МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 14.00.00.000 ПЗ

Група ІПм-23-2

Ляхов Андрій

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Ляхов Андрій Вячеславович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Побудова методології аналізу продуктивності процесу розробки програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Ляхов А.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Бандура Вікторія Валеріївна, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Ляхову Андрію Вячеславовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Побудова методології аналізу продуктивності процесу розробки програмного забезпечення”

керівник проекту (роботи) Бандура Вікторія Валеріївна, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій продуктивності розробки ПЗ

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Теоретичні засади та огляд області підвищення продуктивності проектів розробки ПЗ
2. Особливості визначення даних для побудови методології аналізу продуктивності розробки
3. Моделювання робочого процесу командної розробки
4. Представлення процесів та методології аналізу продуктивності процесу розробки ПЗ

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Інструмент перевірки якості коду CodeScene (рис. 1.1)
2. Представлення помилки в системі Jira (рис. 1.2)
3. Фреймворк Scrum (рис. 1.3)
4. Інтерфейс користувача Jira. Поля праворуч можуть залежати від специфіки компанії (рис. 1.4)
5. Частота випуску продукту (релізи) (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Теоретичні засади та огляд області підвищення продуктивності проектів розробки ПЗ	15.10.2024	виконано
4	Особливості визначення даних для побудови методології аналізу продуктивності розробки	08.11.2024	виконано
5	Моделювання робочого процесу командної розробки	20.11.2024	виконано
6	Представлення процесів та методології аналізу продуктивності процесу розробки ПЗ	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 84 с., 33 рис., 54 джерела.

Тема: Побудова методології аналізу продуктивності процесу розробки програмного забезпечення

Об'єкт дослідження: процеси командної розробки програмного забезпечення в умовах використання гнучких методологій.

Мета роботи: розробка методології аналізу продуктивності процесу розробки програмного забезпечення на основі моделювання робочих процесів, аналізу даних та визначення ключових факторів, що впливають на продуктивність командної роботи.

Предмет дослідження: методи та інструменти аналізу продуктивності процесу розробки програмного забезпечення, зокрема використання моделювання робочих процесів.

Результати дослідження

В роботі виконано розробку систематизованої методології аналізу продуктивності розробки програмного забезпечення, яка базується на моделюванні робочих процесів та використанні процесного майнінгу. Запропоновано нові показники продуктивності, що враховують тривалість та ефективність виконання завдань у процесі командної роботи.

Висновок

Запропоновано методологію аналізу робочих процесів, зокрема процесів обробки помилок та розробки нових функцій. Було змодельовано робочі процеси на основі журналів подій, що дозволило більш точно виявити вузькі місця в роботі команд та запропонувати шляхи їх оптимізації.

ПРОДУКТИВНІСТЬ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ГНУЧКІ МЕТОДОЛОГІЇ, КОМАНДНА РОЗРОБКА, МОДЕЛЮВАННЯ РОБОЧИХ ПРОЦЕСІВ, ПРОЦЕСНИЙ МАЙНІНГ, КОНТРОЛЬ ВЕРСІЙ, УПРАВЛІННЯ ЯКІСТЮ КОДУ

ABSTRACT

Master Thesis: 84 pp., 33 fig., 54 sources.

Thesis Subject: Construction of a methodology for analyzing the productivity of the software development process

The object of research: processes of team software development in conditions of using flexible methodologies.

The purpose of the work: development of a methodology for analyzing the productivity of the software development process based on the modeling of work processes, data analysis and the identification of key factors affecting the productivity of teamwork.

The subject of research: methods and tools for analyzing the productivity of the software development process, in particular, the use of workflow modeling.

Research results

The paper developed a systematized methodology for software development performance analysis, which is based on the modeling of work processes and the use of process mining. New productivity indicators are proposed, which take into account the duration and efficiency of tasks in the process of teamwork.

Conclusion

A methodology for the analysis of work processes, in particular the processes of error handling and the development of new functions, is proposed. Work processes were modeled on the basis of event logs, which made it possible to more accurately identify bottlenecks in the work of teams and suggest ways to optimize them.

PRODUCTIVITY OF SOFTWARE DEVELOPMENT, FLEXIBLE METHODOLOGY, TEAM DEVELOPMENT, MODELING OF WORKING PROCESSES, PROCESS MINING, VERSION CONTROL, CODE QUALITY MANAGEMENT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MDSD - Model-Driven Software Development

JQL - Jira Query Language

CCB - Change and Control Board

TEM - transmission electron microscope

LoC - Lines Of Code

ВСТУП

Актуальність теми.

Актуальність дослідження полягає в необхідності підвищення продуктивності розробки програмного забезпечення в умовах стрімкого розвитку інформаційних технологій та зростаючих вимог до якості програмних продуктів. Сучасні програмні проєкти характеризуються високим рівнем складності, що вимагає оптимізації командних робочих процесів та ефективного управління ресурсами. При цьому, гнучкі методології розробки, такі як Agile і Scrum, широко застосовуються для забезпечення адаптивності та гнучкості процесу розробки, однак відсутність чітких механізмів вимірювання продуктивності робить складним оцінку ефективності командної роботи.

Впровадження новітніх інструментів контролю версій (Git), автоматизації процесів та управління якістю коду не гарантує високої продуктивності без належного аналізу робочих процесів. На сьогоднішній день існує потреба у застосуванні систематизованих методів оцінки продуктивності розробки, які б дозволяли виявляти ключові фактори, що впливають на ефективність роботи команд, зокрема в процесах обробки помилок та розробки нових функцій.

Крім того, нові методи аналізу, такі як процесний майнінг, надають можливість дослідження реальних робочих процесів на основі журналів подій, що дозволяє отримати більш точні результати порівняно з традиційними підходами. Однак, незважаючи на існування цих новітніх підходів, відсутність загальновизнаних методологій для оцінки продуктивності командної роботи у програмній інженерії створює прогалину в науковому та практичному підході до оптимізації робочих процесів.

В умовах зростаючої складності програмних проєктів та зростаючого попиту на високоякісні програмні продукти ефективність командної розробки програмного забезпечення стає ключовим чинником успіху. Гнучкі

методології розробки, такі як Agile і Scrum, а також інструменти для контролю версій і перевірки якості коду, широко використовуються для підвищення продуктивності розробників. Однак, існує потреба в систематизованій методології аналізу продуктивності розробки, яка дозволила б оцінювати і вдосконалювати процеси розробки на основі кількісних та якісних показників. Актуальність даної теми визначається необхідністю впровадження ефективних підходів до підвищення продуктивності команд розробки програмного забезпечення через аналіз робочих процесів та показників їх ефективності.

Таким чином, розробка і впровадження науково обґрунтованої методології аналізу продуктивності розробки програмного забезпечення є актуальним завданням, яке сприятиме підвищенню ефективності командної роботи, поліпшенню якості продуктів та прискоренню їх виведення на ринок. Це дослідження особливо актуальне для компаній, що прагнуть удосконалювати процеси розробки та підтримувати високий рівень конкурентоспроможності на ринку.

Мета дослідження - розробка методології аналізу продуктивності процесу розробки програмного забезпечення на основі моделювання робочих процесів, аналізу даних та визначення ключових факторів, що впливають на продуктивність командної роботи.

Об’єкт дослідження - процеси командної розробки програмного забезпечення в умовах використання гнучких методологій.

Предмет дослідження - методи та інструменти аналізу продуктивності процесу розробки програмного забезпечення, зокрема використання моделювання робочих процесів.

Відповідно до мети роботи було сформовано наступні **задачі:**

- Провести аналіз предметної області та визначити ключові фактори, що впливають на продуктивність розробки програмного забезпечення.

- Дослідити особливості застосування гнучких методологій та інструментів для контролю версій у процесі розробки.
- Розробити методологію аналізу продуктивності розробки програмного забезпечення, включаючи використання журналів подій і моделювання робочих процесів.
- Визначити методи збору та аналізу даних, що впливають на продуктивність розробки.
- Перевірити валідність розроблених показників продуктивності та їх застосовність у реальних проєктах.
- Оцінити ефективність запропонованої методології через порівняння різних робочих процесів командної розробки

Методи дослідження.

1. Аналіз літератури для вивчення сучасних підходів до підвищення продуктивності розробки програмного забезпечення.

2. Моделювання робочих процесів для візуалізації та аналізу процесів розробки та обробки помилок.

3. Процесний майнінг для дослідження фактичних робочих процесів на основі журналів подій.

4. Аналіз даних для виявлення ключових факторів, що впливають на продуктивність.

Наукова новизна отриманих результатів полягає у розробці систематизованої методології аналізу продуктивності розробки програмного забезпечення, яка базується на моделюванні робочих процесів та використанні процесного майнінгу. Запропоновано нові показники продуктивності, що враховують тривалість та ефективність виконання завдань у процесі командної роботи. Доведено, що застосування методів розробки на основі моделей (MDSD) дозволяє підвищити продуктивність команд, що має суттєвий вплив на загальну ефективність проєктів.

Практичне значення магістерської роботи полягає в полягає у можливості застосування розробленої методології для оптимізації робочих

процесів у реальних командах розробників програмного забезпечення. Запропоновані підходи можуть бути використані для оцінки продуктивності команд, виявлення вузьких місць у процесі розробки та підвищення загальної ефективності проєктів

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 84 сторінки, і містить 33 рисунки та список використаних джерел із 54 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ ТА ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ПРОЕКТІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Опис предметної області дослідження

Групи розробників програмного забезпечення постійно прагнуть підвищити ефективність своєї діяльності. Часто в IT компаніях декілька команд працюють над окремими компонентами кінцевого продукту, кожна з яких демонструє різний рівень продуктивності. Важливим завданням є визначення ключових чинників, що впливають на ці відмінності у продуктивності, а також оцінка їхнього впливу. Це має значення для прийняття обґрунтованих рішень щодо оптимізації складу команд і покращення процесів розробки програмного забезпечення в майбутньому. У рамках даного дослідження здійснено моделювання робочого процесу команд з метою виявлення затримок у процесі, а також розроблено метрику для оцінки продуктивності команд. Використовуючи цю метрику, було проведено порівняльний аналіз команд та визначено впливові фактори, такі як якість коду, розмір команди, рівень досвіду команди та підхід до розробки програмного забезпечення на основі моделі.

Багато компаній, зокрема ті, що орієнтовані на розробку програмного забезпечення, прагнуть максимізувати цінність, яку створює їхня робоча сила. У компаніях розробляють вузькоспеціалізоване програмне забезпечення, важливим компонентом їх функціонування відіграє дедалі більшу роль, оскільки багато процесів автоматизуються. Це відображає зміни у користувацькій базі: якщо раніше з обладнанням працювали лише фахівці, то тепер дедалі більше користувачів без спеціальних знань прагнуть взаємодіяти з системою та отримувати аналогічні результати. Така трансформація призвела до зростання кодової бази та поставила нові виклики щодо того, на чому необхідно зосередити зусилля для подальшого

масштабування, забезпечення легкості підтримки, збереження якості та максимізації цінності вкладених зусиль. Для дослідження чинників, які найбільш впливають на ці аспекти, було сформульовано дослідницьке питання:

- Які фактори визначають продуктивність команди розробників програмного забезпечення?

Під час проекту стало зрозуміло, що це питання охоплює занадто багато, щоб відповісти протягом тривалості проекту, тому сформульовано наступні три піддослідницькі питання:

1. Чи можна змоделювати процес розробки програмного забезпечення командою ?

2. Як порівняти продуктивність програмних команд між собою ?

3. Чи може порівняння програмних команд виявити фактори, що сприяють продуктивності ?

Проект розпочався з оцінки комерційного готового інструменту, який міг би допомогти (частково) відповісти на запитання дослідження, результати якого наведено нижче.

CodeScene — це інструмент, який показує якість коду та стосунки між розробниками шляхом майнінгу сховища коду. Інструмент пропонує, які файли мають покращити якість коду, зосереджуючись на гарячих точках, файлах, які часто редагуються. Перевага цього інструменту перед іншими інструментами, які надають інформацію про якість коду, полягає в тому, що він визначає пріоритетність того, з чого почати вдосконалення бази коду, тоді як інші інструменти перевантажують розробника кількістю виявлених проблем якості та показують проблеми якості, які розробники не помічають. бути критичним.

Однак оцінка пропозицій показує, що підхід до гарячих точок не завжди дає змістовні пропозиції. Пропозиції можна розділити на три групи:

- файли, які часто змінюються, оскільки на них зосереджено поточний фокус проекту,

- файли, які часто змінюються через зміни залежностей третіх сторін,

- файли конфігурації, які часто змінюються.

Під час обговорення пропозицій з розробником з одного з досліджених сховищ вони вказали, що згодні з двома з тринадцяти пропозицій. Ці два файли були тестовими, і команда вже знала, що вони мають змінитися, оскільки вони зросли до понад 5000 рядків коду. Розробник зазначив, що, швидше за все, не буде змінювати запропоновані файли через обсяг роботи, який потрібно зробити для цього. Не всі пропозиції легко втілити в дію та вимагають значної кількості переробок без явної переваги, окрім вищої якості коду. Щоб переробка окупилася, фокус проекту повинен залишатися на тій самій частині протягом деякого часу, але це не гарантовано. Категорія файлів, які підхід до точки доступу не може виявити, — це файли, які не змінюються через те, що команда не збирається їх змінювати, оскільки власник залишив команду. Команда хотіла б змінити їх, але через страх порушити функціональність вони вирішили обійти це. Інструмент також показує спільну еволюцію файлів, але це дає очікувані результати, наприклад тестові файли часто змінюються разом із впровадженням.

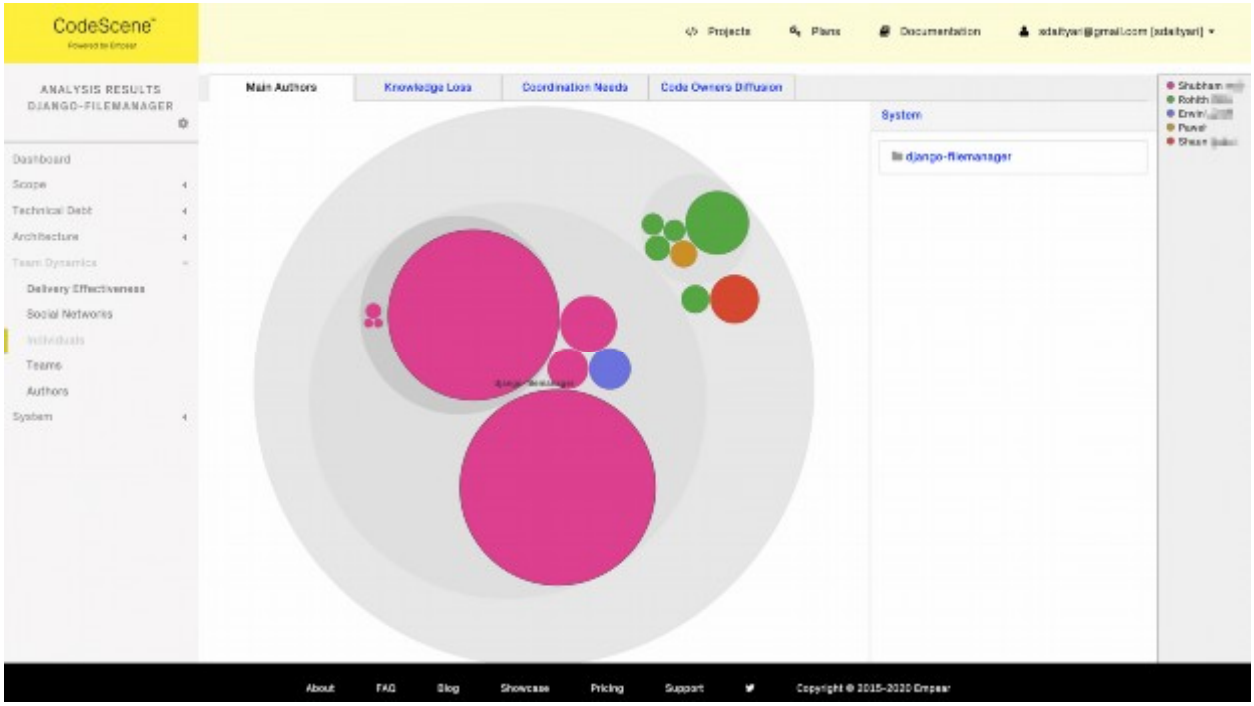

Рис. 1.1. Інструмент перевірки якості коду CodeScene

Командний аналіз надає огляд власності на код у сховищі. Це чітко показує, яким кодом володіють нинішні та колишні члени команди, а також те, що належить людям поза командою. У випадку, коли код належить іншим командам, ніж команда, яка володіє більшістю коду в сховищі, причина полягає в тому, що команда не має можливості реалізувати функцію для іншої команди, тому інша команда робить це самостійно. Випадки, коли це відбувається, низькі, один або два випадки на проект.

Ці екземпляри також потрібно перевірити, оскільки CodeScene дозволяє вводити лише поточну конфігурацію складу команди; тому люди, які змінили команду, спричиняють трохи шуму у візуалізації. Крім того, люди можуть бути призначені лише до однієї команди в CodeScene, тобто люди в кількох командах створюють шум. Першу проблему пом'якшує CodeScene, обмежуючи аналіз права власності на код останніми трьома місяцями, але через це є багато папок сховища, де право власності позначено як непереконливе.

Інтеграція Jira

CodeScene забезпечує інтеграцію з Jira, щоб дозволити візуалізувати, де витрачається час на базу коду. Це досягається шляхом врахування різниці в часі між моментом, коли проблема Jira, з якою пов'язано фіксацію, переходить у стан «виконується» та моментом завершення фіксації. CodeScene додає кількість помилок і функцій і показує це як загальну вартість, вартість функцій і вартість помилок кожної частини програмного забезпечення. Існує також графік сумарної кількості годин, витрачених на проект на місяць. Показані графіки вказують на те, що щось не так, оскільки мінливість між місяцями велика, а також години, витрачені на помилки, дуже низькі для деяких команд.

Подальше дослідження показує, що CodeScene не розглядає зв'язок між проблемами Jira. Це призводить до того, що команди вирішують помилки, пов'язуючи комміт з історією рішення або підзавданням, щоб час, витрачений на цей комміт, зараховувався до розробки функцій замість

виправлення помилок. Щоб час, витрачений на функції та помилки, був правильним, усі коміти виправлення помилок мають бути пов’язані з помилкою, для функцій не має значення, який тип проблеми пов’язано, оскільки будь-який тип проблеми, який не є помилкою, зараховується як розробка функції. Різниця в часі може бути пов’язана з тим, що команди, що працюють над кількома сховищами, щомісяця витрачають різну кількість часу на роботу над репозиторієм. Експерименти зі зміною моделі витрат не дали більш узгодженого обсягу зусиль з часом для кожного сховища. Наприклад, зміна моделі витрат на використання сюжетних балів показала різницю через те, що лише 70–80% історій у досліджуваних командах мали сюжетні бали, а помилки не мали сюжетних балів.

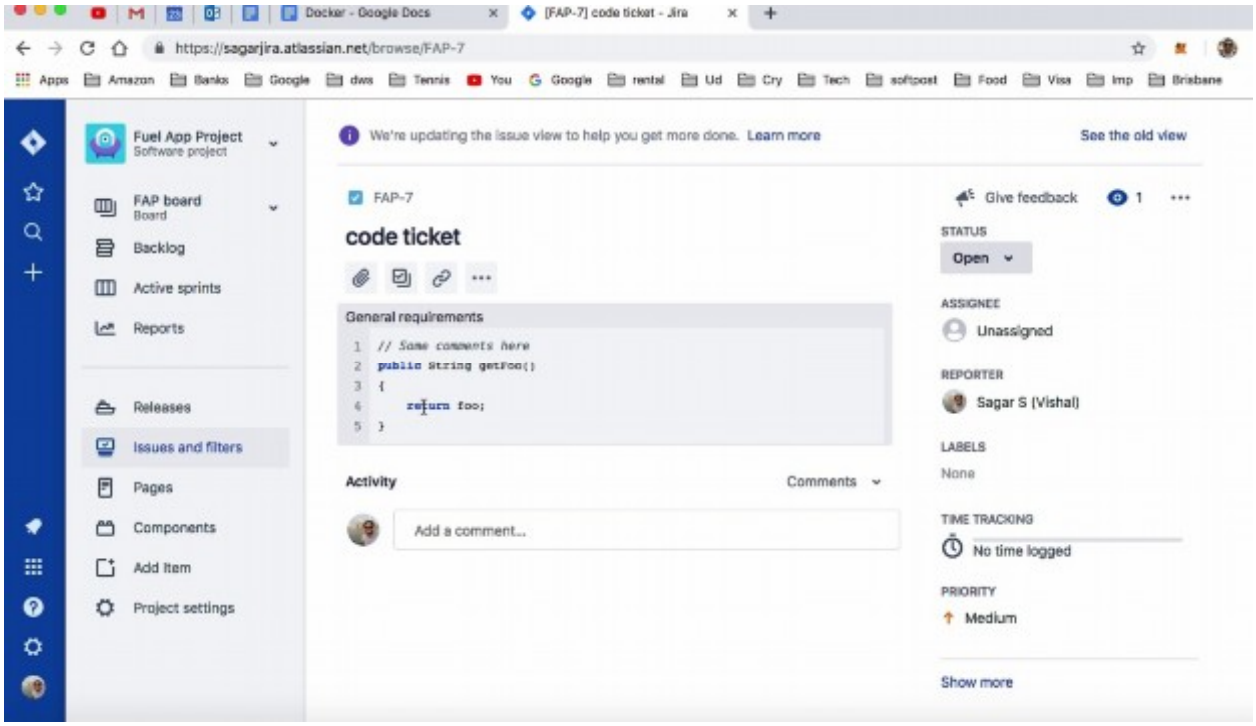

Рис. 1.2. Представлення помилки в системі Jira

Пропозиції, які CodeScene робить для покращення бази коду, правильно визначають, де код часто змінюється, але вдосконалення коду в цих файлах вимагає значних зусиль і не забезпечує явної вигоди в довгостроковій перспективі, оскільки в майбутньому фокус проекту може переміститися на інші функції . Надана інтеграція з Jira не узгоджується з

тим, як команди вирішують помилки та пов’язують свої зобов’язання з проблемами Jira для вирішення проблеми. Через відсутність ключа проблеми до вихідної помилки візуалізація того, скільки часу витрачається на виправлення помилок у різних частинах сховища, є неправильною. Це призводить до того, що CodeScene не може показати, чи файли з нижчою якістю коду вимагають більше часу для виправлення помилок і чи справді технічна заборгованість, показана інструментом, уповільнює команду.

