Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Розробка методу автоматизованої публікації крейтів»
за освітньою програмою **12 Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи ПЗ2221: /Богдан БАЛУШКІН/

Керівник: доц. / Олена КУРОПЯТНИК/

Нормоконтролер: доц. /Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Developing a method for automated publication of crates»

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2221: ___ ____ / Bogdan BALUSHKIN/
(посада) (підпис)

Scientific Supervisor: ___ ____ / Olena KUROPIATNYK /
(посада) (підпис)

Normative controller: ___ ____ / Svitlana VOLKOVA/
(посада) (підпис)

Dnipro – 2024

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем

Кафедра: Комп’ютерні інформаційні технології

Рівень вищої освіти: магістр

Освітня програма: Інженерія програмного забезпечення

Спеціальність: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри ____КІТ

_____ Вадим ГОРЯЧКІН

_____ грудня 2023 р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістра_____

студенту Балушкіну Богдану Віталійовичу_____

1. Тема дипломної роботи: Розробка методу автоматизованої публікації крейтів
   Керівник роботи: Куроп’ятник Олена Сергіївна
   затверджені наказом 1196 ст від 05.12.2022 року
2. Строк подання студентом роботи 12.01.2024 року
3. Вихідні дані до дипломної роботи: крейти, призначенні для тестової публікації.
4. Зміст пояснювальної записки (перелік питань до розробки):
   4.1. Аналітична частина: вступ, збір вимог до програмного забезпечення;
   4.2. Основна частина: розробка методу, зовнішнє і внутрішнє проектування, тестування та налагодження, дослідження ефективності методу та ПЗ, висновки, література;
5. Перелік демонстраційного матеріалу:
   5.1. презентація;
   5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	15.12.22 – 31.12.22	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.01.23 – 07.07.23	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	08.07.23 – 26.11.23	
4	Постановка задачі, технічне завдання	27.11.23 – 03.12.23	30%
5	Виконання досліджень	18.12.23 – 24.12.23	60%
6	Оформлення тез доповідей	25.12.23 – 28.12.23	
7	Оформлення пояснювальної записки	29.12.23 – 07.01.24	
8	Розробка демонстраційних матеріалів	08.01.24 – 14.01.24	100%
9	Подання кваліфікаційної роботи до кафедри	15.01.24 – 17.01.24	
10	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	23.01.24	

Студент _____ /Богдан БАЛУШКІН/

Керівник роботи _____ / Олена КУРОП’ЯТНИК /

РЕФЕРАТ

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 4 додатків:

– вступ – у даному розділі визначається актуальність магістерської роботи, аналізується об'єкт дослідження та його предмет, формуються мета та методи проведення дослідження. Особлива увага приділяється розгляданню наукової новизни та практичного значення роботи, також наведено посилання на джерело, де були опубліковані тези. Складається з 3 сторінок;

– огляд проблеми публікації крейтів – у цьому розділі описуються поточний стан вирішення проблеми, огляд програмних аналогів, що націлені на вирішення проблеми, а також виконується постановка задачі. Складається з 7 сторінок;

– розробка методу – присвячений опису методу та його складових, з детальним описом використаних алгоритмів. Складається з 9 сторінок;

– проектування й розробка програмного продукту – мета цього розділу полягає у визначенні архітектури системи, реалізації програмного продукту на основі розробленого методу та перевірка коректності виконання розробленого продукту. Складається з 28 сторінки;

– дослідження ефективності методу та ПЗ – описує процес оцінки ефективності розробленого методу та програмного забезпечення.

Складається з 10 сторінок;

– висновки. Складається з 1 сторінки;

– список літератури – включає в себе бібліографічний список використаної літератури. Складається з 3 сторінки;

– додатки – містить технічне завдання, керівництво користувача, тези, текст програми.

Таблиць – 27, рисунків – 10, бібліографія – 30.

Ключові слова: Rust, публікація, автоматизація, перевірка, ефективність.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ

Крейт (Crate) – це основна одиниця компіляції в Rust. Це може бути бінарний крейт, який компілюється в виконуваний файл, або бібліотечний крейт, який компілюється в бібліотеку

Віддалений крейт – це крейт, який розміщено на сервері крейтів, наприклад, на crates.io.

Пакет (Package) – це набір одного або декількох крейтів, які надають певний набір функціональності. Пакет містить файл Cargo.toml, який описує, як будувати ці крейти. В одному пакеті може бути до одного виконуваного крейту та будь яка кількість бібліотечних крейтів.

Воркспейс (Workspace) – це набір пакетів, які ділять один і той же Cargo.lock і вихідний каталог. Воркспейси корисні для управління кількома пов’язаними пакетами, які розвиваються разом.

Модуль (Module) – це колекція елементів: функцій, структур, трейтів, блоків impl та інших модулів.

Трейт (Trait) – це колекція методів, визначених для невідомого типу: Self. Вони можуть отримувати доступ до інших методів, оголошених в тому ж трейті. Трейти можуть бути реалізовані для будь-якого типу даних

Фіча (Feature) це механізм мови програмування Rust, для вираження умовної компіляції та необов’язкових залежностей.

ВСТУП

Актуальність роботи

У сучасному світі програмування є однією з найважливіших та найпоширеніших галузей діяльності, яка впливає на розвиток науки, техніки, освіти, культури та інших сфер життя. Програмування дозволяє створювати різноманітні програми та застосунки, які спрощують, прискорюють та покращують роботу людей, а також розширюють їх можливості та перспективи. Однак, програмування також має свої виклики та проблеми, які потребують постійного вдосконалення та інновацій.

Однією з таких проблем є публікація коду, тобто процес, який дозволяє розробникам розповсюджувати свої бібліотеки та програми через інтернет, щоб інші користувачі могли їх використовувати, модифікувати, вдосконалювати та інтегрувати у свої проекти. Це сприяє співпраці, обміну знаннями, повторному використанню, стандартизації та підвищенню якості коду. Однак, цей процес також має свої недоліки, такі як складність, часовитрата, помилки, несумісність, залежності а також ризик порушення авторських прав.

Rust є однією з найшвидших та найбезпечніших мов програмування, яка використовує концепцію крейтів для організації коду. Крейти можуть бути бібліотеками або виконуваними, вони містять модулі, функції, структури, переліки, трейти, макроси та інші елементи коду. Крейти можуть бути публіковані на crates.io, який є офіційним реєстром крейтів Rust.

Однак, публікація крейтів вимагає виконання багатьох кроків, таких як створення облікового запису, налаштування проекту, написання метаданих, перевірка коду, підписання коду, завантаження коду, оновлення версії, синхронізація з GitHub та іншими платформами, вирішення проблем та інше. Ці кроки можуть бути складними, займати багато часу, можливості допустити помилки та можуть залежати один від одного.

Тому, публікація крейтів потребує автоматизації, тобто процесу, який дозволяє розробникам публікувати свої крейти на crates.io швидко, легко, ефективно та безпечно, з мінімальним або відсутнім втручанням людини.

Автоматизація публікації крейтів має багато переваг, таких як зменшення складності, часу, помилок, несумісності, залежності, ризику порушення правил, а також підвищення якості, продуктивності, надійності, безпеки та задоволення розробників та користувачів.

Об’єктом дослідження є процес автоматизованої публікації коду в контексті мови програмування Rust і платформи crates.io.

Предмет дослідження є метод автоматизації публікації крейтів на crates.io.

Мета дослідження полягає в розробці ефективного та точного методу автоматизації публікації крейтів на crates.io. Критерієм ефективності будемо вважати кількість операцій та час, необхідний для публікації крейтів, а критерієм точності буде кількість помилок.

Завдання дослідження включають аналіз ручного процесу публікації, визначення його недоліків, розробку нового методу та програмного забезпечення, що його реалізує для автоматизованої публікації та їхньої оцінки ефективності та точності.

Методи дослідження. Для вирішення поставлених задач було використано наступні методи дослідження:

- аналіз документації: Вивчення документації Rust і crates.io для розуміння поточного процесу публікації крейтів;
- спостереження: Спостереження за процесом публікації крейтів розробниками для виявлення можливих проблем і викликів;
- експеримент як вивчення явища[1], процесу чи об’єкта дослідження з метою виявлення невідомих його властивостей чи якостей або перевірки правильності теоретичних положень, які визначаються певною науковою ідеєю;
- теорія графів, способи представлення графів, підграфи, алгоритми на графах, такі як топологічне сортування, обхід графу.

Ці методи дозволять провести глибоке та всебічне дослідження, що сприятиме досягненню мети дослідження.

Наукова новизна полягає в розробці нового методу автоматизації публікації крейтів на crates.io, який може зменшити складність, час, помилки,

несумісність, залежності, ризику порушення правил, а також підвищити якість, продуктивність, надійність, безпеку та задоволення розробників та користувачів.

Практичне значення полягає в тому, що розроблений метод може бути використаний розробниками Rust для публікації своїх крейтів на crates.io, що сприятиме розвитку спільноти Rust і покращенню якості коду.

Апробація результатів дослідження та публікації.

Результати дослідження були представлені на науково-практичній конференції “Наука і сталий розвиток транспорту 2023", а також "Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті"[2] та опубліковані в збірнику тез доповідей.

1 ОГЛЯД ПРОБЛЕМИ ПУБЛІКАЦІЇ КРЕЙТІВ

Для аналізу проблеми необхідно оцінити сучасний рівень її вирішення, виявити які аспекти проблеми вже розв'язані, які залишаються невирішеними або потребують удосконалення, розглянути існуючі програмні рішення, з'ясувати їх переваги та недоліки, та на підставі зібраної інформації визначити функціонал проектованого програмного продукту.

1.1 Поточний стан вирішення проблеми

На сучасному етапі розробки у мові програмування Rust існують кілька значущих проблем, пов'язаних з процесом публікації крейтів. Перш за все, варто зазначити, що процес публікації вимагає дотримання певних правил та рекомендацій, які можуть бути незрозумілими для багатьох розробників. Це може впливати на якість та доступність нових бібліотек.

Проблема виконання рутинних операцій [3]. Одним з найбільш часовитратних аспектів процесу публікації крейтів є ретельна перевірка Cargo.toml файлу кожного крейта на наявність і коректність необхідних полів і залежностей. Cargo.toml файл є основним конфігураційним файлом для крейта, який містить інформацію про назву, версію, авторів, ліцензію, опис, залежності, цільові платформи, функції, ресурси та інше. Ця інформація є важливою для ідентифікації крейта, його сумісності з іншими крейтами та його функціонування на різних платформах. Якщо Cargo.toml файл містить помилки або пропуски, це може призвести до проблем з публікацією, компіляцією або використанням крейта. Вирішенням цієї проблеми займаються майже усі пов’язані рішеня, котрі будуть розглянуті у наступному розділі.

Проблема зростання складності. При розробці великих проектів зазвичай використовуються воркспейси, які дозволяють групувати декілька крейтів, що можуть залежати один від одного. Однак, для публікації таких проектів потрібно окремо завантажувати кожен крейт у воркспейсі, дотримуючись правильної послідовності, оновлення залежностей та версій. Це може призвести до помилок та невідповідності між крейтами в одному проекті. Було зроблено багато спроб вирішення цієї проблеми, але остаточного рішення вона не має.

Проблема циклічних залежностей. Ще одним викликом, який ускладнює процес публікації крейтів, є відсутність підтримки циклічних залежностей між крейтами, навіть для dev-залежностей, які потрібні тільки для розробки. Dev-залежності - це залежності, які використовуються тільки для тестування, документації, прикладів або інших цілей, пов’язаних з розробкою. Вони не включаються в збірку крейта, а тому не впливають на його поведінку, продуктивність або сумісність. Однак, для публікації крейта, потрібно, щоб він не мав циклічних залежностей з іншими крейтами, навіть якщо ці залежності є dev-залежностями. Це означає, що якщо два крейти використовують один одного для тестування, то вони не можуть бути опубліковані. В основному рішення цієї проблеми покладається на розробника.

Останньою, але не менш важливою проблемою, є високі вимоги до ресурсів, які необхідні для публікації крейтів. Публікація кожного крейта вимагає виконання усіх перевірок і компіляцій, включаючи залежності, що може займати багато часу. Крім того, публікація кожного крейта вимагає створення окремої папки з усіма файлами і залежностями, що може займати багато місця на диску. Це може бути проблематичним для розробників, які мають обмежені ресурси або працюють з великими проектами.

1.2 Огляд програмних аналогів

1.2.1 Використання вбудованого інструменту публікації “cargo publish”

“cargo publish” виступає як ключовий інструмент у мові програмування Rust для публікації крейтів у репозиторії crates.io. Важливо відзначити, що цей інструмент має особливу значущість в екосистемі Rust, оскільки він використовується як стандартний механізм публікації, і інші інструменти зазвичай взаємодіють з ним.

Основна мета “cargo publish” полягає в забезпеченні мінімально необхідного набору дій для виконання процесу публікації. Він включає в себе кроки, які забезпечують стабільність і надійність публікації, але зобов’язує розробника вирішувати усі проблеми пов’язані з перевіркою, та визначенням дій необхідних перед виконанням завантаженням.

Ще однією важливою особливістю “cargo publish” є те, що він є вбудованим інструментом, що постачається разом з cargo - системою управління пакетами для Rust. Це спрощує використання, оскільки розробникам не потрібно встановлювати додаткові інструменти або конфігурувати додаткові параметри.

Важливою рисою “cargo publish” є те, що багато інших інструментів та сценаріїв для публікації крейтів у Rust використовують саме його як основу. Розробники часто максимально використовують “cargo publish”, доповнюючи його за потреби, оскільки він забезпечує надійну та добре вивірену основу для цього процесу.

Таким чином, “cargo publish” виступає не лише як окремий інструмент, але і як ключовий компонент у великій екосистемі управління пакетами Rust, що сприяє простоті та надійності публікації крейтів.

1.2.2 Використання інструменту “cargo release”

“cargo release” являється потужним інструментом для автоматизації процесу випуску крейтів у Rust та управління версіями. Його важливою перевагою є можливість працювати з ворспейсами, оновлювати залежності при зміні версій та перевіряти необхідність у публікації.

Ще однією з переваг цього інструменту є функція автоматичного оновлення “readme” файлу, а також генерація журналу, що відображає зміни у коді, заснованого на історії коммітів.

Додатково, хоча підтримка робочих гілок у Git дозволяє зручно випускати релізи з робочих гілок, це також може створити проблеми, якщо не враховувати потенційні конфлікти при об'єднанні змін у головну гілку.

У великому і динамічному проекті “cargo release” може виявитися корисним, але важливо уважно враховувати його відомі обмеження та визначити, які конкретні функції найбільше підходять для конкретного випадку використання.

Крім того, важливо підкреслити, що “cargo release” зобов’язує розробника самостійно перевіряти та впевнюватись, що він знаходиться на коректній гілці, а його робочий стан синхронізований з віддаленим репозиторієм та не має

конфліктів. Також, інструмент виконує перевірку на необхідність публікації, що може бути реалізовано за допомогою Git або Gitoxide. Важливо відмітити, що “cargo release” також оновлює теги у Dockerfile-ах, що може бути корисно для забезпечення консистентності та відслідковування версій у середовищі контейнеризації. Ці функції спрямовані на полегшення розробникам завдань, пов'язаних з публікацією та управлінням версіями, але водночас вони вимагають від користувача уважності та самостійної перевірки перед використанням.

1.2.3 Використання розширеного інструменту “cargo smart-release”

“cargo smart-release”[4] представляє собою розширений інструмент для автоматизації випуску та управління версіями крейтів у Rust. Його унікальність полягає в тому, що він ефективно враховує циклічні залежності, що, за деякими відгуками, реалізовано більш ефективно порівняно з іншими аналогами.

Однією з важливих особливостей “cargo smart-release” є здатність виявляти необхідність у випуску та не виконувати його, якщо зміни в крейті чи його залежностях невеликі чи не впливають на стабільність. Це сприяє уникненню зайвих релізів та зменшенню версійного шуму.

Крім того, “cargo smart-release” автоматично публікує змінені залежності разом з бажаним для публікації, спрощуючи процес управління залежностями та забезпечуючи їхню консистентність.

Інструмент також використовує git теги для перевірки необхідності публікації. Хоча цей підхід може викликати обмеження, оскільки порівняння по git тегам може не завжди точно відображати всі зміни, це може бути проблемою для деяких випадків, де важлива точність індикації необхідності публікації.

Важливо відзначити, що “cargo smart-release” жорстко прив'язаний до імені "origin" та використовує бібліотеку gitoxide у процесі розробки для взаємодії з git. Це робить його стабільним та потужним інструментом для тих, хто шукає високий рівень автоматизації та управління версіями у Rust-проектах.

1.2.4 Використання оптимізуючого інструменту “cargo workspaces”

“cargo workspaces” у мові програмування Rust представляє собою потужний механізм для управління проектами, які складаються з декількох крейтів та розвивається як єдиний воркспейс. Його унікальність полягає у тому, що він дозволяє об'єднати кілька пакетів під одним деревом та спростити управління залежностями та версіями.

“cargo workspaces” особливо добре підходить у випадках, коли всі модулі воркспейсу призначені для виконання однієї цілі чи пов'язані з однією загальною задачею. Зміни в одному модулі автоматично впливатимуть на інші, що полегшує розробку та утримання великих та складних проектів.

