити за допомогою тестів з мови програмування, що використовуються в експерименті, та / або інших використовуваних технологій.

Суб'єкти експерименту отримують технічні характеристики продукту

Контрольна група починає впроваджувати специфікацію

Експериментальна група також має доступ до специфікації, але групі заборонено починати над нею працювати.

Коли учасник контрольної групи закінчує своє завдання, експериментатор виконує тестування та перевірку продукту.

Виявлені дефекти надсилаються учаснику експерименту для усунення. Кількість дефектів тепер має реєструвати експериментатор. Учасник у цей момент повинен відстежувати кількість чистого часу, який він витрачає на виправлення дефектів.

У той же час інший член пари, учасник експериментальної групи, отримує код товару, написаний випробуваним з контрольної групи, і починає виправляти ті самі дефекти. Він також враховує час, витрачений на виправлення програмного забезпечення.

Якщо учасник контрольної групи написав код ідеально або виявлені дефекти незначні, експериментатор може внести зміни до специфікації програми, і обидва члени пари повинні внести ці зміни.

Коли обидві сторони закінчать свою роботу, експериментатор знову отримує продукт і перевіряє його за допомогою приймально-здавальних випробувань із зазначенням кількості виявлених нових дефектів.

Змінні та формальні гіпотези

Можна виділити дві групи змінних за умовами експерименту.

По-перше, є змінні, що оцінюють час, який кожен учасник витратив на роботу.

Дозволяти T - чистий час, витрачений учасником контрольної групи на внесення змін до програмного продукту.

Тоді T '- це чистий час, витрачений іншим учасником пари, тобто учасником експериментальної групи, на виправлення проблем.

Це час учасників попросили відстежити.

Інша група змінних - це змінні, що оцінюють якість товару.

D - це оцінка дефектів, виявлених після того, як учасник контрольної групи вніс зміни до продукту в цій ітерації.

D ' - це оцінка дефектів, виявлених після того, як учасник експериментальної групи вніс зміни у виріб у цьому тесті.

Дефекти можуть бути різного характеру (деякі можуть бути критичними і впливати на весь додаток, а інші можуть бути незначними і не становити невеликої загрози для користувальницького досвіду). Це слід враховувати при оцінці якості програми.

Пропонується використовувати шкалу серйозності дефектів:

Критичний дефект. 5 балів. Неможливо скористатися функцією програми.

Серйозний дефект. 3 бали. Користуватися цією функцією можливо, але важко.

Незначний дефект. 1 бал. Дефект майже не впливає на користування функцією програми користувачем.

Кожен виявлений дефект оцінює відповідну кількість балів, залежно від його серйозності.

Використовуючи наведені вище змінні, гіпотезу можна висловити формально вже зараз.

Гіпотеза H1:

$$T' > T$$

Учасник, який зазнав робочого руху, збільшив загальний час розвитку.

Коефіцієнт

$$\frac{T' - T}{T} 100$$

дає можливість оцінити часові втрати.

Нульовою гіпотезою H10 є:

$$T' \leq T$$

Гіпотеза H2 полягає в припущенні, що якість кінцевого продукту, який переніс роботу, буде гіршою.

Це:

$$D' > D$$

Поясненням затримок та дефектів може бути наступне: новим розробникам потрібен час, щоб зрозуміти частини вже розробленого продукту.

Відсутність знань про приховані аспекти роботи має призвести до зниження якості продукції. Видимий прояв дефекту кінцевого продукту може бути лише симптомом деяких складних систематичних проблем.

Що ще важливіше, подальша розробка продукту може виконуватися з наявною помилкою в коді. Виправлення однієї помилки в цьому випадку може призвести до появи нових помилок у залежних областях коду.

Загрози дійсності

Специфіка дослідження

Цей експеримент досліджує втрати, які виникають внаслідок переходу роботи від етапу програмування до етапу підтримки програми.

Фактори, що лежать в основі передачі, є універсальними. Це труднощі спілкування, які виникають між розробниками.

Отже, результати цього дослідження можуть бути певною мірою застосовані до передачі роботи на інших стадіях розвитку, таких як переміщення робіт від етапу специфікації до проектування.

Однак такі докази є непрямими.

