Форма № 17

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Кафедра програмних засобів

(найменування кафедри)

КУРСОВИЙ ПРОЄКТ (РОБОТА)

з дисципліни «Об'єктно-орієнтоване програмування»
(назва дисципліни)

на тему: Розробка програмного забезпечення моделювання структур графів з використанням ООП

Студента 1 курсу КНТ-113сп групи
спеціальності 121 Інженерія програмного забезпечення
освітня програма (спеціалізація) інженерія програмного забезпечення
Щедровського І. А.
(прізвище та ініціали)
Керівник асистент, Короткий О. В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала _____
Кількість балів: _____Оцінка: ECTS _____

Члени комісії _____ Короткий О. В.
(підпис) (прізвище та ініціали)
_____ Каплієнко Т.І.
(підпис) (прізвище та ініціали)

2022 рік

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Інститут, факультет Запорізький національний університет НУ «Запорізька політехніка». Факультет комп`ютерних наук і технологій ;
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____С.О. Субботін
"____" _____2022 року

З А В Д А Н Н Я

НА КУРСОВИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

Щедровського Івана Андрійовича
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка програмного забезпечення моделювання структур графів з використанням ООП
керівник проєкту (роботи) Короткий Олександр Володимирович, асистент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від _____
2. Строк подання студентом проєкту (роботи) 21 грудня 2022 року
3. Вихідні дані до проєкту (роботи) розробити додаток згідно теми курсової роботи
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Огляд та аналіз сучасних методів та засобів проектування програмного забезпечення; 2. Проєктування програмного забезпечення системи; 3. Розробка програмного забезпечення системи; 4.Аналіз ефективності програмного забезпечення; 5. Розробка документів на супроводження програмного забезпечення.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	Короткий О.В., асистент		

7. Дата видачі завдання 06 листопада 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів курсового проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1.	Аналіз індивідуального завдання.	1 тиждень	
2.	Аналіз програмних засобів, що будуть використовуватись в роботі.	2 тиждень	
3.	Аналіз структур даних, що необхідно використати в курсової роботі.	3 тиждень	
4.	Затвердження завдання	4 тиждень	
5.	Вивчення можливостей програмної реалізації структур даних та інтерфейсу користувача.	5-9 тиждень	
6.	Аналіз вимог до апаратних засобів	9 тиждень	
7.	Розробка програмного забезпечення	9-13 тиждень	
8.	Проміжний контроль	10 тиждень	Розділи 1-5 ПЗ
9.	Оформлення, відповідних пунктів пояснювальної записки.	10-14 тиждень	Розділи 1-2 ПЗ
10.	Захист курсової роботи.	15 тиждень	

Студент _____ Щедровський І.А.
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ Короткий О.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до курсової роботи містить 107 сторінок, 31 рисунок, 0 таблиць, 2 додатки, 8 джерел.

Пояснювальна записка складається з шести розділів.

Розділ «Огляд та аналіз сучасних методів та засобів проектування програмного забезпечення» містить в собі аналіз сучасних методів до проектування програмного забезпечення та аналіз різних сфер життя.

Розділ «Проектування програмного забезпечення системи» містить аналіз функцій системи, розробку UML діаграм використання, проєктування графічного інтерфейсу та постановку мети.

Розділ «Розробка програмного забезпечення системи» містить розробку структури програми, розробку UML діаграми класів та опис класів програмного забезпечення.

Розділ «Аналіз ефективності програмного забезпечення » містить аналіз застосунку на швидкодія та масштабованість, аналіз ефективності кожного компоненту системи, а також тестування всього програмного забезпечення.

Розділ «Розробка документів на супроводження програмного забезпечення» містить інструкцію для програміста, де є розгортка застосунку на локальній машині та на сервері, а також інструкцію користувача.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, ГРАФИ, TYPESCRIPT, VITE, GIT, АЛГОРИТМИ, ОПТИМІЗАЦІЯ ВІЗУАЛІЗАЦІЇ ГРАФІВ

ЗМІСТ

Вступ

%%

Тема: Дослідження можливостей інтерпретатору BASH у ОС Linux. Розробка комплексу скриптів для автоматичного моніторингу доступу до файлових ресурсів

%%

-якийсь вводний абзац. Тут треба якусь прикольну воду про те, що зараз скрипти особливо для linux систем актуальні як ніколи, і що вони взагалі корисні що капець і всім потрібні-

-Ця курсова робота спрямована на дослідження та розробку... І далі по темі. І ще коротко чому я це хочу -

-Основною метою мого дослідження є створення програмного продукту, який ... буде гарно працювати, робити логи, створювати файли.. хз-

-Ця робота має значення для ... хз чого взагалі, хоч для чогось. Ще можна написати, що "Я вірю, що мій внесок допоможе вирішити виклики, пов'язані з цією проблемою бла-бла-бла" -

1 ОГЛЯД ТА АНАЛІЗ СУЧАСНИХ МЕТОДІВ ТА ЗАСОБІВ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Огляд сучасного стану питання

Розвиток автоматизації операцій за допомогою скриптів суттєво змінив підхід до управління системами та виконання рутинних завдань. Спочатку скрипти застосовувалися для автоматизації простих завдань, таких як резервне копіювання чи налаштування системи, однак сьогодні їх використовують для складніших процесів, зокрема для моніторингу ресурсів, управління конфігураціями та виконання розподілених обчислень. Ефективне застосування Bash та інших shell-інтерпретаторів дозволяє значно скоротити час на виконання операцій і зменшити ймовірність помилок, пов’язаних із ручними діями.

Тенденції в автоматизації операцій із використанням скриптів сприяють інтеграції інструментів автоматизації в інфраструктури. Для управління конфігурацією та оркестрації часто використовуються сучасні засоби, такі як Ansible, Terraform або Docker. Скрипти стають основою для створення складніших процесів автоматизації, що включають управління контейнерами, CI/CD процеси та інтеграцію з хмарними платформами, що дозволяє забезпечити ефективне і масштабоване виконання задач. Основними перевагами автоматизації через скрипти є зниження ймовірності людських помилок, підвищення швидкості виконання завдань і покращення керованості системами.