Результати комерційного інструменту дозволили проаналізувати робочий процес команд, які працюють над різними компонентами програмного забезпечення. В даній роботі детально описується підхід до моделювання та результати процеси всередині команд і відмінності між ними.

1.2. Особливості командної розробки програмного забезпечення

1.2.1. Гнучка методологія

Команди розробників дотримуються гнучкого способу роботи, вносячи поступові зміни в програмне забезпечення для досягнення мети [8]. Це дозволяє ітераційно надавати другорядні функції, що сприяють розвитку основної функції, дозволяючи командам ділитися своєю роботою з іншими командами або клієнтами, щоб отримати відгук про неї раніше, ніж при використанні каскадного підходу. Щоб застосувати цю гнучку методологію на практиці, команди дотримуються методології scrum. Команди працюють спринтами, з кроком один або два тижні, які починаються із сесії планування, під час якої вони вирішують, на чому зосередитися в майбутньому спринті. Фокус значною мірою визначається власником продукту, членом команди, якому доручено визначати пріоритети роботи та пропонувати, що включити в наступні спринти. Щоб вирішити, скільки роботи включити в певний спринт, команди визначають оцінку часу або зусиль для кожного завдання, виражену в сюжетних балах, і включають у спринт стільки завдань, скільки, на їхню

думку, можуть впоратися з story points. Потім робота призначається членам команди, вони працюватимуть над завданнями та, як правило, щодня проводять зустрічі з командою, щоб інформувати інших про прогрес. Спринт завершується ретроспективою, щоб обговорити, як все пройшло, і внести покращення для наступного спринту. Для нагляду за цим процесом у більшості команд є скрам-майстер, особа, відповідальна за те, щоб команда дотримувалася всіх елементів скраму. Огляд структури Scrum показано на рисунку 1.3.

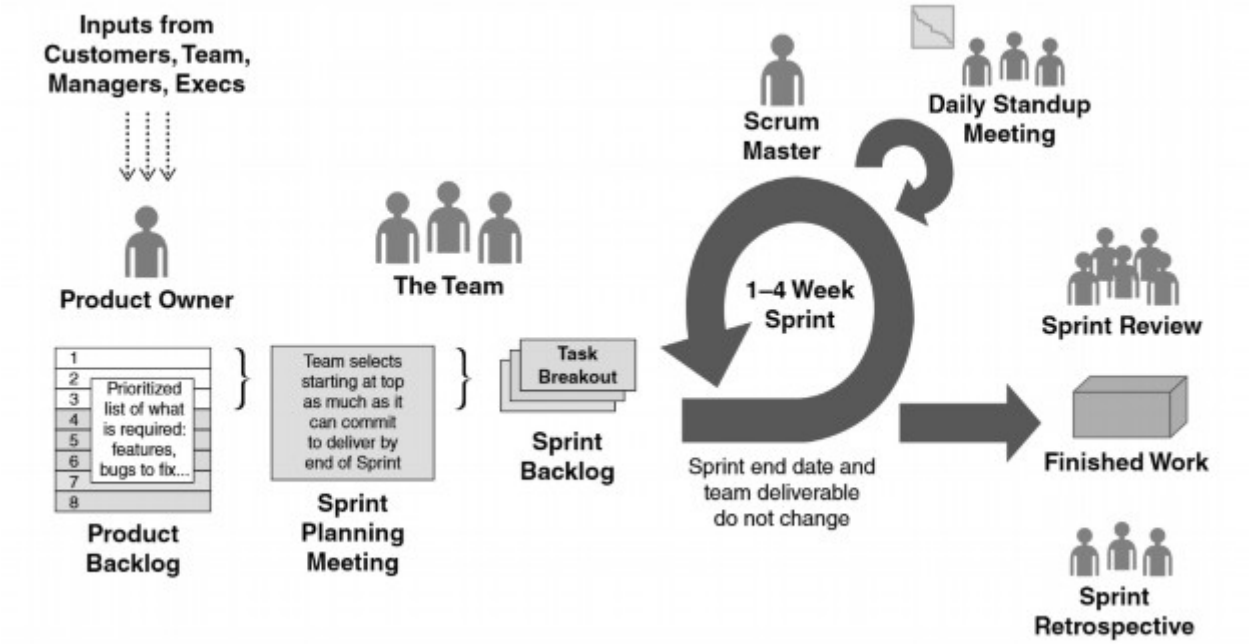

Рис. 1.3. Фреймворк Scrum

1.2.2. Інструменти для розробки

Кілька інструментів використовуються для відстеження виконаної роботи, версії, інтеграції та вимірювання якості. Jira використовується як засіб відстеження помилок [2]. Завданнями, представленими проблемою, можуть бути помилка, яку потрібно виправити, або функція, яку потрібно реалізувати, а також менші завдання, як-от запит у службу підтримки до команди або електронний лист, який потрібно надіслати. Усі завдання, які виконує команда, мають відповідну проблему в Jira та мають ключ, на який

посилаються під час використання іншого програмного забезпечення. Цей ключ проблеми складається з префікса команди, за яким слідує число. Кожна команда має власну дошку в інтерфейсі користувача, де перераховані всі їхні проблеми. На дошках спринту відображаються лише проблеми, над якими працюємо в даний момент, і їх статус. Приклад інформації, яка відображається під час перегляду випуску, зображено на рисунку 1.4.

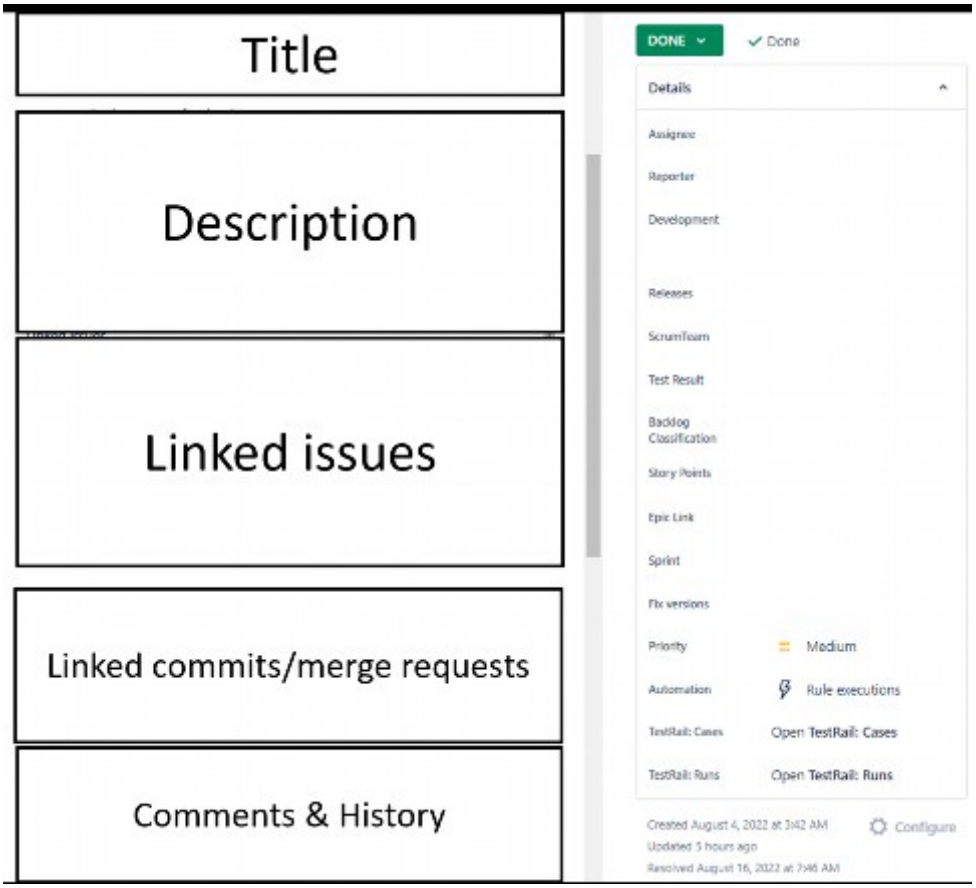

Рис. 1.4. Інтерфейс користувача Jira. Поля праворуч можуть залежати від специфіки компанії

Найважливіші поля знаходяться вгорі: title, description, status, resolution, assignee і reporter.

Нижче ми знаходимо більш конкретну інформацію: класифікація невиконаних завдань, версія виправлення, точки історії, спринт, пріоритет, пов’язані проблеми, коміти та запити на злиття, що стосуються проблеми.

Внизу є місце для коментарів і перегляду змін, внесених у поля проблеми. Опис і можливі значення для нетривіальних полів наведено в таблиці 1.1.

Таблиця 1.1.

Поля проблеми з описом і можливими значеннями

Field	Description	Values
Backlog Classification	The type of work	Technical Debt Technical Innovation Business Value Support
Resolution	The end result of the issue	None (issue still open) Done Canceled
Story points	An estimate of how long it will take to resolve the issue. Some teams do not fill it in for bugs	a number
Status	The state of the issue	popular values: inbox, triage, open, backlog, to investigate, investigated, to do, blocked, in progress, in test, resolved, rejected, done, closed

Існує п’ять типів питань:

- Epic. Велике завдання, яке складається з кількох історій. Цей тип зазвичай використовується для великої функції або загальної ініціативи, яку проводить команда.

- Story. Окреме завдання або завдання, яке є частиною epic. Це має відображати обсяг роботи, який не потрібно розбивати далі, хоча команди можуть це зробити.

Bug. Проблема в програмному забезпеченні.

- Task. Щось, що потрібно зробити ad hoc, історія зазвичай створюється для планової роботи.

- Sub-task. Невелике завдання, яке є частиною більшого завдання.

Проблеми (Issues) структуровані в трирівневу ієрархію, від найбільшого до найменшого:

• Epic

• Bug, Story, Task

• Sub-task

Проблеми можуть бути пов’язані одна з одною, це можуть бути стосунки «батьки-діти», між вищим і нижчим рівнями в ієрархії, наприклад, епос та історія, що вказує на те, що проблема на нижчому рівні в ієрархії є частиною одного на вищому рівні. Крім того, проблеми також можуть бути пов’язані з іншими на тому самому рівні в ієрархії проблем. Ці зв’язки також можуть бути застосовані між проблемами на різних рівнях і можуть бути одним із типів зв’язків проблеми: блоки, зв’язки, дублікати, посилання, клони, стеження, лікування, містить, причини, відокремлення та виявлення.

1.2.3. Репозиторій Git для контролю версій

Репозиторії програмного забезпечення використовують Git як систему контролю версій [15]. Під час виправлення помилки або роботи над функцією розробник створить нову гілку, у якій він працюватиме лише над цим завданням. Гілка складається з серії змін коду, які називаються комітами, які відстежують, які рядки в кожному файлі були змінені. Коли завдання буде виконано, розробник відкриє запит на отримання, сигналізуючи іншим розробникам, що їхня робота готова до перегляду. Відкриття запиту на отримання також запускає збірку, яка автоматично піддається деяким тестам. Запит на злиття показує коментарі зроблені іншими та результати тестування. Якщо тести проходять успішно та інші члени команди схвалюють зміни, зміна коду об’єднується в гілку, для якої було відкрито запит на отримання.

Git дозволяє розробникам працювати над кодом у розподіленому режимі, тобто локальні гілки та коміти на машині розробника не обов’язково надсилаються на віддалений сервер Git. Крім того, функція Git полягає в тому, що історію можна переписати, це застосовується до запитів на отримання шляхом стискання окремих комітів в один комміт. Ці два аспекти

означають, що історію сховищ можна зберегти коротшою і, отже, чистішою, але це також видаляє інформацію про те, скільки комітів було потрібно для вирішення проблеми, а разом з цим також і значення часу. Щоб дізнатися, чому було внесено певну зміну коду , усі повідомлення коміту починаються з ключа проблеми Jira, для якого призначено зміну. Часто репозиторії містять гілку розробки, гілку кандидата на випуск і гілку випуску.

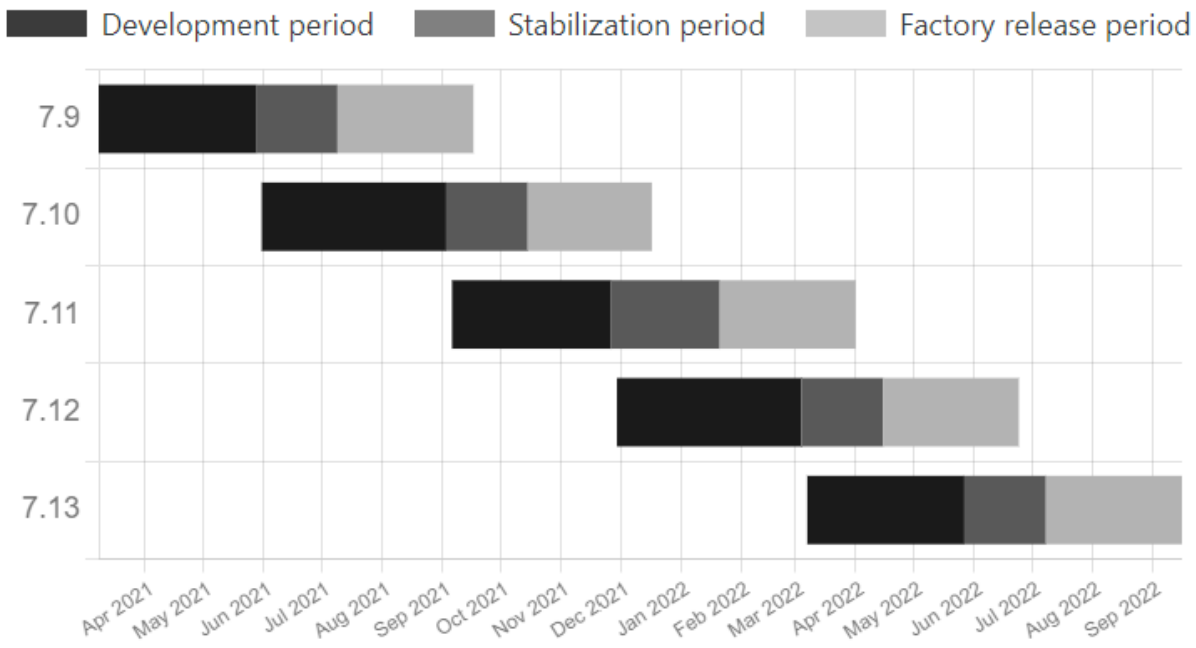

Рис. 1.5. Частота випуску продукту (релізи)

Частота випуску показана на рисунку 1.5. Реліз розділений на три періоди, що відповідають гілкам. Період розробки становить близько трьох місяців, після чого код буде об’єднано з гілкою кандидата на випуск і розпочнеться період стабілізації.

Протягом цього періоду в один місяць програмне забезпечення цієї філії перевіряється більш ретельно, і лише виправлення помилок дозволяється вносити у філію. Інкремент закінчується тим, що код об’єднується з гілкою релізу, і код встановлюється на системах, що створюються на заводі. Цей період випуску також становить місяць, після чого відбувається публічний випуск.

1.3. Засоби та методи перевірки якості коду. Розробка на основі моделі

Якість програмного забезпечення вимірюється за допомогою структури TiCS [28]. Фреймворк перевіряє код на ряд властивостей: покриття коду, абстрактна інтерпретація, цикломатична складність, попередження компілятора, стандарти кодування, дублювання коду, розгортання та безпека. Кожне з цих властивостей отримує власну оцінку, яка з різними вагами вносить свій внесок у загальну оцінку показника якості. Такі бали та підбали надаються для кожного файлу, папки та сховища. Інтерфейс користувача показує це у відсотках літерою у форматі, який зазвичай зустрічається на побутових приладах (рисунок 1.6).

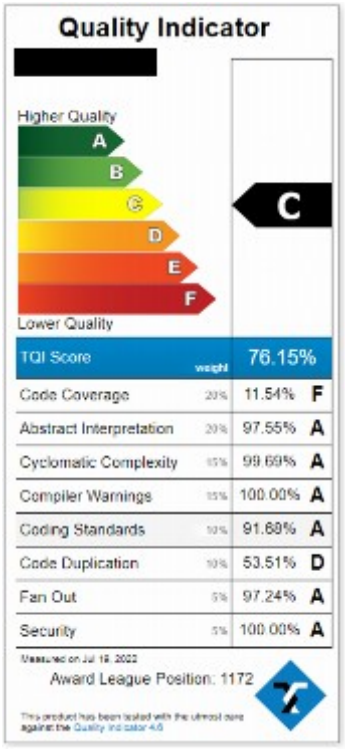

Рис. 1.6. Приклад індикатора якості програмного забезпечення

1.3.1. Інструмент для проектування на основі моделі

Dezyne — це інструмент для проектування, керованого моделлю (Model-Driven Engineering, MDE), який забезпечує автоматизацію процесу

розробки складних програмних систем. Він дозволяє інженерам проектувати, перевіряти та генерувати програмне забезпечення на основі формальних моделей. Основною метою Dezyne є забезпечення коректності системи на ранніх етапах розробки шляхом автоматичного виявлення помилок у моделі ще до етапу програмування.

Основні функції Dezyne:

- Моделювання систем: Інструмент підтримує розробку формальних моделей компонентів системи, що описують їхню поведінку та взаємодію. Це дозволяє створювати системи, що складаються з декількох компонентів, які можуть взаємодіяти між собою через інтерфейси.

- Верифікація: Dezyne автоматично перевіряє моделі на наявність помилок, таких як мертві стани, конфлікти або порушення коректності послідовностей дій. Верифікація допомагає забезпечити правильну роботу системи ще на етапі проектування.

- Автоматична генерація коду: Після моделювання та верифікації системи, Dezyne може автоматично згенерувати код для різних мов програмування, таких як C++ або Java, що відповідає специфікації моделі. Це значно скорочує час на написання коду вручну та знижує ймовірність помилок.

- Підтримка інтерфейсів та асинхронної комунікації: Dezyne дозволяє описувати та перевіряти асинхронну взаємодію між компонентами системи, що є важливим для розробки розподілених систем або систем реального часу.

Переваги використання Dezyne:

- Формальна верифікація: Забезпечує надійність та правильність системи до етапу написання коду.

- Скорочення витрат на тестування та усунення помилок: Помилки можуть бути виявлені на ранніх етапах, що значно знижує вартість виправлень.

- Підвищення продуктивності: Завдяки автоматизації процесів, таких як генерація коду, розробники можуть зосередитись на більш важливих завданнях.

Dezyne ідеально підходить для розробки складних програмних систем, таких як вбудовані системи, системи з високими вимогами до надійності та масштабовані архітектури, де важливо забезпечити коректність взаємодії між компонентами і уникати помилок ще на етапі проектування. Таким чином, Dezyne є потужним інструментом для інженерів, які прагнуть впроваджувати підхід проектування, керованого моделями, з метою підвищення якості та надійності програмного забезпечення.

Dezyne — це інструмент, який використовується для розробки програмного забезпечення на основі моделі (MDSD). Усі нові дії MDSD відбуваються в Dezyne, але деякі старіші моделі є моделями ASD, які були попередниками Dezyne. Інструмент орієнтований на впровадження програмного забезпечення одночасного керування для систем, керованих подіями, і спрямований на скорочення зусиль на тестування та підвищення якості програмного забезпечення шляхом формальної перевірки моделей [32].