Інші стадії розвитку можуть мати свої особливості, тому не слід, наприклад, покладатися на припущення, що затримка часу роботи буде такою ж, як у цьому експерименті.

Цей експеримент може лише опосередковано засвідчити факт того, що така затримка відбудеться.

Невідповідність процесу

Успіх експерименту залежить від правильної поведінки експериментатора та учасників.

Учасники повинні вести належний облік чистого часу, проведеного під час редагування програмного забезпечення.

Правильна підготовка експериментаторів та розробка правильних приймальних тестів є критично важливими.

Випробовувані повинні бути чітко поінформовані про важливість належного відстеження часу. Вони також повинні бути поінформовані про те, як це правильно робити - який час враховувати, а що ні.

Використання автоматизованих рішень може допомогти вирішити проблему.

Іншим можливим рішенням є проведення експерименту під особистим наглядом експериментатора. Це може бути можливим під час звичайного університетського лабораторного заняття, якщо зміни, які мають бути впроваджені, можуть бути внесені протягом дня.

Варіативні процеси розробки програмного забезпечення

Якщо хтось із членів пари використовує автоматичне тестування (до або після написання коду), це вплине на якість коду та час, витрачений на проведення таких редагувань.

Наприклад, прийняття тестової розробки може призвести до меншої кількості дефектів, але також можливо, що в деяких умовах це може подовжити час розробки.

Якщо інший член пари взагалі не проводить тестування, експериментальні дані можуть бути спотворені.

Тому важливим завданням експериментатора є організація процесу розробки програмного забезпечення для всіх учасників та переконання, що вони використовують однакову методологію.

Подальші дослідження

Мінімальною та достатньою умовою істинності гіпотез H1 та H2 є наявність втрат після внесення змін для учасника, який використовував переказ роботи.

Однак план експерименту також може бути використаний для оцінки того, як швидко зменшаться такі втрати від робочого руху.

Зробити цей поточний експеримент можна відносно просто.

Після того, як учасники внесуть зміни в продукт, експериментатор може знову змінити специфікацію продукту і попросити учасників внести ці зміни. Цей процес внесення змін до специфікації можна повторити n тестами.

Кожного разу учасники повинні відстежувати точний час, який вони витрачають на завдання. Експериментатор повинен вести записи про кількість та якість виявлених дефектів.

Ця схема надасть можливість оцінити швидкість адаптації нового розробника до програмного продукту, що знаходиться під розробкою, після події робочого руху.

Можливо, втрати часу та якості з часом зменшаться, оскільки новий розробник продовжить працювати з тією ж інформацією (продуктом), і ніякого нового перенесення роботи не відбудеться.

Дефекти

Вступ

Існування дефектів кінцевого продукту є дуже небажаним явищем. Це впливає на взаємодію з користувачем під час використання програми, що, в свою чергу, погіршує репутацію компанії та зменшує її прибуток. Певні помилки програмного забезпечення можуть призвести до втрати дорогого обладнання та навіть втрати життя.

Поки неможливо позбутися всіх дефектів, різні методики можуть мати різний час виявлення дефектів.

Типовою практикою розробки програмного забезпечення є наявність окремого відділу забезпечення якості (QA), який має завдання виявити можливі дефекти кінцевого програмного продукту.

Дефекти, виявлені в продукті на етапі перевірки, повертаються програмістам для усунення. Цей цикл повторюється, доки виріб не виявиться готовим до випуску.

Модель водоспаду має окремий етап перевірки, призначений для пошуку дефектів виробу, що розробляється.

Однак наявність окремого етапу тестування критикується концепцією ощадливого розвитку програмного забезпечення[7].

Стверджується, що перевірка якості після написання коду призводить до нашарування помилок, а також додає затримку в процесі розробки. Наявність окремої команди тестувальників стверджується, що зменшує мотивацію програмістів виправляти помилки на місці.

Загальна ідея полягає в тому, що чим пізніше виявляються дефекти у виробі, тим більше витрат необхідно виправити. Ось чому правильна організація процесу розробки повинна бути спрямована на якнайшвидше виявлення дефектів.

Як рішення цієї проблеми пропонується концепція тестової розробки або TDD. Основна ідея підходу полягає в написанні автоматизованих тестів до того, як буде написаний власне код.