Також спостерігається зростаюча популярність мов високого рівня для автоматизації, таких як Python чи Ruby, які доповнюють можливості shell-скриптів у більш складних випадках, зберігаючи при цьому високий рівень продуктивності та гнучкості. Популяризація DevOps-методологій підштовхує до створення інтегрованих рішень для автоматизації, що дозволяють працювати

з різними інструментами в єдиному робочому процесі, зокрема через створення контейнеризованих середовищ для скриптів і сервісів.

BASH (Bourne Again SHell) — це командний інтерпретатор для операційних систем UNIX-подібного типу.

Цей командний інтерпретатор є покращеною версією традиційної оболонки Bourne shell (sh) і надає безліч додаткових можливостей, таких як підтримка командних історій, розширеного редагування команд, а також функціональність для програмування (цикли, умови тощо).

BASH підтримує скрипти, які можуть бути виконані на багатьох UNIX-системах без змін, і забезпечує зручний інтерфейс для взаємодії з операційною системою в режимі командного рядка. Він відповідає стандарту POSIX і включає елементи з інших оболонок, таких як Korn shell (ksh) і C shell (csh) [1.1].

Основними перевагами Bash є:

– Широке поширення та підтримка
– Простота та можливості
– Гнучкість
– Велика кількість інтегрованих інструментів

Недоліками Bash є:

– Проблеми з сумісністю на різних системах
– Продуктивність
– Обмежена підтримка багатопотоковості

BASH є одним з найпоширеніших та стандартних інтерпретаторів команд у більшості Linux-систем та інших Unix-подібних операційних системах, що забезпечує його популярність і доступність. Цей інтерпретатор активно використовується в різних серверних середовищах, на хмарних платформах та в контейнерах, що робить його незамінним інструментом для системних адміністраторів і розробників.

Популярність BASH можна пояснити його простотою та багатофункціональністю. Цей shell пропонує безліч можливостей для автоматизації завдань, обробки текстових даних і управління файлами. Завдяки

своїй гнучкості BASH здатен задовольняти потреби користувачів, починаючи від виконання простих операцій і до розробки складних сценаріїв.

Крім того, BASH інтегрується з численними інструментами та утилітами, які є частиною стандартного набору Linux. Це дозволяє користувачам без зусиль взаємодіяти з іншими програмами, автоматизувати їх роботу і оптимізувати різноманітні процеси. Велика кількість документації та активна спільнота користувачів надають можливість швидко освоїти інтерпретатор і використовувати його потужний функціонал для виконання складних завдань.

BASH має певні проблеми з сумісністю на різних системах, оскільки поведінка команд і сценаріїв може змінюватися залежно від версії інтерпретатора та налаштувань операційної системи. Це може призвести до непередбачуваних результатів, особливо при перенесенні скриптів між різними дистрибутивами або версіями Linux.

Продуктивність BASH також обмежена, особливо коли мова йде про виконання складних або ресурсомістких операцій. Обробка великих обсягів даних або виконання великих скриптів може відбуватися значно повільніше, ніж у таких мовах програмування, як Java або C, що обмежує застосування BASH у великих та складних системах.

Ще одним недоліком є обмежена підтримка багатопотоковості. BASH не забезпечує паралельне виконання вбудованих команд без використання додаткових інструментів або складних обхідних шляхів. Це ускладнює створення високопродуктивних багатозадачних скриптів та обмежує можливості обробки паралельних задач без застосування сторонніх інтерпретаторів чи утиліт.

BASH часто порівнюють з іншими оболонками, такими як Zsh та Fish. Однією з основних переваг BASH є його сумісність з більшістю операційних систем і те, що він є стандартом у більшості дистрибутивів Linux. Натомість Zsh надає більші можливості для кастомізації, зокрема автодоповнення, кольорове

підсвічування та підтримку плагінів, що покращує зручність і продуктивність роботи.

Fish також пропонує зручні функції, такі як автоматичне автозавершення та підсвічування синтаксису без необхідності додаткових налаштувань. Fish орієнтований на зручність для початківців, оскільки його синтаксис є простішим і інтуїтивно зрозумілим. Однак через обмежену сумісність і нестандартний синтаксис, Fish може бути менш гнучким у складніших сценаріях порівняно з Bash або Zsh.

1.2 Формулювання мети досліджень

Метою цього дослідження є вивчення можливостей BASH-інтерпретатора для роботи з файлами та моніторингу доступу до них у середовищі Linux. Зокрема, планується оцінити ефективність застосування BASH-скриптів для автоматизації створення файлів за шаблонами, аналізу їх вмісту та відслідковування операцій читання й запису.

В рамках роботи буде розроблено набір скриптів, що дозволять налаштовувати та запускати різні процеси за допомогою параметрів і опцій. Особлива увага буде зосереджена на розробці рішень для моніторингу змін у файлових системах і збору даних про доступ до файлів.

Основне завдання полягає в розробці оптимальних методів для автоматизованого контролю доступу до файлів, що надасть зручний і ефективний доступ до необхідної інформації для системних адміністраторів і розробників у середовищі Linux.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Постановка мети

Основною метою цієї курсової роботи є розробка комплексу скриптів на основі інтерпретатора BASH, які демонструють можливості цієї мови програмування. Також передбачається створення функціоналу для автоматичного моніторингу доступу до файлових ресурсів у операційній системі Linux.

У рамках роботи планується дослідити можливості BASH, зокрема його здатності ефективно керувати великими об'ємами даних, зручно та швидко отримувати необхідну інформацію для подальшого аналізу.

2.2 Аналіз функцій системи

Для демонстрації базових можливостей BASH, а також задач, у яких цей інтерпретатор є зручним і ефективним, буде розроблено кілька скриптів:

- скрипт для створення файлів на основі аналізу шаблонного файлу;
- скрипт для аналізу файлів, який надаватиме коротку інформацію про їх загальний стан за вказаним шляхом;
- скрипт для автоматичного моніторингу доступу до файлових ресурсів, який записуватиме в протокол усі дії з файлами, які виконуються в обраній директорії.

Перший скрипт буде відповідати за створення групи вихідних файлів на основі одного шаблонного файлу.