Однією з ключових переваг “cargo workspaces” є те, що він забезпечує уніфікований процес управління залежностями для всього воркспейсу. Всі пакети воркспейсу можуть спільно використовувати один і той же Cargo.toml файл, що спрощує відстеження та синхронізацію залежностей.

Додатково, “cargo workspaces” надає можливість легко керувати внутрішніми залежностями між модулями воркспейсу та гнучко конфігурувати окремі модулі, які можуть використовувати спільні ресурси чи функціонал.

Також важливо відзначити, що “cargo workspaces” забезпечує високий рівень ізоляції між модулями, що дозволяє зберігати їх незалежними та неспільними, якщо це необхідно.

Отже, “cargo workspaces” є інструментом, який сприяє ефективній розробці та управлінню великими проектами у мові програмування Rust, зокрема тоді, коли всі модулі воркспейсу орієнтовані на вирішення спільної задачі.

1.2.5 Можливості, що не було виявлено в жодному аналогічному програмному продукті

Жодне з розглянутих рішень не передбачає автоматичної перевірки тестів перед публікацією файлів. Крім того, вони не показують, які саме файли будуть опубліковані, і розробник повинен виконувати окремі команди для цього. Однак, ці команди можуть бути невідомі розробнику, або він може забути про них і опублікувати небажані або конфіденційні файли.

1.3 Призначення методу

Функціональне призначення методу автоматизованої публікації крейтів полягає у автоматизованій публікації крейтів використовуючи спеціальні програмні продукти та рішення, який виконує необхідні перевірки, контролює зміну версій та фіксацію змін, виконує автоматичне тестування, розв’язує циклічні залежності, компілює та публікує крейти. Метод автоматизованої публікації крейтів приймає на вхід список крейтів, які потрібно опублікувати, і виконує усі необхідні кроки для їх публікації, повертаючи результат у вигляді повідомлення про успішну або невдалу публікацію кожного крейта.

Експлуатаційне призначення методу полягає у спрощенні та прискорені процесу публікації крейтів, уникнувши ручного виконання необхідних дій. Метод дозволяє розробникам фокусуватися на створенні якісного коду, а не на технічних деталях публікації. Розроблений метод може бути застосований до будь-яких крейтів, які відповідають вимогам публікації, а також до воркспейсів, котрі складаються з кількох крейтів.

1.4 Постановка задачі

Розробити метод для автоматизованої публікації крейтів, провести проектування, розробку й тестування програмного забезпечення, що реалізовуватиме розроблений раніше метод, провести тестування розробленого додатку, також необхідно провести дослідження, для підтвердження ефективності розроблених методу та програмного забезпечення порівняно з ручним виконанням та за допомогою аналогічних рішень.

Висновки до розділу 1

У цьому розділі було розглянуто чотири основні проблеми, які виникають при публікації крейтів: рутинні перевірки Cargo.toml файлів, зростаюча складність, циклічні залежності та вимоги до ресурсів. Було проаналізовано, як існуючі рішення вирішують або не вирішують ці проблеми, а також виявлено їх переваги та недоліки.

З огляду на результати аналізу, зроблено висновки, подані нижче.

Проблема рутинних перевірок Cargo.toml файлів була частково вирішена деякими рішеннями, але не всіма. Ця проблема впливає на якість та надійність публікації крейтів, а також на час, необхідний для їх підготовки.

Проблема зростаючої складності була вирішена краще, але з компромісами. Ця проблема впливає на зручність та ефективність публікації крейтів, а також на їх сумісність та залежності. Тому необхідно розробити рішення, яке б забезпечувало оптимальну та гнучку організацію крейтів, а також їх оновлення та підтримку.

Проблема циклічних залежностей була залишена на користувача, або частково вирішена у деяких рішеннях. Ця проблема впливає на стабільність та безпеку публікації крейтів. Тому необхідно розробити рішення, яке б забезпечувало виявлення та усунення циклічних залежностей, а також їх запобігання.

Проблема з вимогами до ресурсів може стати критичною при спробі опублікувати велику кількість крейтів. Ця проблема впливає на швидкість та доступність публікації крейтів, а також на їх розподіленість та масштабованість. Тому необхідно розробити рішення, яке б забезпечувало мінімізацію та оптимізацію використання ресурсів, а також їх адаптацію до різних умов.

Таким чином, огляд проблеми показав, що існуючі програмні аналоги не відповідають повною мірою поставленим вимогам, а також що є можливості для подальшого покращення та вирішення недоліків. Отже, завдання з дослідження, розробки методу та програмного продукту, який буде реалізовувати розроблений метод, є актуальним.

2 РОЗРОБКА МЕТОДУ

Цей розділ присвячений розробці методу, який задовольняє функціональні вимоги поставленої задачі, а також надає додаткові можливості для підвищення якості та ефективності її виконання. У розділі будуть розглянуті основні стадії процесу публікації крейтів, а також описано метод та його складові алгоритми, що реалізують ці стадії.

2.1 Опис методу

Розроблюваний метод потрібен для автоматизації процесу публікації крейтів на платформі crates.io. Складається з наступних етапів: визначення крейтів, що необхідно опублікувати та порядок їх публікації, виконання публікації та перевірка успішності виконання. Використовує алгоритми у наступному порядку:

1. Отримати від користувача перелік крейтів, що бажані до публікації
2. Впевнитись, що користувач виконав авторизацію на crates.io
3. Побудувати граф проекту, враховуючи отриманий перелік крейтів з залежностями
4. Фільтрація графу
5. Топологічне сортування отриманого графу
6. Виконання публікації крейтів з отриманої послідовності
7. Перевірка успішності завантаження

2.2 Алгоритми роботи з графом

В даній роботі пропонується представлення проекту у вигляді графу, де вершинами є крейти, а ребрами є відношення цих крейтів. Орієнтація графа демонструє ієрархічну структуру залежностей, а ваги ребер вказують на тип залежності.у

2.2.1 Побудова графа

Граф [5] є візуалізацією структури воркспейсу, який складається з різних крейтів та їхніх залежностей. Для побудови графу потрібно спочатку отримати список учасників воркспейсу, які можуть бути локальними крейтами, що знаходяться у межах поточного воркспесу на файловій системі користувача, або

зовнішніми крейтами, що завантажуються з репозиторію або іншого джерела. Потім потрібно створити вузли з крейтами та їхніми залежностями, при цьому уникаючи дублювань, які можуть виникнути через звертання до одного крейту декілька разів. Нарешті, потрібно створити ребра між вузлами, які показують зв'язок між крейтами та їхніми залежностями. Ребро направлене від крейту до його залежності, яка визначається за допомогою Cargo.toml файлу крейта.

2.2.2 Фільтрація графу

Для фільтрації графу необхідно отримати околиці вершин з крейтів, що користувач бажає опублікувати. «Окіл вершини» – це породжений підграф всіх суміжних вершин обраної вершини, а «породжений підграф» це інший граф, утворений з підмножини вершин графа разом з усіма ребрами, що з’єднують пари вершин з цієї підмножини.

Наприклад, на рис. 2.1 показано повний граф воркспейсу, а користувачу треба здійснити публікацію крейту 3. Отже, немає потреби публікувати крейти 1 та 2, на відміну від крейтів 4 та 5, які мають важливе значення, оскільки є залежностями для крейту, який планується до публікації. Тому підграф, що показаний на рис. 2.2, буде результатом фільтрації цього графа.

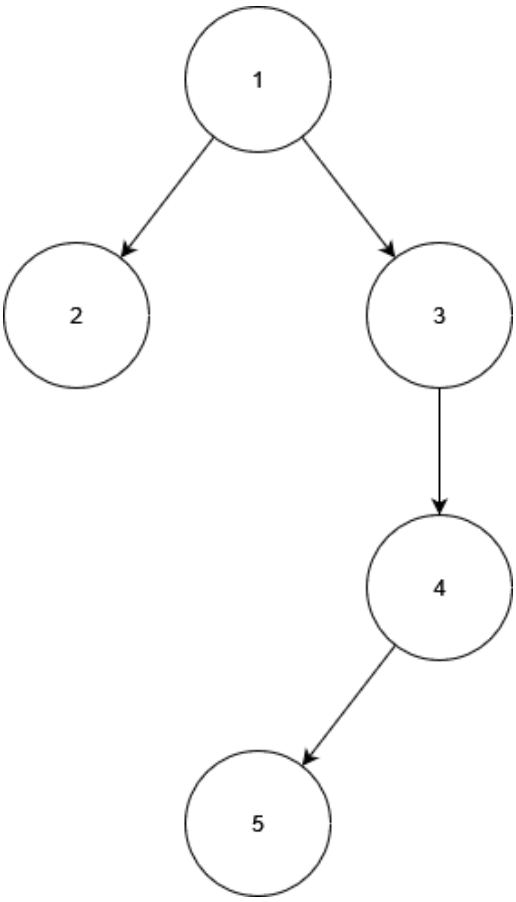

Рисунок 2.1 – Повний граф

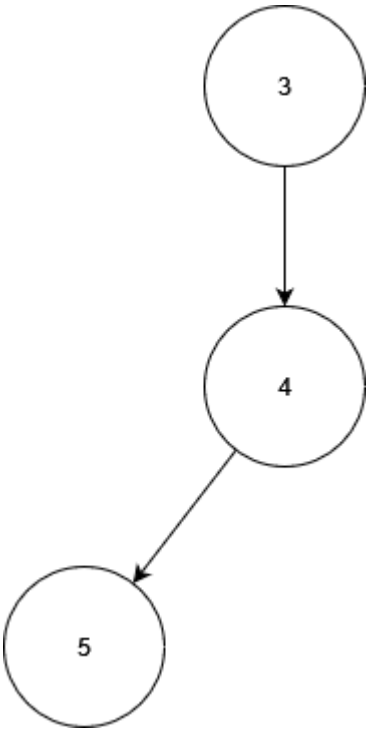

Рисунок 2.2 – Породжений підграф

Далі необхідно прибрати з графу вершини, що позначають крейти, що не заплановано публікувати, зменшуючи тим самим обсяг роботи та уникаючи непотрібної публікації. Для цього необхідно пройти по отриманому підграфу в ширину, перевірити кожен Cargo.toml файл на наявність поля publish, якщо це поле вказано ця вершина не повинна бути опублікована у загальнодоступному регістрі crates.io. Також, з графу слід прибрати віддалені крейти, бо користувач не має права публікувати крейти інших користувачів, тобто з графу слід прибрати ті крейти, що не належать поточному воркспейсу, котрі могли бути попередньо додані через залежності. Крім того, з графу слід прибрати залежності, що необхідні тільки для тестування, документації або інших цілей що не впливають на функціональну частину крейту та виконат перевірку необхідності публікації для отриманого підграфу.

Наприклад, на рис. 2.3, граф проекту має віддалений крейт, позначений червоним, та залежність, що потрібна лише для розробки, показану пунктиром. Якщо не застосувати фільтрацію, що описана, то виникне неможливість у встановленні послідовності публікації крейтів через циклічність залежностей, адже для публікації крейту 1 треба спочатку опублікувати крейт 3, а для публікації крейту 3 треба спочатку опублікувати крейт 1. Тому підграф, що демонструється на рис. 2.4, буде результатом фільтрації графу проекту, з якого вилучені всі віддалені залежності та залежності, що необхідні тільки для розробки.

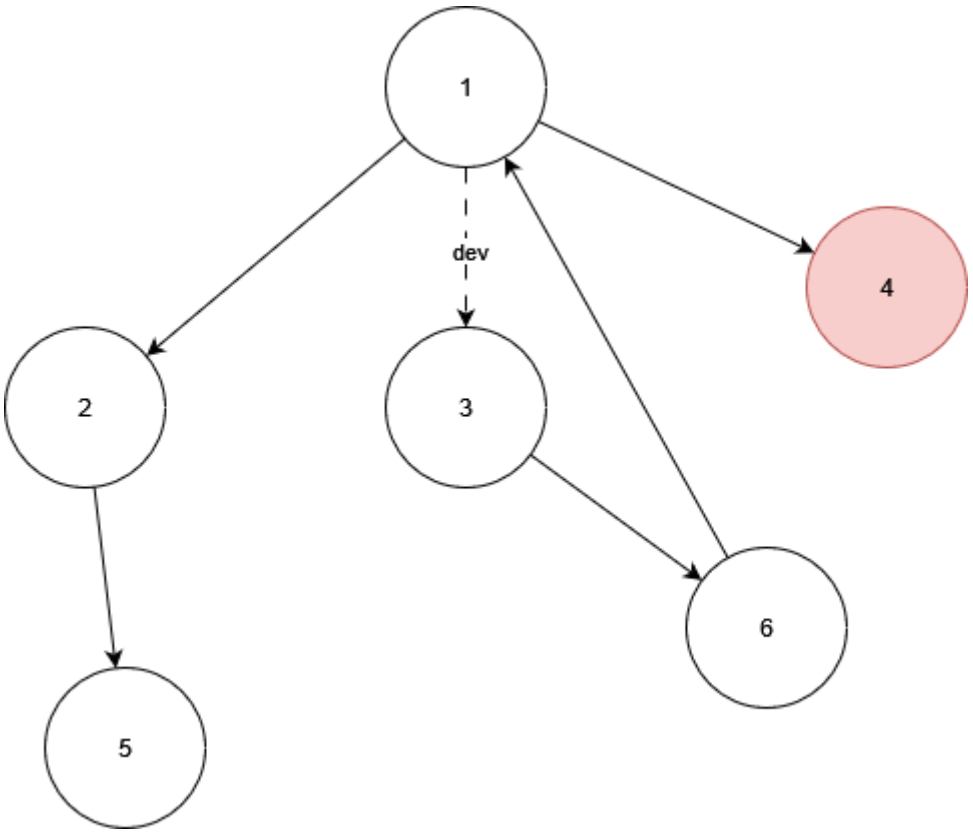

Рисунок 2.3 – Повний граф проекту

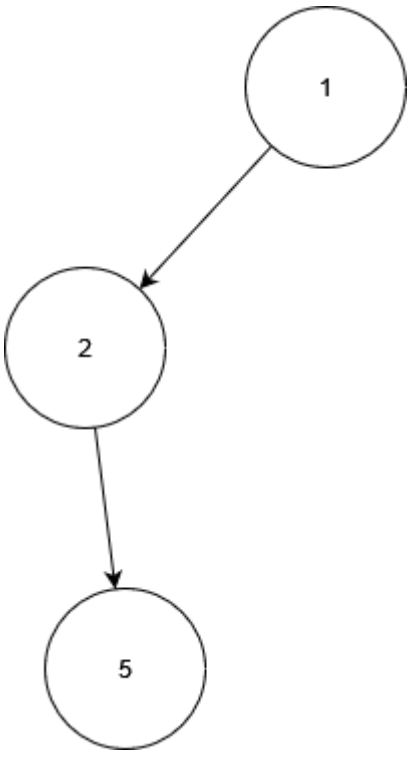

Рисунок 2.4 – Відфільтрований граф

2.2.3 Топологічне сортування

Топологічне сортування графу це лінійне упорядкування його вершин так, що для кожного спрямованого ребра, вершина у яку направлене ребро, буде йти після вершини з якої було спрямоване це ребро. Точніше, топологічне сортування – це обхід графа, в якому кожна вершина відвідується тільки після того, як відвідуються всі її залежності. Топологічне упорядкування можливе тільки тоді, коли граф не має спрямованих циклів, тобто якщо він є спрямованим ациклічним графом.

Наприклад, є граф, зображений на рис. 2.5, у якому є вершина, що не має зв’язків зовсім та вершина 4, з якою пов’язані одразу дві вершини, а саме 2 та 5.

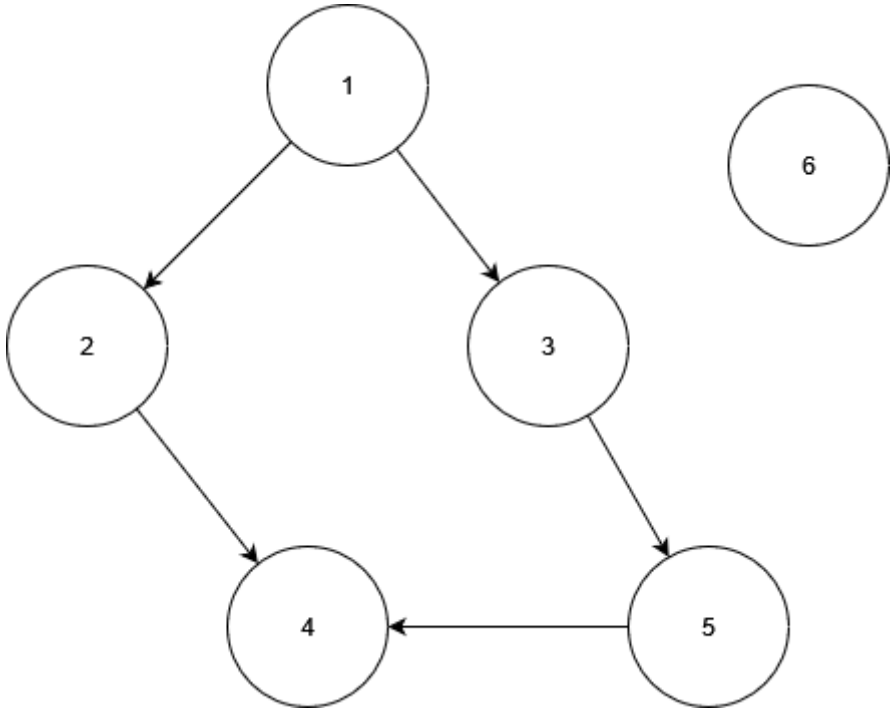

Рисунок 2.5 – Повний граф, призначений для топологічного сортування

Результатом топологічного сортування для наведеного графу може бути одна з наступних послідовностей: [6, 4, 5, 2, 3, 1], [6, 4, 5, 3, 2, 1], [4, 6, 5, 2, 3, 1], [4, 6, 5, 3, 2, 1], [4, 6, 2, 5, 3, 1], [6, 4, 2, 5, 3, 1].