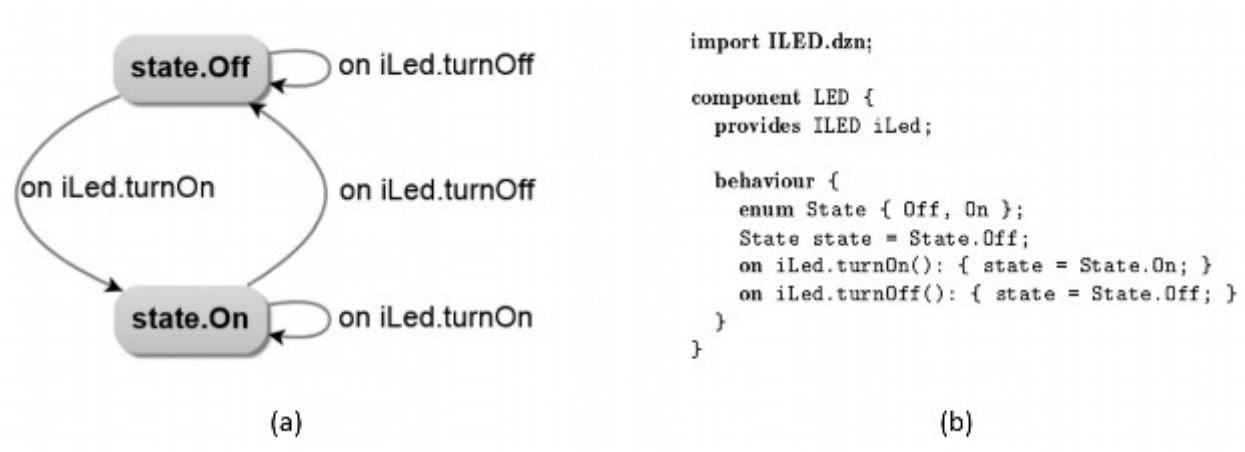

Рисунок 1.7. Вимикач світла засобами Dezyne

Dezyne підходить для програмного забезпечення, яке слідує потоку керування, подібному до кінцевого автомата. Приклад вимикача світла,

змодельованого в Dezyne, наведено на рисунку 1.7. Мова Dezyne надає синтаксис для визначення станів, надання початкового стану та визначення поведінки для зміни станів. На рисунку 1.7 показана результуюча діаграма стану. Після цього Dezyne може перевірити модель на повноту, відсутність блокувань і взаємоблокувань.

1.3.2. Організація та архітектура програмної системи

Вважатимемо, що компанія розділила свою вузькоспеціалізовану програмну систему на три основні частини, зображені на рисунку 1.8:

1) апаратний рівень, що містить мікропрограмне забезпечення пристрою та драйвери,

2) сервер, частина програмного забезпечення, яка працює на обладнанні та залежить від драйверів апаратного забезпечення,

3) програми, які взаємодіють із сервером, який зазвичай працюють в іншій системі та забезпечують інтерфейс користувача.

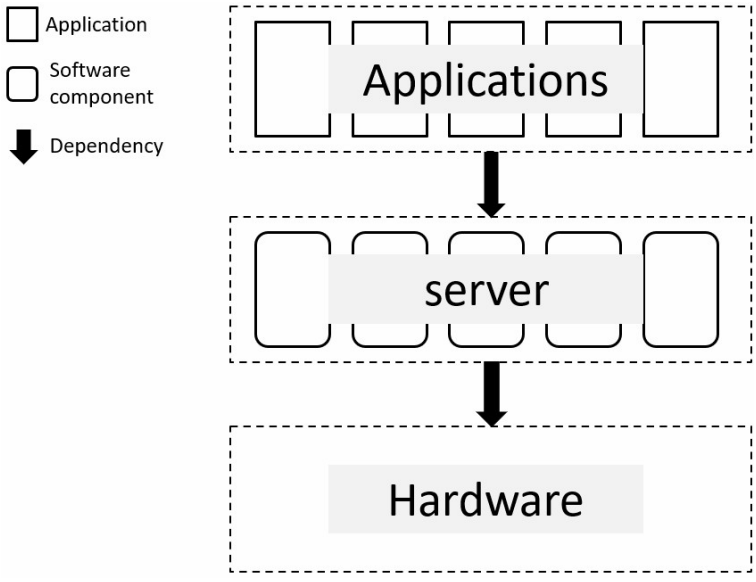

Рис. 1.8. Архітектура проекту розробки вузькоспеціалізованого ПЗ. Залежності всередині шарів не зображені

Існують ще деякі додатки, які працюють на обладнанні, але ведеться робота над їх відокремленням. Програми інтерфейсу користувача інтенсивно

спілкуються з сервером і набагато менше один з одним. Однак компоненти сервера значною мірою залежать один від одного.

Кожна команда працює над власною програмою або частиною сервера, і тому має кілька основних сховищ, яким вони зобов’язуються. Однак їм іноді потрібно вносити зміни в інші сховища, щоб змусити інші системи працювати з їх власними змінами програмного забезпечення. Щоб пом’якшити вплив змін на інші місця, основні підсистеми роз’єднані офіційно узгодженими інтерфейсами. Ці підсистеми мають окреме сховище інтерфейсів, яке рідко змінюється та зберігає інтерфейс відокремленим від реалізації.

Висновки до розділу

У першому розділі роботи розглянуто теоретичні засади та проведено огляд предметної області, пов'язаної з підвищенням продуктивності проектів розробки програмного забезпечення. Було проведено опис предметної області дослідження. Було проведено аналіз специфіки сучасної розробки програмного забезпечення, зокрема, акцент зроблено на командній роботі в ІТ-проектах, яка є одним із важливих елементів успіху проектів. Підкреслено роль скоординованого підходу та важливість взаємодії між учасниками команд у рамках методологій розробки.

Визначено особливості командної розробки програмного забезпечення. Розглянуто різні методи організації командної роботи, зокрема, було детально досліджено гнучкі методології розробки, такі як Agile. Виявлено, що такі підходи сприяють підвищенню продуктивності завдяки адаптивності до змін, швидкому циклу зворотного зв’язку та тісній взаємодії з клієнтами.

Досліджено інструменти для розробки. Проаналізовано важливість сучасних інструментів для ефективної командної розробки. Визначено, що використання спеціалізованих інструментів, таких як системи управління

проєктами, середовища інтегрованої розробки та засоби автоматизації, позитивно впливають на продуктивність команд.

Розглянуто Git-репозиторій для контролю версій. Обговорено роль Git як ключового інструменту для контролю версій у розробці програмного забезпечення. Зазначено, що використання Git забезпечує ефективне управління змінами в коді, співпрацю між розробниками та надає можливість відстежувати історію проєкту. Визначено засоби та методи перевірки якості коду. Важливою частиною розробки є забезпечення високої якості коду. У дослідженні підкреслено значення автоматичних тестів, кодування стандартів, аналізаторів коду та інших методів забезпечення якості програмного забезпечення.

Представлено інструменти для проектування на основі моделей, які допомагають формалізувати та автоматизувати процес розробки. Особливу увагу приділено інструментам для моделювання архітектури програмних систем, що сприяють підвищенню структурованості та продуктивності розробки. Описано роль правильної архітектури та організації компонентів системи у забезпеченні її масштабованості, підтримуваності та продуктивності.

Таким чином, висновки першого розділу свідчать про важливість комплексного підходу до розробки програмного забезпечення, що включає командну роботу, застосування сучасних методологій, інструментів і технологій для підвищення продуктивності та якості кінцевого продукту.

РОЗДІЛ 2. ОСОБЛИВОСТІ ВИЗНАЧЕННЯ НАБОРІВ ДАНИХ ДЛЯ ПОБУДОВИ МЕТОДОЛОГІЇ АНАЛІЗУ ПРОДУКТИВНОСТІ РОЗРОБКИ

2.1. Представлення методики та інструментів збору даних проекту

2.1.1. Команди розробників

Переважно групи розробників програмного забезпечення зосереджені навколо конкретного апаратного забезпечення або функціональності. Для аналізу ми зберемо дані про 24 команди. Усі ці команди працюють над серверними компонентами, щоб зберегти різницю між тим, над чим працюють команди, на вони показані для кожного під компонента, що показано на рисунку 3.1 . Вибір лише команд серверів виключає з аналізу групи додатків вищого рівня та інтерфейсу користувача, оскільки вони дуже відрізнялися за кількістю помилок і функціями, які вони впроваджували.

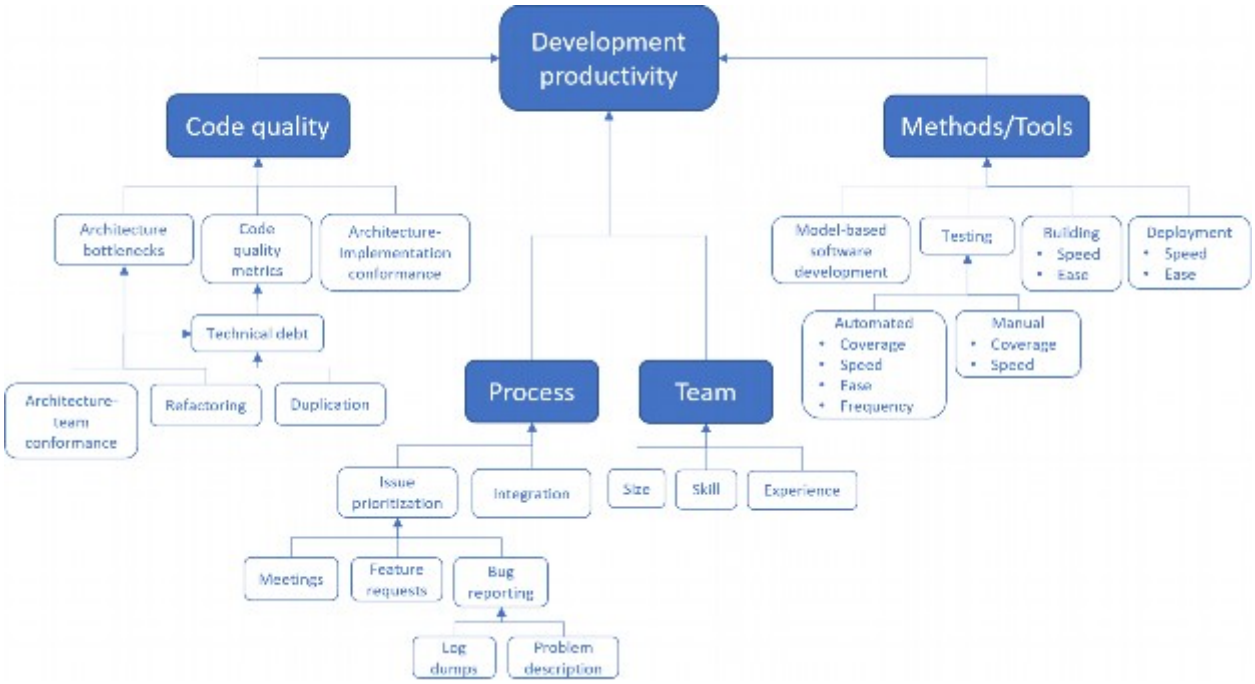

Рис. 2.1. Фактори, що впливають на швидкість розробки

В таблиці 2.1. показано ранжування команд за підсистемами розробки.

Таблиця 2.1. Команди, упорядковані за підсистемами

Sub-system	Team
SA	*TA, TB, TC, TG, TO, TT, TY*
SB	*TD, TE, TN, TP*
SC	*TF, TI, TJ, TK, TL, TM, TR, TV*
SD	*TH, TQ, TU*
SE	*TS, TX*

2.1.2. Репозиторії коду для розробки

Репозиторії коду для розробки — це сховища, де програмісти зберігають, керують та обмінюються своїм кодом. Вони є важливою частиною процесу розробки програмного забезпечення і зазвичай використовують системи контролю версій для відстеження змін у коді. Основні функції репозиторіїв коду включають:

- Зберігання коду: Можливість зберігати вихідний код проєкту, включаючи всі його версії та гілки.

- Контроль версій: Відстеження історії змін у коді, що дозволяє повернутися до попередніх версій у разі потреби.

- Спільна робота: Декілька розробників можуть одночасно працювати над одним проєктом, вносячи зміни та коментуючи роботу інших.

- Інтеграція з іншими інструментами: Репозиторії часто інтегруються з іншими інструментами розробки, такими як CI/CD (безперервна інтеграція/доставка) для автоматизації тестування та розгортання.

Популярні сервіси для зберігання репозиторіїв коду:

- GitHub: Один з найпопулярніших сервісів для зберігання коду з підтримкою Git, що використовується як для приватних, так і для публічних проєктів.

- GitLab: Подібний до GitHub, але з додатковими можливостями для CI/CD.

- Bitbucket: Сервіс, що підтримує як Git, так і Mercurial, з акцентом на корпоративні команди.

В даному випадку, обрані команди виконують більшість своїх зобов'язань у 74 сховищах коду. Близько 75 репозиторіїв пропущено, оскільки вони мають дуже мало комітів, є сховищами інтерфейсу, які рідко змінюються, або є застарілими компонентами. Вони були пропущені через те, що вони додавали значних витрат часу на пошук даних.

2.1.3. Терміни релізів

Реліз у контексті розробки програмного забезпечення – це процес випуску нової версії програмного продукту, яка стає доступною для користувачів. Це кульмінація довгого процесу розробки, тестування та вдосконалення.

Реліз включає в себе:

- Нові функціональні можливості: Додавання нових функцій, які розширюють можливості програми.

- Виправлення помилок: Усунення виявлених помилок та багів, що впливали на роботу програми.

- Оптимізація: Покращення швидкодії, стабільності та безпеки програми.

- Зміни в інтерфейсі: Оновлення дизайну та структури користувацького інтерфейсу для більшої зручності.

Існують наступні типи релізів:

- Major release: Великий реліз з кардинальними змінами, новими функціями та можливостями.

- Minor release: Менший реліз, який фокусується на виправленні помилок та незначних доповненнях.

- Patch release: Дуже невеликий реліз, призначений для швидкого усунення критичних помилок.

Реліз – це важлива віха в життєвому циклі програмного продукту. Він дозволяє доставляти цінність користувачам, збирати зворотний зв'язок та підтримувати конкурентноспроможність. Ефективне управління процесом релізу є ключовим фактором успіху будь-якого програмного проекту.

Аналізується робота команд протягом семи випусків. Це робиться для того, щоб аналіз відповідав поточному робочому процесу, а також тому, що наприкінці проекту року всі команди почали позначати свої зобов’язання із відповідним ключем проблеми Jira.

Спочатку проаналізований проміжок часу який було розбито на 3 місяці, але через частоту випусків, яка впливає на час здійснення комітів, ці періоди призвели до занадто великої мінливості продуктивності команд. Тому було вибрано узгодити періоди часу з періодом розробки випуску та додати близько одного місяця до початку та кінця початкового періоду часу.

2.1.4. Представлення помилок в Jira

Jira – це потужний інструмент для відстеження завдань та управління проектами, який широко використовується для обліку та управління помилками програмного забезпечення. Він пропонує гнучку конфігурацію та різноманітні функції, що дозволяють ефективно керувати процесом виявлення, фіксації та усунення помилок.

Алгоритм використання Jira для обліку помилок є наступним:

1. Створення задач: Кожна виявлена помилка реєструється як окрема задача зі своїм унікальним ідентифікатором.

2. Присвоєння атрибутів: Кожній задачі присвоюються детальні атрибути, такі як:

- Тип помилки: Баг, дефект, пропозиція покращення тощо.

- Пріоритет: Визначає терміновість виправлення.

- Статус: Відображає поточний етап життєвого циклу помилки (нова, у роботі, перевірена, закрита).

- Причина: Короткий опис причини виникнення помилки.

- Кроки відтворення: Послідовність дій, які необхідно виконати для відтворення помилки.

- Очікуваний результат: Опис того, що повинно відбуватися за нормальних умов.

- Фактичний результат: Опис того, що відбувається насправді.

- Прикріплені файли: Скріншоти, лог-файли та інша довідкова інформація.

3. Призначення відповідальних: Кожній задачі призначається виконавець, відповідальний за виправлення помилки.

4. Створення гілок: Для кожної помилки можна створити окрему гілку в системі контролю версій (наприклад, Git), що дозволяє ізольовано працювати над її усуненням.

5. Інтеграція з іншими інструментами: Jira легко інтегрується з іншими інструментами розробки, такими як системи контролю версій, системи безперервної інтеграції та іншими.

6. Створення робочих процесів: За допомогою конфігурації робочих процесів можна визначити різні стани, переходи між ними та автоматичні дії, які виконуються при зміні стану задачі.

7. Звіти та аналітика: Jira дозволяє створювати різноманітні звіти, які допомагають оцінити ефективність процесу управління помилками, виявити тренди та визначити області для покращення.

Переваги використання Jira для обліку помилок:

- Централізоване управління: Всі помилки зберігаються в одному місці, що полегшує їх відстеження та управління.

- Прозорість: Кожен член команди може бачити поточний стан кожної помилки.

- Автоматизація: Багато рутинних завдань, таких як перехід між станами, можна автоматизувати.

- Інтеграція: Легко інтегрується з іншими інструментами розробки.

- Гнучкість: Може бути налаштована під потреби будь-якого проекту.

Комбінація команд та часових інтервалів призвела до виникнення майже 30 тисяч помилок у Jira, які були або створені, або вирішені протягом зазначеного періоду. Після фільтрації клонів, дублікатів та врахування всіх задач, пов'язаних з іншими задачами як єдине ціле, залишається 16509 окремих задач, з яких 14472 були створені, а 13973 — вирішені протягом зазначеного часу. Клони виключаються через те, що деякі команди дублюють задачу, коли зміни необхідні в кількох репозиторіях. Для коректного підрахунку витрачених зусиль, час, витрачений на клони, враховується при обчисленні часу, витраченого на оригінальну задачу. Якщо задачі пов'язані між собою (наприклад, через тип посилань "дублюється" або "пов'язано з"), весь набір задач розглядається як одна задача при пошуку відповідних комітів. Це явище не є поширеним для функціональних особливостей, однак для помилок може існувати кілька історій і підзадач, створених для виправлення однієї й тієї ж помилки.

2.1.5. Етап збору даних

Щоб вирішити, про що збирати дані, можливі впливи на продуктивність розробки зібрані на рисунку 2.1. Фактори продуктивності також описані в [9], але обмежуються тими, які не вимагають використання опитувань, виключаючи такі фактори, як мотивація та якість управління. Чотири основні фактори, які гіпотетично впливають на продуктивність:

- Якість коду;
- Процес;
- Характеристика команди;
- Методи та інструменти.

На ці основні фактори додатково впливають більш конкретні аспекти, дані про які ми можемо збирати, які пов’язані з основними аспектами на рисунку 2.1. Було неможливо зібрати дані про всі конкретні аспекти,

оскільки деяка інформація недоступна, не доступна для всіх команд, або її неможливо було зібрати протягом періоду часу проекту, тобто відповідність архітектури, ручне та автоматичне тестове покриття та час побудови. Тестування на основі моделі використовувалося лише однією командою, тому подальше дослідження не проводилося. Кожен період часу в межах проміжку часу має власний набір значень, отже, для кожної команди існує п’ять наборів значень.

2.2. Опис характеристик даних, що впливають на продуктивність розробки

Нижче наведено огляд найважливіших полів даних, для яких збирається інформація.

Кількість помилок і функцій

Кількість помилок і функцій на команду протягом періоду, які мають принаймні одну пов’язану фіксацію.

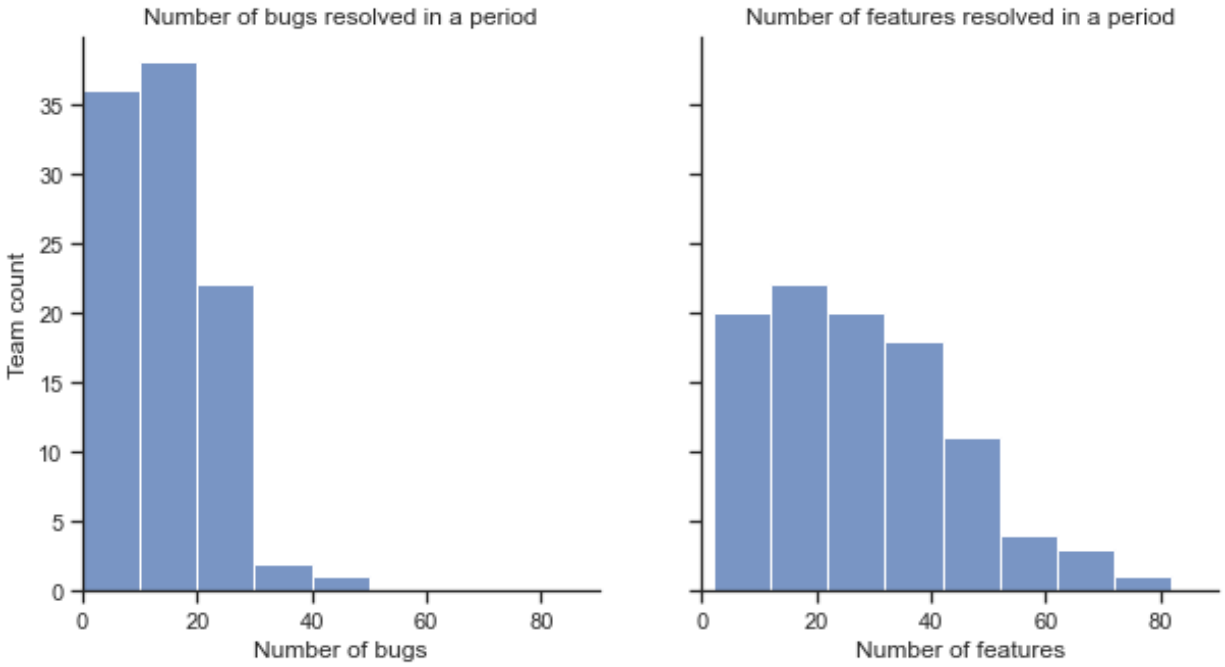

Рис. 2.2. Кількість виправлених помилок і функцій за період розподілу

Рисунок 2.2 показує, що майже всі команди вирішують від 0 до 30 помилок протягом періоду, за кількома винятками в діапазоні від 30 до 50 помилок. Кількість функцій, реалізованих протягом періоду, здебільшого становить від 0 до 50, при цьому кількість пар команда-період постійно зменшується зі збільшенням кількості функцій. Крім того, для функцій є кілька винятків на високому рівні в діапазоні від 50 до 80.