Ці тести постійно проводяться розробником, поки він пише код.

Коли код є і всі тести проходять, функція програми вважається реалізованою.

Вважається, що такий підхід дозволяє раніше виявляти та виправляти помилки.

Передумови та супутні роботи

Тестова розробка також відома як тест-перший підхід від екстремального програмування.

У попередньому експерименті можна було оцінити втрати, які виникають внаслідок передачі продукту для випробування в іншому відділі.

Існує кілька досліджень, присвячених оцінці ефективності підходу, розробленого на основі тестів.

Таблиця 3.2

Короткий зміст попередніх робіт, що оцінюють ефективність тестового розвитку.

Вивчення	Дизайн	Предмети	Вплив TDD на якість	Вплив TDD на продуктивність кодування
М. Мюллер та О. Хагнер [25]	Контрольований експеримент	Студенти (19 учасників)	Без змін.	Без змін
Е. М. Максимільєнта Л. Вільямс [26]	Приклад	IT-фахівці	TDD краще.	TDD трохи гірший.
Бобі Джордж і Лорі Вільямс [27]	Контрольований експеримент	IT-фахівці (12 учасників)	TDD краще.	TDD гірше.

Едвардс [28]	Контрольований експеримент	Студенти (59 учасників)	TDD краще.	Не оцінюється.
Ердогм, Морісіо, Торчіано[29]	Контрольований експеримент	Студенти (24 учасника)	Без змін.	TDD краще.

Результати Мюллера та Хагнера занадто неоднозначні, щоб чітко судити про переваги або недоліки TDD, але автори відзначають, що було покращено розуміння програми для тих учасників, які використовували TDD.

Максимілієн та Вільямс оцінили прийняття концепції TDD до відділу IBM. Вони виявили покращення якості продукту (покращення на 40-45%), але продуктивність програмістів була трохи нижчою, ніж очікувалося.

В офіційному експерименті Джорджа та Вільямса використовувались професійні програмісти на тесті. Експеримент показав підвищення якості продукції на 18%, тоді як продуктивність впала на 14%.

Експерименти Ердогмуса, Морісіо, Торчіано не виявили підвищення якості програмного забезпечення, але виявили певний приріст продуктивності студентів.

Це та деякі інші дослідження відзначають інші позитивні ефекти від використання TDD, за винятком підвищення якості. Наприклад, використання TDD дозволило краще зрозуміти код завдяки декомпозиції завдань.

Всі експерименти на TDD різняться в деталях. Наприклад, в експерименті Мюллера та Хагнера учасники отримували заглушки методів і повинні були відновлювати тіла методу.