Цей скрипт може бути особливо корисним при створенні електронних конспектів у форматі, коли один файл містить лише одну ідею. Це дозволяє розбивати великий обсяг інформації на дрібні нотатки, які можуть бути пов’язані між собою.

Для опису зв’язків між нотатками використовується аналогія з компасом:

– північ, коли нотаток, або ж ідея, твердження, походить з іншого нотатку. Наприклад, деякі особливості синтаксису BASH походять зі зворотної сумісності з UNIX [2.1];

– захід, коли ідея нотатку є схожою на ідею іншого нотатку. Гарним прикладом тут є вивчення різних філософських течій, а саме їх перетинів;

– схід, коли ідея нотатку є протилежною ідеї іншого нотатку. Тут також гарним прикладом може бути філософія, в якій є багато протилежностей в підходах;

– південь, коли нотаток дає початок іншим нотаткам. Наприклад, нотаток про те, що таке функції в програмуванні може стати відправною точною нотатку про рекурсію.

Оскільки таких маленьких нотаток може бути достатньо багато, не завжди зручно відразу їх записувати кожен в окремий файл. Іноді зручно записати їх всі в одному великому файлі, а потім розбити на маленькі. Для реалізації такого підходу і створений цей скрипт.

Цей скрипт буде орієнтований для роботи з файлами у форматі Markdown, проте може підтримувати й інші файлові формати.

Markdown – це полегшена мова розмітки, створена з метою зручного написання та сприйняття тексту [2.2].

Шаблонний файл - це файл, в якому записана інформація про вихідні файли, які будуть створені під час виконання скрипту.

Вихідні файли - це набір файлів, які є результатом роботи скрипту.

Кожен вихідний файл може включати:

– компас зв'язків, який визначає зв’язки цього файлу з іншими;

– теги - для зручного пошуку з-поміж всіх файлів,

– поле джерела інформації, де вказується посилання або назва джерела інформації, щоб можна було до нього повернутись в майбутньому.

Для зручності створення вихідних файлів, будуть використовуватись шаблони вихідних файлів - файли, в яких записана синтаксична структура майбутнього вихідного файлу. В цих шаблонах можна буде вказати положення тегів, тексту, джерела та елементів компаса зв'язків відносно друг друга.

Основна мета використання шаблонів вихідних файлів – щоб мати можливість створювати декілька видів вихідних файлів без необхідності змінювати внутрішній код скрипту.

Шаблонний файл складається з блоків.

Найперший блок - це блок метаданих, які будуть використані для створення вихідних файлів. Тут можна буде вказати загальне джерело та загальний тег всіх нотаток. Також можна вказати шаблон вихідного файлу. Якщо шаблон не задано в цьому блоці, буде використовуватися значення, яке можна вказати в опціях скрипту або у файлі конфігурації

Останній блок, який буде позначений спеціальним символом - це блок коментарів. Він розташовуватиметься в кінці документа, і весь його вміст буде ігноруватися.

Інші блоки містять інформацію про вихідні файли. У них можна вказати назву файлів, їхній вміст, а також додаткові теги та джерело для кожного конкретного файлу. Крім того, тут буде вказано компас зв'язків для кожного файлу.

Цей скрипт матиме свій файл конфігурації, який буде знаходитись в стандартній директорії поряд зі всіма іншими конфігураціями системи. У цьому файлі користувач може вказати значення за замовчуванням, щоб при подальшому запуску скрипту не потрібно було передавати аргументи кожен раз.

У конфігурації можна буде вказати назву шаблону для вихідних файлів за замовчуванням, а також директорію, в якій ці файли зберігатимуться. Для цих обох параметрів будуть реалізовані опції для вказування їх під час запуску скрипту.

Другий скрипт відповідатиме за аналіз файлів у директорії, вказаній користувачем, та надання базової інформації про них, такої як розмір, дата редагування тощо.

Після виконання цей скрипт повинен надати користувачу такі дані:

– загальна кількість файлів;
– загальна кількість директорій;
– загальна вага всіх файлів;
– кількість порожніх файлів, тобто таких, які вважать 0 біт;
– найстаріший файл;
– згруповані файли за mime типом, який автоматично визначається системою;
– згруповані файли за розширенням;
– 5 найважчих файлів;
– 5 нещодавно оновлених файлів.

Найстаріший файл та файли, що були нещодавно оновлені, визначатимуться на основі дати останнього оновлення.

Групування файлів, наприклад, за розширенням, має на увазі знаходження кількості файлів які мають однакове розширення та відображення цих даних у вигляді таблиці.

Цей скрипт буде корисний для швидкого аналізу деякої директорії, щоб побачити цікаву інформацію про неї. Такі скрипти досить популярні для linux та запускаються досить рідко

Третій скрипт буде відповідати за автоматичний моніторинг доступу до файлових ресурсів, відстежуючи будь-які зміни у файлах і директоріях, а також записуючи ці зміни до спеціального протоколу.

Для безпеки користувача скрипт не вказуватиме точні зміни, а лише повідомлятиме, що відбулася зміна в одному з файлів або директорій..

Цей скрипт матиме власний файл конфігурації, який зберігатиметься в стандартній директорії поряд з іншими конфігураціями системи. Користувач

зможе вказати значення за замовчуванням, щоб не передавати опції при кожному запуску скрипту.

За замовчуванням режим моніторингу читання файлів та відкриття директорій не буде активований, оскільки це може призвести до створення великої кількості записів у протоколі, які можуть бути не дуже важливими для користувача.

Щоб ввімкнути режим відстежування читання файлів разом з режимом відстежування змін файлів користувачу скрипту потрібно буде вказати опцію, яка за це відповідає під час запуску скрипту, або ж вказати це в конфігурації.

Також за замовчуванням при кожному запуску скрипту старі дані у протоколі видаляються та запис починається з чистого файлу.

Щоб вимкнути режим перезапису файлів та увімкнути режим доповнення, користувачеві потрібно буде вказати опцію, яка за це відповідає під час запуску скрипту, або ж налаштувати це в конфігурації.