Час помилки та час функції

Кількість робочих днів, витрачена на усунення помилок і функцій. Це сумарний час, протягом якого питання мають статус «виконується». Як показано на рисунку 2.3, кількість днів перевищує кількість робочих днів у періоді розробки, оскільки кілька розробників працюють паралельно, і розробник також може одночасно виконувати кілька завдань. На впровадження функцій витрачається більше часу, ніж на помилки, причому час, витрачений на функції, розподіляється від 0 до 400 робочих днів, а час, витрачений на помилки, зазвичай становить від 0 до 160 робочих днів.

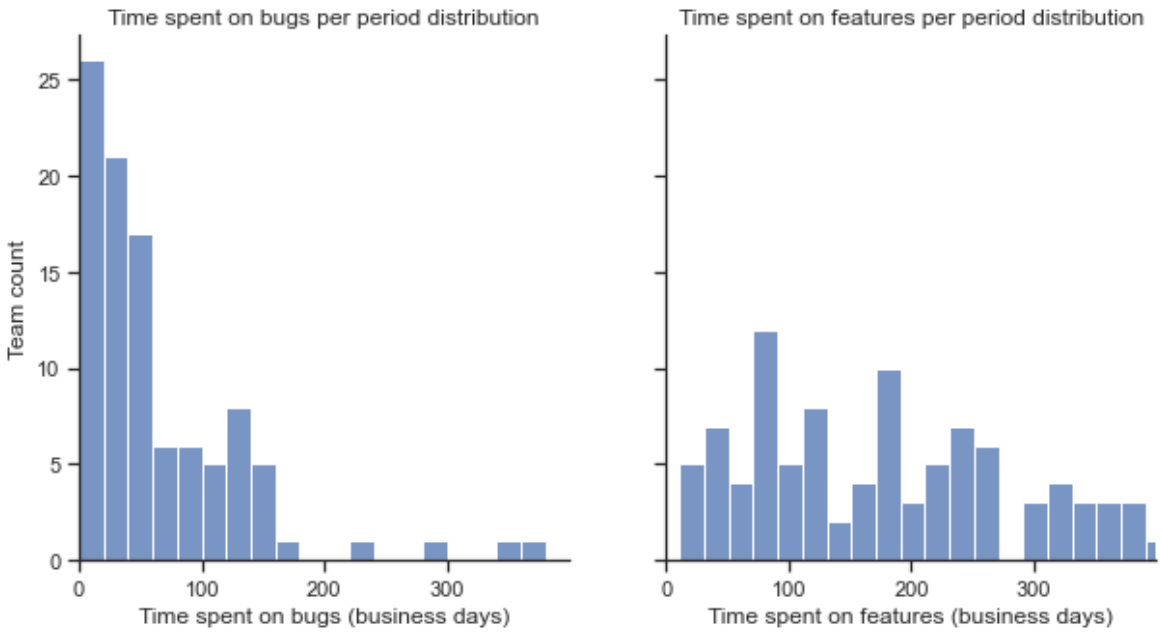

Рис. 2.3. Розподіл часу, витраченого на помилки та функції за періодами

Для якості коду використовуються значення QI (quality indicator), і ми використовуємо рядки коду для кількісного визначення розміру проекту.

Показник QI для команди протягом періоду – це середній показник QI сховища, над яким вони працюють протягом цього періоду, оскільки команди не працюють в одному сховищі. Це середньозважене значення відповідно до кількості комітів, зроблених у кожному сховищі. Поряд із загальним показником QI обчислюється середньозважене значення кожного показника, що впливає на загальний бал.

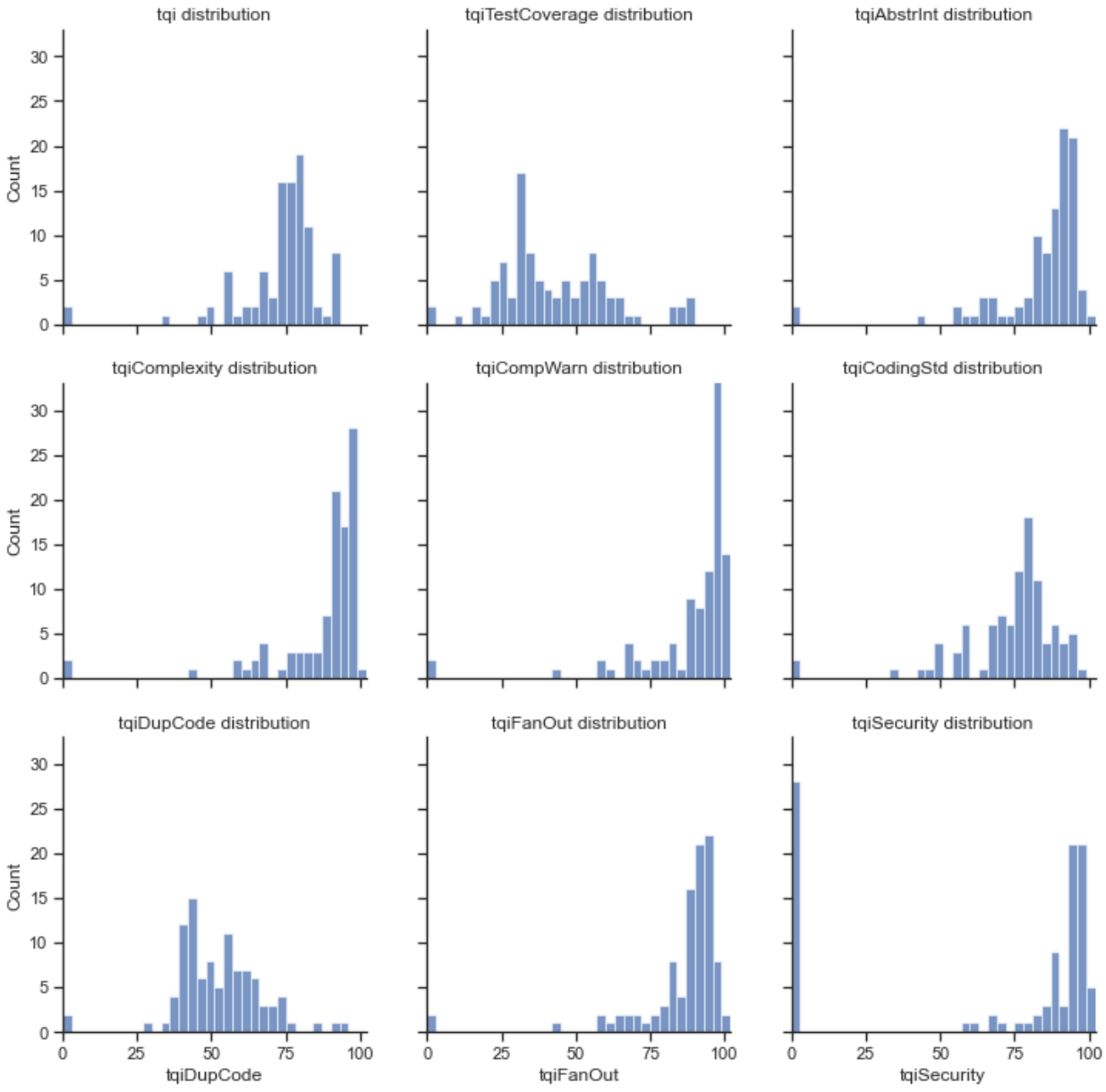

Рис. 2.4. Розподіл показників quality indicator

На рисунку 2.4 показано розподіл кожного показника QI, усі значення знаходяться в діапазоні від 0 до 100. Усі показники знаходяться на верхньому

кінці шкали, за винятком охоплення тестом, показників дубльованого коду та показника безпеки. Ці три бали є тими, які найбільше відрізняють команди з точки зору якості коду, але часто значення тестового покриття та безпеки є неточними, і тому їх не слід брати до уваги. Найнижчі показники QI – це значення з перших кроків у наборі даних, оскільки керівництво змушує всі команди досягти певного рівня якості, що призводить до збіжності показників TQI з часом, як показано на рисунку 2.5.

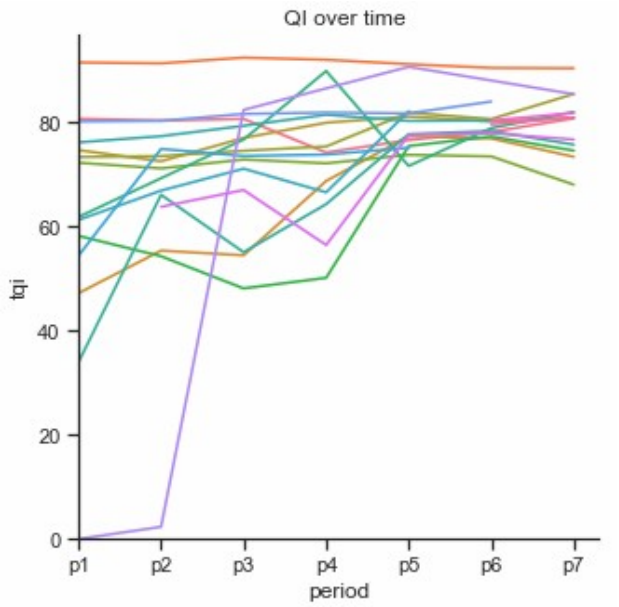

Рис. 2.5. Зміна показників QI з часом

LoC (кількість рядків коду, включаючи порожні рядки та коментарі), ELoC (кількість рядків коду, виключаючи порожні рядки та коментарі) та GLoC (кількість рядків коду, включаючи згенерований код) є зваженими середніми значеннями для репозиторіїв, над якими працює команда, відповідно до кількості комітів, здійснених у кожному з них. На рисунку 3.6 представлено розподіл ELoC, де спостерігається розмежування між командами, що працюють над підсистемою SA, які мають високе середнє значення ELoC, та іншими підсистемами, для яких кількість ELoC є значно меншою.

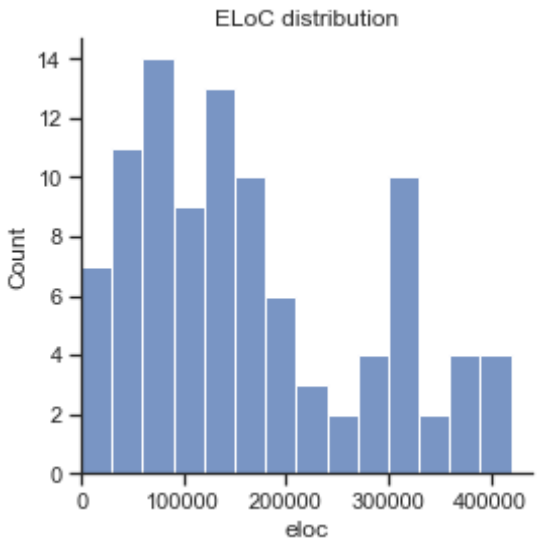

Рис. 2.6. Розподіл ELoC. Середні рядки коду сховища, над якими працює команда

Кількість проблем із певною класифікацією відставання може вказувати, яким типам функцій команда надає пріоритет.

Усім проблемам функцій присвоюється одна з чотирьох класифікацій відставання: бізнес-цінність, підтримка, технічні інновації або технічний борг. Кожна класифікація відставання має власне поле в наборі даних, яке дорівнює кількості проблем із пов'язаною фіксацією за період часу з цією класифікацією відставання, поділеною на загальну кількість проблем із фіксацією за період часу. На рисунку 2.7 показано, що дуже мало зобов'язань спрямовано на надання підтримки, причому більшість команд мають від нуля до п'яти відсотків своїх зобов'язань для цього значення відставання. Наступне значення відставання з найменшим числом commits — це технічний борг, причому більшість команд здійснюють зміну коду від 0 до 15% часу для усунення технічного боргу. Другим найбільш відданим значенням відставання є технічні інновації, оскільки більшість пар командних періодів вкладають від 0 до 20% своїх функцій для цього значення відставання. Найбільша вартість відставання – це бізнес-цінність, команди витрачають до 80% своїх зобов'язань на цю вартість відставання.

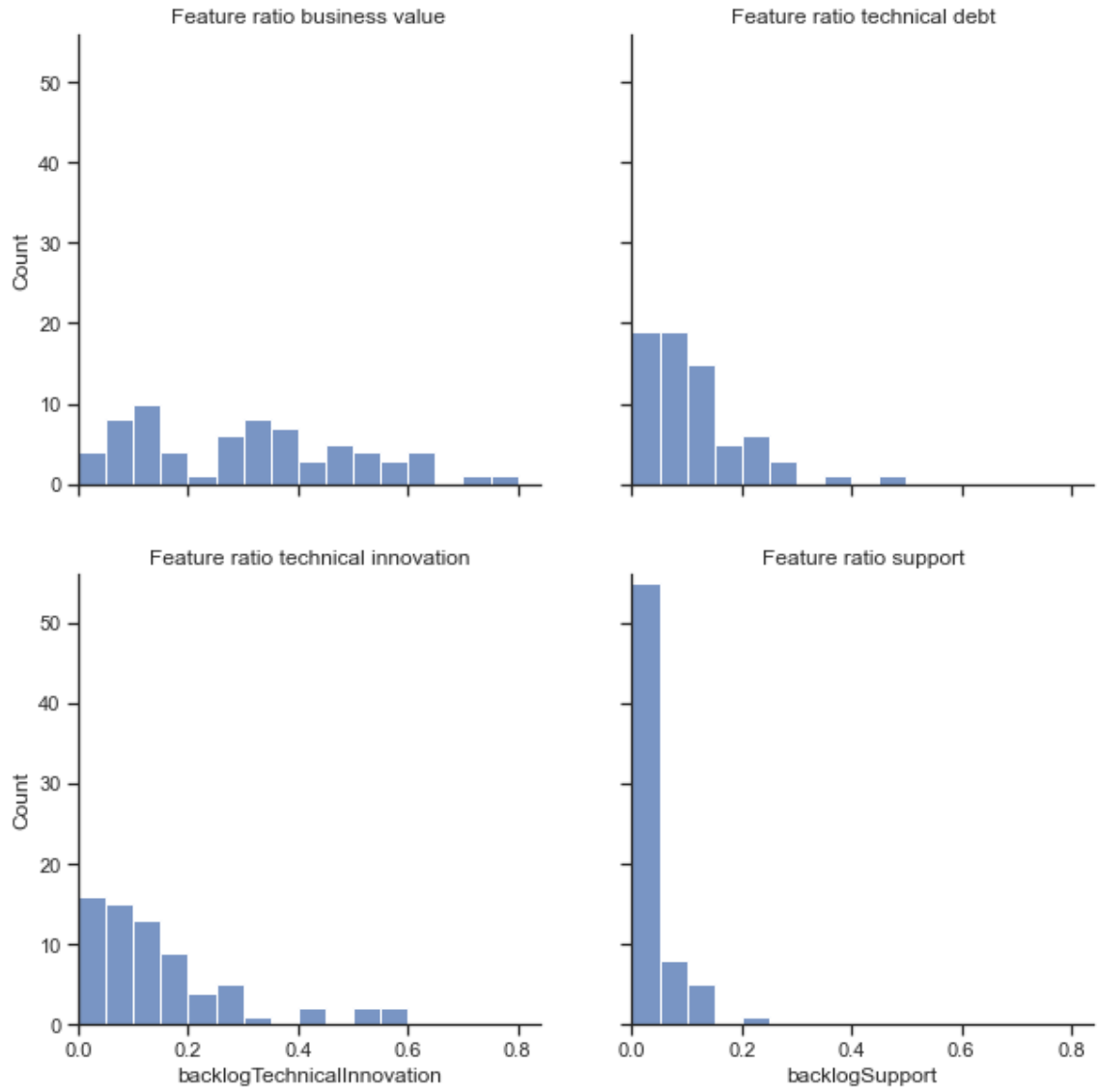

Рис. 2.7. Структура беклогу за категоріями

Ми збираємо кілька показників розміру команди та обчислюємо середній досвід команди, щоб кількісно визначити рівень її кваліфікації.

Значення досвіду відображає середній час у днях між кожним комітом і першим комітом того самого автора в одному сховищі. На відміну від інших полів у наборі даних, кількість днів включає вихідні та святкові дні. На рисунку 2.8 показано розподіл досвіду пар команда-період. Найменш досвідчені команди мають середній досвід близько одного року, тоді як найбільш досвідчені команди мають в середньому три з половиною роки досвіду.

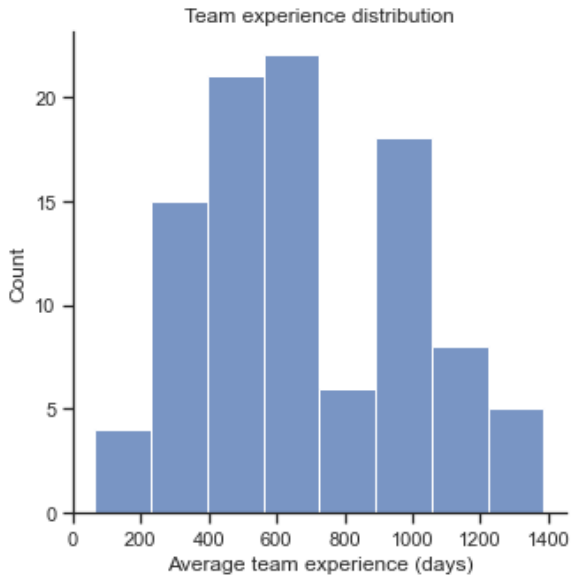

Рис. 2.8. Розподіл досвіду

Committers – це термін, який найчастіше перекладається як комітери або автори комітів. Коммітер – це людина, яка вносить зміни до програмного коду та зберігає ці зміни в репозиторії контролю версій.

Кількість унікальних авторів комітів, які зробили щонайменше три коміти за цей період часу. На рисунку 2.9 ми бачимо, що команди складаються з двох до тринадцяти людей, причому більшість команд складаються з п'яти або шести людей.

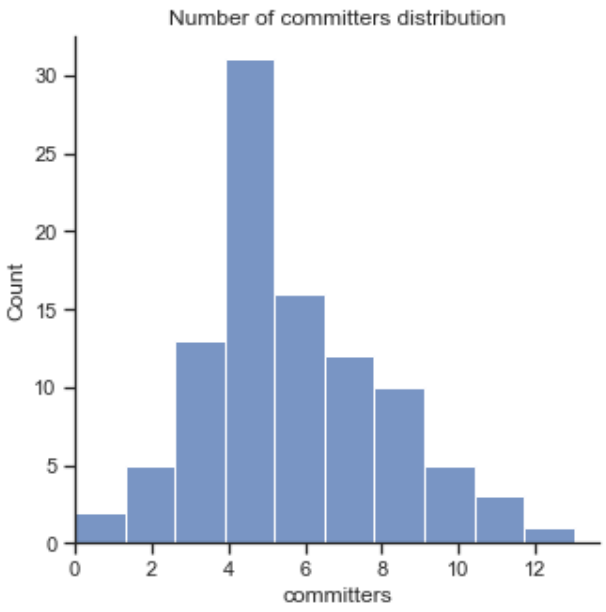

Рис. 2.9. Розподіл кількості комітетів

Для використовуваних методів і інструментів збирається лише інформація про розробку програмного забезпечення на основі моделі. Розробка програмного забезпечення на основі моделі

Поле розробки на основі моделі містить число від 0 до 1, що відображає відсоток комітів, у яких редагується модель Dezyne.