Усі варіанти, що наведені, є правильними результатами топологічного сортування, оскільки вони задовольняють умову, за якою кожен елемент, що має залежності, йде після елемента, від якого залежить.

2.3 Перевірка необхідності публікації

Для того, щоб уникнути надлишкових публікацій та змін версій крейтів, які можуть призвести до плутанини, необхідно порівняти опубліковану версію з локальною версією. Це можна зробити шляхом створення попередньої упаковки крейта, який планується публікувати, за допомогою утиліти cargo package. Результатом цієї операції є файл з розширенням “crate”, який містить всі файли крейта. Потім необхідно завантажити з регістру опубліковану версію крейта, який представляє аналогічний файлу з розширенням “crate”. Наступним кроком є відкриття обох файлів як архівів та порівняння вмісту кожного файлу локального крейта з відповідним файлом в опублікованій версії. Якщо виявлено будь-які відмінності, то це означає, що крейт потребує публікації.

Однак, порівняння файлів не є єдиним критерієм для визначення необхідності публікації. Також слід враховувати, чи були внесені зміни у залежностях крейта, які призведуть до їх переопублікації. У такому випадку, крейт також має бути оновлений.

2.4 Виконання публікації крейту

Для публікації крейту слід дотримуватися деяких вимог [6], зокрема, мати ліцензію та опис. У файлі Cargo.toml необхідно вказати ліцензію у полі license або посилання на файл ліцензії у полі license-file. У разі використання файлу ліцензії, він має бути присутнім у проекті. Крім того, у файлі Cargo.toml та у файлі README.md має бути наданий опис крейту. Перед публікацією крейту необхідно переконатися, що усі зміни у проекті зафіксовані за допомогою системи контролю версій git, оскільки незафіксовані зміни можуть призвести до неконсистентності. Також слід провести тестування крейту та виконати підвищення версії, після чого можна виконати вивантаження упакованого крейту за допомогою команди “cargo publish”, результатом котрої буде повністю готовий, завантажений на crates.io крейт.

2.5 Підвищення версії крейтів

Далі буде розглянуто декілька варіантів зміни версій крейтів. Один з них полягає в тому, щоб дозволити користувачеві самостійно обирати стратегію

зміни версії, але за замовчуванням застосовувати підвищення першого не нульового числа у версії. Цей варіант є найпростішим, але не враховує можливі проблеми, пов'язані зі зміною зовнішнього інтерфейсу або вхідних та вихідних даних. Інший варіант передбачає більш детальне дослідження цих проблем і порівняння зовнішнього інтерфейсу перед і після зміни. Якщо інтерфейс не змінився, але вхідні або вихідні дані відрізняються, то це вважається ломаючою зміною, яка потребує підвищення вищого порядку версії. Цей варіант є складнішим, але більш точним і безпечним.

Крім того, слід розглянути питання оновлення версій залежностей у залежних крейтах. Якщо зміна у крейті впливає на його залежні крейти, то слід оновити версію залежності у цих крейтах, використовуючи аналогічну стратегію, за якою було підвищено крейт. Це дозволить уникнути конфліктів версій і забезпечити сумісність. Загальна стратегія каже, що якщо зміна була серйозна і порушує зворотну сумісність, то слід підвищувати мажорну версію, якщо серйозна, але не порушує сумісність, то мінорну, якщо не серйозна і не порушує, то патч. Однак, визначення серйозності та порушення зворотної сумісності є доволі складними. Так, видалення публічно доступного елементу точно порушує зворотну сумісність, але зміна внутрішньої реалізації, що повертає інший результат, також порушує зворотну сумісність. Отже, вирішено обрати компромісне рішення, з можливістю залишити користувачу вибір, а також врахувати потенційну помилку, коли локально може бути застаріла версія.

2.6 Тестування

Для того, щоб гарантувати сумісність рішення користувача з підтримуваними версіями Rust і різними комбінаціями фіч, необхідно тестувати його за всіма можливими умовами. Ручне тестування є часомістким і схильним до помилок, тому автоматизація цього процесу значно полегшить розробнику публікацію крейтів.

Висновки до розділу 2

У цьому розділі було розроблено та описано метод автоматизованої публікації крейтів на crates.io.

Метод базується на побудові та фільтрації графу проекту, який відображає залежності між крейтами. Він складається з наступниї стадій: отримання вхідних даних від користувача, авторизація на crates.io, фільтрація графу, топологічне сортування, публікація крейтів, та перевірка результату. Запропонований метод задовольняє функціональні вимоги поставленої задачі та надає додаткові можливості для підвищення якості та ефективності її виконання. До цих можливостей належать врахування залежностей між крейтами, уникнення публікації крейтів без змін, забезпечення послідовності публікації крейтів, та автоматизація процесу публікації крейтів.

Ці можливості дозволяють уникнути конфліктів версій та забезпечити сумісність крейтів, зменшити обсяг передаваних даних та зберегти зрозумілість версіонування, зберегти логіку та структуру проекту, зменшити кількість людських помилок, пришвидшити виконання, та спростити використання.

Метод може бути використаний для публікації будь-яких крейтів, які відповідають формату crates.io, та може бути адаптований до різних сценаріїв використання. Метод має також деякі обмеження та виклики, пов'язані з необхідністю мати доступ до інтернету та авторизації на crates.io, враховувати можливі зміни в інтерфейсі та політиці crates.io, та забезпечити безпеку та конфіденційність даних, які передаються та зберігаються на сервері. Метод відкриває перспективи для подальшого розвитку, такі як розширення функціональності методу, оптимізація алгоритмів методу, та інтеграція методу з іншими регістрами, окрім crates.io.

Таким чином, у цьому розділі було представлено метод автоматизованої публікації крейтів, котрий передбачається бути ефективним та надійним рішенням для задачі публікації крейтів на crates.io.

3 ПРОЕКТУВАННЯ Й РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

У цьому розділі буде описано процес проектування, розробки та тестування програмного продукту, що реалізовуватиме розроблений у другому розділі метод.

3.1 Формалізація задачі

На рівні зовнішнього проектування задача формалізована за допомогою діаграми варіантів використання. Система взаємодіє з користувачем, який є актором, через варіанти використання. Варіанти використання описують функціональність, яку система надає акторам.

Варіанти використання системи зображені на рис. 3.1.

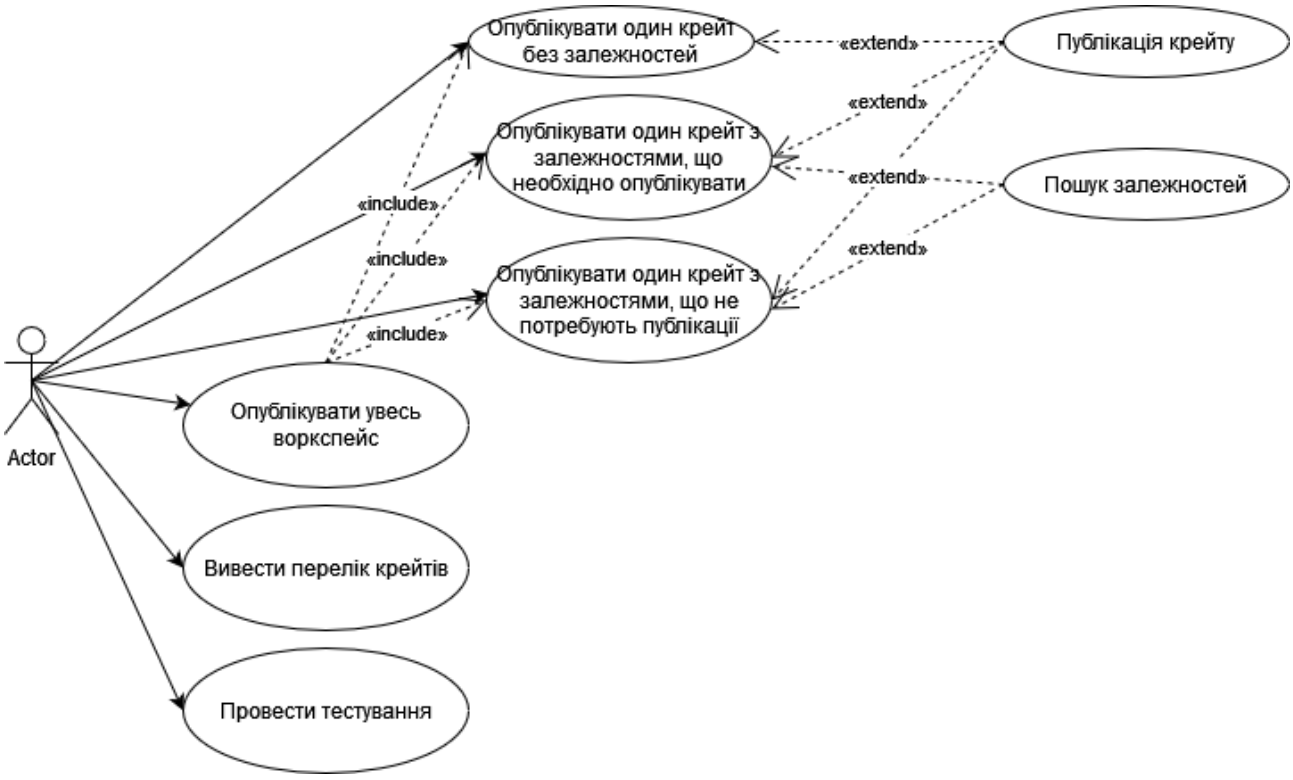

Рисунок 3.1 – Варіанти використання системи

Мінімальна послідовність дій необхідна для публікації крейту:

- перевірити наявність ліцензії та опису крейту;
- перевірити, що зміни зафіксовані;
- перевірити необхідність публікації;
- підвищити версію;
- зафіксувати зміну версії;

- виконати завантаження крейту до репозиторію crates.io.

При піблікації одного крейту без залежностей повинен виконуватися мінімальна необхідна послідовність дій для публікації крейту.

Публікація одного крейту з залежностями, що необхідно опублікувати потребує наступних дій: побудова графу залежностей від обраного графу, фільтрація графу за необхідністю у публікації та виконання мінімально необхідної послідовності дій для публікація для кожного крейту з графу дотримуючись послідовності.

Для публікації усього воркспейсу необхідно перелічити у аргументах програмного продукту усі крейти з воркспейсу, після чого для кожного виконуватиметься послідовність дій представлена у пунктах з публікацією крейту з залежностями чи без.

Для виведення переліку крейтів користувач може обрати формат виводу у вигляді дерева чи топологічно відсортований список та обрати фільтр, котрий залишатиме тільки локальні крейти.

При проведенні тестування користувачу необхідно надати шлях до крейту для якого треба провести тестування, після чого він буде протестований з усіма комбінаціями фіч цього крейту й буде відображено результати виконання усіх тестів.

3.2 Базова архітектура системи

У процесі проектування було виконано декомпозицію модулів системи на три рівня згідно з моделлю «Модель-вигляд-котролер», результати якої наведені у табл. 3.1.

Таблиця 3.1 – Результати декомпозиції модулів, за архітектурним шаблоном MVC

Model	Controller	View
Tools	Endpoints	Commands
Cargo		Wca
Git		
Manifest		
Version		
Package		
workspace		

Обрані парадигми програмування: об’єктно-орієнтована (у вигляді процедурної з об’єктною системою, що базується на типажах реалізації) та функціональна.

За допомогою архітектурного шаблону MVC програма розділяється на три відокремлені, проте взаємозалежні компоненти. Компонент Модель виконує необхідний функціонал, компонент Вигляд забезпечує отримання, перетворення та передавання вхідних даних від користувача до компонента Контролер, який за правилами бізнес логіки управляє компонентами Моделі для досягнення мети.

3.3 Внутрішнє проектування

У цьому розділі буде описано процес внутрішнього проектування системи, а саме, наведено опис компонентів та їх взаємодію, перелік необхідних шаблонів проектування та приклади їх використання для розроблюваного проекту також буде наведено проектування динаміки системи, де буде наведено як описані компоненти будуть взаємодіяти між собою.

Результатом цього розділу вважається готовий проект за яким можна реалізувати програмний продукт.

3.3.1 Використані шаблони проектування

Перелік шаблонів проектування необхідний для реалізації проекту: прототип, адаптер, фасад, ланцюжок обов’язків та будівельник.

Далі наведено описи цих шаблонів проектування та приклади їх використання для розроблюваного проекту.

Прототип – це породжувальний патерн проектування, що дає змогу копіювати об’єкти, не вдаючись у подробиці їхньої реалізації.

Проблема: якщо потрібно створити копію об’єкта, можна спробувати створити новий об’єкт того ж класу і перенести в нього всі значення полів з оригінального об’єкта. Але цей спосіб має декілька недоліків. По-перше, до деяких полів може не бути доступу, якщо вони є приватними. По-друге, точний тип об’єкта може не бути відомий, якщо він визначається за інтерфейсом, а не за класом.

Рішення: за допомогою патерну Прототип дозволяється копіювання об’єктів самими собою, надаючи їм спільний інтерфейс, який містить метод clone. Цей метод повинен бути реалізований кожним класом, який підтримує клонування. Таким чином, копія об’єкта може бути створена, не залежно від його типу і внутрішньої структури. Об’єкт, який служить зразком для клонування, називається прототипом.

Приклад: є структура, що використовує інші, котрі імплементують інтерфейс Clone, у якості своїх полів:

```
pub struct CmdReport
{
  /// Command that was executed.
  pub command : String,
  /// Path where command was executed.
  pub path : PathBuf,
  /// Stdout.
  pub out : String,
  /// Stderr.
  pub err : String,
}
```

Для того, щоб надати цій структурі можливість копіювати, для неї слід імплементувати відповідний інтерфейс Clone наступним чином:

```
impl Clone for CmdReport
{
    fn clone(&self) -> Self
    {
      Self
      {
        command: self.command.clone(),
        path: self.path.clone(),
        out: self.out.clone(),
```

```
            err: self.err.clone(),
        }
    }
}
```

Або за допомогою макроса, що імплементує це автоматично.

Для копіювання цього об’єкта достатньо буде викликати метод clone, котрий поверне копію об’єкта.

Адаптер – це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом.

Проблема: програма використовує формат файлів XML для обміну даними, але потрібно використати бібліотеку, яка використовує інший формат файлів, який не підтримується поточним рішенням. Одним з можливих рішень є переписання бібліотеки для підтримки необхідного формату, але це може призвести до значних часових витрат, а також додаткових проблем з підтримкою цього коду. Також може бути проблема з закритим кодом бібліотеки.

Рішення: створено адаптер, який трансформує інтерфейс або дані одного об’єкта так, щоб вони стали «зрозумілими» іншому об’єкту. Один з об’єктів згорнуто адаптером таким чином, що інший об’єкт не знає про його існування. Наприклад, за допомогою адаптера створено об’єкт, який викликає функції з іншого програмного продукту та надає програмний інтерфейс для розроблюваного ПЗ.

Приклад: компонент Cargo, котрий є адаптером до програмного продукту cargo:

```
pub fn package< P >( path : P, dry : bool ) -> Result< CmdReport > { … }
pub fn publish< P >( path : P, dry : bool ) -> Result< CmdReport > { … }
```

Методи котрого приймають аргументи у програмних типах, запускають утиліту cargo з відповідними параметрами та конвертують результат цієї утиліти в структури програмного продукту.

Фасад – це структурний патерн проектування, який надає простий інтерфейс до складної системи, бібліотеки або фреймворку.

Проблема: потрібно працювати з багатьма об’єктами складної бібліотеки чи фреймворку. Це вимагає самостійного контролю за правильністю викликів методів, ініціалізацією об’єктів, тощо. Це призводить до того, що бізнес логіка сильно залежить від деталей реалізації, що ускладнює розуміння та підтримку коду.

Рішення: використати патерн Фасад, який надає простий інтерфейс для роботи зі складною підсистемою. Фасад може бути абстрактним представленням системи, тобто не реалізовувати весь функціонал, а лише використовувати необхідну частину підсистеми.

Приклад: компоненти з компоненту endpoint.

```
pub fn publish( patterns : Vec< String >, dry : bool ) -> Result< PublishReport, ( PublishReport, Error ) >
{
  // Використання структур та методів з інших компонентів
  // для взаємодії з воркспейсом та крейтами.
  // А саме, для виконання публікації.
}
```

Ланцюжок обов’язків – це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

Проблема: потрібно робити багато послідовних перевірок, які впливають на подальше виконання. Це можна зробити за допомогою умовних операторів, але це призведе до того, що код стане дуже великим, нечитабельним та складним для підтримки. З кожною новою перевіркою ця проблема тільки збільшується.