Цей скрипт буде корисний в ситуації, коли в системі щось постійно або іноді змінюється, але не дуже зрозуміло що саме, і тому хочеться це прослідкувати. Також цей скрипт буде корисним для відстежування дій стороннього користувача на вашій машині. Наприклад, якщо поставити такий скрипт на сервері, то можна побачити, що хтось змінював/читав якісь секретні файли

2.3 Проєктування командного інтерфейсу

Проєктування командного інтерфейсу буде базуватись на визначення аргументів та опцій, які буде приймати скрипт.

У shell-скриптах аргументи — це значення, які передаються в скрипт або функцію під час її виконання. Ці значення можуть бути використані як вхідні дані для скрипту або функції для виконання операцій чи прийняття рішень. Shell-скрипти можуть приймати аргументи у вигляді аргументів командного рядка або аргументів функцій.

Аргументи командного рядка — це значення, які передаються в shell-скрипт під час його виконання з командного рядка. Вони зазвичай використовуються для надання вхідних даних або опцій для скрипту. Аргументи командного рядка передаються в скрипти як змінні, причому перший аргумент зберігається у змінній $1, другий аргумент — у $2 і так далі

Опції в контексті shell-скриптів — це спеціальні параметри або прапори, які змінюють поведінку скрипту або команди. Вони зазвичай передаються після команди або аргументів і починаються з одного або двох дефісів (- або --), щоб відрізняти їх від звичайних аргументів.

Опції використовуються для налаштування виконання скрипту або команди, наприклад, для вибору режиму роботи, вказівки файлів для обробки, зміни формату виведення або активації додаткових функцій [2.3].

Скрипт для створення файлів на основі шаблонного буде мати два обов'язкових позиційних аргументи.

Перший позиційний аргумент - це шлях до шаблонного файлу.

Другий позиційний аргумент - це директорія, в якій будуть створюватись вихідні файли після виконання скрипту.

Цей скрипт може мати свій файл конфігурації, який буде називатись config.conf, розташований поряд зі всіма іншими конфігураціями в домашній директорії користувача, в директорії .config, в директорії з назвою скрипту.

В конфігурації можна буде вказати шлях до директорії з шаблонами для вихідних файлів та назву шаблона вихідних файлів за замовчуванням.

Цей скрипт буде приймати три опції.

Перша опція - шлях до директорії, в якій будуть зберігатись шаблони для вихідних файлів.

Друга опція - назва шаблону для вихідного файлу за замовчуванням.

Найбільш пріоритетною назвою шаблона вихідних файлів буде та, яка вказана в шаблонному файлі. Наступною по пріоритету буде назва з опціонального аргументу, а вже потім з файлу конфігурації.

Третій опціональний аргумент - команда про допомогу, яка є у всіх скриптах, "-h" або "--help". Ця команда буде показувати підказку користувачеві як користуватись скриптом.

Шаблонний файл буде поділений на блоки.

Найперший блок - це метадані для створення всіх вихідних файлів. В цьому блоці користувач може вказати які теги проставити для всіх файлів, використовуючи слово "tags", який використовувати шаблон для створення вихідних файлів, використовуючи "file_template", а також проставити звідки була взята ця інформація використовуючи "source".

Блок, який починається з однієї решітки - це блок коментарів. Він повинен буде останнім, оскільки весь його контент буде повністю ігноруватись аж до кінця файлу.

Всі інші блоки будуть містити інформацію про вихідні файли. Кожен файл може мати свої метадані, які будуть перезаписувати метадані шаблонного файлу. Метадані будуть знаходитись перед назвою файлу.

Користувач може вказати для конкретного файлу які самі теги йому встановити, використовуючи "tags" та вказати джерело інформації використовуючи "source". При вказанні tags та source на рівні файлу - це значення буде використовуватись замість значення з метаданих шаблонного файлу.

Також користувач може вказати які інші файли, або нотатки, передували цьому використовуючи "north", які нотатки є схожими використовуючи "west", які є протилежними використовуючи "east", а також які нотатки походять від поточного використовуючи "south". В шаблоні вихідного файлу вони мають таку саму назву, але з приставкою "note_".

Щоб вказати назву файлу потрібно використовувати три символи решітки, це позначення заголовка третього рівня в Markdown форматі, пробіл і після написати назву файлу.

Весь текст який йде після назви файлу до роздільника блоків, або блоку коментарів, буде текстом файлу, який в шаблоні вихідного файлу називається "note_text".

Скрипт, який буде відповідати за аналіз файлів, буде мати один обов'язковий позиційний аргумент - шлях до директорії, з підтримкою regex, до файлів які потрібно проаналізувати

А також цей скрипт буде мати одну опцію, а саме команду про допомогу, "-h" або "--help", яка буде показувати підказку користувачеві як користуватись скриптом.

Після виконання скрипту, користувачеві в терміналі буде відображено наступне:

- загальна кількість файлів;
- загальна кількість директорій;
- загальна вага всіх файлів;
- кількість пустих, тобто таких, які вважать 0 біт, файлів;
- найстаріший файл;
- згруповані файли за mime типом, який автоматично визначає система;
- згруповані файли за розширенням;
- 5 найважчих файлів;
- 5 нещодавно оновлених файлів.

Третій скрипт відповідає за слідкуванням за файлами та записом змін до протоколу. Цей скрипт повинен мати один обов'язковий позиційний аргумент, який буде вказувати шлях, за яким потрібно слідкувати.

Також він може мати файл конфігурації, який називається config.conf, який розташований в домашній директорії користувача, в директорії .config, в директорії з назвою скрипту.

Цей скрипт буде мати декілька опцій, наприклад "-l", або ж "--log", який буде вказувати, в який саме файл записувати протокол взаємодії з файлами та

директоріями. Якщо цей аргумент не вказаний, то дані будуть записуватись в файл, шлях до якого вказаний в файлі конфігурації.

Якщо ж файлу конфігурації не існує, то буде використане значення, яке записане за замовчуванням напряму всередині скрипту.

Другою опцією є прапор про доповнення протоколу, а саме "--no-overwrite". Це означає, що користувачу не потрібно буде передавати його значення.

Якщо опцію було передано при запуску скрипту, або ж вона вказана в файлі конфігурації як ввімкнена, то буде виконано додавання нових даних в кінець протоколу замість його повного перезапису при запуску скрипту.