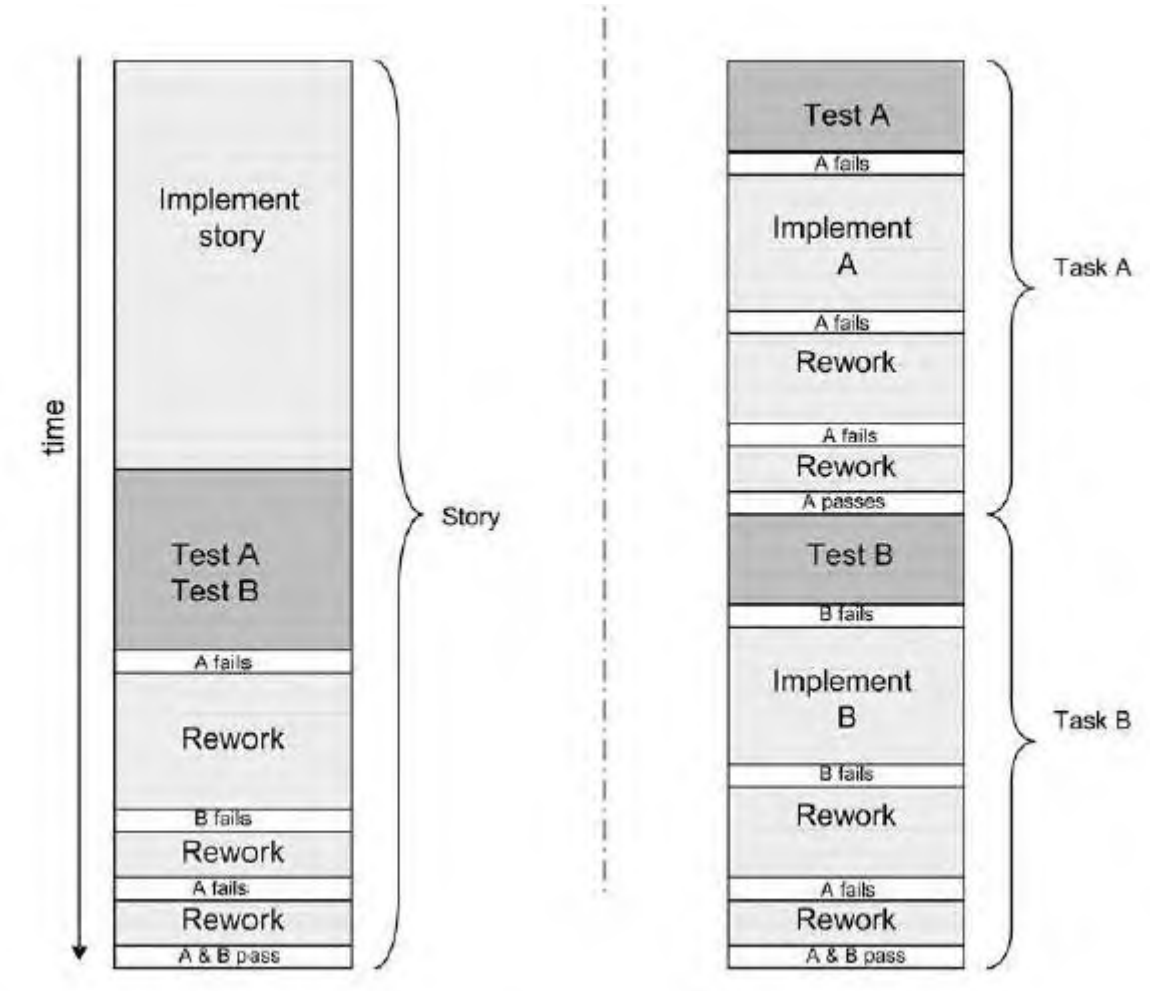
У експерименті Джорджа та Вільямса для учасників контрольної групи не було обов'язково писати тести, тому деякі учасники не застосовували жодного автоматизованого тестування.

Цілі експерименту

Як видно з попередніх досліджень, використання тестової розробки зазвичай позитивно впливає на якість програми (вона була кращою в 3 з 5 випробувань, а в інших - ніяких змін).

Однак вплив TDD на продуктивність розробників не є ясним.

Деякі дослідники вважають це TDD погіршує абсолютну продуктивність кодування, виміряну рядками коду, тоді як інші впевнені, що використання TDD може навіть прискорити процес розробки, покращивши розуміння коду.



Малюнок 3.2 Порівняння процесів останнього та першого тестування в експерименті Ердогмуса [29].

Типовим планом експерименту для оцінки TDD було наступне:

1. Учасники отримали тестові завдання.

2. Половина учасників виконує завдання за допомогою TDD, а інша половина - без нього.

3. Вимірюється ефективність кодування та якість кінцевої програми.

Недоліком такої схеми є те, що учасникам експерименту не доводиться виправляти дефекти, виявлені експериментатором.

Отже, експерименти не вимірювали загальну загальну ефективність процесу розробки, що використовує TDD; натомість вони виміряли місцевий програміст продуктивності написання коду.

Хоча така безпосередня ефективність насправді може бути нижчою в процесі розробки, що використовує TDD, загальний ефект використання TDD на процес розробки може бути позитивним.

Це насправді може призвести до підвищення продуктивності.

Продуктивність може покращитися за рахунок економії часу, витраченого на передачу продукту до відділу контролю якості, тестування його там і передачу роботи назад програмістам.

Запропоновано ще один експеримент для оцінки впливу використання TDD на весь процес розробки програмного забезпечення.

Мета цього експерименту полягає у порівнянні тестової розробки та тестування після завершення розробки (Тест Останній підхід) для того, щоб перевірити, чи зменшується загальний час розробки при використанні TDD.