Рішення: створено окремі класи, які мають один метод виконання, де робляться необхідні перевірки. Ці класи зв’язані в один ланцюжок, де результат

виконання одного передається до наступного. Таким чином, після кожної перевірки, визначається, чи може об’єкт продовжити виконання, чи повернути помилку. Це дозволяє уникнути залежності від порядку та кількості перевірок.

Приклад:

```
pub fn perform< S >( &self, program : S ) -> Result< (), Error >
where
 S : AsRef< str >
{
 let program = program.as_ref();

 let raw_program = self.parser.program( program ).map_err( | e | Error::Validation( ValidationError::Parser( e ) ) )?;
 let grammar_program = self.grammar_converter.to_program( raw_program ).map_err( | e | Error::Validation( ValidationError::GrammarConverter( e ) ) )?;
 let exec_program = self.executor_converter.to_program( grammar_program ).map_err( | e | Error::Validation( ValidationError::ExecutorConverter( e ) ) )?;

 if let Some( callback ) = &self.callback_fn
 {
  callback.0( program, &exec_program )
 }

 self.executor.program( exec_program ).map_err( | e | Error::Execution( e ) )
}
```

У цьому прикладі, вхідними даними є рядок, а у результаті повністю коректна програма, що буде виконана. Ланцюжком є перетворення рядку у результат після виконання парсингу, після чого цей об’єкт перевіряється на граматичну коректність, а наприкінці граматично вірна програма перетворюється на виконувану.

Будівельник – це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Проблема: потрібно ініціалізувати складний об’єкт з багатьма полями та вкладеними об’єктами. Це може призвести до того, що створено конструктор з великою кількістю аргументів та складною реалізацією для перетворення цих аргументів на необхідний об’єкт, особливо якщо більшість полів мають значення за замовчуванням. Або ще гірше, створено безлічі методів, що встановлюють окремі поля цього об’єкта, через що користувач може використовувати ці методи неправильно і порушувати логіку використання.

Рішення: використано патерн Будівельник, який пропонує виокремити конструювання об’єкта в окремий клас, завдання якого буде «будівництво» об’єкта. У цього класу повинні бути методи, що будуть вказувати(«будувати») об’єкт по крокам, вони можуть бути досить комплексними, головне щоб їх мета була побудувати кінечний об’єкт. А у кінці, після виклику усіх необхідних методів, повинен бути викликаний метод, що поверне готовий об’єкт початкового класу.

Приклад:

```
let publish_command = wca::Command::former()
   .hint( "Publish package on `crates.io`." )
   .long_hint( "Publish package on `crates.io`." )
   .phrase( "publish" )
   .subject( "A path to package. Should be a directory with file `Cargo.toml`.", Type::List( Type::String.into(), ',' ), true )
   .property( "dry", "Run command dry. Default is false.", Type::String, true )
   .property( "verbosity", "Setup level of verbosity.", Type::String, true )
   .property_alias( "verbosity", "v" )
   .form();
```

3.3.1 Опис компонентів та їх взаємодії

У проекті буде створено наступні компоненти:

- Command – визначає типи команд, які може приймати та виконувати ПЗ;
- Wca відповідає за парсинг та обробку команд, отриманих від користувача. Він перевіряє, чи є команда дійсною, чи відповідає вона конфігурації ПЗ, та чи є необхідні ресурси для її виконання;
- Endpoint реалізує бізнес-логіку команд, які обробляє ПЗ. Він викликає відповідні функції та методи для кожної команди, та повертає результати або помилки;
- Manifest зберігає інформацію про Cargo.toml файл, який є маніфестом крейта. Він дозволяє читати, змінювати та зберігати дані про назву, версію, залежності, цільові платформи та інші параметри крейта;
- Package надає функціонал для роботи з крейтами, які є пакетами;
- Workspace надає функціонал для роботи з крейтами, які є воркспейсами;
- CratesTools надає функціонал для завантаження та читання запакованих крейт-файлів. Запакований крейт-файл - це архівний файл, який містить вихідний код пакета та його метадані. Цей компонент дозволяє отримувати крейти з репозиторію crates.io, розпаковувати їх, переглядати їх вміст та аналізувати їх структуру;
- Version необхідний для модифікації версій крейтів;
- Cargo виступає у ролі огортки для утиліт cargo. Цей компонент дозволяє використовувати cargo для виконання різних операцій з крейтами, таких як створення, компіляція, тестування, публікація тощо;
- Git – виступає у ролі огортки для утиліт git. Цей компонент дозволяє використовувати git для роботи з віддаленими репозиторіями крейтів, таких як GitHub, GitLab, Bitbucket тощо;
- wTools надає додаткові інструменти для полегшення розробки проекту;
- Tools є збірником інструментів для виконання поставленої мети, що не відносяться до бізнес-логіки проекту. Ці інструменти включають функції для роботи з графами, процесами та шляхами у файловій системі.

На рис. 3.2 наведено взаємодію перелічених вище компонентів.

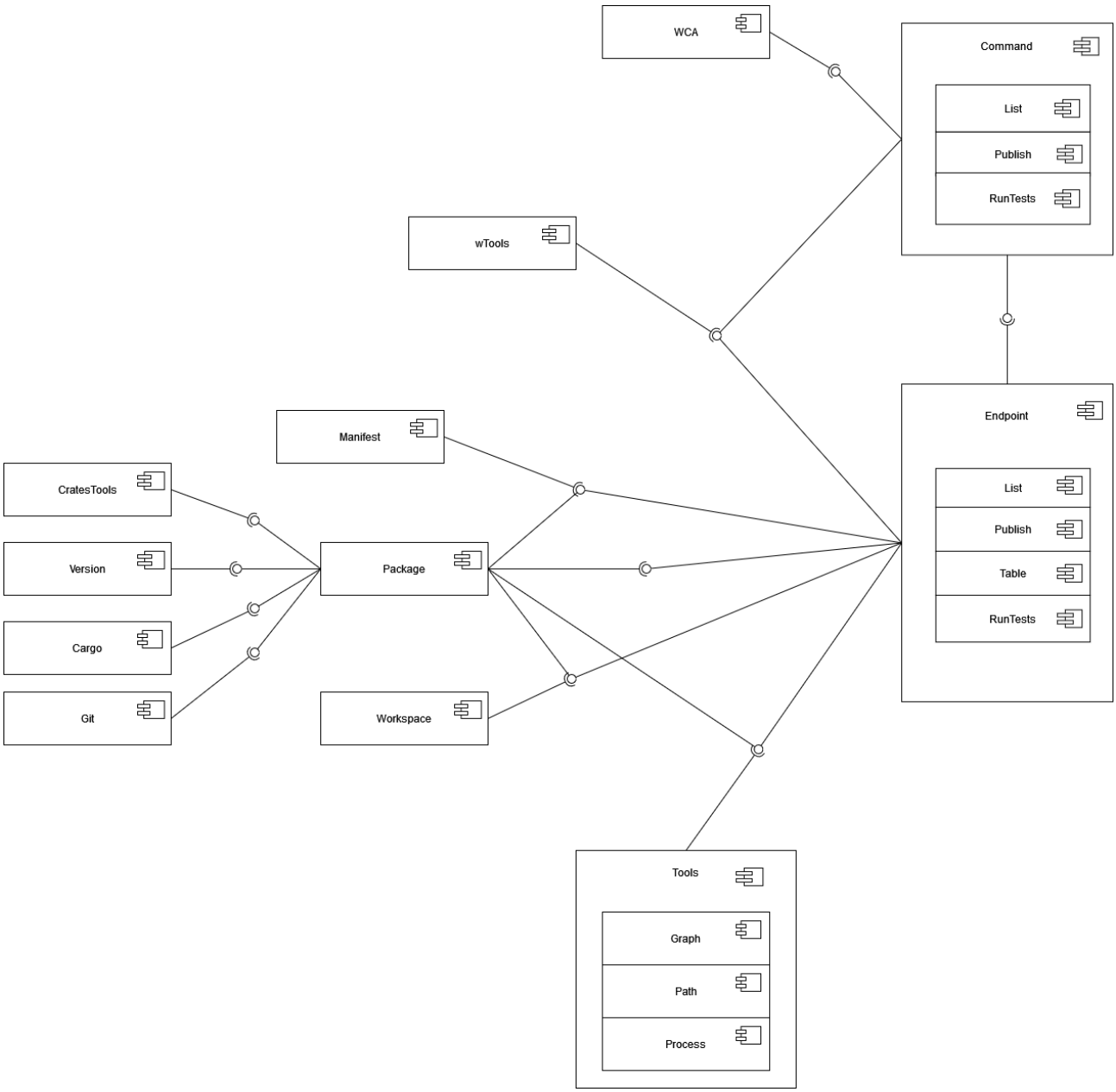

Рисунок 3.2 – Діаграма компонентів

3.3.3 Проектування динаміки системи

Ініціалізація ПЗ зображена на рис. 3.3. виконується за будь яким виконанням програми. А саме, виконується ініціалізація усіх доступних команд, вказується функції, що повинні бути виконанні для кожної команди, та відбувається обробка вхідних даних користувача, результатом якої є виконані відповідні команди, що викликав користувач.

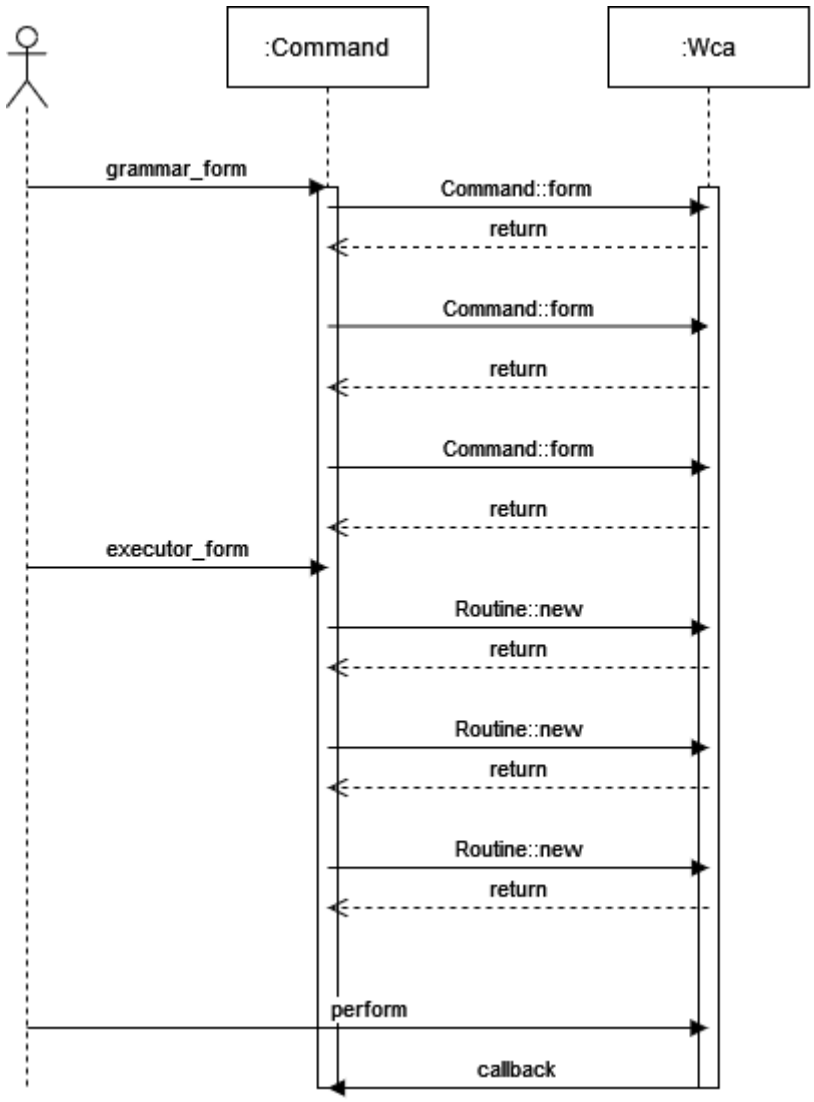

Рисунок 3.3 – Діаграма динаміки системи для ініціалізації та виконання команд

Поведінка системи під час виконання команди publish, яка призводить до опублікування крейту, показана на рис. 3.4. Діаграма демонструє процес публікації тільки для одного крейту, проте для публікації більшої кількості крейтів відмінність буде тільки у частині, де викликається функція publish_single, яка повторюється стільки разів, скільки потрібно. Це можливо тому, що всі крейти, які мають бути опубліковані, встановлюються на стадії побудови графу, а потім лише здійснюється публікація кожного крейту у відповідній послідовності.

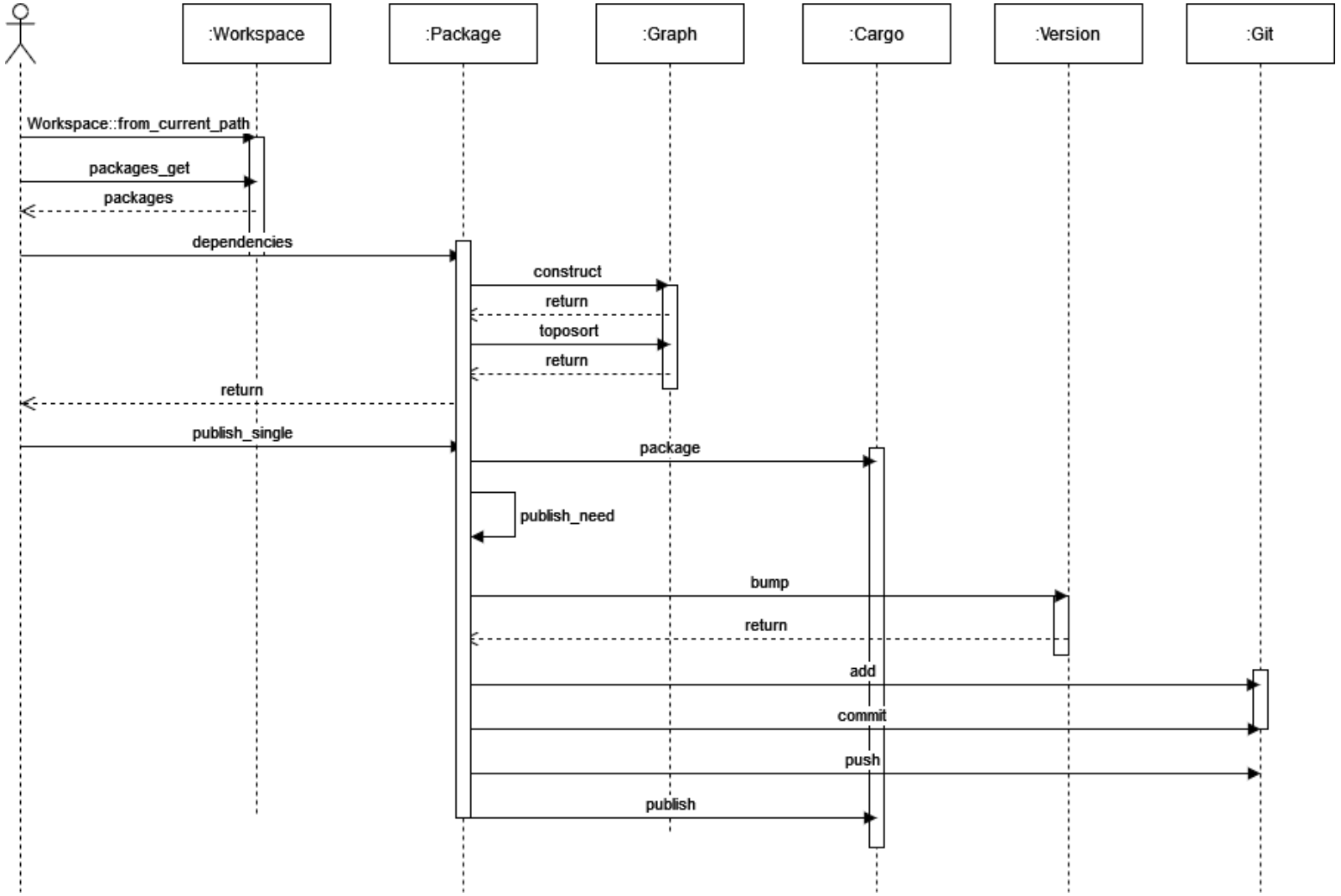

Рисунок 3.4 – Діаграма динаміки для виконання публікації

3.4 Реалізація програмного продукту

У даному розділі буде детально описано реалізацію програмного продукту з використанням мови програмування Rust. Вибір Rust як основної мови розробки обумовлено рядом факторів, які будуть розглянуті нижче, а також будуть розглянуті переваги використання цієї мови в контексті даного проекту.

3.4.1 Вибір мови програмування

З метою реалізації розроблюваного продукту була обрана мова програмування Rust, обрана з урахуванням кількох ключових аспектів. Перш за все, безпека програмування у Rust вирізняється високим рівнем завдяки вбудованим механізмам контролю пам'яті та системі власностей, що сприяє уникненню типових помилок.

Одним із факторів вибору є швидкодія, яку Rust гарантує завдяки своєму системному рівню доступу та можливостям оптимізації коду. Мова також володіє потужною системою типів та вбудованими механізмами конкурентності, що сприяють легкості масштабування розроблених рішень.