Третьою опцією, "--read" або "-r", є прапор про запис в протокол дій по читання контенту файлів на ряду з їх редагуванням.

Третьою опцією є команда про допомогу, "-h" або "--help", яка буде показувати підказку користувачеві як користуватись скриптом.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

Оскільки BASH є командною мовою програмування, усі скрипти складаються з набору команд. У цих скриптах можна використовувати всі команди, доступні в командній оболонці, тобто команди операційних систем UNIX, Linux, а також команди SHELL. Крім того, можна використовувати додатково встановлені інструменти та утиліти.

Для всіх опцій в розроблених скриптах було прийнято стандарт, згідно з яким кожна опція має як довгу, так і коротку версію.

Для взяття опцій всередині скрипту було використано команду shift, яка є стандартним і найкращим підходом для цієї мети. Альтернативним варіантом є використання getopts, але цей підхід має деякі недоліки, зокрема складніший синтаксис і меншу гнучкість. Крім того, він не підтримує довгі опції (наприклад, --option) і взагалі є не вбудованою командою в BASH [3.1].

Також для всіх опцій був прийнятий стандарт, коли спочатку пишеться назва опції, а потім через пробіл, її значення, якщо воно потрібно. Альтернативним варіантом є використання символу дорівнює між назвою та значенням замість пробілу, але цей підхід менш поширений.

Позиційні ж аргументи беруться за їх положенням в рядку. Перед тим, як знаходити позиційні аргументи, спочатку з рядку відфільтровуються всі опції. Це потрібно, оскільки у користувача повинна бути можливість писати опції та аргументи чергуючи їх, й це не повинно викликати помилок. Після фільтрації аргументи беруться по їх положенню в рядку. Варто зауважити, що роздільниками в даному випадку виступають пробіли.

Запис та перезапис інформації в файлі реалізований через перенаправлення виводу, тобто символи ">" та ">>". Команда ">" робить перезапис інформації в файлі. А команда ">>" доповнює інформацію без її перезапису [3.2].

Використання перенаправлення виводу показано на рисунку 3.1.

```
echo "Hello, World!" > output.txt
cat output.txt
# Hello, World!

echo "Another line" >> output.txt
cat output.txt
# Hello, World!
# Another line

echo "Overwritten content" > output.txt
cat output.txt
# Overwritten content
```

Рисунок 3.1 – Приклад використання перенаправлення виводу

Для читання інформації з файлу було використано стандартний для unix механізм перенаправлення вводу, символ "<". Перенаправлення вводу означає, що контент файлу, назва якого вказана після "<", буде перенаправлений в команду, яка стоїть зліва від "<" [3.3].

За допомогою цього функціоналу можна зробити, наприклад, зручне читання інформації з файлу, яке показано на рисунку 3.2.

```
echo "Hello, World!" > output.txt
cat output.txt
# Hello, World!

echo < output.txt
# Hello, World!
```

Рисунок 3.2 – Приклад читання інформації з файлу порядково через перенаправлення вводу

Другим варіантом прочитати файл є використання pipeline, але цей спосіб є менш поширеним в BASH скриптах.

Конвеєр, або ж pipeline - це послідовність однієї чи кількох команд, розділених одним із операторів керування «|» або «|&».

Вихід кожної команди в pipeline підключений через оператор керування до входу наступної команди. Тобто кожна команда читає вихід попередньої команди [3.4].

Перший скрипт, який виконує створення файлів на основі шаблонного, складається з трьох основних частей:

– читання аргументів;
– взяття інформації з шаблонного файлу;
– створення та запис інформації в вихідні файли.

Варто зауважити, що частина друга, взяття інформації шаблонного файлу, а також частина третя, створення та запис інформації в вихідні файли, будуть виконуватись паралельно, а не послідовно. Це означає, що як тільки скрипт має всю необхідну та достатню інформацію для створення файлу – цей файл буде створений.

Взяття інформації з шаблонного файлу становить собою великий while-цикл який проходиться по кожному рядку файлу та перевіряє її спеціальним чином.

Щоб не витрачати час на блок коментарів, весь контент після рядку, який містить тільки символ решітки пропускається.

Існує спеціальний лічильник номера блоку.

Якщо це перший, або ж нульовий блок, то інформація з нього записується до глобальних змінних, які більше не будуть змінюватись.

Після цих двох перевірок йде взяття інформації про окремий файл. Це реалізовано через умовні оператори з REGEX, який дозволяє перевірити, чи є якийсь рядок під-рядком іншого рядку.

На рисунку 3.3 зображений приклад коду, який виконує взяття даних параметра tags та записує її до змінної.

```
while read line; do
  # ...

  if [[ $line =~ ^tags:\ (.+) ]]; then
      note_tags="${BASH_REMATCH[1]}"
  fi

  # ...
done < $INPUT_FILE
```

Рисунок 3.3 – Взяття інформації про параметр tags з рядка файлу

Після того, як закінчився один блок, або ж код дійшов до блоку коментарів - викликається функція збереження файлу.

Ця функція створює директорію в якій будуть зберігатись вихідні файли, якщо її ще немає.

Читає шаблон вихідного файлу, замінює всі відомі параметри через awk та створює файл.

Awk - це утиліта, яка дозволяє програмісту писати крихітні, але ефективні програми у вигляді операторів, що визначають текстові шаблони, які слід шукати в кожному рядку документа, і дії, які слід виконати, коли збіг знайдено в рядку. Awk здебільшого використовується для сканування та обробки шаблонів. Вона шукає в одному або декількох файлах рядки, які відповідають заданим шаблонам, а потім виконує відповідні дії.

Awk - це скорочення від імен розробників - Aho, Weinberger та Kernighan [3.5].

Код функції, яка замінює деякий шаблон на потрібне значення, а також використання цієї функції, показано на рисунку 3.4.

```
change_string_in_text () {
    echo "$1" | awk -v note_text="$2" -v pattern="$3" '{gsub("\\{\\{"
pattern "\\}\\}", note_text); print}'
}
```

Рисунок 3.4 – Функція заміни деякого шаблону на значення

Другий скрипт, для аналізу інформації в директорії, є досить простим для розуміння та швидким, завдяки можливостям bash. Цей скрипт повністю побудований на команді find та використані її з іншими командами через pipeline, такими як sort, uniq, head, awk, du, wc тощо.