Гіпотези

Єдина гіпотеза H з Експеримент полягає в тому, що загальний час розробки продукту буде меншим при використанні розробленої за допомогою тестування.

Передбачається, що в інших експериментах група, яка не використовувала TDD, виявилася більш продуктивною за рахунок прихованого перенесення частини роботи на фазу тестування, що не було оцінено в дослідженні.

Нульова гіпотеза H0 полягає в наступному: використання тестової розробки не зменшує загальний час розробки продукту.

Предмети

Студенти старших курсів, знайомі з мовою програмування, яка використовується в експерименті, можуть грати роль суб'єктів у цьому експерименті.

Дизайн експерименту

Цей експеримент являє собою простий однофакторний експеримент. Єдиним фактором тут є спосіб організації процесу розвитку.

Учасники будуть розділені на дві рівні групи: одна, яка використовує TDD (експериментальна група), і інша, яка цього не робить (контрольна група).

Загальний рівень навичок між групами повинен бути приблизно однаковим, тому на початку експерименту необхідно оцінити підготовку кожного учасника.

План експерименту

Перший крок - для експериментатора. Він повинен розробити специфікацію продукт, що підлягає розробці.

Специфікація створюється у вигляді окремих заяв про товар (історії користувачів). Такі твердження легко перевірити пізніше, щоб оцінити якість кінцевої продукції.

Друге завдання для експериментатора - це розробити приймальні тести для кінцевого продукту. У плані експерименту такі приймальні тести відіграють ключову роль, отже, успіх всього дослідження залежить від якості цих тестів.

Пізніше експериментатор оцінює рівень підготовки учасників.

Рівень підготовки випробовуваних можна оцінити за допомогою тестування на мові програмування, що використовується в експерименті.

Суб'єкти проходять інструктаж.

Випробувані поділяються на групи.

Поділ учасників на групи проводиться з урахуванням рівня кваліфікації учасників, щоб групи знаходились приблизно на одному рівні підготовки.

Суб'єкти експерименту отримують специфікацію товару і починають працювати над ним.

Учасники експерименту повинні відстежувати чистий час, який вони витрачають на розробку продукції.

Експериментальна група слідкує за процесом розробки з TDD, який може бути представлений наступним алгоритмом:

Виберіть історію користувача для реалізації.

Написати тест, який перевіряє дану історію користувача.

Продовжуйте писати код для реалізації історії користувача, доки весь продукт не пройде всі автоматизовані тести.

Продовжуйте переробляти код і запускати тести, поки всі тести не будуть успішно завершені.

Процес повторюється для кожної нової історії користувача.

Контрольна група використовує стандартний процес розробки та створює тести лише після реалізації функції програми.

Тестування повинно бути обов'язковим для суб'єктів контрольної групи.

Експериментатор приймає і тестує кінцевий продукт.

Якщо програмний продукт має дефекти, експериментатор повертає назад розробника для виправлення. Він інформує розробника про історію користувача, яка була реалізована неправильно.

Учасники не повинні заздалегідь знати, що експериментатор поводитиметься так; інакше вони можуть знайти мало мотивації для тестування та виправлення помилок у додатку самостійно.

Піддослідні виправляють дефекти.

Вони також відстежують час, витрачений на виправлення дефектів.

Експериментатор ще раз перевіряє програмний продукт

Процес повторюють, поки з кінцевого продукту не будуть усунені всі дефекти.

Змінні та формальні гіпотези

Змінні цього експерименту такі:

Час T - загальна кількість часу, яку член контрольної групи витрачає на розробку та виправлення помилок продукту.

Час T '- це загальна кількість часу, яку член експериментальної групи витрачає на розробку та виправлення помилок продукту.

Часи F та F 'відносяться до кількості часу, необхідного для внесення поправок до продукту відповідно членами контрольної та експериментальної груп.

У цьому випадку гіпотезу H можна представити наступним чином:

$$T > T'$$

Очікується, що є виграш у часі, який може статися через зменшення часу, витраченого на виправлення дефектів, тобто:

$$F > F'$$

Нульова гіпотеза H0:

$$T \leq T'$$

Загрози дійсності

Відсутність робочого руху

Цей експеримент не враховує час, необхідний для перенесення роботи між командами розробників. У реальній ситуації відділ контролю якості витратить трохи на ретельне тестування.