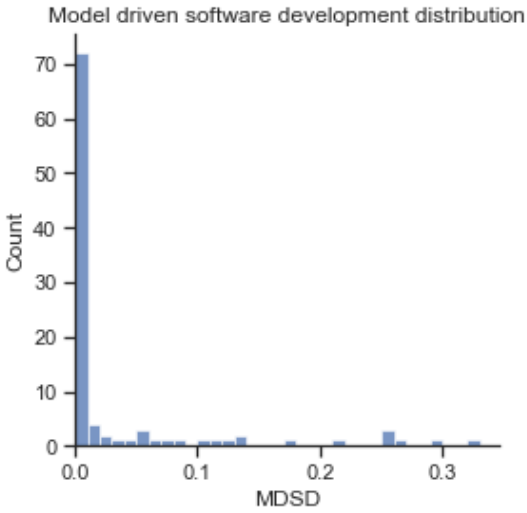

Рис. 2.10. Розподіл моделей в модельно-орієнтованій розробці програмного забезпечення

Рисунок 2.10 показує, що більшість команд не використовують MDSD (Model Driven Software Development), а команди, які редагують моделі, редагують їх до третини своїх комітів. Це вказує на те, що значна частина їх роботи над програмним забезпеченням відбувається в рукописних файлах, і це відповідає тому, що Dezyne не можна застосовувати для всіх випадків використання.

2.3. Структура завдань в Jira

З початкового дослідження в Jira ми знаємо, що команди здебільшого використовують типи епічних, сюжетних, помилкових та підзавдань. Щоб знати, чи використовується також тип проблеми із завданням, і формувати

запити щодо зв’язків між проблемами Jira. Jira надає веб-інтерфейс, з яким можна взаємодіяти за допомогою запитів, подібних до SQL, які називаються запитами Jira Query Language (JQL).

```
type = Task order by created DESC
```

Команда TZ, TTA і TTB є проектами TEM, які використовують завдання типу проблеми, але всі команди в остаточному наборі даних не використовують цей тип.

```
type = Sub-task and parent in (EMPTY) order by created DESC
```

Наведений вище запит дає результати, які належать одній команді, але ця команда не включена в набір даних. Це вказує на те, що всі проблеми типу підзавдання в наборі даних мають батьківські завдання і не є окремими завданнями.

Щоб детальніше пояснити зв’язок між проблемами, ми обговоримо зв’язки між помилками та функціями окремо в наступних розділах.

2.3.1. Структура помилки

Bug Structure (структура помилки) в Jira — це організаційна модель, яка допомагає керувати та відстежувати помилки в програмному забезпеченні. Структура багу містить кілька основних елементів:

- Заголовок (Summary): Короткий опис помилки, який має передавати суть проблеми, щоб її було легко ідентифікувати серед інших багів.

- Опис (Description): Деталізований опис помилки, що включає:

  - Очікувану поведінку (expected behavior).

  - Фактичну поведінку (actual behavior).

- Кроки для відтворення помилки (steps to reproduce).

- Інформацію про середовище (версія програмного забезпечення, операційна система, браузер тощо).

- Пріоритет (Priority): Вказує на важливість виправлення багу, зазвичай у межах таких категорій: Low, Medium, High, Critical.

- Стан (Status): Поточний стан помилки. Типові статуси можуть включати:

- Open (відкрито) — помилка зареєстрована, але ще не вирішена.

-In Progress (в процесі) — робота над виправленням ведеться.

- Resolved (вирішено) — помилку виправлено, але ще не перевірено.

- Closed (закрито) — помилку перевірено та остаточно виправлено.

- Reopened (перевідкрито) — помилка знову відкрита після перевірки.

- Призначений виконавець (Assignee): Особа або команда, відповідальна за виправлення помилки.

- Звітник (Reporter): Особа, яка зареєструвала баг і виявила помилку.

- Коментарі (Comments): Простір для обговорення помилки, де користувачі можуть залишати додаткові відомості, запитувати інформацію або обговорювати можливі рішення.

- Тип посилань (Issue Links): Вказує на відносини помилки з іншими задачами або багами, наприклад, "дубль" (duplicate), "залежить від" (depends on) або "пов'язано з" (relates to).

- Прикріплені файли (Attachments): Файли, які можуть допомогти в діагностиці або відтворенні помилки, наприклад, скріншоти, лог-файли, відео.

- Підзадачі (Sub-tasks): Окремі задачі, які можуть бути створені в рамках багу для полегшення його виправлення або розбиття на дрібніші частини.

- Компоненти (Components): Частини програмного забезпечення або модулі, до яких стосується помилка.

- Мітки (Labels): Ключові слова або теги, що дозволяють швидше знайти або відфільтрувати баги.

- Версії (Versions):

- Affects Version: Вказує, в якій версії програми виникла помилка.

- Fix Version: Версія, в якій помилка буде або вже була виправлена.

Ця структура дозволяє організовано і чітко управляти процесом виявлення, виправлення і перевірки помилок в Jira, забезпечуючи прозорість та ефективність роботи команд.

Команди діють на основі звітів про помилки подібним чином. Спочатку помилка досліджується, що включає оновлення опису з додатковою інформацією про те, як відтворити помилку, оцінку часу, необхідного для її виправлення, а також визначення, які програмні компоненти вона зачіпає. Після завершення дослідження рада з контролю змін (CCB) команди вирішує, чи слід виправляти помилку або відхилити її. Основними причинами для відхилення є те, що помилка більше не є актуальною, її виправлення вже в процесі, або ж це очікувана поведінка системи. Якщо помилка не відхилена, вона отримує виправлення, а посилання на коміти та запити на злиття додаються на сторінку помилки в Jira. Після цього помилці надається статус тестування для перевірки її виправлення, а згодом — статус "виконано".

Деякі команди дотримуються вищезгаданого процесу, документуючи всі дії в оригінальному звіті про помилку, але є також команди, які створюють окремі задачі для дослідження та вирішення. Наприклад, команди TA, TI та TJ створюють окрему задачу для дослідження, але при цьому оновлюють опис початкового звіту про помилку. TI та TJ також створюють одну або більше задач для вирішення, залежно від розміру помилки. TA, однак, використовує оригінальний звіт про помилку для відстеження ходу виправлення. У деяких випадках, коли обсяг роботи для виправлення

помилки є незначним, помилку виправляють під час дослідження. Також є команди, які створюють клон звіту про помилку для кожного репозиторію, де потрібні зміни. Типи посилань на задачі, такі як "розділено на" або "клон", використовуються для позначення зв'язків між помилкою та іншими задачами, пов'язаними з дослідженням або вирішенням. Різні типи посилань враховуються, якщо пов'язані задачі починаються зі слів "виправити" або "дослідити". Усі підзадачі, пов'язані із задачами, що стосуються помилки, також вважаються частиною процесу її дослідження або вирішення.

Як видно з прикладу на рисунку 2.11, баги також можна пов’язати з епіками. Ми не будемо використовувати цей зв’язок, оскільки ці епіки не пов’язують пов’язані помилки, а зазвичай використовуються для групування всіх помилок, вирішених у процесі розробки.

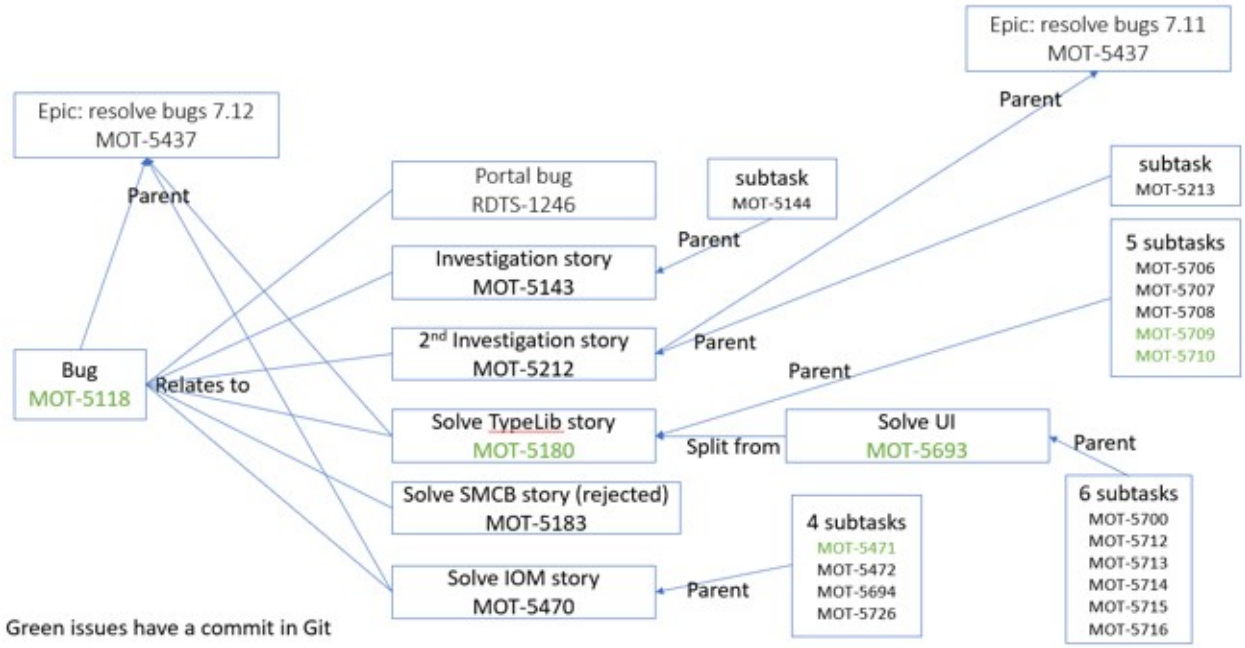

Рис. 2.11. Приклад помилки та всі пов’язані з нею проблеми

Висновки до розділу

У цьому розділі було детально розглянуто методику збору даних для аналізу продуктивності процесу розробки програмного забезпечення. На основі проведеного аналізу можна зробити такі висновки:

Роль команд розробників у процесі збору даних: Для ефективного аналізу продуктивності важливо враховувати специфіку організації роботи команд. Кожна команда має свої підходи до вирішення проблем, зокрема, у Jira деякі команди розбивають завдання на підзадачі або створюють окремі історії для дослідження і вирішення помилок. Це дозволяє більш детально структурувати робочий процес, однак ускладнює загальний аналіз, оскільки потрібно враховувати різні підходи до документування дій.

Репозиторії коду є ключовим джерелом інформації для оцінки обсягів виконаної роботи. Важливими показниками є LoC, ELoC та GLoC, які дозволяють оцінити продуктивність розробників на основі кількості рядків коду, включаючи чи виключаючи коментарі та згенерований код. Оскільки обсяг внесків команди в кожен репозиторій впливає на зважену середню кількість рядків коду, важливо враховувати всі внески при оцінці загальної продуктивності.

Продуктивність розробки тісно пов'язана з етапами випуску нових версій програмного забезпечення. Релізні цикли визначають як короткострокові, так і довгострокові цілі, що спрямовують діяльність команд. Терміни релізів також визначають межі для збору і аналізу даних про виконану роботу.

Структура багів у Jira визначає, як збираються і обробляються дані про помилки. Процес від дослідження багу до його виправлення та перевірки включає кілька етапів, що документуються в системі. Деякі команди об'єднують схожі задачі для полегшення пошуку та обробки пов'язаних комітів, інші — створюють окремі завдання для кожного репозиторію, де потрібні зміни. Такий підхід дає можливість гнучко організовувати робочий процес, але потребує уважного аналізу для правильної оцінки продуктивності.

Характеристики даних, що впливають на продуктивність: Дані, зібрані з різних систем, таких як Jira та репозиторії коду, повинні бути проаналізовані комплексно. Зокрема, слід враховувати кількість і типи

завдань, складність їх вирішення, а також ресурси, які були залучені для виконання кожної задачі. Важливим аспектом є розподіл задач за різними підсистемами, оскільки це впливає на різні показники продуктивності.

Загалом, збір та аналіз даних з різних джерел, таких як Jira і репозиторії коду, дозволяє отримати повнішу картину процесу розробки і точніше оцінити продуктивність команд. Проте різні підходи до організації завдань, структур багів і методів розробки потребують детального аналізу для коректної інтерпретації результатів.

РОЗДІЛ 3. ПРЕДСТАВЛЕННЯ ПРОЦЕСІВ ТА МЕТОДОЛОГІЇ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Моделювання робочого процесу командної розробки

Для моделювання робочого процесу команд обрано нотацію мережі Петрі [1]. У цьому позначенні кола позначають місця, які можна розглядати як стан, а квадрати позначають переходи, дії, які можна виконувати з місця. Стрілки з'єднують місця з переходами і навпаки, вказуючи, які переходи можна зробити з місця, а до яких місць можна дістатися з переходу.

Місця представляють стани, в яких є проблеми Jira. Їх можна реконструювати з журналу історії, пов’язаного з проблемою Jira. Переходи — це дії, які зберігаються в журналі історії. Щоб зробити графіки робочих процесів простими, робочі цикли помилок і функцій переглядаються окремо, оскільки ці робочі цикли значно відрізняються. Щоб ще більше спростити графіки, ми будемо переглядати лише хід основної історії помилки або функції, оскільки включення пов’язаних проблем і підзавдань створює безлад. Крім того, відображаються не всі дії, виконані з історією помилки чи функції, оскільки для багатьох проблем виконується понад 100 дій у результаті автоматизованих повідомлень від програмного забезпечення, з яким інтегровано Jira. Дії, які не можна порівняти між командами, виключено, наприклад, мітки, додані до проблем, використовуються не всіма командами та відрізняються за назвами в різних командах, тому їх не враховано.

Для моделювання робочого процесу мережа Петрі ми називаємо алгоритм 1 і як вхідні дані надайте йому всі проблеми Jira, які ми хочемо мати в моделі. Цей алгоритм створює порожній графік і циклічно обходить задані проблеми та додає їх до робочого циклу за допомогою виклику

алгоритму 2. Нарешті, він викликає алгоритм 3 і повертає побудований графік.

Алгоритм 2 приймає проблему та графік як вхідні дані. Він отримає представлення стану проблеми та перевірить, чи це представлення вже має місце на графі мережі Петрі G. Якщо ні, він додасть його, після чого алгоритм перегляне всі дії в історії проблеми. Ці дії зберігаються у зворотному порядку, тобто першою остання дія. Він повертає дію над представленням s для отримання попереднього представлення стану s' і додає його до графа, якщо місце з представленням s' ще не існує. Якщо попереднє представлення s' ще не з'єднане з представленням s переходом з дією, перехід також додається до графа. Алгоритм оновлює s до поточного представлення s ' і переходить до наступної дії в історії. Коли досягається остання дія в циклі, тобто перша дія проблеми в хронологічному порядку, досягається початковий стан проблеми, і алгоритм повертає результуючий графік.

Algorithm 1 RecoverWorkflow

```
1: procedure RecoverWorkflow(issues)
2:     G ← empty graph
3:     for issue in issues do
4:         G ← AddIssueToWorkflow(issue, G)
5:     G ← CorrectWorkflow(G)
6:     return G
```

В кінці алгоритму 1 ми викликаємо алгоритм 3 через існування шляхів у моделі, які не спостерігаються в даних. Приклад показано на рисунку 3.1, де зображені шляхи двох задач — задачі номер один і задачі номер два. У верхній частині рисунка обидві задачі опиняються в одному і тому ж стані після виконання різних дій над ними, а потім над кожною з них виконується різна дія, яка переводить їх із подібного стану в різні стани. У моделі, створеній після рядків з другого по четвертий алгоритму 1, кожне зіткнення зі станом має одне місце в моделі. Це означає, що модель допускає шлях від

стану, в якому починається задача один, до стану, в якому закінчується задача два, хоча такий шлях ніколи не виникає в даних.

Algorithm 2 AddIssueToWorkflow

```
1: procedure AddIssueToWorkflow(issue, G)
2:     s ← getCurrentState(issue)
3:     if s not in G then
4:         add s to G
5:     for action in issue.history do
6:         s′ ← revert action on s
7:         if s′ not in G then
8:             add s′ to G
9:         if s′ not connected to s in G then
10:            add transition from s′ to s in G
11:        s ← s′
12:    return G
```

Тому для коректування моделі ми розділяємо місце, через яке проходять обидві задачі, на два окремі місця з однаковим представленням стану, що призводить до графа, показаного в нижній частині рисунка 3.1, і таким чином підвищує точність моделі, описаної в [30].

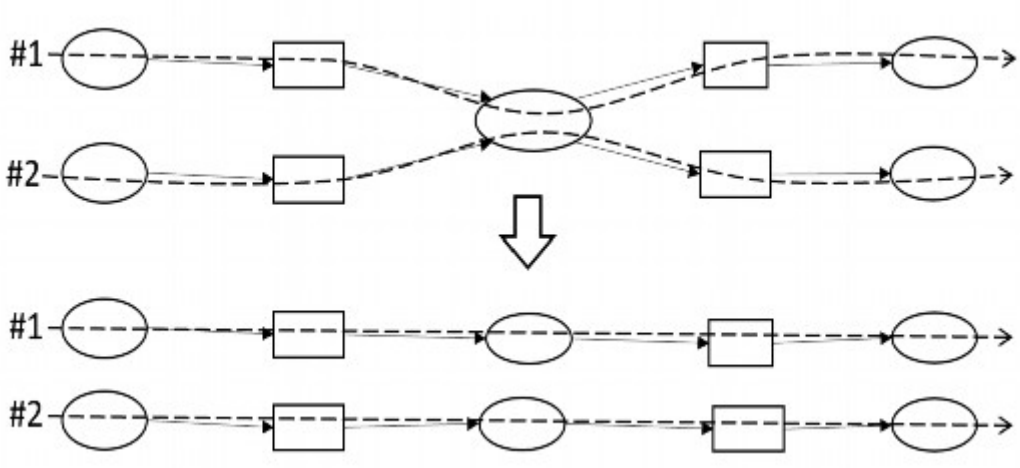

Рис. 3.1. Приклад шляху в моделі, який не спостерігається в даних

Щоб виправити ці неспостережувані шляхи в даних, алгоритм 3 перебирає всі місця на результуючому графіку алгоритму 1 рядки з другого по чотири та обчислює набір потужностей вихідних переходів. Цей набір потужностей сортується за збільшенням розміру, і набір без переходів

видаляється, а також набір усіх переходів, тому що некорисно відокремлювати жоден або всі переходи з місця. Алгоритм проходить цикл по кожному набору вихідних переходів у наборі потужності вихідних переходів OTPowerSet і отримує набір усіх вхідних переходів, через які проходять шляхи, які закінчуються в одному з переходів, включених до набору вихідних переходів. Якщо всі шляхи, що проходять через вхідні переходи, збігаються з тими, що виходять через вихідні переходи, перехід(и) відокремлюється від місця. Проблеми, які проходять через місце, не розглядаються, а радше шляхи проблеми, оскільки проблеми можуть проходити через місце кілька разів. Якщо в місці залишився лише один вхідний або вихідний перехід або набір вихідних переходів дорівнює кількості вихідних переходів, більше переходів не буде розділено. Оскільки попередньо відокремлені переходи можуть бути в інших наборах набору потужностей, алгоритм переходить до наступного елемента набору потужностей, якщо ці переходи більше не пов’язані з місцем у рядку 9. Виклик splitPlace у рядку 14 створює дублікат місце, передає в нього вхідні та вихідні переходи та обробляє дублювання або переміщення власних переходів.

Algorithm 3 CorrectWorkflow

```
1:  procedure CorrectWorkflow(G)
2:      for place in G do
3:          OTPowerSet ← powerSet(place.outgoingTransitions) sorted by ascending size
4:          remove set of all transitions and set of no transitions from OTPowerSet
5:          for outTransitions in OTPowerSet do
6:              if |place.incomingTransitions| <= 1 or |place.outgoingTransitions| <= 1 then
7:                  break
8:              if not all outTransitions in place.outgoingTransitions then
9:                  continue
10:             if |outTransitions| = |place.outgoingTransitions| then
11:                 break
12:             inTransitions ← place.incomingTransitions t where every t.paths.outTransition
    in outTransitions
13:             if inTransitions.paths = outTransitions.paths then
14:                 splitPlace(place, inTransitions, outTransitions)
15:     return G
```

Щоб включити більше проблем у графік, повернутий алгоритмом 1 , зміни, внесені алгоритмом 3, спочатку повинні бути скасовані, що означає,

що всі місця з однаковим представленням стану об’єднуються. Це пов’язано з тим, що алгоритм 2 має як передумову унікальність усіх представлень стану місць у вхідному графі G. Після додавання додаткових проблем алгоритм 3 можна знову викликати на G для видалення шляхів, які не спостерігаються в даних.

Для визначення станів проблеми враховуються такі поля; тип проблеми, вирішення, статус, правонаступник, версія виправлення, пов’язані коміти та поля проекту. Не всі значення цих полів безпосередньо беруться як представлення стану, для деяких полів перевіряється, чи вони заповнені та скільки значень вони містять. Наприклад, для типу проблеми, який береться безпосередньо, але для поля версії виправлення підраховується кількість заповнених версій. Якщо представлення двох станів однакове, стани вважаються рівними в нашій моделі робочого процесу. Це призводить до того, що певні дії не переводять проблему в інший стан, тому в робочому процесі також відбуваються самопереходи, наприклад, коли оновлюється опис проблеми. Усі дії в результаті зміни полів, не врахованих для представлення стану, призводять до самопереходу.