Цей скрипт гарно показує сильні сторони bash, а саме велику кількість команд, які можуть зробити багато щось дуже швидко не витрачаючи ресурси на написання великого скрипту, а також можливість об'єднувати ці команди в послідовності.

Наприклад, щоб знайти загальну вагу всіх файлів в по деякому шляху можна або написати великий скрипт, який буде проходитись по всім файлам, для кожного брати вагу та додавати її до загальної, або використати команду "du", яка розшифровується як "disk usage", використання диску, та передати туди флаги "-s", який буде сумувати вагу кожного окремого файлу в одну змінну, а також прапор "-h", який замінить кількість байтів на зрозумілі людині кілобайти, мегабайти тощо [3.6]. Приклад використання команди du показаний на рисунку 3.5.

```
du -h /home/ltlaitoff/test

# Output:
44K    /home/ltlaitoff/test/data
2.0M   /home/ltlaitoff/test/system design
24K    /home/ltlaitoff/test/table/sample_table/tree
28K    /home/ltlaitoff/test/table/sample_table
32K    /home/ltlaitoff/test/table
98M    /home/ltlaitoff/test
```

Рисунок 3.5 – Приклад використання команди

Команда find в Linux дозволяє знайти всі записи, наприклад тільки директорії, або ж тільки файли, за деяким критерієм, наприклад розміром, по деякому шляху. Пошук відбувається по всім вкладеним директоріям

автоматично. За допомогою цієї команди, наприклад, можна знайти всі пусті файли та вивести інформацію про них.

Команда sort в Linux сортує, поєднує та порівнює всі рядки з наданих фалів, або ж з pipeline. Цю команду дуже зручно використовувати, наприклад, щоб відсортувати всі файли по даті збереження, а потім командою head взяти тільки перші п’ять [3.7].

Команда head дає змогу з файлу, або pipeline, взяти тільки першу частину. Наприклад, можна взяти тільки перші 20 строк деякого файлу.

Приклад використання команд find, sort, head та awk для взяття п’яти файлів, які були оновленні останніми показаний на рисунку 3.6

```
find "$INPUT_PATH" -type f -exec stat --format='%Y %n' {} + | sort -n -r |
head -n 5 | awk '{print strftime("%b %d %H:%M", $1), substr($0, index($0,
$2))}'
```

Рисунок 3.6 – Приклад знаходження п’яти файлів, які були оновленні останніми

Команда uniq видаляє всі рядки з вводу, які повторюються. Для цієї команди важливо, щоб повторюванні рядки йшли один за одним, оскільки інакше вона не буде працювати [3.8].

Команда wc використовується для підрахунку кількості символів, слів або ж рядків в вводі. Найбільш поширене використання – для підрахунку саме кількості рядків [3.9].

Саме pipeline в bash надають йому таку силу та популярність, оскільки, наприклад щоб порахувати скільки файлів по якомусь шляху потрібно написати досить просту команду find, яка бере по шляху всі записи з типом файл, а потім через pipeline через команду wc -l знайти кількість файлів. Приклад взяття загальної кількості файлів по деякому шляху показана на рисунку 3.7.

```
find "$INPUT_PATH" -type f | wc -l
```

Рисунок 3.7 – Приклад знаходження кількості файлів по деякому шляху

Третій скрипт, який відстежує зміни в файлах та директоріях, реалізований за допомогою inotifywait.

inotify — це механізм в Linux, який дозволяє процесам отримувати сповіщення про зміни в файловій системі, такі як додавання, видалення, зміна файлів та директорій. Він надає ефективний спосіб моніторингу подій у файловій системі без необхідності постійно опитувати стан файлів (polling). Механізм підтримує різноманітні події, які можуть бути відслідковані для вказаних файлів або директорій, включаючи створення, видалення, зміни вмісту файлів, а також зміни атрибутів [3.10].

Події в Linux — це конкретні зміни або дії в системі, на які можуть реагувати програми чи процеси.

У контексті inotify події можуть бути пов'язані з файлами або директоріями в файловій системі, і кожна подія має унікальний код, що визначає тип зміни. Наприклад, подія IN_CREATE вказує на створення нового файлу або директорії, а подія IN_MODIFY означає зміну вмісту файлу.

Події можуть бути поєднані для моніторингу комплексних процесів, наприклад, за допомогою комбінування подій IN_MODIFY та IN_MOVE для відслідковування змін і переміщення файлів у певній директорії [3.11].

inotifywait — це утиліта командного рядка, що дозволяє інтерфейсу користувача підключатися до механізму inotify для відслідковування подій у реальному часі. Вона дозволяє вказати директорію або файл для моніторингу та виводить повідомлення, коли відбувається певна подія, така як зміна вмісту файлів, створення чи видалення файлів.

inotifywait є потужним інструментом для автоматизації різних процесів, які потребують реагування на зміни в файловій системі. Наприклад, його можна використовувати для автоматичної обробки файлів після їх створення або

зміни. Важливою особливістю є те, що inotifywait блокує виконання процесу до того часу, поки не буде зафіксовано певну подію, що робить його зручним для інтеграції в скрипти та автоматизацію [3.12].

4 АНАЛІЗ ЕФЕКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз ефективності програмного забезпечення

Перший скрипт, який відповідає за створення вихідних файлів по шаблонному, буде працювати дуже швидко, оскільки він не задіє багато пам’яті та ресурсів машини.

Оскільки читання шаблонного файлу відбувається порядково шаблонний файл може мати будь-який розмір.

А також через те, що вихідні файли створюються відразу після того, як відома вся інформація для їх створення, тобто ми взяли всю інформацію з одного блоку, в пам’яті скрипту не зберігається інформація про кожен вихідний файл. Після того, як вихідний файл був створений інформація про нього в пам’яті скрипту очищується, а потім туди записуються данні про новий блок. Таким чином, ми можемо створювати велику кількість файлів при цьому не використовуючи багато пам’яті.