У цьому експерименті експериментатор тестує проект відразу, тому програміст, який писав код, не відчуває затримок.

Тому фактичний час перебування у виробничому середовищі буде навіть більшим, ніж у цьому дослідженні.

Ось чому цей експеримент не має на меті оцінити абсолютну суму збитків, які можуть виникнути. Метою цього експерименту є лише дати відповідь на питання, чи зменшує використання TDD загальний час розробки.

Невідповідність процесу

Успіх експерименту залежить від того, наскільки добре експериментатор та учасники виконують свої обов'язки.

Важливо, щоб кожен учасник точно дотримувався плану розробки, а саме під час процесу створення та виконання тестів.

Є кілька способів досягти цього. Перш за все, учасники повинні усвідомлювати важливість дотримання протоколу експерименту.

Інший метод був використаний в експерименті Ердогмуса, Морісіо та Торчіано [29] .

Наприкінці експерименту автори використовували анонімну анкету, просячи суб'єкта повідомити, чи чесно вони виконували експериментальні обов'язки та правильно відстежували час. Результати "ні" відповідь були усунені.

Висновки

Деякі відходи, виділені у розділі 2, були експериментально перевірені різними програмними дослідженнями, інші - ні.

Хоча опис деяких відходів та пошкоджень, які вони наносять, є інтуїтивно переконливим і сприймається як істинне багатьма авторами, які їх описують, багато з описаних відходів ніколи не отримували експериментальної перевірки.

Крім того, деякі експериментальні випробування можуть допомогти кількісно визначити шкоду, заподіяну відходами, а також дати краще розуміння причин, що лежать в основі їх виникнення.

Були запропоновані та офіційно описані плани експериментів. Ці експерименти можуть бути використані в освітньому середовищі для тестування та оцінки певних відходів.

Експеримент з перемиканням завдань призначений для оцінки втрат часу, що виникають внаслідок процесу розробки програмного забезпечення, що використовує перемикання завдань. Інша мета експерименту - знайти взаємозв'язок між такими втратами та складністю виконуваних завдань.

Експеримент з робочим рухом має на меті перевірити твердження, що наявність робочого руху від однієї команди розробників до іншої веде до затримок у випуску продукції та знижує якість продукції.

Основною метою експерименту з дефектами є вдосконалення великої кількості інших експериментів, що оцінюють тестову розробку з метою вирішення питань щодо продуктивності розробки програмного забезпечення з використанням TDD.

Хоча завдання попередніх експериментів було оцінити швидкість розвитку, вони робили це, оцінюючи місцеві показники на одному етапі розвитку, а саме впровадження.

Цей експеримент має на меті оцінити вплив тестової розробки на загальну ефективність всього процесу розробки.

ВИСНОВКИ

Ця робота вивчає процес розробки програмного забезпечення з використанням екологічного підходу.

Перш за все, досліджуються концепція сталого розвитку та пов’язані з ними концепції сталого підприємництва, зелених IT та інших.

Було відзначено, що концепція сталого розвитку може бути застосована до будь-якої організаційної структури, тому таким чином концепція сталого розвитку може використовуватися разом із розробкою програмного забезпечення.

Цей документ широко використовує концепцію сталого розвитку для подальших досліджень. Проаналізовано три аспекти сталого розвитку: соціальний, економічний та екологічний.

Надано короткий огляд розробки програмного забезпечення як інформаційної технології.

Доступно багато методологій розробки програмного забезпечення. Метою кожного з них є вдосконалення процесу розробки програмного забезпечення. Деякі методології зосереджуються на одному аспекті розвитку, а інші - на іншому.

Обговорювались нові модні гнучкі методології. Зокрема, концепція Lean IT приділяла велику увагу завдяки концепції розробки програмного забезпечення.

Сформульовано та описано концепцію відходів розробки програмного забезпечення. В оригінальному дослідженні Lean IT дослідження відходи аналізувались без особливого систематичного огляду причин їх існування.

Крім того, відходи від розробки програмного забезпечення не класифікувались.

Отже, інженер-програміст, який бажає вдосконалити своє середовище розробки програмного забезпечення, має єдиний вибір використання відходів, описаних в оригінальному дослідженні.