Важливим аспектом є активна та зростаюча спільнота розробників Rust, що гарантує наявність необхідної допомоги та сторонніх бібліотек для швидкого впровадження рішення. Крім того, мова має ефективні механізми для роботи з многозадачністю, що є важливим аспектом у контексті розробки методу автоматизованої публікації крейтів.

Узагальнюючи, вибір мови програмування Rust для розробки даного продукту обумовлений необхідністю забезпечення високої безпеки, продуктивності та масштабованості, роблячи її оптимальним вибором для досягнення цілей.

3.4.2 Опис середовища розробки

Rust Rover – це IDE для Rust, яка розроблена компанією JetBrains і заснована на їхній платформі IntelliJ. Rust Rover – це окремий продукт, який не потребує встановлення інших IDE від JetBrains, таких як CLion або IntelliJ IDEA.

Rust Rover має багато переваг як IDE для Rust, таких як:

• швидка і зручна робота з проектами Rust, які можна створювати, відкривати, імпортувати і запускати з IDE;

• потужні засоби редагування, відлагодження, тестування, рефакторингу і документування коду;

• підтримка Cargo, Rustup, Rustfmt і інших інструментів Rust;

• інтеграція з Git, GitHub, Docker, Kubernetes і іншими сервісами і платформами;

• розширюваність за допомогою багатьох доступних плагінів від JetBrains і спільноти;

• налаштовуваний інтерфейс і поведінка за допомогою різних параметрів і тем.

Але у Rust Rover також є деякі недоліки, наприклад:

• платний і потребує підписки для повного функціонування;

• важкий і може споживати багато ресурсів системи;

• має деякі помилки і недоліки, оскільки знаходиться в стадії Early Access Preview.

3.4.3 Використані бібліотеки

Використані бібліотеки: petgraph, semver, rayon, anyhow.

Petgraph[7] – це бібліотека структур даних для графів, яка надає кілька типів графів, алгоритми на цих графах та функціональність для виводу графів у форматі graphviz. Графи – це колекції вузлів та ребер між вузлами. Кожен вузол та ребро може мати довільні пов’язані дані, а ребра можуть бути або направленими, або ненаправленими. Типом графа, що викликає найбільший інтерес, є Graph, який є направленим або ненаправленим графом з власними змінними доступними довільними вузлами та ребрами.

Бібліотека petgraph має такі особливості:

- підтримує кілька типів графів, таких як Graph, StableGraph, GraphMap, MatrixGraph та CSR, кожен з яких має свої переваги та недоліки в залежності від сценарію використання;

- надає різноманітні алгоритми на графах, такі як пошук у ширину, пошук

у глибину, Дейкстри, A*, топологічне сортування, мінімальне охоплююче дерево, тар'яновські компоненти, максимальний потік та інші.

Крейт semver[8] призначений для реалізації інтерпретації семантичного версіонування, що використовується cargo. Семантичне версіонування – це настанова для того, як призначати і збільшувати номери версій. Вона широко застосовується в екосистемі cargo/crates.io для Rust. Крейт semver дозволяє аналізувати і порівнювати версії, що містять основні, додаткові і патч-компоненти, а також необов'язкові ідентифікатори попередніх випусків і метаданих збірки.

Rayon[9] – це бібліотека для паралельної обробки даних у мові Rust. Вона дуже легка і зручна для впровадження паралелізму у існуючий код. Вона також гарантує відсутність гонок даних. Rayon надає високорівневі паралельні конструкції, такі як паралельні ітератори, паралельне сортування, паралельне розширення колекцій та інші. Rayon також дозволяє створювати власні паралельні завдання за допомогою функцій join та scope. Rayon працює з глобальним пулом потоків, який може бути налаштований або замінений власним пулом за допомогою ThreadPoolBuilder.

Крейт anyhow[10] є бібліотекою, яка спрощує обробку помилок у додатках Rust. Вона надає тип помилки anyhow::Error, який базується на об'єкті трайту std::error::Error. За допомогою цього типу помилки можна повертати будь-яку помилку, яка реалізує трайт std::error::Error, використовуючи оператор ?. Також можна додавати контекст до помилок, щоб полегшити їх виправлення. Крім того, anyhow::Error підтримує downcasting, який дозволяє отримати доступ до базового типу помилки. Крейт anyhow працює з будь-яким типом помилки, який має реалізацію std::error::Error, включаючи ті, що визначені у власному крейті. Для створення одноразових повідомлень про помилки можна використовувати макрос anyhow!, який створює anyhow::Error. Макрос bail! надає скорочення для передчасного повернення з помилкою.

3.5 Тестування розробленого ПЗ

Будуть розроблені позитивні тести, що підтверджуватимуть опис функції, а також методом передбачення помилки.

3.5.1 Специфікація функцій

Наведення специфікацій функцій, що будуть протестовані. Усі вхідні та вихідні дані є у явному вигляді, крім випадків з поточненням вигляду вхідних чи вихідних даних.

1) Функція для знаходження усіх залежностей обраного крейту

Приймає на вхід:

- вказівник на об’єкт воркспейсу. Тип даних: Workspace;

- вказівник на об’єкт крейту, залежності котрого необхідно отримати. Тип даних: Package;

- об’єкт з опціями для отримання залежностей. Тип даних DependenciesOptions.

Вихідними даними є:

у явному вигляді: список з ідентифікаторами крейтів, що є залежностями вказанного крейду;

у не явному вигляді: завантажує інформацію про воркспейс.

Текст функції:

```
pub fn dependencies( workspace : &mut Workspace, manifest : &Package, opts: DependenciesOptions ) -> wtools::error::Result< Vec< CrateId > >
 {
  let mut graph = HashMap::new();
  let root = _dependencies( workspace, manifest, &mut graph, opts.clone() )?;

  let output = match opts.sort
  {
   DependenciesSort::Unordered =>
   {
    graph
    .into_iter()
    .flat_map( | ( id, dependency ) |
    {
     dependency
     .into_iter()
```

```
            .chain( Some( id ) )
          })
          .unique()
          .filter( | x | x != &root )
          .collect()
        }
        DependenciesSort::Topological =>
        {
          graph::toposort( graph::construct( &graph ) ).into_iter().filter( | x | x != &root ).collect()
        },
      };

      Ok( output )
    }
```

2) Функція для перевірки необхідності публікації крейту

Приймає на вхід: крейт, необхідність публікації котрого необхідно перевірити. Тип даних: Package

Вихідними даними є: значення true – якщо публікація необхідна, false – якщо ні

```
pub fn publish_need( package : &Package ) -> bool
    {
      // These files are ignored because they can be safely changed without affecting functionality
      //
      // - `.cargo_vcs_info.json` - contains the git sha1 hash that varies between different commits
      // - `Cargo.toml.orig` - can be safely modified because it is used to generate the `Cargo.toml` file automatically, and the `Cargo.toml` file is sufficient to check for changes
      const IGNORE_LIST : [ &str; 2 ] = [ ".cargo_vcs_info.json", "Cargo.toml.orig" ];

      let name = package.name();
      let version = package.version();
      let local_package_path = local_path( &name, &version, package.crate_dir() );

      let local_package = CrateArchive::read( local_package_path ).expect( "Failed to read local package. Please, run `cargo package` before." );
```

```
    let remote_package = match CrateArchive::download_crates_io( name, version )
    {
     Ok( archive ) => archive,
     Err( ureq::Error::Status( 403, _ ) ) => return true,
     _ => /* return an error */ panic!( "Failed to load remote package" ),
    };

    let filter_ignore_list = | p : &&Path | !IGNORE_LIST.contains( &p.file_name().unwrap().to_string_lossy().as_ref() );
    let local_package_files : Vec< _ > = local_package.list().into_iter().filter( filter_ignore_list ).sorted().collect();
    let remote_package_files : Vec< _ > = remote_package.list().into_iter().filter( filter_ignore_list ).sorted().collect();

    if local_package_files != remote_package_files { return true; }

    let mut is_same = true;
    for path in local_package_files
    {
     // unwraps is safe because the paths to the files was compared previously
     let local = local_package.content_bytes( path ).unwraр();
     let remote = remote_package.content_bytes( path ).unwrap();

     is_same &= local == remote;
    }

    !is_same
   }
```

3) Функція для підвищення версії крейту

Приймає на вхід: значення версії. Тип даних: Version

Вихідними даними є: змінене значення версії

```
pub fn bump( self ) -> Self
  {
   let mut ver = self.0;
   if ver.major != 0
   {
    ver.major += 1;
    ver.minor = 0;
    ver.patch = 0;
   }
   else if ver.minor != 0
   {
```

```
    ver.minor += 1;
    ver.patch = 0;
   }
   else
   {
    ver.patch += 1;
   }

   Self( ver )
  }
```

4) Функція для виконання мінімально необхідної послідовності дій, для публікації крейту

Приймає на вхід:

- посилання на крейт, що необхідно опублікувати. Тип даних: Package;
- опція, що визначає чи потрібно виконувати зміни, чи імітувати виконання. Тип даних: bool.

Вихідними даними є: звіт з виконаними (чи запланованими до виконання) діями, для виконання публікації

```
pub fn publish_single( package : &Package, dry : bool ) -> Result<
PublishReport, ( PublishReport, Error ) >
{
  let mut report = PublishReport::default();
  if package.local_is()
  {
   return Ok( report );
  }

  let package_dir = &package.crate_dir();

  let output = cargo::package( &package_dir, false ).context( "Take information
about package" ).map_err( | e | ( report.clone(), e ) )?;
  if output.err.contains( "not yet committed")
  {
   return Err(( report, anyhow!( "Some changes wasn't committed. Please,
commit or stash that changes and try again." ) ));
  }
  report.get_info = Some( output );

  if publish_need( &package )
  {
   report.publish_required = true;
```

```
    let mut files_changed_for_bump = vec![];
    // bump version in the package manifest
    let new_version = version::bump( &mut package.manifest(), dry ).context(
"Try to bump package version" ).map_err( | e | ( report.clone(), e ) )?;
    files_changed_for_bump.push( package.manifest_path() );

    let package_name = package.name();

    // bump the package version in dependents(so far, only workspace)
    let workspace_manifest_dir : AbsolutePath = Workspace::with_crate_dir(
package.crate_dir() ).workspace_root().try_into().unwrap();
    let workspace_manifest_path = workspace_manifest_dir.join( "Cargo.toml" );

    // qqq: should be refactored
    if !dry
    {
      let mut workspace_manifest = manifest::open(
workspace_manifest_path.as_ref() ).map_err( | e | ( report.clone(), e ) )?;
      let workspace_manifest_data =
workspace_manifest.manifest_data.as_mut().unwrap();
      workspace_manifest_data
      .get_mut( "workspace" )
      .and_then( | workspace | workspace.get_mut( "dependencies" ) )
      .and_then( | dependencies | dependencies.get_mut( &package_name ) )
      .map
      (
        | dependency |
          {
            let previous_version = dependency.get( "version" ).and_then( | v |
v.as_str() ).unwrap().to_string();
            if previous_version.starts_with( '~' )
            {
              dependency[ "version" ] = value( format!( "~{new_version}" ) );
            }
          }
      );
      workspace_manifest.store().unwrap();
    }

    files_changed_for_bump.push( workspace_manifest_path );
    let files_changed_for_bump : Vec< _ > =
files_changed_for_bump.into_iter().unique().collect();
    let objects_to_add : Vec< _ > = files_changed_for_bump.iter().map( | f |
f.as_ref().strip_prefix( &workspace_manifest_dir ).unwrap().to_string_lossy()
).collect();
```

```
        report.bump = Some( BumpReport { package_name : package_name.to_string(), new_version : new_version.clone(), changed_files : files_changed_for_bump.clone() } );

        let commit_message = format!( "{package_name}-v{new_version}" );
        let res = git::add( workspace_manifest_dir, objects_to_add, dry ).map_err( | e | ( report.clone(), e ) )?;
        report.add = Some( res );
        let res = git::commit( package_dir, commit_message, dry ).map_err( | e | ( report.clone(), e ) )?;
        report.commit = Some( res );
        let res = git::push( package_dir, dry ).map_err( | e | ( report.clone(), e ) )?;
        report.push = Some( res );

        let res = cargo::publish( package_dir, dry ).map_err( | e | ( report.clone(), e ) )?;
        report.publish = Some( res );
       }

      Ok( report )
     }
```

5) Функція для публікації крейтів, рекурсивно враховуючи їх залежності

Приймає на вхід:

- перелік шляхів до крейтів, що необхідно опублікувати. Тип даних: Vec<String>;
- опція, що визначає чи потрібно виконувати зміни, чи імітувати виконання. Тип даних: bool.

Вихідними даними є: звіт з виконаними (чи запланованими до виконання) діями, для виконання публікації

```
pub fn publish( patterns : Vec< String >, dry : bool ) -> Result< PublishReport, ( PublishReport, Error ) >
 {
  let mut report = PublishReport::default();

  let mut paths = HashSet::new();
  // find all packages by specified folders
  for pattern in &patterns
  {
   let current_path = AbsolutePath::try_from( std::path::PathBuf::from( pattern ) ).map_err( | e | ( report.clone(), e.into() ) )?;
   paths.extend( Some( current_path ) );
```

```
  }

  let mut metadata = if paths.is_empty()
  {
   Workspace::from_current_path().map_err( | e | ( report.clone(), e.into() ) )?
  }
  else
  {
   // patterns can point to different workspaces. Current solution take first random path from list
   let current_path = paths.iter().next().unwrap().clone();
   let dir = CrateDir::try_from( current_path ).map_err( | e | ( report.clone(), e.into() ) )?;

   Workspace::with_crate_dir( dir )
  };
  report.workspace_root_dir = Some( metadata.workspace_root().try_into().unwrap() );

  let packages_to_publish : Vec< _ >= metadata
  .load()
  .packages_get()
  .iter()
  .filter( | &package | paths.contains( &AbsolutePath::try_from( package.manifest_path.as_std_path().parent().unwrap() ).unwrap() ) )
  .cloned()
  .collect();
  let mut queue = vec![];
  for package in &packages_to_publish
  {
   let local_deps_args = DependenciesOptions
   {
    recursive: true,
    sort: DependenciesSort::Topological,
    ..Default::default()
   };
   let deps = package::dependencies( &mut metadata, &Package::from( package.clone() ), local_deps_args )
   .map_err( | e | ( report.clone(), e.into() ) )?;

   for dep in deps
   {
    if !queue.contains( &dep )
    {
     queue.push( dep );
```

```
          }
         }
         let crate_id = CrateId::from( package );
         if !queue.contains( &crate_id )
         {
          queue.push( crate_id );
         }
        }

        for path in queue.into_iter().filter_map( | id | id.path )
        {
         let current_report = package::publish_single( &Package::try_from( path.clone() ).unwrap(), dry )
          .map_err
          (
           | ( current_report, e ) |
           {
            report.packages.push(( path.clone(), current_report.clone() ));
            ( report.clone(), e.context( "Publish list of packages" ).into() )
           }
          )?;
          report.packages.push(( path, current_report ));
        }

        Ok( report )
       }
```

3.5.2 Опис тест-кейсів

У наступному переліку тест-кейсів, крейти мають умовні назви, котрі повинні бути унікальними у межах crates.io репозиторію.

Розглянемо тест-кейси для функції «dependencies»:

- chain_of_three_packages

Передумови: воркспейс з трьома крейтами a, b, c, де a залежить від b, а b залежить від c.

Кроки для виконання: створити об’єкт Workspace за шляхом до воркспейсу, викликати тестовану функцію з воркспейсом, шляхом до крейту a та об’єктом DependenciesOptions зі значенням за замовчуванням.

Очікувані результати: список з ідентифікаторів двох крейтів, а саме b та c у довільній послідовності.

- chain_of_three_packages_topologically_sorted

Передумови: воркспейс з трьома крейтами a, b, c, де a залежить від b, а b залежить від c.

Кроки для виконання: створити об’єкт Workspace за шляхом до воркспейсу, викликати тестовану функцію з воркспейсом, шляхом до крейту a та об’єктом DependenciesOptions де значення поля sort є DependenciesSort::Topological, а усі інші значення за замовчуванням.

Очікувані результати: список з ідентифікаторів двох крейтів, а саме b та c у топологічно відсортованій послідовності, де на першомі місці повинен бути крейт c, а на другому крейт b.

- package_with_remote_dependency

Передумови: воркспейс з двома крейтами a, b, де a залежить від b та довільного, доступного, віддаленого крейту.

Кроки для виконання: створити об’єкт Workspace за шляхом до воркспейсу, викликати тестовану функцію з воркспейсом, шляхом до крейту a та об’єктом DependenciesOptions зі значенням за замовчуванням.

Очікувані результати: список з одним ідентифікатором крейту b

- workspace_with_cyclic_dependency_a

Передумови: воркспейс з двома крейтами a, b, де a залежить від b та b залежить від a.

Кроки для виконання: створити об’єкт Workspace за шляхом до воркспейсу, викликати тестовану функцію з воркспейсом, шляхом до крейту a та об’єктом DependenciesOptions зі значенням за замовчуванням.

Очікувані результати: список з одним ідентифікатором крейту b.

- workspace_with_cyclic_dependency_b

Передумови: воркспейс з двома крейтами a, b, де a залежить від b та b залежить від a.

Кроки для виконання: створити об’єкт Workspace за шляхом до воркспейсу, викликати тестовану функцію з воркспейсом, шляхом до крейту b та об’єктом DependenciesOptions зі значенням за замовчуванням.