Єдина проблема з цим скриптом може бути, коли контент файлу буде занадто великий, оскільки весь контент зберігається в пам’яті та редагується також в ній. Тут можна зробити оптимізацію та також парсити контент по строково, що може сильно прискорити скрипт.

Другий скрипт, для аналізу файлів, виконує багато пошуків файлів, оскільки це єдиний варіант як його можна написати. Хоч пошук файлів в Linux є досить швидким, але при запускі скрипту на велику кількість, наприклад, 100 тисяч або більше файлів він може тормозити. Це можна спробувати оптимізувати, але це буде складно.

Третій скрипт, який слідкує за файлами, працює швидко, оскільки використовуємо події в Linux замість того, щоб перевіряти всі файли кожну секунду, що називається polling. Навіть при запуску цього скрипту на мільйони файлів він буде працювати швидко.

4.2 Тестування програмного забезпечення

Якщо користувач не вказує деякі обов’язкові аргументи, або ж вказує не правильні значення, то скрипти повинні видавати помилки.

Наприклад, якщо в перший скрипт не передати один з аргументів, він повинен показати помилку, яка показана на рисунку 4.1.

=image=

Рисунок 4.1 – xxx

Якщо ж в перший скрипт передати не правильний шлях до шаблонного файлу або ж директорії вихідних файлів також повинна бути помилка, показана на рисунку 4.2.

=image=

Рисунок 4.2 – xxx

При виклику першого скрипту з не правильним значенням опцій також повинна бути помилка, показана на рисунку 4.3.

=image=

Рисунок 4.3 – xxx

При передачі не правильної опції повинна бути помилка, показана на рисунку 4.4.

=image=

Рисунок 4.4 – xxx

Це все базові перевірки, але їх достатньо, щоб впевнитись, що все працює правильно.

Другий скрипт має тільки один обов’язковий аргумент. Якщо його не передати повинна бути помилка, яка показана на рисунку 4.5.

=image=

Рисунок 4.5 – xxx

Або ж якщо передати в нього не правильне значення, то повинна бути помилка показана на рисунку 4.6.

=image=

Рисунок 4.6 – xxx

При передачі не правильної опції повинна бути помилка, показана на рисунку 4.7.

=image=

Рисунок 4.7 – xxx

В третьому скрипті при не передачі обов’язкового аргументу повинна бути помилка, показана на рисунку 4.8.

=image=

Рисунок 4.8 – xxx

При передачі не правильного значення цього аргументу повинна бути помилка, показана на рисунку 4.9.

=image=

Рисунок 4.9 – ххх

При передачі не правильного значення в опцію --log повинна бути помилка показана на рисунку 4.30.

=image=

Рисунок 4.30 – ххх

5 РОЗРОБКА ДОКУМЕНТІВ НА СУПРОВОДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Інструкція програміста

Перед початком розробки скриптів необхідно виконати ініціалізацію - встановити та налаштувати. Процес ініціалізації аналогічний тому, що використовується для звичайних користувачів.

Для редагування BASH-скриптів можна використовувати будь-який текстовий редактор, здатний відкривати файли з розширенням sh. Наприклад, neovim або nano, як найбільш поширені.

Bash-скрипт - це звичайний текстовий файл, де розширення sh використовується радше як загальноприйнята конвенція, а не обов’язкова вимога. Файл може взагалі не мати розширення, і його запуск залишатиметься повністю коректним.

Для запуску Bash-скриптів необхідно мати встановлений дистрибутив Linux, наприклад, Arch Linux. Також можна використовувати Bash у Windows, однак цей підхід є складнішим і може супроводжуватися численними проблемами, тому це швидше є винятком, та краще встановити Linux.

Розроблені скрипти не мають окремого вбудованого режиму відлагодження. Однак, для детального відстеження виконання коду можна скористатися командами set -x та set +x, які дозволяють побачити покрокове виконання команд у скрипті.

5.2 Інструкція користувачеві

Перед запуском скриптів необхідно виконати їх встановлення та налаштування.

Спочатку скрипти потрібно розмістити у вибраній директорії для зберігання, яка може бути як домашньою директорією користувача, так і будь-якою іншою.

Потім слід зробити ці скрипти виконуваними, щоб вони могли запускатися в системі. Для цього використовується команда chmod, як показано на рисунку 5.1.

=image=

Рисунок 5.1 – Приклад використання chmod команди

Після встановлення скриптів необхідно створити файли конфігурацій, які розміщуються в домашній директорії користувача, у папці .config, у піддиректорії з назвою скрипту. Конфігураційний файл має назву config.conf.

Наприклад, для першого скрипту файл конфігурації розташовуватиметься за шляхом "~/.config/file-parser/config.conf".

У кожному файлі конфігурації потрібно зазначити стандартні значення відповідно до документації. Після цього скрипти готові до запуску.

Для запуску скрипту в Linux необхідно вказати шлях до нього, а потім передати відповідні аргументи та опції

Розглянемо запуск першого скрипту для створення файлів на основі шаблонного, який називається "file-parser".

Цей скрипт має два обов’язкових аргументи, а саме шлях до шаблонного файлу та шлях до директорії, в якій будуть створюватись вихідні файли. Опції ж є не обов’язковими, якщо їх не передавати то будуть використовуватись значення з файлу конфігурації.

Розглянемо приклад запуску цього скрипту без вказання опцій.

На рисунках 5.2 та 5.3 показаний файл конфігурації та шаблон вихідного файлу який буде використовуватись.

=image=

Рисунок 5.2 – Налаштований файл конфігурації першого скрипту

=image=

Рисунок 5.3 – Шаблон вихідного файлу за замовчуванням

Тепер, після запуску скрипту передаючи в нього шаблонний файл, та директорію в якій будуть створюватись вихідні файли, ці файли будуть використовувати шаблон з конфігурації, який показаний на рисунку 5.3

Приклад запуску скрипту показаний на рисунку 5.4.

=image=

Рисунок 5.4 – Приклад запуску першого скрипту

Тепер ми можемо перевірити, чи все правильно працює. Для цього порівняємо данні шаблонного файлу, рисунок 5.5, список вихідних файлів, рисунок 5.6 та контент першого вихідного файлу, рисунок 5.7.