Причиною впровадження цього алгоритму на основі стану для відновлення робочого циклу є те, що заздалегідь не було відомо, чи будуть переплетені процеси під час відновлення моделі процесу для набору проблем. Державна реконструкція дозволяє мати кілька початкових і кінцевих місць. Також співробітники мають сторінки з номерами в Jira, де вони можуть переглядати проблеми, які знаходяться в певному стані, прикладами є сторінки, на яких показано всі проблеми в поточному спринті, або сторінка, на якій показано лише проблеми, призначені певній особі. Реконструкція процесу на основі стану може показати, коли проблеми втрачають увагу співробітників у переглядах Jira.

Щоб отримати уявлення про те, хто виконує яку дію, до переходів додається колір. Якщо дію завжди виконує одна роль у проекті, перехід має один колір, якщо дію виконує кілька ролей, кольори переходу пропорційні

кількості разів, коли кожна роль виконувала дію. Пояснення того, який колір до якої ролі належить, показано на рисунку 3.2.

Рис. 3.2. Представлена легенда акторів

В якості алгоритму компонування використовується алгоритм компонування CoSE-Bilkent [10]. Після створення макета необхідно вручну переставити місця та переходи, щоб зробити графіки більш читабельними. Удосконалена та швидша версія цього алгоритму макета з підтримкою обмежень також була протестована, але не призвела до кращих макетів [4].

CoSE-Bilkent – це алгоритм, який використовується для автоматичного розміщення елементів (наприклад, вузлів графу) на екрані таким чином, щоб вони були максимально читабельними та зрозумілими. Іншими словами, він допомагає створювати візуально привабливі діаграми та графіки.

Алгоритм працює в кілька етапів:

- Ініціалізація: Всі елементи розміщуються на екрані випадковим чином.

- Обчислення сил: Для кожного елемента обчислюються сили, які діють на нього. Ці сили можуть бути відштовхуючими (між елементами) або притягуючими (між елементами, які пов'язані між собою).

- Оновлення позицій: Елементи переміщуються в напрямку результуючої сили.

- Перевірка на збіг: Перевіряється, чи не накладаються елементи один на одного. Якщо так, то вони розсуваються.

- Повтор: Кроки 2-4 повторюються до тих пір, поки не буде досягнуто бажаного результату.

Початкові місця позначені чорною рамкою, кінцеві місця позначені зеленими смугами, якщо випуски, які там закінчуються, виконані, і позначені червоними смугами, якщо випуски, які там закінчуються, відхилені.

Отриманий графік і, отже, макет містить усі стани та дії, які були виконані, в результаті чого модель є надто детальною та дуже великою, що містить послідовності станів і дій, відвіданих лише в одному екземплярі. В [17] називають ці спагетті-моделі, викликані менш структурованими процесами або частинами процесів. Вони вирішують це, видаляючи ребра та вузли, кластеризуючи їх разом із сусідами. У цій реалізації ми відфільтровуємо стани та дії, які не відвідують більше п’яти-десяти відсотків проблем, що зменшує розмір моделі та дозволяє визначити потоки проблем.

3.2. Дослідження робочого процесу обробки помилок

На рисунку 3.3 показано робочий процес команди щодо помилок, створених або вирішених у піврічний період. Щоб зменшити кількість безладу, показано лише стани та переходи, які відбувалися принаймні в п’яти відсотках часу. Проблеми починаються зліва в одному з двох початкових станів. Ці стани представляють портал, на якому надсилаються помилки. Звідти вони залучаються до проекту власником продукту, позначеним синім кольором. У більшості випадків проблема переміщується безпосередньо з папки «Вхідні» до категорії «дослідити», але деякі проблеми одразу відхиляються, а інша невелика кількість проблем потрапляє безпосередньо до «виконання».

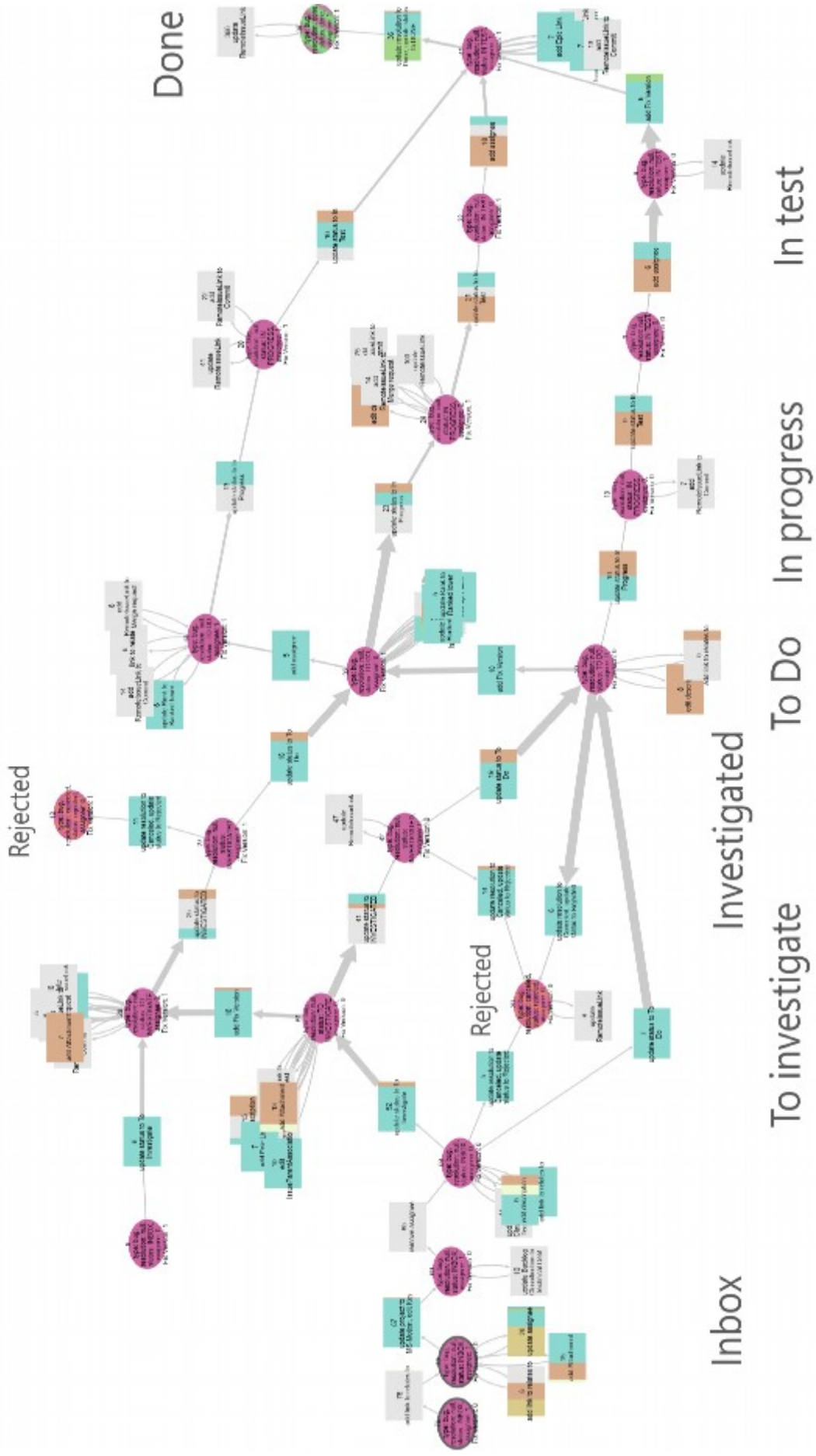

Рис. 3.3. Робочий процес обробки помилок

Проблеми витрачають значно більше часу в стані «розслідувати», ніж в інших станах, на що вказують товстіші стрілки, що входять і виходять з місця. Середній час у «розслідуванні» становить сімнадцять днів, після чого вони переходять до «досліджено», а власник продукту приймає рішення, чи перемістити помилку в «виконання», чи відхилити її. Це відбувається протягом одного-двох днів.

Щойно помилки отримають статус «виконати», проблеми витрачають значну кількість часу на очікування, поки над ними почнуть працювати, що знову позначається товстішими стрілками. Два верхніх шляхи праворуч від місць зі статусом «виконати» представляють шляхи, якими проходять відтворювані помилки, вони отримують версію виправлення, перш ніж перейти з «виконання», а іноді також одержують правонаступника. Нижній шлях представляє періодичні помилки, помилки, які трапляються рідко і користувачеві важко відтворити, вони отримують версію виправлення безпосередньо перед тим, як перейти до завершення.

У всіх трьох шляхах після статусу «виконати» помилки переходять у «в процесі», а потім у «в тесті», і призначають уповноваженого, відповідального за тестування, але нижній шлях, який містить періодичні помилки, витрачає набагато довше в «в test", оскільки вони чекають кілька тижнів, щоб побачити, чи вони більше не відбуваються після внесення змін. Під час цієї частини робочого процесу власник продукту не стільки ініціює переходи, а інженери, помаранчеві, переносять проблеми на наступний крок. Член команди забезпечення якості програмного забезпечення (зелений) ініціює перехід від «тестування» до виконання.

Є багато переходів, які також виконуються ботом, показано сірим кольором, вони вказують на дії, виконані в програмному забезпеченні, інтегрованому з Jira, наприклад, коміти або злиття в Git, пов’язані з комітом. Автоматичні переходи між станами запускаються правилами, які встановлюють команди, як переміщення проблеми до досліджуваної, якщо історія, пов’язана з комітом, що починається на «Дослідити», позначена як

завершена. Автоматизовані переходи не завжди виконуються автоматично, оскільки вони не запускаються належним чином. У групі це трапляється, коли хтось переміщує підзавдання розслідування до «виконується», а потім виконує, але забув перемістити історію розслідування до «розвивається».

На рисунку 3.4 показано процес роботи з помилками команди TJ за той самий проміжок часу, що й у попередньому прикладі. Робочий процес здебільшого однаковий, оскільки команда TJ також використовує стандартні статуси робочого циклу помилок TEM, але є деякі невеликі відмінності. Поруч із початковими станами, що представляють портал для надсилання помилок, TJ також має початковий стан, який показує помилки, створені безпосередньо в рамках командного проекту Jira. Подальше розслідування показує, що про ці помилки повідомляють їх інженер-випробувач і члени команди. Оскільки п’ять відсотків шляхів виникнення проблем не показано на рисунку 3.3, це не означає, що проблем, створених інженером-випробувачем у групі TI, немає , але там це рідше. Команда TJ також додає версію виправлення, поки помилка має статус папки "Вхідні", раніше, ніж команда TI. Ще одна відмінність — відсутність багів, які тривалий час перебувають у «тесті». Середній час у «тесті» становить 3 дні, а медіана трохи менше 1 дня, що набагато менше, ніж на попередньому графіку. Крім того, проблеми в команді TJ можуть переходити до завершення без призначення виконавця, чого не відбувається в команді TI, але, здається, це трапляється лише для простіших помилок, оскільки помилки без виконавця переходять до виконання швидше. Коли дивитися на час двох шляхів від «розслідувати» до «виконано» з правонаступником, немає різниці, в якій точці шляху додається цеснова.

У робочому процесі команди TI проблеми витрачають більшість часу на «дослідження» та «виконання». У той час як тривале очікування не викликає занепокоєння у «виконанні», оскільки команда не може працювати над усім одразу. Цікаво дослідити, що є причиною довшого часу очікування в «розслідувати».

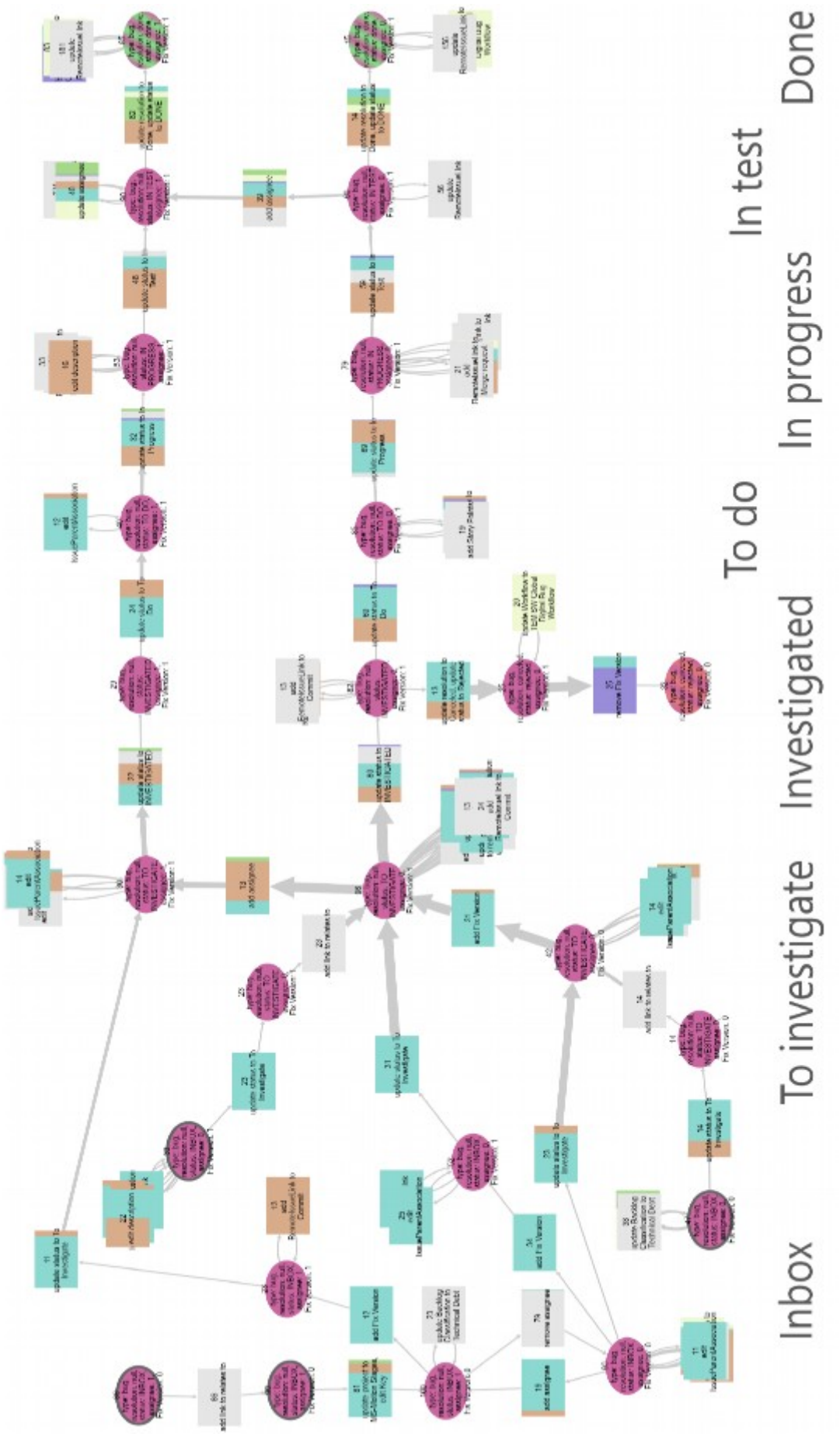

Рис. 3.4. Робочий процес обробки помилок TJ

Команда TJ не має тривалого часу очікування в «виконання», що означає, що коли помилку досліджено і не відхилено, її виявляють майже негайно, середній час, який їхні помилки витрачають на «виконання», становить менше одного дня.

3.3. Представлення робочого процесу розробки нової функції

На відміну від робочого процесу для помилок, робочий процес для нових функцій складається з дуже невеликої кількості статусів.

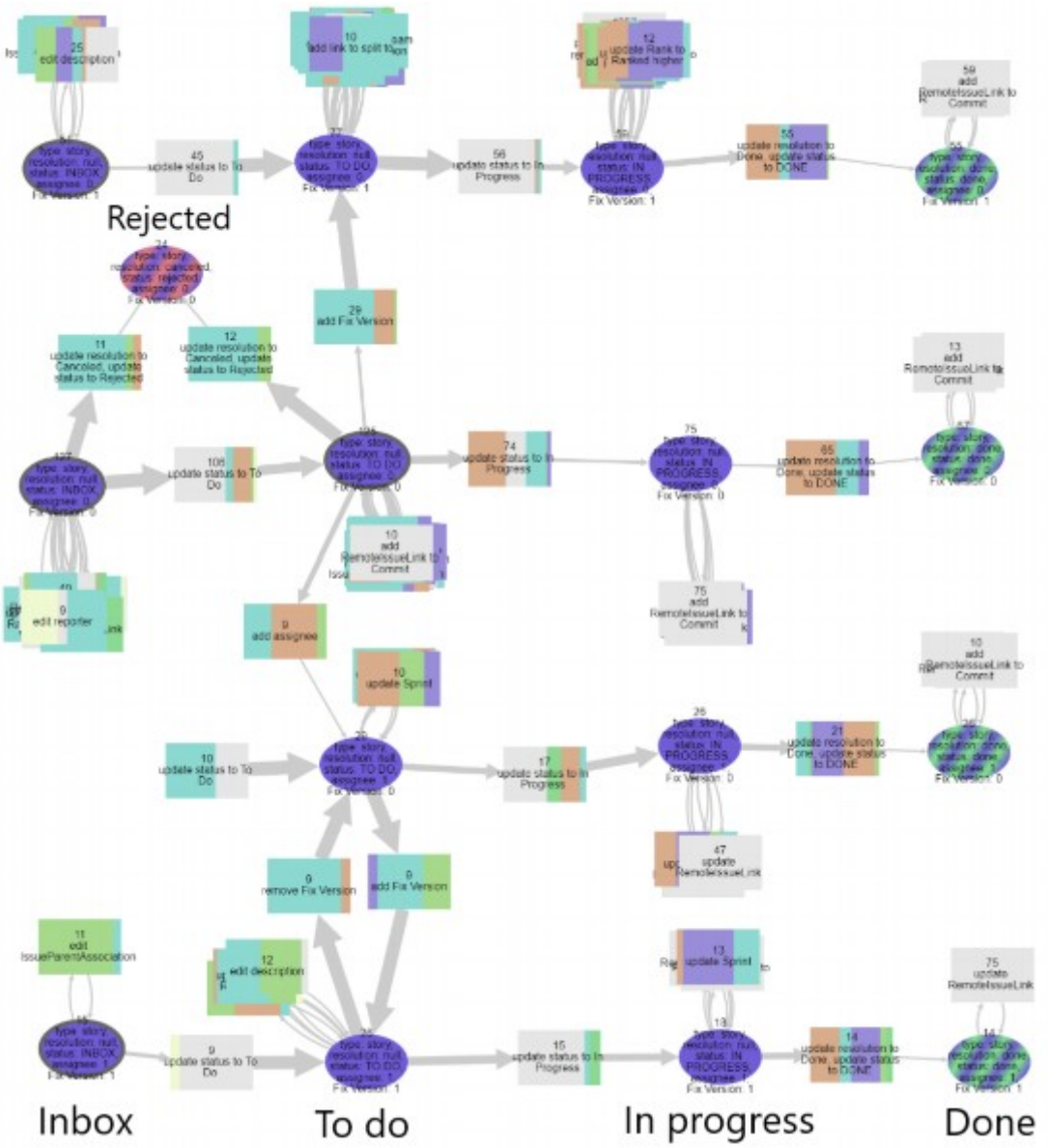

Рис. 3.5. Робочий процес розробки нової функції ТВ

На рисунку 3.5 показано реконструйований робочий процес команди ТВ, де завдання переміщуються зліва направо через граф. Завдання починаються в одному з трьох початкових станів — з або без зазначеної версії виправлення та виконавця. Завдання можуть бути позначені як виконані без заповнення обох полів, але це переважно з адміністративних причин, коли процес затримано; середній час виконання таких завдань — чотири секунди. На відміну від цього, середній час виконання завдань, що мають зазначену версію виправлення або виконавця, становить десять днів.

Відмінністю від роботи з помилками є те, що функції в команді ТВ не перевіряються іншою особою перед завершенням, тому у робочому процесі немає статусу "тестування". Хоча тестування не є частиною основного процесу, іноді його все ж проводять у вигляді підзадачі до основної історії. Наявність чотирьох шляхів, що ведуть до завершення завдань безпосередньо зі статусу "виконати", свідчить про те, що робочий процес цієї команди є гнучким, поля часто залишаються незаповненими, і порядок дій не є суворим. Як і у випадку з помилками, функції часто залишаються в стані "виконати" тривалий час.