Очікувані результати: список з одним ідентифікатором крейту a.

Розглянемо текст-кейси для функції «publish_need»:

- no_changes

Передумови: один вже опублікований крейт, яким володіє тестувальник, цей крейт повинен повністю співпадати з опублікованою версією.

Кроки для виконання: виконати запакування наведеного крейту за допомогою команди cargo package, викликати тестовану функцію з об’єктом Package котрий відповідає наведеному крейту.

Очікувані результати: тестована функція повинна повернути значення false.

- with_changes

Передумови: один вже опублікований крейт, яким володіє тестувальник, цей крейт повинен повністю співпадати з опублікованою версією.

Кроки для виконання: внести зміни, що впливають на функціонал крейту(наприклад: додати нову функцію), виконати запакування наведеного крейту за допомогою команди cargo package, викликати тестовану функцію з об’єктом Package котрий відповідає наведеному крейту.

Очікувані результати: тестована функція повинна повернути значення true.

Розглянемо текст-кейси для функції «Version::bump»:

- patch

  Вхідні дані: об’єкт зі значенням версії «0.0.0».

  Очікувані вихідні дані: об’єкт зі значенням версії «0.0.1».

- minor_without_patches

  Вхідні дані: об’єкт зі значенням версії «0.1.0».

  Очікувані вихідні дані: об’єкт зі значенням версії «0.2.0».

- minor_with_patch

  Вхідні дані: об’єкт зі значенням версії «0.1.1».

  Очікувані вихідні дані: об’єкт зі значенням версії «0.2.0».

- major_without_patches

  Вхідні дані: об’єкт зі значенням версії «1.0.0».

  Очікувані вихідні дані: об’єкт зі значенням версії «2.0.0».

- major_with_minor

  Вхідні дані: об’єкт зі значенням версії «1.1.0».

  Очікувані вихідні дані: об’єкт зі значенням версії «2.0.0».

- major_with_patches

  Вхідні дані: об’єкт зі значенням версії «1.1.1».

  Очікувані вихідні дані: об’єкт зі значенням версії «2.0.0».

Розглянемо текст-кейси для функції «publish_single»:

- publish_new_crate

Передумови: один ще не опублікований крейт, з назвою, що не присутня у репозиторії crates.io, з верією «0.1.0».

Кроки для виконання: викликати тестовану функцію з об’єктом Package котрий відповідає наведеному крейту, та значенням false для опції dry.

Очікувані результати: у репозиторії crates.io завантажений відповідний крейт з верією «0.1.0».

- publish_new_crate_version

Передумови: один вже опублікований крейт з версією «0.1.0» та мати внесені та зафіксовані, за допомогою git, зміни, що впливають на функціональність.

Кроки для виконання: викликати тестовану функцію з об’єктом Package котрий відповідає наведеному крейту, та значенням false для опції dry.

Очікувані результати: у репозиторії crates.io завантажений відповідний крейт з верією «0.2.0».

- try_to_publish_crate_without_changes

Передумови: один вже опублікований крейт, що повністю співпадає з опублікованою версією.

Кроки для виконання: викликати тестовану функцію з об’єктом Package

котрий відповідає наведеному крейту, та значенням false для опції dry.

Очікувані результати: жодних змін не призведено, відображено повідомлення, про те що крейт не має змін.

- try_to_publish_crate_without_commiting_changes

Передумови: один вже опублікований крейт та мати внесені але не зафіксовані, за допомогою git, зміни, що впливають на функціональність.

Кроки для виконання: викликати тестовану функцію з об’єктом Package котрий відповідає наведеному крейту, та значенням false для опції dry.

Очікувані результати: жодних змін не призведено, відображено повідомлення, про те що зміни не були зафіксовані.

Розглянемо текст-кейси для функції «publish»:

- publish_crate_without_dependencies

Передумови: один крейт та мати внесені й зафіксовані, за допомогою git, зміни, що впливають на функціональність та не мати жодної залежності.

Кроки для виконання: викликати тестовану функцію зі списком з одним елементом, що містить шлях котрий відповідає наведеному крейту, та значенням false для опції dry.

Очікувані результати: опубліковано один, вказаний крейт.

- publish_crate_with_dependecies_not_required_to_publish

Передумови: воркспейс з двома крейтами a, b, де a залежить від b. Крейт a має внесені й зафіксовані, за допомогою git, зміни, що впливають на функціональність, а крейт b – не має залежностей.

Кроки для виконання: викликати тестовану функцію зі списком з одним елементом, що містить шлях котрий відповідає крейту a, та значенням false для опції dry.

Очікувані результати: опубліковано один, крейт a, а крейт b залишається без змін.

- publish_crate_with_dependecies_that_required_publishing

Передумови: воркспейс з трьома крейтами a, b, c, де a залежить від b, а b

залежить від c. Й у кожному з перелічених крейтів присутні зафіксовані зміни.

Кроки для виконання: викликати тестовану функцію зі списком з одним елементом, що містить шлях котрий відповідає крейту a, та значенням false для опції dry.

Очікувані результати: опубліковані усі три крейти у наступній послідовності: c, b, a.

3.5.3 Аналіз результатів тестування

У результаті проведення процедури тестування була виявлена логічна невідповідність у процесі публікації нового крейту. Згідно з очікуваними результатами тесту publish_new_crate, опублікована версія функції publish_single мала б мати позначку версії «0.1.0». Однак, після виконання необхідних кроків, було встановлено, що опублікований крейт має позначку версії «0.2.0».

Одним із можливих способів вирішення даної проблеми є додавання додаткової перевірки, яка б визначала, чи існує опублікована версія з такою ж позначкою версії, як і поточна. У випадку, коли крейт з такою позначкою версії не був опублікований раніше, зміна версії не мала б відбуватися.

Виконання всіх інших тест кейсів пройшло з очікуваними результатами, що свідчить про достатню надійність розробленого ПЗ для виконання більшості задач з публікації крейтів.

Висновки до розділу 3

Результатом цього розділу є розроблений програмний продукт на основі розробленого раніше методу.

У процесі розробки була визначені архітектура проекту та використовувані шаблони проектування, за якими можливо реалізувати проект за допомогою різних мов програмування. Після чого було визначено мову програмування, середовище та необхідні залежності для імплементації описаного продукту. Також підчас написання імплементації для програмного продукту були розроблені тести, що перевіряли та підтримували стійкість написаного коду.

У підсумку, ці підходи забезпечили високий рівень надійності та стабільності програмного продукту, демонструючи важливість ретельного тестування та керування залежностями у розробці якісного програмного забезпечення.

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МЕТОДУ ТА ПЗ

Метою цього розділу є описати хід та процес виконання експериментів, порівняти виконання експериментів з використанням розробленого методу та ПЗ, з використанням аналогічних рішень та вручну. У результаті проведення цих експериментів очікується визначити переваги розробленого методу над аналогами з точки зору обчислювальної складності та часової ефективності.

4.1 Опис експериментів

Наведемо перелік експериментів, що дозволять оцінити швидкість, обчислювальну складність та ефективність розробленого методу.

4.1.1 Публікація великої кількості крейтів

Мета експерименту: цей експеримент має на меті визначити швидкість публікації великої кількості крейтів та порівняти її з швидкістю аналогічних підходів.

Умова: воркспейс повинен бути повністю налаштований та включати велику кількість крейтів готових до публікації, також повинні бути відсутні зовнішні перешкоди, що можуть вплинути на процес публікації

Задача: необхідно опублікувати всі крейти в найкоротші терміни.

Метрики оцінки: швидкість публікації. Вимірюється у кількості опублікованих крейтів на одиницю часу.

Цей експеримент дозволить оцінити ефективність розробленого методу у порівнянні з існуючими підходами, зокрема в аспектах швидкості та масштабування процесу публікації.

4.1.2 Публікація крейтів у складному (з великою кількістю перетинів) воркспейсі

Мета експерименту: цей експеримент спрямований на визначення точності публікації крейтів у складному воркспейсі, де існує велика кількість взаємозв'язків та перетинів між крейтами. Мета полягає у порівнянні цієї точності з точністю аналогічних підходів.

Умови: воркспейс, у якому всі крейти тісно пов’язані між собою, утворюючи складний граф залежностей, також повинні бути відсутні зовнішні перешкоди, що можуть вплинути на процес публікації.

Задача: необхідно опублікувати один обраний крейт, при цьому врахувати всі зміни та залежності, які виникають у гілці з цим крейтом й опублікувати їх також.

Метрики оцінки: точність публікації, котра визначається через врахування всіх залежностей та змін без помилок або пропусків, порівняно з іншими методами.

Цей експеримент має велике значення для перевірки здатності методу коректно опрацьовувати складні залежності та зміни у великих та взаємопов'язаних системах, що є критично важливим для забезпечення якості та надійності публікацій.

4.1.3 Публікація усіх крейтів з воркспейсу

Мета експерименту: цей експеримент спрямований на оцінку ефективності публікації усіх крейтів з воркспейсу. Мета полягає у порівнянні ефективності цього методу з аналогічними підходами.

Умови: воркспейс містить велику кількість крейтів, у кожному з яких є зміни, що вимагають публікації, також повинні бути відсутні зовнішні перешкоди, що можуть вплинути на процес публікації.

Метрики оцінки: кількість дій, необхідних для публікації всіх крейтів.

4.2 Проведення експериментів

4.2.1 Публікація великої кількості крейтів

Розглянемо виконання експерименту вручну:

Перелік дій: змінити значення версії у кожному Cargo.toml файлі усіх крейтів, зафіксувати зміни(виконати команду git commit -am’<Повідомленні>’), для кожного крейту виконати команду cargo publish -p <Назва крейту>, де замість «<Назва крейту>» підставлено назву крейту, що публікується та представлена у Cargo.toml файлі.

Таблиця з результатами виконання, з замірами часу необхідного для публікації усіх крейтів наведена у табл. 4.1

Таблиця 4.1 – Результати виконання експерименту вручну

Кількість крейтів	Середній час публікації (у секундах)
1	16
5	118
25	666
125	-

Розглянемо виконання експерименту розробленим методом:

Перелік дій: виконати команду publish у розробленому програмному забезпеченню, та передати перелік усіх крейтів як аргумент.

Таблиця з результатами виконання, з замірами часу необхідного для публікації усіх крейтів наведена у табл. 4.2

Таблиця 4.2 – Результати виконання експерименту розробленим методом

Кількість крейтів	Середній час публікації (у секундах)
1	4
5	21
25	100
125	524

Розглянемо виконання експерименту за допомогою утиліти cargo release:

Перелік дій: змінити версію для кожного крейту на вищу, зафіксувати зміни(виконати команду git commit -am’<Повідомленні>’), виконати команду cargo release -x, слідувати інструкціям.

Таблиця з результатами виконання, з замірами часу необхідного для публікації усіх крейтів наведена у табл. 4.3

Таблиця 4.3 – Результати виконання експерименту за допомогою «cargo-release»

Кількість крейтів	Середній час публікації(у секундах)
1	18
5	123
25	671
125	-

Розглянемо виконання експерименту за допомогою утиліти cargo workspace:

Перелік дій: виконати команду cargo ws publish, слідувати наведеним інструкціям.

Таблиця з результатами виконання, з замірами часу необхідного для публікації усіх крейтів наведена у табл. 4.4

Таблиця 4.4 – Результати виконання експерименту за допомогою «cargo-workspace»

Кількість крейтів	Середній час публікації(у секундах)
1	20
5	45
25	226
125	1130

4.2.2 Публікація крейтів у складному (з великою кількістю перетинів) воркспейсі

Розглянемо виконання експерименту вручну:

Перелік дій: визначити порядок публікації крейтів, шляхом перегляду усіх залежності усіх крейтів воркспейсу, обрати крейти, що не мають не опублікованих залежностей, вони мають бути опубліковані в першу чергу, після повторювати алгоритм до тих пір поки задача не буде виконана. Для публікації кожного крейту необхідно виконати наступну послідовність дій: змінити значення версії у кожному Cargo.toml файлі усіх крейтів та у блоці залежностей залежних крейтів, зафіксувати зміни(виконати команду git commit -am’<Повідомленні>’), для кожного крейту виконати команду cargo publish -p <Назва крейту>, де замість «<Назва крейту>» підставлено назву крейту, що публікується та представлена у Cargo.toml файлі.

Таблиця з результатами виконання, з підрахунком кількості помилок під час публікації усіх крейтів наведена у табл. 4.5

Таблиця 4.5 – Результати виконання експерименту вручну

Кількість крейтів	Середня кількість помилок
1	1
5	8
25	54
125	-

Розглянемо виконання експерименту розробленим методом:

Перелік дій: виконати команду publish у розробленому програмному забезпеченню, та передати перелік усіх крейтів як аргумент.

Таблиця з результатами виконання, з підрахунком кількості помилок під час публікації усіх крейтів наведена у табл. 4.6. З якої можна побачити, що у середньому випадку корситувач, за допомогою розробленого методу не допускає помилок.

Таблиця 4.6 – Результати виконання експерименту розробленим методом

Кількість крейтів	Середня кількість помилок
1	0
5	0
25	0
125	0

Розглянемо виконання експерименту за допомогою утиліти cargo release:

Перелік дій: змінити версію для кожного крейту на вищу, зафіксувати зміни, виконати команду cargo release -x, слідувати інструкціям.

Таблиця з результатами виконання, з підрахунком кількості помилок під час публікації усіх крейтів наведена у табл. 4.7

Таблиця 4.7 – Результати виконання експерименту за допомогою «cargo-release»

Кількість крейтів	Середня кількість помилок
1	1
5	1
25	9
125	-

Розглянемо виконання експерименту за допомогою утиліти cargo workspace:

Перелік дій: виконати команду cargo ws publish, слідувати наведеним інструкціям.

Таблиця з результатами виконання, з підрахунком кількості помилок під час публікації усіх крейтів наведена у табл. 4.8. З якої можна побачити, що у

середньому випадку корситувач, за допомогою утиліти cargo-workspaces не допускає помилок.

Таблиця 4.8 – Результати виконання експерименту за допомогою «cargo-workspaces»

Кількість крейтів	Середня кількість помилок
1	0
5	0
25	0
125	0

4.2.3 Публікація усіх крейтів з воркспейсу

Розглянемо виконання експерименту вручну:

Перелік дій: визначити порядок публікації крейтів, шляхом перегляду усіх залежності усіх крейтів воркспейсу, обрати крейти, що не мають не опублікованих залежностей, вони мають бути опубліковані в першу чергу, після повторювати алгоритм до тих пір поки задача не буде виконана. Для публікації кожного крейту необхідно виконати наступну послідовність дій: змінити значення версії у кожному Cargo.toml файлі усіх крейтів та у блоці залежностей залежних крейтів, зафіксувати зміни(виконати команду git commit -am’<Повідомленні>’), для кожного крейту виконати команду cargo publish -p <Назва крейту>, де замість «<Назва крейту>» підставлено назву крейту, що публікується та представлена у Cargo.toml файлі.

Таблиця з результатами виконання, з підрахунком кількості дій необхідних для публікації усіх крейтів наведена у табл. 4.9

Таблиця 4.9 – Результати виконання експерименту вручну

Кількість крейтів	Середня кількість дій
1	3
5	28
25	75
125	-

Розглянемо виконання експерименту розробленим методом:

Перелік дій: виконати команду publish у розробленому програмному забезпеченню, та передати перелік усіх крейтів як аргумент.

Таблиця з результатами виконання, з підрахунком кількості дій необхідних для публікації усіх крейтів наведена у табл. 4.10

Таблиця 4.10 – Результати виконання експерименту розробленим методом

Кількість крейтів	Середня кількість дій
1	1
5	1
25	1
125	1

Розглянемо виконання експерименту за допомогою утиліти cargo release:

Перелік дій: змінити версію для кожного крейту на вищу, зафіксувати зміни(за допомогою команд git commit -am’<Повідомленні>’ та git push), виконати команду cargo release -x, слідувати інструкціям.

Таблиця з результатами виконання, з підрахунком кількості дій необхідних для публікації усіх крейтів наведена у табл. 4.11

Таблиця 4.11 – Результати виконання експерименту за допомогою «cargo-release»

Кількість крейтів	Середня кількість дій
1	5
5	18
25	57
125	-

Розглянемо виконання експерименту за допомогою утиліти cargo workspace:

Перелік дій: виконати команду cargo ws publish, слідувати наведеним інструкціям.

Таблиця з результатами виконання, з підрахунком кількості дій необхідних для публікації усіх крейтів наведена у табл. 4.12

Таблиця 4.12 – Результати виконання експерименту за допомогою «cargo-workspace»

Кількість крейтів	Середня кількість дій
1	3
5	3
25	3
125	3

4.3 Результати експериментів

Результати експериментів були узагальнені та представлені у вигляді таблиці та графіку з метою наглядності та зручності порівняння. Зазначені візуальні засоби дозволяють оперативно оцінити різницю в ефективності між ручним методом, існуючими аналогами та розробленим методом. Усі проведені експерименти виконані особою, яка ознайомлена з процесом публікації крейтів. Отже, для користувача, який не має відповідного досвіду, представлені результати можуть бути применшені.