=image=

Рисунок 5.5 – Данні шаблоного файлу

=image=

Рисунок 5.6 – Список вихідних файлів

=image=

Рисунок 5.7 – Контент першого вихідного файлу

Також цей скрипт можна запустити з опціями, наприклад, вкажемо деякий шлях до шаблону вихідного файлу при запуску, рисунок 5.8.

=image=

Рисунок 5.8 – Запуск першого скрипту з вказанням шаблону вихідного файлу

На рисунку 5.9 показаний шаблон вихідного файлу, який був використаний. А на рисунку 5.10 показаний вихідний файл.

=image=

Рисунок 5.9 – Шаблон вихідного файлу

=image=

Рисунок 5.10 – Вихідний файл

В цьому скрипті також реалізована команда допомоги, яка показана на рисунку 5.11.

=image=

Рисунок 5.11 – Команда про допомогу в першому скрипті

Назва другого скрипту, який слідкує за файлами – "file-stat". Для запуску цього скрипту потрібно вказати тільки один обов’язковий аргумент – шлях, в якому потрібно зробити аналіз файлів.

Приклад запуску цього скрипту на директорію показаний на рисунку 5.12.

=image=

Рисунок 5.12 – Запуск другого скрипту для аналізу контента директорії

Приклад запуску цього скрипту з regex в якості шляху показаний на рисунку 5.13.

=image=

Рисунок 5.13 – Запуск другого скрипту для аналізу файлів по regex

Також цей скрипт має команду про допомогу. Її запуск показаний на рисунку 5.14.

=image=

Рисунок 5.14 – Команда про допомогу другого скрипту

Третій скрипт, який виконує слідкування за файлами, називається "file-watcher".

В цьому скрипті є один обов’язковий аргумент – шлях, за файлами якого потрібно слідкувати.

Для того, щоб показати можливості цього скрипту його потрібно запустити, наприклад, вказавши шлях до фалу в якому буде зберігатись протокол, відредагувати декілька файлів та переглянути протокол. Це показано на рисунку 5.15.

=image=

Рисунок 5.15 – Приклад запуску скрипта для стеження за файлами

При повторному запуску скрипту файл протоколу буде перезаписаний, щоб це не відбувалось можна вказати опцію "--no-overwrite", яка буде дозаписувати інформацію в протокол. Це показано на рисунку 5.16.

=image=

Рисунок 5.16 – Приклад дозапису данних в файл протоклу

Щоб відслідковувати читання файлів можна вказати опцію "--read". Це показано на рисунку 5.17.

=image=

Рисунок 5.17 – Приклад запису читання файлів

В цьому скрипті також реалізована команда допомоги, яка показана на рисунку 5.18.

=image=

Рисунок 5.18 – Команда про допомогу в третьому скрипті

ВИСНОВКИ

%%

Тема: Дослідження можливостей інтерпретатору BASH у ОС Linux. Розробка комплексу скриптів для автоматичного моніторингу доступу до файлових ресурсів

%%

%%

Тут треба описати, що всі цілі були виконанні, все добре, все good

А взагалі можна пройтись по пунктам і описати, що я виконав аналіз сучасних методів, мету, проектування, розробку, аналіз з тестуванням та написання документації окремо для розробників та користувачів

%%

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

0 -

i1 - https://chemerys.site/blog/shcho-take-markdown-rozmitka/

[1.1] - https://www.gnu.org/software/bash/

[2.2] - https://chemerys.site/blog/shcho-take-markdown-rozmitka/

[2.3] - https://olanipekunayo2012.medium.com/shell-script-functions-and-arguments-52957f960844

[3.1] - http://mywiki.wooledge.org/BashFAQ/035

[3.2] - output redirection

[3.3] - input redirection

[3.4] - https://www.gnu.org/software/bash/manual/html_node/Pipelines.html

[3.5] - https://devzone.org.ua/post/komanda-awk-v-unixlinux-z-prykladamy

[3.6] – du

[3.7] - https://ss64.com/bash/sort.html

[3.8] - https://ss64.com/bash/uniq.html

[3.9] - https://ss64.com/bash/wc.html

[3.10] – inotify

[3.11] – events in linux

[3.12] - inotifywait

1. Hatch, S.V. Computerized Engine Controls / S.V. Hatch. – Boston: Cengage Learning, 2016. – 688 p.

2. Czichos, H. Measurement, Testing and Sensor Technology. Fundamentals and Application to Materials and Technical Systems / H. Czichos. – Berlin: Springer, 2018. – 213 p.

3. Kaźmierczak, J. Data Processing and Reasoning in Technical Diagnostics / J. Kaźmierczak, W. Cholewa. – Warszawa: Wydawnictwa Naukowo-Techniczne, 1995. – 186 p.

4. Diagnostics as a Reasoning Process: From Logic Structure to Software Design / [M. Cristani, F. Olivieri, C. Tomazzoli, L. Vigano, M. Zorzi] // Journal of Computing and Information Technology. – 2018. – Vol. 27 (1). – P. 43-57.

5. Wieczorek, A.N. Analysis of the Possibility of Integrating a Mining Right-Angle Planetary Gearbox with Technical Diagnostics Systems / A.N. Wieczorek // Scientific Journal of Silesian University of Technology. Series Transport. – 2016. – Vol. 93. – P. 149-163.

6. Tso, B. Classification Methods for Remotely Sensed Data / B. Tso, P.M. Mather. – Boca Raton : CRC Press, 2016. – 352 p.

7. Oppermann, A. Regularization in Deep Learning – L1, L2, and Dropout [Electronic resource]. – Access mode: https://www.deeplearning-academy.com/p/ai-wiki-regularization.

8. Classic Regularization Techniques in Neural Networks [Electronic resource]. – Access mode: https://medium.com/@ODSC/classic-regularization-techniques-in-neural-networks-68bccee03764.

ДОДАТОК А

=todo=

ДОДАТОК Б

Слайди презентації:

Слайд 1

Слайд 2

Слайд 3

Слайд 4

Слайд 5

Слайд 6

Слайд 7

Слайд 8

Слайд 9