Цей документ надав більш систематичний огляд та категоризацію програмних відходів.

Для цього процес розробки програмного забезпечення був розбитий на три компоненти: процеси, проміжні з'єднання між процесами та кінцевий продукт.

Використовуючи цей підхід, можна класифікувати відходи розробки програмного забезпечення. Це також надає інженеру програмного забезпечення спосіб знайти нові відходи розробки програмного забезпечення в його робочому процесі.

Виявлено такі відходи:

Частково виконана робота

Надмірні особливості

Додаткові процеси

Код одноразового використання

Робочий рух

Дефекти

Повторне набуття знань

Перемикання завдань

Нестандартизований товар

Затримки

Управлінська діяльність

Екологічно чистий продукт

Незворотні рішення

Ці відходи були проаналізовані та описані з використанням наявних попередніх досліджень.

Ще важливіші причини їх появи були виявлені та описані. Були вивчені практики та організаційні форми розробки програмного забезпечення, які можуть покращити ситуацію.

Деякі з таких рішень вимагають незначних доопрацювань, тоді як інші можуть бути неможливі повністю позбутися.

Відходи, виявлені в процесі розробки програмного забезпечення, були класифіковані за категоріями.

Перш за все їх класифікували за допомогою концепції сталого розвитку.

Переважна більшість відходів можна віднести до соціальних та економічних аспектів сталого розвитку.

Відходи також класифікуються за допомогою компонентів процесу розробки програмного забезпечення, виявлених раніше.

Відходи групуються приблизно однаково за цими трьома компонентами.

Відходи, що входять до складу процесу, зазвичай є загальними проблемами управління, і багато з них можна віднести до інших організацій, а не лише до тих, що використовують розробку програмного забезпечення.

Іншою важливою групою відходів є відходи, пов'язані зі зв'язком між процесами розвитку.

Було виявлено, що існує значна кількість таких відходів. Це доводить, що розробка програмного забезпечення значною мірою покладається на обмін інформацією.

Agile методологія критикує традиційну методологію водоспаду здебільшого для зв’язку між процесами розробки програмного забезпечення цього тижня.

Остання частина цієї роботи зосереджена на доведенні дискусії про організацію розробки програмного забезпечення до об'єктивних підстав.

Хоча деякі твердження щодо відходів розробки програмного забезпечення здаються інтуїтивно вірними, важливо зосередитись на пошуку об'єктивних підтверджень таких тверджень.

Цього бракує деяким худорлявим IT-авторам.

Для поліпшення цієї ситуації було розроблено три експерименти. Ці експерименти зосереджені на роботі з наступними відходами: перемикання завдань, робочий рух та дефекти.

Експерименти були описані офіційно, тому в основному кожен опис експерименту може використовуватися незалежно.

Використовуючи ці плани, можна провести експеримент безпосередньо в університетському середовищі.

Це може забезпечити підтвердження та оцінку деяких відходів розробки програмного забезпечення. Це може допомогти перевести проблему із загальної дискусії на більш наукову підробку.

Що ще важливіше, такі плани експериментів можна використовувати, щоб отримати більше розуміння щодо відходів. Це можливо, оскільки плани експериментів містять чіткі гіпотези та містять формули, що дозволяють кількісно оцінювати відходи.

Вони також надають більше інформації про причини відходів програмного забезпечення, оскільки експерименти імітують основні практики та умови, які можуть призвести до появи відходів.

Все це може бути використано на виробничому майданчику для підвищення ефективності процесу розробки програмного забезпечення.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Andreas M. Antonopoulos Mastering Bitcoin*: *Unlocking Digital Cryptocurrencies* / *Andreas M. Antonopoulos – K.*: *NGITS,* 2014. – 420 с.

2. *Don Tapscott, Alex Tapscott Blockchain Revolution*: *How the Technology Behind Bitcoin is Changing Money, Business, and the World* / *Don Tapscott, Alex Tapscott Blockchain – K.* : *Information Systems,* 2016. – 328 с.

3. Аммус, С. *The Bitcoin Standard* [Електронний ресурс] / С. Амусс. – Режим доступу: *http*://*www.altcoincalendar.info*/*r*/*article*/*h*72. – 28.04.2018.