Таблиця з загальними результатами виконання наведених експериментів, для кількості крейтів меншою за 26, представлена у табл. 4.13

Таблиця 4.13 – Загальні результати проведення експериментів

Метод	Середня кількість дій	Середня кількість помилок	Середній час (у хвилинах)
Вручну	35	21	4,45
Розроблений метод	1	0	0,7
cargo release	27	4	4,51
cargo workspace	3	0	1,21

Також, для кращого візуального сприйняття та порівняння результатів був створений графік відображений на рис. 4.1 (менше значення – краще).

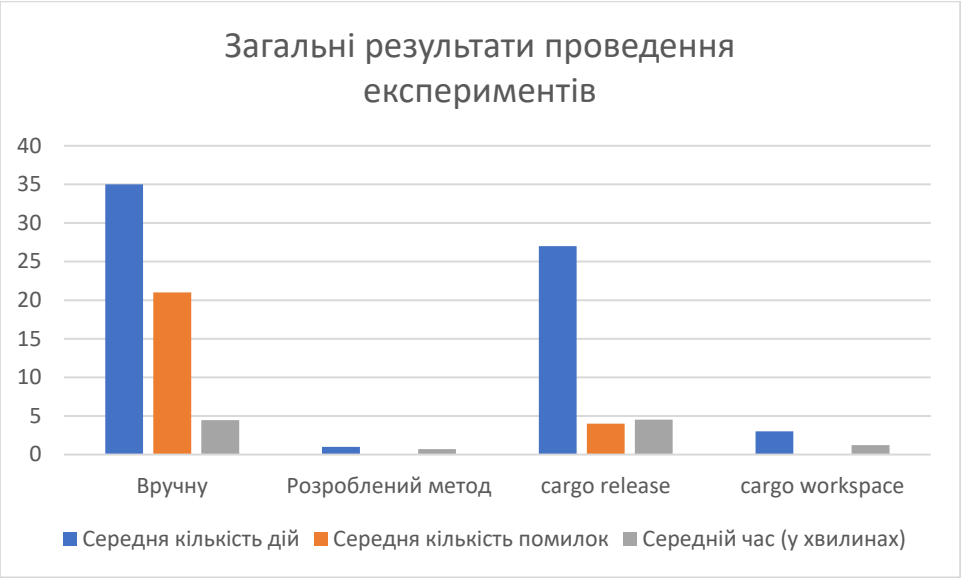

Рисунок 4.1 – Візуалізація результатів експериментів

У ході дослідження було отримано значущі результати, які висвітлюють ефективність та потенційні можливості розробленого методу автоматизованої публікації крейтів.

Спостереження показали, що використання розробленого методу значно прискорює процес публікації крейтів у порівнянні із традиційним, ручним підходом. Час, необхідний для автоматизованої публікації, зменшується у 6 разів, що свідчить про великий потенціал в економії робочого часу.

Точність розробленого методу також була вище, ніж у ручного підходу, з відсутністю помилок та недоліків, за рахунок максимально спрощеного інтерфейсу користувача. Це свідчить про високий рівень автоматизації та точності, враховуючи встановлені критерії публікації.

Стабільність та надійність розробленого методу були підтверджені під час різноманітних експериментів, що охоплювали різні умови робочого середовища. Система ефективно впоралася з зовнішніми факторами та виявила високу стійкість у реальних умовах використання.

Порівняння з альтернативними методами підтвердило переваги розробленого підходу в термінах ефективності, швидкості та точності публікації крейтів.

Загальні результати дослідження свідчать про те, що розроблений метод автоматизованої публікації крейтів є ефективним та перспективним рішенням, володіючи значним потенціалом для вдосконалення та розширення застосування у різних галузях.

Висновки до розділу 4

В результаті проведеного дослідження можна зробити кілька ключових висновків щодо розробленого методу автоматизованої публікації крейтів.

По-перше, розроблений метод виявився ефективним інструментом для значного зменшення часу, необхідного для публікації крейтів. Середній час, необхідний для використання розробленого методу, дуже значуще зменшився до 0,7 хвилини, що є важливим покращенням у порівнянні із 4,45 хвилинами при ручному використанні. Його висока швидкість дозволяє покращити продуктивність та знизити трудомісткість процесу, викорінюючи непотрібний ручний труд.

Другий важливий аспект стосується точності розробленого методу. В порівнянні з традиційним ручним підходом, система виявила високий рівень точності та низький рівень помилок. Під час виконання публікації розробленим методом не було виявлено жодної помилки, що досягається за рахунок повної автоматизації процесу виконання публікації. Це дозволяє покращити якість та консистентність публікації крейтів.

Третій висновок стосується стабільності та надійності розробленого методу в різних умовах. Система успішно впоралася з різноманітними викликами та продемонструвала стійкість, забезпечуючи надійність у реальних умовах використання.

Крім того, порівняння з альтернативними методами підтвердило конкурентоспроможність розробленого підходу. Його переваги в ефективності та точності дозволяють розглядати його як важливий та перспективний інструмент у сфері автоматизованої публікації крейтів.

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

В даній роботі була вирішена актуальна проблема публікації крейтів у веб-розробці. Оглянуті програмні аналоги, такі як "cargo publish", "cargo release", "cargo smart-release" та "cargo workspaces", виявились несуттєвими для вирішення проблеми. Запропоновано та розроблено власний метод автоматизованої публікації крейтів, що включає в себе побудову графа, фільтрацію, топологічне сортування та ефективну перевірку необхідності публікації.

У процесі проектування та розробки програмного продукту були використані формалізація задачі, базова архітектура системи, внутрішнє проектування та реалізація. Детально розглянуті шаблони проектування, опис компонентів та їх взаємодій, проектування динаміки системи та розроблені алгоритми, що забезпечують ефективну роботу системи.

Тестування програмного продукту включало в себе специфікацію функцій, створення тест-кейсів та аналіз результатів. Отримані результати підтвердили високу якість та надійність розробленого методу та програмного забезпечення.

Дослідження ефективності методу та програмного продукту, проведене через експерименти, підтвердило їх високий рівень продуктивності та демонструє переваги в порівнянні з існуючими аналогами. Результати роботи становлять важливий внесок у розвиток сучасних засобів автоматизації публікації крейтів у веб-розробці, що може знайти застосування в широкому спектрі проектів та сприяти подальшому розвитку цієї галузі.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Методологія наукових досліджень: навчальний посібник / В. І. Болдак, О. В. Кузьмін, О. В. Лозовий, О. В. Строкова; за заг. ред. В. І. Болдака. – К.: Видавничий дім «Київ. політехнік», 2021. – 131 с. [Електронний ресурс]. Режим доступу: https://ela.kpi.ua/bitstream/123456789/45110/1/Metodolohiia-2021.pdf
2. Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті:Тези XVII Міжнародної науково-практичної конференції (Дніпро, 13-14 грудня 2023 р.). – Д.: УДУНТ, 2023. – 152 с
3. Методологія наукових досліджень: навчальний посібник / В. І. Болдак, О. В. Кузьмін, О. В. Лозовий, О. В. Строкова; за заг. ред. В. І. Болдака. – К.: Видавничий дім «Київ. політехнік», 2021. – 131 с. [Електронний ресурс]. Режим доступу: https://ela.kpi.ua/bitstream/123456789/45110/1/Metodolohiia-2021.pdf
4. cargo-smart-release. [Електронний ресурс]. Режим доступу: https://crates.io/crates/cargo-smart-release
5. Теорія графів: навчальний посібник / В. В. Пасічник. – К.: Видавничий дім «Київ. політехнік», 2020. – 131 с. [Електронний ресурс]. Режим доступу: https://ela.kpi.ua/bitstream/123456789/35854/1/Teoriia_hrafiv.pdf
6. Publishing on crates.io - The Cargo Book. [Електронний ресурс]. Режим доступу: https://doc.rust-lang.org/cargo/reference/publishing.html
7. petgraph - Rust. [Електронний ресурс]. Режим доступу: https://docs.rs/petgraph/latest/petgraph/
8. semver - Rust. [Електронний ресурс]. Режим доступу: https://docs.rs/semver/latest/semver/
9. rayon - Rust. [Електронний ресурс]. Режим доступу: https://docs.rs/rayon/latest/rayon/
10. anyhow - Rust. [Електронний ресурс]. Режим доступу: https://docs.rs/anyhow/latest/anyhow/
11. Клабнік С., Ніколс К. The Rust Programming Language / С. Клабнік, К.

Ніколс ; [пер. з англ.]. - [Б. м.] : [б. в.], 2023. - 588 с.

12. UML 2. (. D. G. UML 2002-- the Unified Modeling Language: Model engineering, concepts, and tools : 5th International Conference, Dresden, Germany, September 30-October 4, 2002 : proceedings / 2002 (2002 Dresden Germany) UML. – Berlin : Springer, 2002. – 447 с.
13. ДСТУ ГОСТ 7.1-2006. Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання : чинний з 2007-07-01. – К. : Держспоживстандарт України, 2007. – 47 с. (Система стандартів з інформації, бібліотечної та видавничої справи) (Національний стандарт України).
14. Тестування програмного забезпечення [Текст]: навчальний посібник / Авраменко А.С., Авраменко В.С. Косенюк Г.В. – Черкаси : ЧНУ імені Богдана Хмельницького, 2017. – 284 с.
15. Publishing My First Rust Crate - DEV Community. [Електронний ресурс]. Режим доступу: https://dev.to/vovacodes/publishing-my-first-rust-crate-cni
16. Guru99. What is Test Driven Development (TDD)? Example. [Електронний ресурс]. Режим доступу: https://www.guru99.com/test-driven-development.html
17. Agile Alliance. What is Test Driven Development (TDD)? [Електронний ресурс]. Режим доступу: https://www.agilealliance.org/glossary/tdd/
18. Simpliaxis. Mastering the Principles of Test-Driven Development (TDD). [Електронний ресурс]. Режим доступу: https://www.simpliaxis.com/resources/principles-of-test-driven-development
19. The Top 13 Graph Theory and Algorithm Books for Fundamentals (Neo4j Blog). [Електронний ресурс]. Режим доступу: https://neo4j.com/blog/top-13-resources-graph-theory-algorithms/
20. Graph Data Structure And Algorithms - GeeksforGeeks. [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
21. Topological Sorting - GeeksforGeeks. [Електронний ресурс]. Режим

доступу: https://www.geeksforgeeks.org/topological-sorting/

22. Graph Data Structure And Algorithms - GeeksforGeeks. [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
23. Tools - Rust Programming Language. [Електронний ресурс]. Режим доступу: https://www.rust-lang.org/tools
24. 58 Rust Resources Every Learner Should Know in 2023 - DEV Community. [Електронний ресурс]. Режим доступу: https://dev.to/apollolabsbin/58-rust-resources-every-learner-should-know-in-2023-12m9
25. The top free resources to learn the Rust language - Hack2skill Station. [Електронний ресурс]. Режим доступу: https://h2s.hashnode.dev/6-free-resources-to-learn-rust-in-2022
26. The 5 best resources to learn Rust - DEV Community. [Електронний ресурс]. Режим доступу: https://dev.to/nathan20/the-4-best-resources-to-learn-rust–3238
27. Rust Documentation List | Rust Wiki. [Електронний ресурс]. Режим доступу: https://rustwiki.org/en/
28. Hardware and Software Solutions for Energy-Efficient. [Електронний ресурс]. Режим доступу: https://www.hindawi.com/journals/sp/2021/5514284/
29. Software Efficiency Matters - Efficient Go. [Електронний ресурс]. Режим доступу: https://www.oreilly.com/library/view/efficient-go/9781098105709/ch01.html
30. Analysis of the Effectiveness and Efficiency of Software Development Tools. [Електронний ресурс]. Режим доступу: https://www.atlantis-press.com/article/125963738.pdf2

ДОДАТКИ

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ

Проректор
Українського державного університету науки і технології
Анатолій РАДКЕВИЧ

Willbe

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1275-01-ЛЗ

Завідувач кафедри КІТ

_____Вадим ГОРЯЧКІН

Керівник розробки

_____Олена КУРОП’ЯТНИК

Виконавець

_____Богдан БАЛУШКІН

Нормоконтролер

_____Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО

44165850.1275-01

Willbe

Технічне завдання

Листів 13

2024

АНОТАЦІЯ

Документ 44165850.1275-01 І3 01 «Розробка методу автоматизованої публікації крейтів. Технічне завдання» входить до складу програмної документації на програму, що реалізовує додаток для автоматичної публікації крейтів.

У даному документі представлено технічне завдання. Програма написана на мові Rust. Об'єм пам'яті, що займають програми комплексу, складає 10 ГБ. Конфігурація комп'ютера стандартна. Комплекс функціонує в середовищі MS WINDOWS 10 або вище.

ВСТУП

У сучасному світі програмування є однією з найважливіших та найпоширеніших галузей діяльності, яка впливає на розвиток науки, техніки, освіти, культури та інших сфер життя. Програмування дозволяє створювати різноманітні програми та застосунки, які спрощують, прискорюють та покращують роботу людей, а також розширюють їх можливості та перспективи. Однак, програмування також має свої виклики та проблеми, які потребують постійного вдосконалення та інновацій.

Однією з таких проблем є публікація коду, тобто процес, який дозволяє розробникам розповсюджувати свої бібліотеки та програми через інтернет, щоб інші користувачі могли їх використовувати, модифікувати, вдосконалювати та інтегрувати у свої проекти. Це сприяє співпраці, обміну знаннями, повторному використанню, стандартизації та підвищенню якості коду. Однак, цей процес також має свої недоліки, такі як складність, часовитрата, помилки, несумісність, залежності а також ризик порушення авторських прав.

Rust є однією з найшвидших та найбезпечніших мов програмування, яка використовує концепцію крейтів для організації коду. Крейти можуть бути бібліотеками або виконуваними, вони містять модулі, функції, структури, переліки, трейти, макроси та інші елементи коду. Крейти можуть бути публіковані на crates.io, який є офіційним реєстром крейтів Rust.

Однак, публікація крейтів вимагає виконання багатьох кроків, таких як створення облікового запису, налаштування проекту, написання метаданих, перевірка коду, підписання коду, завантаження коду, оновлення версії, синхронізація з GitHub та іншими платформами, вирішення проблем та інше. Ці кроки можуть бути складними, займати багато часу, можливості допустити помилки та можуть залежати один від одного.

Тому, публікація крейтів потребує автоматизації, тобто процесу, який дозволяє розробникам публікувати свої крейти на crates.io швидко, легко, ефективно та безпечно, з мінімальним або відсутнім втручанням людини. Автоматизація публікації крейтів має багато переваг, таких як зменшення

складності, часу, помилок, несумісності, залежності, ризику порушення правил, а також підвищення якості, продуктивності, надійності, безпеки та задоволення розробників та користувачів.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 21 серпня 2023 р. ректора Українського державного університету науки і технологій “Про призначення наукових керівників та затвердження тем магістерських робіт” за спеціальністю 121 “Інженерія програмного забезпечення» факультету “Комп’ютерних технологій і систем” по кафедрі “Комп’ютерні інформаційні технології” та завдання на виробничу практику за договором №832303 на проведення виробничої практики студентів вищих навчальних закладів від 21 серпня 2023 р.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення програмного продукту полягає у автоматизованій публікації крейтів використовуючи спеціальні програмні продукти та рішення, який виконує необхідні перевірки, контролює зміну версій та фіксацію змін, компілює та публікує крейти. Програмний продукт автоматизованої публікації крейтів приймає на вхід список крейтів, які потрібно опублікувати, і виконує усі необхідні кроки для їх публікації, повертаючи результат у вигляді повідомлення про успішну або невдалу публікацію кожного крейта

Експлуатаційне призначення методу полягає у спрощенні та прискорені процесу публікації крейтів, уникнувши ручного виконання необхідних дій. Програмний продукт дозволяє розробникам фокусуватися на створенні якісного коду, а не на технічних деталях публікації. Розроблений програмний продукт може бути застосований до будь-яких крейтів, які відповідають вимогам публікації, а також до воркспейсів, котрі складаються з кількох крейтів.

3 ВИМОГИ ДО ПРОГРАМИ

3.1 Вимоги до функціональних характеристик

Програма має визначати які крейти пов’язані з обраними до публікації, визначати які крейти необхідно публікувати та у якій послідовності й виконувати їх публікацію.

Програма повинна виконувати наступні функції:

- отримувати перелік крейтів, що бажано опублікувати;
- перевіряти залежності та версії;
- виконати публікацію;
- повідомити користувача про виконані дії та успішність виконання.

Вхідні дані: перелік крейтів.

Вхідні дані надаються користувачем шляхом вказання шляху до директорій з Cargo.toml файлами проектів, що необхідно опублікувати як аргументи командного рядку.

Вихідні дані:

- опубліковані відповідні крейти у репозиторії crates.io;
- повідомлення з результатами виконання публікації для кожного крейту.

3.2 Вимоги до надійності

Для забезпечення надійного функціонування необхідно забезпечити наявність архівного коду тексту програми на зовнішньому носії.

3.3 Умови експлуатації

Для забезпечення надійної роботи програмного продукту користувачу необхідно дотримуватись таких умов:

– програмний засіб повинен використовуватись у приміщеннях, призначених для роботи з ЕОМ з відповідними кліматичними умовами;

– працювати з програмним засобом може людина, що має навички роботи з ПК та ознайомлена з посібником користувача.

3.4 Вимоги до складу і параметрів технічних засобів

Програмний продукт розрахований для функціонування на пристроях, що мають такі мінімальні характеристики:

– процесор: Intel Core 2 Duo;
– місце на диску: 128 Mb;
– об’єм оперативної пам’яті: 1 Gb;
– маніпулятори: клавіатура;
– інше устаткування: монітор.