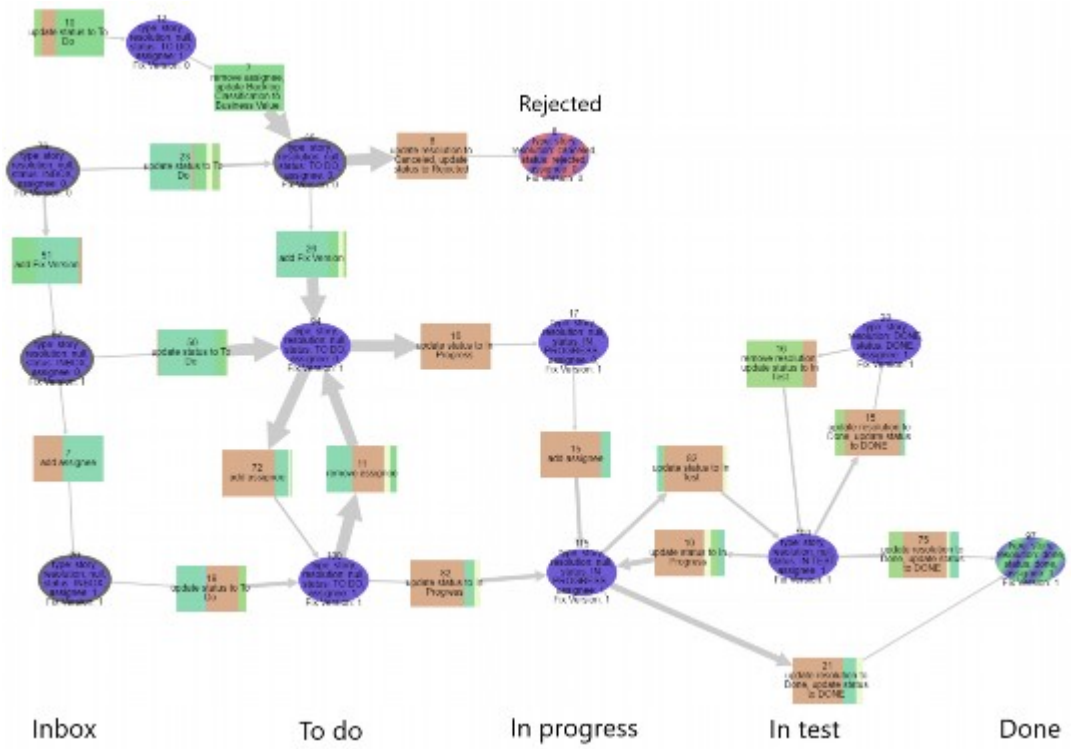

Рис. 3.6. Робочий процес розробки нової функції TN

На відміну від команди TB, робочий процес команди TN, представлений на рисунку 3.7, є більш фіксованим. Усі завдання, що завершуються, мають зазначену версію виправлення та призначеного виконавця, і лише ті функції, для яких було додано версію виправлення, переходять у статус "в процесі". Якщо виконавець не був призначений до зміни статусу на "в процесі", це здійснюється, поки функція ще перебуває у цьому статусі.

На відміну від команди TB, у команди TN є додатковий етап тестування перед завершенням завдань. Цікаво, що в близько десяти відсотках випадків завдання повертаються зі статусу "виконано" на "тестування", і середній час повернення завдання до тестування становить три дні. Невідомо, чи це адміністративна помилка, чи ці завдання дійсно потребують доопрацювання.

3.4 Порівняння з процесним майнінгом на основі журналу подій

Результати нашого алгоритму були порівняні з результатами, отриманими за допомогою двох існуючих інструментів процесного майнінгу, [16]. Рисунки 3.7, 3.8 і 3.9 створені на основі тих самих журналів подій команди TI, що й на рисунку 3.3. Рисунок 3.7 показує результат, отриманий за допомогою Disco, яке використовує метод fuzzy miner. Цей метод створює один стан для кожної дії, що означає, що, наприклад, коли виконавець додається до задачі, що знаходиться в статусі "виконати" або "тестування", дія призводить до того самого місця на графі. Це призводить до того, що багато станів зв'язуються з численними іншими станами, оскільки будь-яка дія, окрім певних переходів статусів, може статися, коли задача перебуває в будь-якому статусі в Jira. На рисунку 3.8 показано результат роботи індуктивного майнера ProM. Як і в попередньому прикладі, результат містить одне початкове та кінцеве місце, що робить ці алгоритми придатними для відображення лише одного процесу за раз.

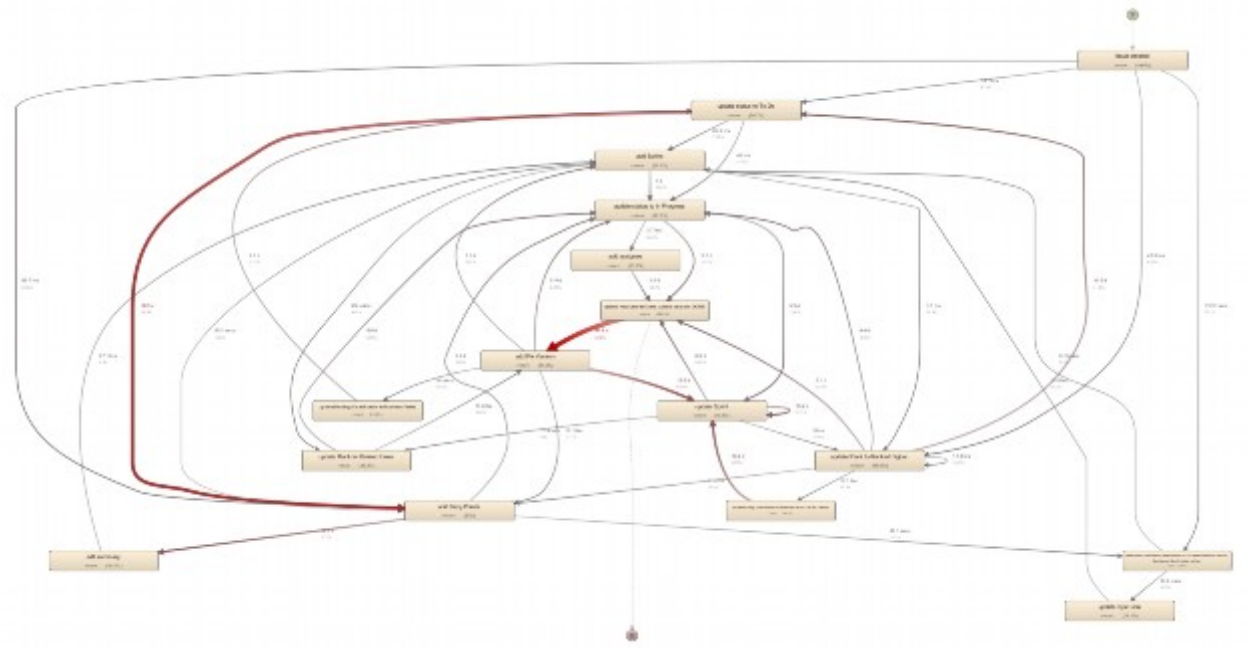

Рис. 3.7. Робочий процес роботи з помилками Team TI, створений Disco, показує 20% найчастіших дій і 10% найчастіших шляхів

Було б більш зрозуміло, якби ці графи показували, що завдання, які походять з команди та з порталу, починаються з різних місць. Ще одна відмінність полягає в тому, що індуктивний майнер не показує циклів самопереходу, що призводить до наявності багатьох необов'язкових дій у ланцюжку в отриманій мережі Петрі. Ці переходи зображаються як зв'язок між двома місцями і чорний перехід, який також з'єднує ті самі місця.

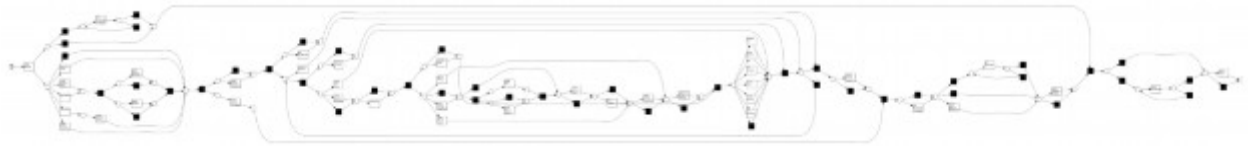

Рис. 3.8. Робочий процес роботи з помилками Team TI, створений за допомогою індуктивного майнера Petri Net

З трьох майнерів індуктивний майнер дає найбільш структурований результат із чітким порядком дій. Рисунок 3.9 демонструє результат роботи майнера систем переходів у ProM. Результат є так званим "спагеті-процесом", оскільки будь-яка дія, виконана в будь-якій частині робочого процесу, веде

до того самого місця. Це створює місця з тією ж кількістю зв'язків, що й на рисунку 3.7.

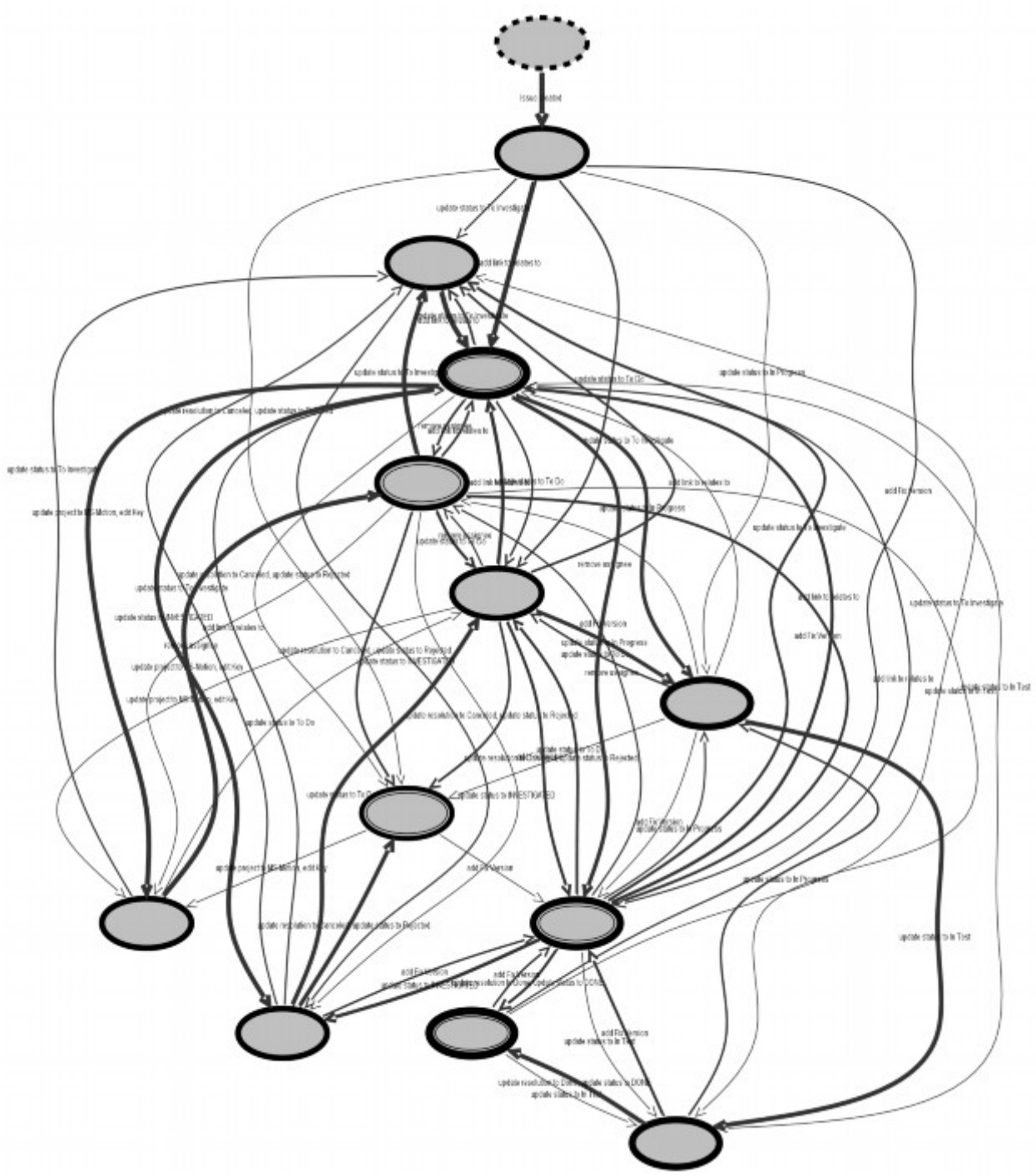

Рис. 3.9. Робочий процес роботи з помилками Team TI, створений за допомогою майнера системи переходів

Отже, у обох робочих процесах, пов'язаних з помилками, власник продукту відіграє важливу роль у прийнятті рішення про те, коли задачі

переходять на наступний етап. У робочих процесах для нових функцій власник продукту має найбільше значення під час уточнення беклогу, коли задачі знаходяться в стані "вхідні" або "виконати". Надалі в робочому процесі для функцій інженери беруть на себе управління завданнями. У всіх прикладах, розглянутих в аналізі, відфільтровується від п'яти до десяти відсотків завдань, що не відповідають робочому процесу команди. Це свідчить про те, що команди дотримуються загального підходу до роботи в більшості випадків. Лише в небагатьох випадках вони відхиляються від цього підходу. Це можливо, оскільки в Jira правила накладаються лише на зміни статусу, а інші поля можуть змінюватися в будь-який момент на будь-яке інше значення. У прикладах робочих процесів для помилок завдання проводять більшу частину часу в станах "для розслідування" та "виконати". Статус "виконати" не є активним, тоді як у стані "для розслідування" проводиться робота в межах розслідування. Однак через відсутність окремого статусу розслідування неясно, скільки часу займає цей етап у представлених графіках. Загалом, статус "у процесі" займає лише невелику частку часу всього процесу виправлення помилок, і більшість часу витрачається на очікування початку робіт, що показано товстішими стрілками, які вказують на вхід та вихід із стану "виконати". Те ж саме стосується й функцій: більша частина часу витрачається в статусі "виконати", а не в статусі "у процесі", де активно ведеться робота.

Нарешті, обраний підхід до реконструкції та візуалізації робочого процесу успішно створює структуровані результати, де інші техніки процесного майнінгу призводять до так званих "спагеті-процесів".

3.5. Методологія визначення продуктивності розробки програмного забезпечення

З реконструкції робочого процесу ми дізналися, що проблеми проводять більшу частину свого часу в неактивних станах, у станах, де

проблеми накопичуються, поки над ними не буде вибрано роботу. Врахування такого часу не є корисним для порівняння команд, оскільки час до розгляду питання залежить від того, скільки питань відкрито та яка здатність команди їх розглядати. Для помилок час, витрачений на етап «дослідження», також не вважається часом, активно витраченим на дослідження, оскільки ці значення вказують на те, що над ними не працюють постійно. Щоб компенсувати вплив цих значень часу, коли члени команди не працюють активно над помилкою чи функцією, під час визначення продуктивності ми будемо враховувати лише те, що відбувається під час статусу проблеми «виконується».

Визначити швидкість команди розробників програмного забезпечення є досить не просто. В [27] припускають, що ця метрика навіть відрізняється між точкою зору розробника та менеджера, причому розробники більш схильні ґрунтуватися на тому, на що вони витрачають свій час, тоді як менеджери більше думають про якість або продуктивність. Ми вважаємо, що показники більше відповідають точці зору керівників. Щоб прийти до числа, яке ми можемо використовувати для порівняння команд, у нас є кілька обмежень.

- Цей показник повинен враховувати розмір команди, більші команди можуть мати більший результат, але ми хочемо нормалізувати цей результат відповідно до кількості членів команди.
- Міра повинна бути незалежною від мови програмування та стилю. Мова, яка використовується в різних командах, відрізняється, і немає стандартного стилю кодування для команд.
- Впровадження виправлень помилок і робота над функціями повинні мати однакову вагу. Команди намагаються забезпечити найбільшу цінність для клієнтів. Це може змусити їх реалізувати функцію, яку хоче клієнт, замість виправлення помилки, яка не заважає клієнтам. Також є невеликі розбіжності між командами щодо того, що слід класифікувати як помилку. Деякі команди створюють помилку для завдань, які не є дефектами

програмного забезпечення, а також для виправлення неправильних вимог або нових ідей.

Незалежність від розміру команди спрямовує нас до метрики, яка ділить результат на вклад, зроблений для його досягнення. Це називається продуктивністю.

$$productivity = \frac{output}{input}$$

Це залишає нам пошук відповідних заходів для вихідних і вхідних даних. В [11] і [24] надають огляд найбільш часто використовуваних показників. Найпопулярнішим є рядки коду на людину або на одиницю часу. Це не вважалося практичним заходом для порівняння команд, оскільки існує велика різниця в мовах програмування, які використовуються, і в компанії немає стилю кодування, що робить проекти однієї мови непорівнянними. Подальші проекти, в яких використовується розробка програмного забезпечення на основі моделі, змінюють моделі, що ускладнює порівняння з командами, які пишуть код вручну.

Метрика, яка намагається вирішити невідповідність стилю кодування, — це функціональні точки. Однак не всі види діяльності з розробки можуть бути виражені у функціональних точках, тобто зміна конфігураційних файлів або змінення керованих моделями програмних інженерних моделей. Якщо відійти від результатів, виміряних за аспектами вихідного коду, найбільш використовуваними показниками є кількість комітів і кількість реалізованих функцій. Ми не хочемо робити упереджене ставлення до помилок або функцій, тому як результат ми візьмемо кількість виправлених помилок і функцій. Як вхідні дані ми беремо кількість днів, за винятком вихідних і святкових днів, протягом яких вирішені помилки та проблеми мали статус «виконується». Запропонована остаточна формула визначення продуктивності показана нижче.

$$productivity = \frac{|bugs| + |features|}{\sum_{issue \in (bugs \cup features)} businessDaysOnInProgressStatus(issue)}$$

Також розглядалися інші параметри продуктивності, як-от співвідношення між часом, витраченим на функції, і часом, витраченим на помилки, як показано на рисунку 3.10, але це спричинило викиди на нижньому та високому рівнях, що також є недоліком співвідношень.

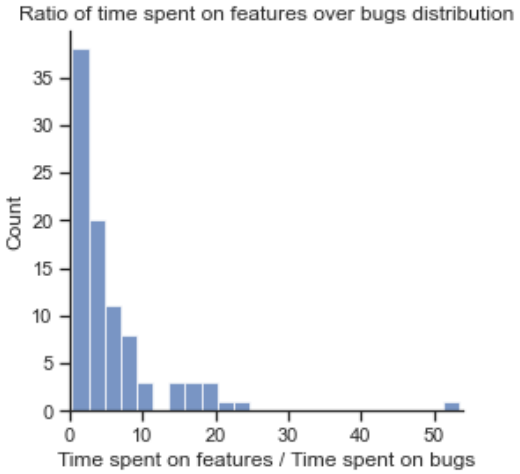

Рис. 3.10. Розподіл часу між розробкою функцій та виправленням помилок

Крім того, співвідношення спотворене до нижнього рівня. Для порівняння, розподіл нашої метрики продуктивності все ще зміщений у бік нижнього рівня, але більш центрований, як показано на рисунку 3.11.

Швидкість — ще один варіант, але команди не мають однакового визначення сюжетних точок, а сюжетні точки часто не призначаються помилкам, що унеможливлює також включати проекти обслуговування в аналіз.

В [24] пропонується об’єднати кілька показників для створення одного значення, яке краще відображає продуктивність програмного проекту. Наведено приклади таких формул із зважуванням деяких метрик, які обговорювалися раніше, але метрики, вибрані як частина таких формул,

залежать від типу створеного програмного забезпечення, і тому їх не можна застосовувати для порівняння дуже різних проектів.

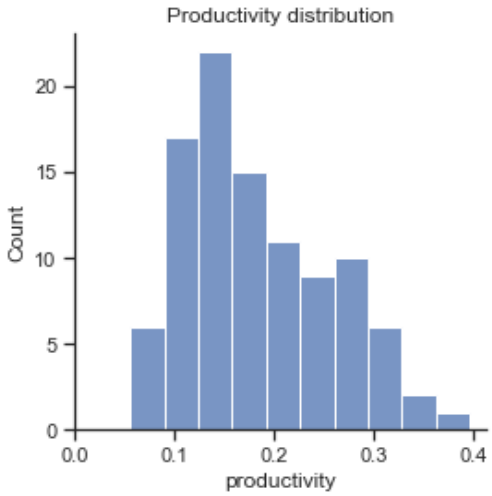

Рис. 3.11. Розподіл продуктивності згідно запропонованої формули

В [14] також пропонують використовувати кілька показників щонайменше з трьох із наступних параметрів: задоволеність, продуктивність, активність, спілкування та ефективність. Це призводить до більш репрезентативного показника продуктивності команди, оскільки враховується більше аспектів процесу розробки. Також вони стверджують, що використання кількох метрик запобігає формуванню метрик на поведінку команди порівняно з використанням лише однієї метрики. На жаль, описана структура для запису продуктивності вимагає даних про аспекти, про які ми не маємо даних, тому їх не можна було використовувати.