4. Антонопулос А. *Mastering Bitcoin* [Електронний ресурс] / А.Антонопулос. – Режим доступу: *https*://*github.com*/*bitcoin_book*/*bitcoin_book*. – 21.01.2021.

5. Брикман Е. *Bitcoin by Analogy* [Електронний ресурс] / Е. Брикман. – Режим доступу: *http*://*brikis*98.*blogspot.com*/2014/04/*bitcoin-by-analogy.html*. – 09.01.2021.

6. Взлёты и падения биткоина, *The Economist. Buttonwood's notebook. Tales from the crypto* [Електронний ресурс]. – Режим доступу: *https*://*www.economist.com*/*blogs*/*buttonwood*/2018/01/*tales-crypto*-1. – 15.01.2021.

7. Винья П., Кейси М. Эпоха криптовалют. Как биткоин и блокчейн меняют мировой экономический порядок [Електронний ресурс] / П.Винья, М.Кейси. – Режим доступу: *https*://*piratbit.ru*/*topic*/251351. – 14.01.2021.

8. График курса биткоина [Електронний ресурс]. – Режим доступу: *https*://*ru.investing.com*/*crypto*/*bitcoin*/*btc-usd*. – 11.01.2021.

9. Кунета М. Мнение: почему протокол биткоина уже не остановить [Електронний ресурс] / М.: Кунета. – Режим доступу: *https*://*whattonews.ru* /*reviews*/13939/*mnenie-pochemu-protokol-bitkoina-uzhe-ne-ostanovit*/. – 30.04.2018.

10. Лагард К. Работа с темной стороной криптомира [Електронний ресурс] / К. Лагард. – Режим доступу: *http*://*www.inf.org*/*external*/*ukranian*/*np*/*blog*/2018/ 031318*r.pdf*. – 10.01.2021.

11. Накамото С. *P2P e-cash system* [Електронний ресурс] / С. Накамото. – Режим доступу: *https*://*bitcoin.org*/*bitcoin_new.pdf*. – 12.01.2021.

12. Таблица негативных факторов [Електронний ресурс]. – Режим доступу: *https://cyberleninka.ru/article/n/provedenie-swot-analiza-dlya-otsenkif - vliyay-uschih-na-razvitie-kriptovalyuty*. – 20.01.2021.

13. Таблица сильных и слабых сторон криптовалют [Електронний ресурс]. – Режим доступу: *https://cyberleninka.ru/article/n/provedenie-swot-analizadlya-otsenki-faktorov-vliyayuschih-na-razvitie-kriptovalyuty*. – 20.01.2021

14. Таблица факторов формирования матрицы возможностей [Електронний ресурс]. – Режим доступу: *https://cyberleninka.ru/article/n/provedenie-swotanaliza-dlya-otsenki-faktorov-vliyayuschih-na-razvitie-kriptovalyuty*. – 20.01.2021.

15. Тапскотт А. Тапскотт Д. Технология блокчейн [Електронний ресурс] / Т. Тапскотт, Д. Тапскотт. – Режим доступу: *http://tornado.org.ru/details.php?id*=32893 – 16.01.2021.

16. Украина переведет все государственные данные на блокчейн [Електронний ресурс] – Режим доступу *https://hightech.fm*/2017/04/14/*us-ukraine-bitfury-blockchain*.

17. Целищев, П. Проведение *SWOT*-анализа для оценки факторов, влияющих на развитие криптовалюты [Електронний ресурс] / П. Целищев. – Режим доступу: *https://cyberleninka.ru/article/n/provedenie-swot-analiza-vliyayuschih-na-razvitie-kriptovalyuty*. – 20.01.2021.

18. ДСТУ 3008-95 Документація. Звіти у сфері науки і техніки. Структура і правила оформлення.

19. НД ТЗІ 1.1-003-99. Термінологія у області захисту інформації в комп'ютерних системах від несанкціонованого доступу. // Департамент спеціальних телекомунікаційних систем і захисту інформації Служби безпеки України. – Київ, 1999.

20. ДСТУ 3008-95 Документація. Звіти у сфері науки і техніки. Структура і правила оформлення.

21. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.

Додаток А

Лістинг коду основного програмного модуля