3.5 Вимоги до інформаційної і програмної сумісності

Для роботи програмного продукту, на ЕОМ повинна бути встановлена операційна система, сумісна із стандартом POSIX.

Для успішної компіляції програми, на операціній системі повинні бути наявними утиліти rustup, rustc та cargo.

3.6 Вимоги до маркування і упаковки

Програмний продукт поширюється у вигляді програмного коду із необхідністю подальшої компіляції. Програмний продукт повинен включати інструкції для компіляції програми.

3.7 Вимоги до транспортування і зберігання

Програмний продукт поширюватиметься мережею із хмарного сховища.

4 ЕКОНОМІЧНІ ПОКАЗНИКИ

Розрахунки економічних показників для цієї розробки не ведуться, оскільки вона не є комерціійною.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу програмної документації мають входити:

– керівництво користувача;
– текст програми.

Вся документація до програмного забезпечення повинна задовольняти вимоги державного стандарту до оформлення програмної документації ГОСТ 19.101-77 [1].

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії та етапи розробки програми приведені в табл. 6.1.

Таблиця 6.1 Стадії та етапи розробки

Стадія	Зміст роботи	Термін виконання
Проектування	Проектування структури програмного коду	04.09.2023 — 08.09.2023
Підготовка середовища для розробки	Завантаження та встановлення необхідних бібліотек, підготовка системи компіляції програми	08.09.2023 — 15.09.2023
Розробка	Розробка додатку	15.09.2023 — 22.09.2023
Тестування	Написання юніт тестів та їх запуск	22.09.2023 — 28.09.2023

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙОМУ

Контроль за виконанням роботи здійснює керівник практики від університету Андрющенко В.О.

Прийом здійснюється комісією у складі, визначеному університетом.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38 с.

ДОДАТОК Б

Керівництво користувача

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

Willbe

Керівництво користувача. Керівництво з публікації крейтів

ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1275-01 ІЗ 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олена КУРОП’ЯТНИК
Виконавець
_____ Богдан БАЛУШКІН
Нормоконтролер
_____Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО

44165850.1275-01 ІЗ 01

Willbe

Керівництво користувача
44165850.1275-01 ІЗ 01

Листів 11

2024

АНОТАЦІЯ

Документ 44165850.1275-01 І3 01 «Розробка методу автоматизованої публікації крейтів. «Керівництво користувача» входить до складу програмної документації на програму, що реалізовує додаток для автоматичної публікації крейтів.

У даному документі представлено технічне завдання. Програма написана на мові Rust. Об'єм пам'яті, що займають програми комплексу, складає 10 ГБ. Конфігурація комп'ютера стандартна. Комплекс функціонує в середовищі MS WINDOWS 10 або вище.

ВСТУП

У сучасному світі програмування є однією з найважливіших та найпоширеніших галузей діяльності, яка впливає на розвиток науки, техніки, освіти, культури та інших сфер життя. Програмування дозволяє створювати різноманітні програми та застосунки, які спрощують, прискорюють та покращують роботу людей, а також розширюють їх можливості та перспективи. Однак, програмування також має свої виклики та проблеми, які потребують постійного вдосконалення та інновацій.

Однією з таких проблем є публікація коду, тобто процес, який дозволяє розробникам розповсюджувати свої бібліотеки та програми через інтернет, щоб інші користувачі могли їх використовувати, модифікувати, вдосконалювати та інтегрувати у свої проекти. Це сприяє співпраці, обміну знаннями, повторному використанню, стандартизації та підвищенню якості коду. Однак, цей процес також має свої недоліки, такі як складність, часовитрата, помилки, несумісність, залежності а також ризик порушення авторських прав.

Rust є однією з найшвидших та найбезпечніших мов програмування, яка використовує концепцію крейтів для організації коду. Крейти можуть бути бібліотеками або виконуваними, вони містять модулі, функції, структури, переліки, трейти, макроси та інші елементи коду. Крейти можуть бути публіковані на crates.io, який є офіційним реєстром крейтів Rust.

Однак, публікація крейтів вимагає виконання багатьох кроків, таких як створення облікового запису, налаштування проекту, написання метаданих, перевірка коду, підписання коду, завантаження коду, оновлення версії, синхронізація з GitHub та іншими платформами, вирішення проблем та інше. Ці кроки можуть бути складними, займати багато часу, можливості допустити помилки та можуть залежати один від одного.

Тому, публікація крейтів потребує автоматизації, тобто процесу, який дозволяє розробникам публікувати свої крейти на crates.io швидко, легко, ефективно та безпечно, з мінімальним або відсутнім втручанням людини. Автоматизація публікації крейтів має багато переваг, таких як зменшення

складності, часу, помилок, несумісності, залежності, ризику порушення правил, а також підвищення якості, продуктивності, надійності, безпеки та задоволення розробників та користувачів.

Для того, щоб скористатися цим процесом, користувач повинен мати достатній рівень підготовки, який включає розуміння що таке крейти та де вони повинні бути опублікувані, а також знання системи контроля версій git. Крім того, користувач повинен ознайомитися з експлуатаційною документацією, яка містить "Керівництво користувача. Керівництво з публікації крейтів".

2 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

Функціональне призначення програмного продукту полягає у автоматизованій публікації крейтів використовуючи спеціальні програмні продукти та рішення, який виконує необхідні перевірки, контролює зміну версій та фіксацію змін, компілює та публікує крейти. Програмний продукт автоматизованої публікації крейтів приймає на вхід список крейтів, які потрібно опублікувати, і виконує усі необхідні кроки для їх публікації, повертаючи результат у вигляді повідомлення про успішну або невдалу публікацію кожного крейта

Експлуатаційне призначення методу полягає у спрощенні та прискорені процесу публікації крейтів, уникнувши ручного виконання необхідних дій. Програмний продукт дозволяє розробникам фокусуватися на створенні якісного коду, а не на технічних деталях публікації. Розроблений програмний продукт може бути застосований до будь-яких крейтів, які відповідають вимогам публікації, а також до воркспейсів, котрі складаються з кількох крейтів.

Для забезпечення сталого функціонування програми користувачеві і програмісту необхідно дотримуватися таких умов:

- програма повинна використовуватись у приміщеннях, які відповідають умовам роботи ЕОМ [1];

- кваліфікація робочого персоналу з даною програмою повинна бути на рівні досвідченого користувача ЕОМ, операційною системою Windows. Робітник повинен бути ознайомлений з керівництвом користувача;

- програмний комплекс повинен використовуватись в приміщеннях, з наступними кліматичними умовами: температура 21-25 °С, відносна вологість повітря 40-60%.

ЕОМ повинен мати такі мінімальні характеристики:

- 4 ГБ оперативної пам’яті;

- 2 ТБ пам’яті на жорсткому диску;

- серверний процесор (кількість ядер 2, базова тактова частота 3.7 GHz, максимальний об’єм пам’яті 32 GB);

- клавіатура;

- миша;

- монітор;

- інтерфейс USB.

Для користувачів програмний продукт повинен використовуватись на комп’ютерах з такими вимогами:

- 2 ГБ оперативної пам’яті;

- клавіатуру;

- мишу;

- монітор.

Програмний продукт розроблений для всіх видів операційних систем сімейства Windows 10 та наступні версії. Середовище розробки: JetBrains RustRover.

3 ПІДГОТОВКА ДО РОБОТИ

Перед початком роботи переконайтесь у наявності файлу “will.exe”, котрий був отриманий у результаті виконання компіляції розробленого програмного продукту, на вашому комп’ютері та пропишіть до нього шлях у змінній оточення “PATH”.

4 ОПИС ОПЕРАЦІЙ

Для отримання підказки по доступним командам програми введіть у терміналі “will .help”, результат виконання буде аналогічним до зображеного на рис. 4.1.

```
PS F:\> will .help
Help command


help <properties> - prints information about existing commands

list <Path> <properties> - List workspace packages.

publish <List(String, ',')> <properties> - Publish package on `crates.io`.

readme.health.table.generate - Generate table for main Readme.md file

tests.run <Path> <properties> - Run tests in a specified crate

workflow.generate - Generate workflow for modules
```

Рисунок 4.1 – Відображення допомоги користувачу

Для отримання більш деталізованої допомоги по окремій команді, викличте програму наступним чином: “will .help <Назва команди>”, де замість “<Назва команди>” вказано назву команди, по якій користувачу потрібно отримати деталізовану допомогу, результат виконання наведеного прикладу виконання програми, для команди publish зображено на рис. 4.2.

```
PS F:\> will .help publish
publish <subjects>  <properties> - Publish package on `crates.io`.

Subjects:
        - A path to package. Should be a directory with file `Cargo.toml`. [List(String, ',')]
Properties:
        dry - Run command dry. Default is false. [String]
        verbosity - Setup level of verbosity. [String]
```

Рисунок 4.2 – Деталізована допомога по визначеній команді

Для виконання команди публікації без внесення змін, користувач повинен перейти у директорію з проектом та викликати програму наступним чином: “will .publish”, чи “will .publish <Відносний шлях до крейти>”, де замість “<Відносний шлях до крейти>” вказано відносний шлях до директорії з Cargo.toml файлом крейту, що необхідно опублікувати. Приклад результату виконання зображено на рис. 4.3.

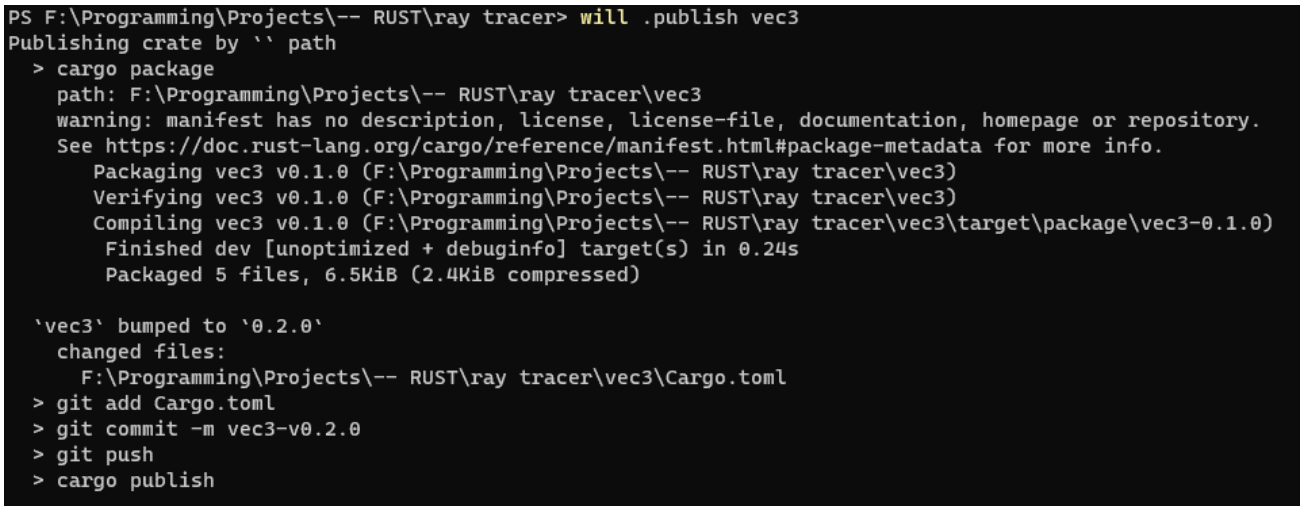

Рисунок 4.3 – Виконання команди публікації

Для виконання публікації з внесенням змін та завантаженням обраного крейту до репозиторію crates.io слід викликати програму наступним чином: “will .publish dry:0” чи “will .publish <Відносний шлях до крейти> dry:0” відповідно.

5 АВАРІЙНІ СИТУАЦІЇ

Якщо програмний засіб буде запускатися з пристрою який не відповідає необхідним технічним характеристикам, буде виведено відповідне повідомлення користувачеві.

Якщо користувач закриє програмний засіб, то при наступному запуску програми, його попередній результат не збережеться.

Якщо програмний засіб під час роботи буде поводити некоректно чи із помилкою, від користувача потребується перезапустити додаток для відновлення коректної роботи програмного засобу.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38 с.

ДОДАТОК В

Текст програми

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

Willbe

Текст програми
44165850.1275-01 12 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олена КУРОП'ЯТНИК
Виконавець
_____ Богдан БАЛУШКІН
Нормоконтролер
_____Світлана ВОЛКОВА

2024

ДОДАТОК Г

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

МІНІСТЕРСТВО ІНФРАСТРУКТУРИ УКРАЇНИ

УКРАЇНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ НАУКИ ТА ТЕХНОЛОГІЙ

СХІДНИЙ НАУКОВИЙ ЦЕНТР ТРАНСПОРТНОЇ АКАДЕМІЇ НАУК

ABSTRACTS
OF THE XVII INTERNATIONAL CONFERENCE
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
13-14, December, 2023

СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ

ТЕЗИ

XVII МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

13-14 ГРУДНЯ 2023

ДНІПРО
2023

Міністерство освіти і науки України

Міністерство інфраструктури України

Український державний університет науки та технологій

Східний науковий центр транспортної академії наук

TEMPUS: CITISET & SEREIN & CRENG

ТЕЗИ
XVII Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ»

ABSTRACTS
of the XVII International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»

13.12.2023 – 14.12.2023

Дніпро
2023

УДК 658.512.2:681.3.06

Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті: Тези XVII Міжнародної науково-практичної конференції (Дніпро, 13-14 грудня 2023 р.). – Д.: УДУНТ, 2023. – 152 с.

У збірнику представлені тези доповідей XVII Міжнародної науково-практичної конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті», яка відбулася 13-14 грудня 2023 року в Українському державному університеті науки та технологій в онлайн форматі. Розглянуто результати теоретичних і експериментальних досліджень, а також проблемні питання функціонування та перспективи розвитку інформаційних технологій транспорту, промисловості й освіти.
Збірник призначений для науково-технічних працівників залізниць, підприємств транспорту, викладачів вищих навчальних закладів, докторантів, аспірантів і студентів.

Адреса редакційної колегії:
49010, м. Дніпро, вул. Лазаряна, 2, УДУНТ

Тези доповідей друкуються мовою оригіналу в редакції авторів.

Автоматизація публікації крейтів

Балушкін Б. В., Куроп'ятник О. С.,
Український державний університет науки і технологій

Мова Rust передбачає модульну структуру коду. Для її організації використовують крейти – модулі або пакети коду, які можуть бути використані для додавання функціональ- ності до проекту Rust або для створення самостійних застосунків. Вони є основним спосо- бом поділу коду в екосистемі Rust і можуть містити бібліотеки або виконувані файли. Для повторного використання крейти можуть публікуватися на crates.io, який є офіційним реєс- тром пакетів для Rust.

Процес публікації крейтів включає в себе перевірку коду на відповідність стандартам якості Rust, забезпечення правильного використання залежностей між крейтами та відповідність версійній політиці. Публікація воркспейсів у Rust може бути особливо складною, оскільки воркспейси часто містять кілька крейтів, які мають бути координовано опубліковані з урахуванням взаємних залежностей та сумісності версій. Це вимагає ретельного планування та керування залежностями для забезпечення стабільності та інтеграції воркспейсу як єдиного цілого.

Для спрощення публікації крейтів в частині від слідкування залежностей та версій пропонується метод її автоматизації. Він передбачає такі етапи:

– отримання списку бажаних до публікації крейтів від користувача та перевірка його авторизації на сайті;
– аналіз проекту і побудова на його основі орієнтованого графу, де вершини відповіда- ють крейтами, а дуги вказують на залежності між ними;
– фільтрація графа, що передбачає видалення вершин і дуг, які відповідають крейтем з позначкою publish та залежностям за їхньою участю, а також дуги, що відповідають допоміжним залежностям, утвореним під час розробки і які не впливають на функ- ціональність;
– топологічне сортування графу. Результатом сортування буде така послідовність вер- шин: в першу чергу йдуть ті, що не мають залежностей зовсім, потім ті що залежать, від перших і так далі. При сортуванні відбувається зміна версій за умови якщо зале- жність або сам вузол були змінені;
– публікація крейтів на crates.io у порядку, отриманому після сортування.

Наведені етапи реалізуються за допомогою програмного забезпечення мовою Rust, розробка якого триває. Вхідними даними для нього є перелік шляхів до Cargo.toml файлів крейтів та дані для авторизації. Вихідними даними є опубліковані крейти на crates.io.

Прогнозується, що автоматизований метод та розроблене на його основі програмне забезпечення надасть ряд переваг, порівняно з ручною публікацією, що значно поліпшить процес розробки завдяки:

– зменшенню часових витрат на відстеження залежностей між модулями та пакетами;
– зменшенню помилок процесу публікації за рахунок автоматизованому відстеження версій;
– забезпеченню консистентності у публікації, що важливо для підтримки якості екосис- теми Rust.

Автоматичне керування залежностями гарантує, що всі необхідні пакети будуть опубліковані в правильному порядку, без зайвих зусиль з боку розробників. Це також сприяє розвитку спільноти Rust, оскільки розробники можуть легше ділитися своїми творіннями та співпрацювати з іншими. В подальшому можливе розширення автоматизації за рахунок інтеграція програмного забезпечення, що реалізує метод, з іншими інструментами розробки, такими як системи CI/CD.