3.6. Перевірка валідності запропонованого показника продуктивності

Щоб оцінити валідність нашого показника продуктивності як хорошого показника для порівняння команд, нам потрібно перевірити, чи самі команди мають приблизно однакове значення продуктивності з часом. Якщо значення продуктивності команди має високу мінливість з часом, це означає, що

показник не в змозі відобразити продуктивність команди, оскільки спосіб роботи, склад команди та інші фактори, що впливають на продуктивність команди, майже однакові між послідовними проміжки часу. Рисунок 3.12 показує результат нашої формули продуктивності застосовано до семи останніх повних періодів розробки.

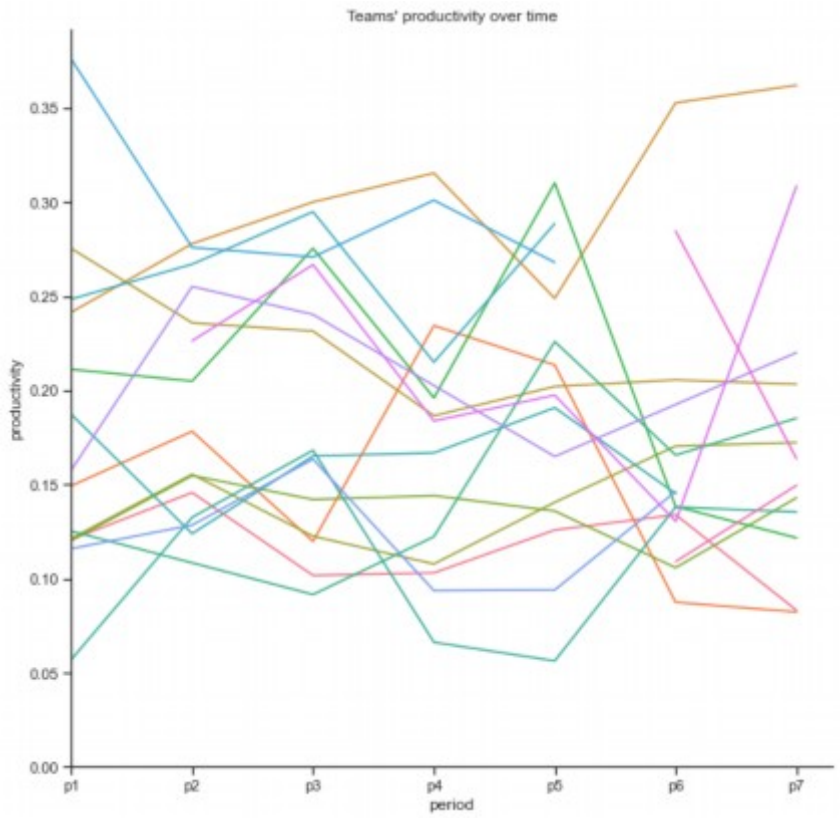

Рис. 3.12. Представлення продуктивності команд у часі

Щоб отримати рисунок 3.12, з набору даних було вилучено кілька команд. Команди TE, TU, TQ, TX, TK, TM і TL мали періоди без помилок або з лише невеликою кількістю помилок і функцій, що призводило до дуже нестабільного рівня продуктивності. Команда TA була вилучена через те, що шостий період був першим періодом її активності. Було також вилучено два періоди роботи команди TV, оскільки це відносно нова команда, і її продуктивність значно відрізнялася від інших трьох періодів.

Є команди з досить стабільною продуктивністю, такі як TB, TG, TH і TF, але також є команди з різницею в два рази між періодами, такі як TJ і TN.

Для команди TJ це можна пояснити тим, що в третій і четвертий періоди вони не мали стільки помилок і функцій, як у п'ятий і шостий, тому якщо одна з помилок або функцій нерегулярна, це має великий вплив на їх продуктивність. Тоді як для команди TN такого можливого пояснення не знайдено.

Різниця в продуктивності між послідовними періодами команд є значно кращою, ніж у першій спробі визначення продуктивності, коли були взяті періоди по три місяці. Це призвело до великих різниць у продуктивності команд у часових періодах, що призвело до вибору узгодження часових періодів з частотою випуску програмного забезпечення.

Однією з причин цього може бути те, що більшість комітів відбувається наприкінці циклу випуску. Можливим покращенням для вирішення варіацій у першій спробі та другій спробі може бути згладжування значень продуктивності шляхом взяття середнього значення поточного, наступного та попереднього часового періоду. Таке покращення дозволило б розділити часові періоди на менші періоди.

Рисунок 3.13 показує кореляцію Kendall’s Tauміж метриками QI у вигляді теплової карти.

Чим темніший червоний колір, тим сильніша позитивна кореляція, чим темніший синій колір, тим сильніша негативна кореляція. Більш світлі відтінки обох кольорів вказують на слабшу позитивну або негативну кореляцію.

Оскільки результати для безпеки та охоплення тестування не є точними, вони виключені. Щоб усунути їх вплив на загальний бал TQI, обчислюється adjustedTqi з решти метрик з їх оригінальними вагами з формули TQI, і тому adjustedTqi є значенням між 0 і 75.

Кореляція між дублюванням коду та іншими підметриками TQI виділяється як слабша. Бал за дотримання стандартів кодування TQI має другу найслабшу кореляцію з іншими метриками, на рівні 0,44. Інші кореляції перевищують 0,5.

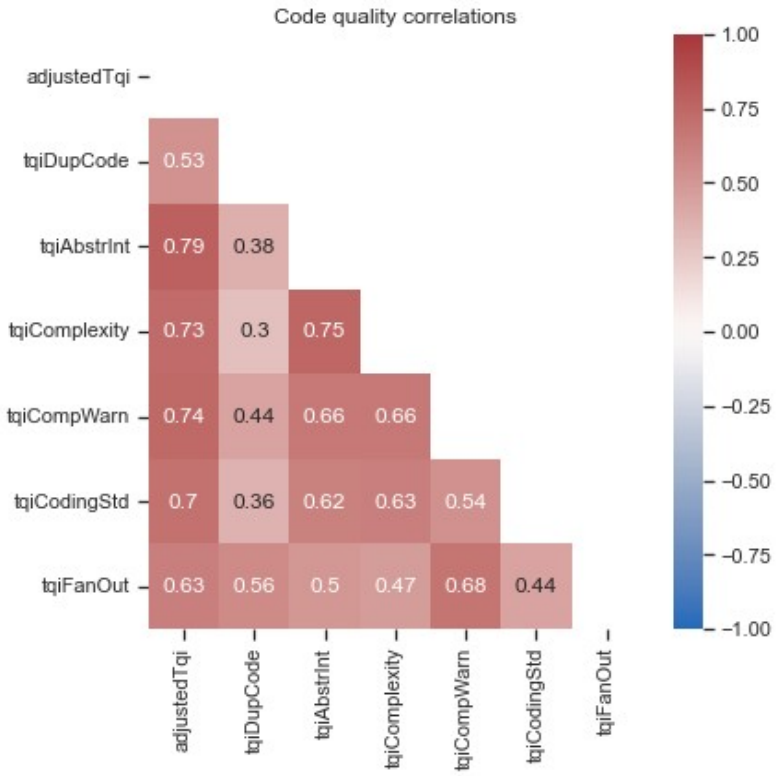

Рис. 3.13. Кореляції якості коду

LoC, ELoC і GLoC, як і очікувалося, мають високу кореляцію, понад 0,98. При порівнянні їх з мірами QI вони мають кореляцію -0,48 з дублюванням коду QI, що вказує на те, що чим більше рядків коду, тим більше дублюваного коду. Інші метрики QI мають дуже слабку кореляцію з кількістю рядків коду.

Рисунок 3.14 показує середню продуктивність і середній QI періодів один і два, а також періодів шостий і сьомий для проектів, які мали значне збільшення QI (немає проектів, які знизили QI). Для кожної команди на малюнку крайній лівий бал є середнім значенням за періоди один і два, а крайній правий бал є середнім значенням за періоди шостий і сьомий. Дві команди мають підвищення продуктивності, тоді як інші чотири команди мають приблизно однакове значення або зниження продуктивності. Це також свідчить про те, що підвищення якості коду не корелює зі збільшенням продуктивності.

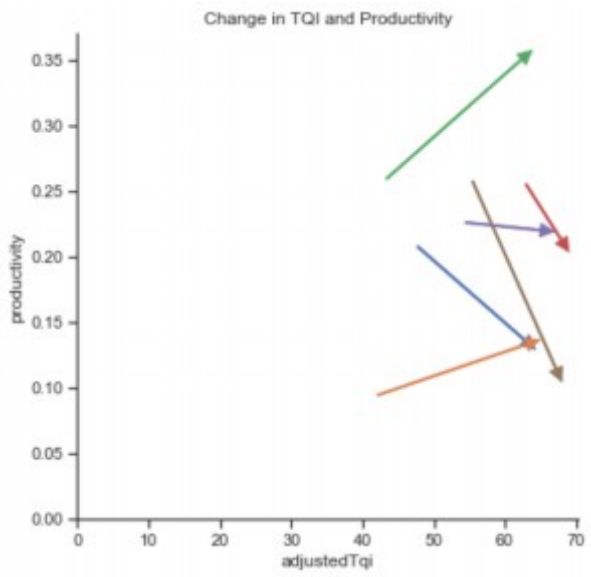

Рис. 3.14. Зміна QI та продуктивності між середнім значенням періоду 1 і 2 та середнім значенням періоду 6 і 7

Висновки до розділу

Отже, в цьому розділі представлено процеси і методологію для оцінки продуктивності розробки програмного забезпечення. Виконано моделювання робочого процесу командної розробки і описано, як структурувати командну роботу, щоб підвищити продуктивність. Акцент робиться на взаємодії між членами команди та їхньою роллю в загальному процесі. Дослідження робочого процесу обробки помилок фокусується на ідентифікації та аналізі типових проблем, які виникають у процесі тестування та усунення помилок. Правильна організація цього процесу може значно скоротити час на його виконання. Представлення робочого процесу розробки нової функції детально розглядає процес інтеграції нових функцій у програмний продукт, звертаючи увагу на ключові етапи, які мають найбільший вплив на загальну ефективність команди.

Порівняння з процесним майнінгом на основі журналу подій аналізує підходи до моніторингу та аналізу реальних робочих процесів шляхом використання методів процесного майнінгу, що допомагає краще розуміти, як процеси насправді виконуються.

Методологія визначення продуктивності розробки програмного забезпечення пропонує інструменти та методи для оцінки ефективності розробки ПЗ. Ці підходи дозволяють виявити вузькі місця та точки для оптимізації. Перевірка валідності запропонованого показника продуктивності містить експериментальні перевірки запропонованих показників продуктивності. Вони підтверджують або спростовують їхню достовірність і практичність в реальних умовах розробки.

Загалом, в даному розділі виконано комплексне дослідження процесів командної розробки програмного забезпечення, зосередженим на підвищенні продуктивності через моделювання, аналіз і вдосконалення робочих процесів.

ВИСНОВКИ

У магістерській роботі досліджено теоретичні засади підвищення продуктивності проєктів з розробки програмного забезпечення та запропоновано методологію аналізу ключових процесів, що впливають на ефективність командної роботи.

Перший розділ присвячений аналізу предметної області та особливостей командної розробки програмного забезпечення. Показано, що впровадження гнучких методологій, таких як Agile і Scrum, позитивно впливає на координацію та адаптацію команди до змінних умов проєкту. Важливим компонентом підвищення продуктивності є використання сучасних інструментів для розробки, таких як системи контролю версій (Git), які забезпечують ефективну співпрацю та контроль за змінами в коді.

Дослідження продемонструвало, що власник продукту відіграє ключову роль у прийнятті рішень стосовно переходу завдань між етапами життєвого циклу, що відповідає ролям у Scrum. Це стосується як робочих процесів з усунення помилок, так і впровадження нових функцій, де роль інженерів стає домінуючою на етапі управління проблемами.

Значна увага приділяється методам перевірки якості коду та проектуванню на основі моделей (Model-Driven Development). Використання таких підходів дозволяє підвищити рівень автоматизації процесів розробки, а також забезпечує більш структуроване управління складними проєктами. Організація архітектури програмної системи на основі моделей сприяє підвищенню надійності та повторюваності процесу розробки.

Другий розділ присвячено методам збору та аналізу даних для оцінки продуктивності процесу розробки. Детально описані підходи до визначення характеристик даних, що впливають на ефективність командної роботи, зокрема структура завдань в Jira та помилок, що дозволяє більш детально оцінити процеси розробки та управління завданнями. Запропоновано підходи

до організації збору даних про розробку з використанням репозиторіїв коду та термінів релізів, що забезпечує точніший аналіз продуктивності.

У третьому розділі розглянуто методологію аналізу робочих процесів, зокрема процесів обробки помилок та розробки нових функцій. Було змодельовано робочі процеси на основі журналів подій, що дозволило більш точно виявити вузькі місця в роботі команд та запропонувати шляхи їх оптимізації. Методологія визначення продуктивності розробки програмного забезпечення включає як кількісні, так і якісні показники, що дозволяє об'єктивно оцінити ефективність розробки.

Проведено перевірку валідності запропонованого показника продуктивності, що підтвердило його придатність для оцінки різних аспектів командної роботи. Використання даного підходу дозволяє оптимізувати процеси розробки та підвищити загальну продуктивність команд програмістів.

Таким чином, робота робить вагомий внесок у розробку науково обґрунтованих методів аналізу продуктивності програмної розробки, що може бути застосовано для підвищення ефективності в різних контекстах розробки програмного забезпечення.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. W.M.P. van der Aalst and C. Stahl. Modeling Business Processes — A Petri Net-Oriented Approach. Master's thesis, Eindhoven University of Technology, 2011.
2. Atlassian. Jira Issue & project tracking software. https://www.atlassian.com/software/jira
3. Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context — motorola case study. In Lionel Briand and Clay Williams, editors, Model Driven Engineering Languages and Systems, pages 476–491, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
4. Hasan Balci and Ugur Dogrusoz. fcose: a fast compound graph layout algorithm with constraint support. IEEE Transactions on Visualization and Computer Graphics, 28(12):1–1, 2021.
5. Joseph D Blackburn, Gary D Scudder, and Luk N VanWassenhove. Improving speed and productivity of software development: a global survey of software developers. IEEE transactions on software engineering, 22(12):875–885, 1996.
6. Frederick P Brooks Jr. The mythical man-month: essays on software engineering. Pearson Education, 1995.
7. Jo˜ao Caldeira and Fernando Brito e Abreu. Software development process mining: Discovery, conformance checking and enhancement. In 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC), pages 254–259, 09 2016.
8. C.G. Cobb. The Project Manager's Guide to Mastering Agile: Principles and Practices for an Adaptive Approach. Wiley, 2015.
9. Suzana Cˆandido de Barros Sampaio, Emanuella Aleixo Barros, Gibeon Soares de Aquino, Mauro Jos´e Carlos e Silva, and Silvio Romero de Lemos Meira. A review of productivity factors and strategies on software

development. In 2010 Fifth International Conference on Software Engineering Advances, pages 196–204, 2010.

10. Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. A layout algorithm for undirected compound graphs. Information Sciences, 179(7):980–994, 2009.
11. Carlos Henrique C. Duarte. The quest for productivity in software engineering: A practitioners systematic literature review. In 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP), pages 145–154, 2019.
12. Johri van Eerd. Dezyne tutorial. https://doc.verum.com/dezyne/manual/dezyne-tutorial.pdf
13. Norman E Fenton and Martin Neil. Software metrics: roadmap. In Proceedings of the Conference on the Future of Software Engineering, pages 357–370, 2000.
14. Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Tom Zimmermann, Brian Houck, and Jenna Butler. The space of developer productivity: There's more to it than you think. ACM Queue, 19(1):1–29, March 2021. 34
15. Git. Git. https://git-scm.com/
16. Christian W. G¨unther and Anne Rozinat. Disco: discover your processes. In Niels Lohmann and Simon Moser, editors, Proceedings of the Demonstration Track of the 10th International Conference on Business Process Management (BPM 2012), CEUR Workshop Proceedings, pages 40–44. CEUR-WS.org, January 2012. Demonstration Track of the 10th International Conference on Business Process Management, BPM Demos 2012
17. Christian W. G¨unther and Wil M. P. van der Aalst. Fuzzy mining – adaptive process simplification based on multi-perspective metrics. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, Business

Process Management, pages 328–343, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

18. Silvia Jaqueline Urrea-Contreras, Brenda L. Flores-Rios, Mar´ıa Ang´elica Astorga-Vargas, and Jorge E. Ibarra-Esquer. Process mining perspectives in software engineering: A systematic literature review. In 2021 Mexican International Conference on Computer Science (ENC), pages 1–8, 2021.
19. Damodaram Kamma and G. Sasi Kumar. Effect of model based software development on productivity of enhancement tasks – an industrial study. In 2014 21st Asia-Pacific Software Engineering Conference, volume 1, pages 71–77, 2014.
20. Rita Marques, Miguel Mira da Silva, and Diogo R. Ferreira. Assessing agile software development processes with process mining: A case study. In 2018 IEEE 20th Conference on Business Informatics (CBI), volume 01, pages 109–118, 2018.
21. Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In IFIP Central and East European Conference on Software Engineering Techniques, pages 252–266. Springer, 2007.
22. Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. Developer-driven code smell prioritization. In Proceedings of the 17th International Conference on Mining Software Repositories, MSR ’20, page 220–231, New York, NY, USA, 2020. Association for Computing Machinery.
23. Kai Petersen. Measuring and predicting software productivity : A systematic map and review. Information and Software Technology, 53(4):317–343, 2011.
24. Wouter Poncin, Alexander Serebrenik, and Mark van den Brand. Process mining software repositories. In 2011 15th European Conference on Software Maintenance and Reengineering, pages 5–14, 2011.

25. Vladimir Rubin, Christian W. G¨unther, Wil M. P. van der Aalst, Ekkart Kindler, Boudewijn F. van Dongen, and Wilhelm Sch¨afer. Process mining framework for software processes. In Qing Wang, Dietmar Pfahl, and David M. Raffo, editors, Software Process Dynamics and Agility, pages 169–181, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
26. M.A. Storey, B Houck, and T Zimmermann. How developers and managers define and trade productivity for quality. In Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering. ACM, may 2022.
27. Edith Tom, Ayb¨uke Aurum, and Richard Vidgen. An exploration of technical debt. Journal of Systems and Software, 86(6):1498–1516, 2013. 45
28. Wil M. P. van der Aalst. Process Mining: Data Science in Action. Springer, Heidelberg, 2 edition, 2016.
29. B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The prom framework: A new era in process mining tool support. In Gianfranco Ciardo and Philippe Darondeau, editors, Applications and Theory of Petri Nets 2005, pages 444–454, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
30. Basili, V. R., & Caldiera, G. (1995). "A Methodology for Collecting Valid Software Engineering Data." IEEE Transactions on Software Engineering.
31. Boehm, B. W. (1981). Software Engineering Economics. Prentice Hall.
32. Sommerville, I. (2010). Software Engineering. Addison-Wesley.
33. Kitchenham, B. (1996). "Software Metrics: Measurement for Software Process Improvement." International Journal of Project Management.
34. Cusumano, M. A., & Selby, R. W. (1998). Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology. Free Press.
35. Humphrey, W. S. (1995). A Discipline for Software Engineering. Addison-Wesley.
36. Pressman, R. S. (2005). Software Engineering: A Practitioner’s Approach. McGraw-Hill.

37. CMMI Product Team. (2002). Capability Maturity Model Integration (CMMI), Version 1.1. Carnegie Mellon Software Engineering Institute.
38. Petersen, K., Wohlin, C., & Baca, D. (2009). "The Waterfall Model in Large-Scale Development." International Conference on Software Process Improvement and Capability Determination.
39. DeMarco, T. (1986). Controlling Software Projects: Management, Measurement, and Estimation. Yourdon Press.
40. Dybå, T., & Dingsøyr, T. (2008). "Empirical Studies of Agile Software Development: A Systematic Review." Information and Software Technology.
41. Brooks, F. P. (1995). The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley.
42. Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley.
43. Boehm, B., & Turner, R. (2004). Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley.
44. Poppendieck, M., & Poppendieck, T. (2003). Lean Software Development: An Agile Toolkit. Addison-Wesley.
45. Curtis, B., Kellner, M. I., & Over, J. (1992). "Process Modeling." Communications of the ACM.
46. Royce, W. W. (1970). "Managing the Development of Large Software Systems." Proceedings of IEEE WESCON.
47. Wieringa, R. J. (2014). Design Science Methodology for Information Systems and Software Engineering. Springer.
48. McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.
49. Fuggetta, A. (2000). "Software Process: A Roadmap." Proceedings of the International Conference on Software Engineering.
50. Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). "Agile Software Development Methods: Review and Analysis." VTT Publications.

51. Rubin, K. S. (2012). Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley.
52. Jørgensen, M. (2007). "Estimation of Software Development Work Effort: Evidence on Expert Judgment and Formal Models." International Journal of Forecasting.
53. Kruchten, P. (1995). "The 4+1 View Model of Architecture." IEEE Software.
54. Haug, J. A., & Stigler, M. (2000). "Performance Metrics in Software Process Improvement Programs." Journal of Software Process: Improvement and Practice.