КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Автоматизована система виявленя елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики»

Дородний Павло Геннадійович
(прізвище, ім’я та по батькові студента повністю)

Київ 2024 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
к.т.н., доцент Гончаренко Т.А.

„___” _____2024 року

ПОЯСНЮВАЛЬНА ЗАПИСКА ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Автоматизована система виявленя елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики»

Виконав: студент 4-го курсу, групи КН-20-2

Спеціальності: 122 «Комп`ютерні науки»

Освітня програма: «Інформаційні управляючі системи та технології»
(шифр і назва напряму підготовки, спеціальності)

Дородний П. Г.
(прізвище та ініціали)

Керівник к.т.н., доцент Гончаренко Т. А.
к.т.н., доцент Горда О.В.
(прізвище та ініціали)

Рецензент _____
(прізвище та ініціали)

Київ, 2024 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій
Кафедра: інформаційних технологій
Освітній рівень: «бакалавр» за ОП
Спеціальність: 122 «Комп`ютерні науки»
Освітня програма: Інформаційні управляючі системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
к.т.н., доцент Гончаренко Т.А.
_____
„___” _____2024 року

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

Дородний П. Г.
Тема роботи: Автоматизована система виявленя елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики
затверджена наказом ректора КНУБА № 433/2 від « 29 » лютого 2024 р.
2. Керівник роботи: Гончаренко Тетяна Андріївна, к.т.н, доцент
кафедри інформаційних технологій проектування та прикладної математики
Горда Олена Володимирівна. к.т.н. доцент кафедри інформаційних технологій проектування та прикладної математики
3. Строк подання студентом роботи до захисту:
4. Зміст пояснювальної записки за розділами:
Р.1. Аналіз предметної області та постановка задачі
Р.2. Алгоритмічне та математичне забезпечення
Р.3. Розробка програмного забезпечення
Р.4. Ергономіка інформаційних технологій
5. Інформаційні слайди:
С.1. Приклад ботів
С.2. Область видимості
С.3. Лексична область видимості
С.4. Кінцевий автомат станів
С.5. Підготовлені дані
С.6. Діаграма класів
С.7. Результат тестового прикладу

6. Календарний план виконання атестаційної випускної роботи

Види робіт та їх зміст	Дата виконання

Р. 1. Аналіз предметної області та постановка задачі	Січень 2024 р.
Р. 2. Алгоритмічне та математичне забезпечення	Лютий 2024 р.
Р. 3. Розробка програмного забезпечення	Березень 2024 р.
Тестовий приклад програми	Квітень 2024 р.
Р. 4. Ергономіка інформаційних технологій	Травень 2024 р.
Остаточне оформлення роботи	Травень 2024 р.
Направлення роботи на рецензування	Червень 2024 р.
Попередній захист роботи на кафедрі	Червень 2024 р.

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта, представника комісії	дата	підпис
Розділ 1	Гончаренко Т.А.		
Розділ 2	Гончаренко Т.А.		
Розділ 3	Гончаренко Т.А.		
Розділ 4	Гончаренко Т.А.		

8. Дата видачі завдання: 16.01.2024

Зав.кафедри _____ Гончаренко Т. А.
(підпис) (прізвище та ініціали)

Керівник _____ Гончаренко Т. А.
(підпис) (прізвище та ініціали)

Бакалавр _____ Дородний П. Г.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дородний П. Г. Автоматизована система виявленя елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики.

Атестаційна випускна робота бакалавра за спеціальністю: 122 «Комп’ютерні науки», спеціалізація: «Інформаційні управляючі системи і технології». – Київський національний університет будівництва та архітектури. – Київ, 2024.

Робота присвячена можливості автоматизації та прискорення розпізнання інформаційно-психологічних операцій у соціальних мережах, шляхом створення системи яка використовує розширений лексичний аналіз. Інструментами реалізації розроблюваної системи виступають Node.js , React.js та javascript.

Ключові слова: інформаційна війна, лексичний аналіз, інформаційно-психологічна операція, javascript , Node.js , React.js

SUMMARY

Dorodnyi P. G. Automated system for detecting elements of informational and psychological operation using advanced vocabulary analysis. Attestation graduation thesis of a bachelor in the specialty: 122 "Computer science", specialization: "Information management systems and technologies". - Kyiv National University of Construction and Architecture. - Kyiv, 2024. The work is devoted to the possibility of automating and accelerating the recognition of informational and psychological operations in social networks by creating a system that uses advanced lexical analysis. Node.js, React.js, and javascript are the implementation tools of the developed system. Keywords: information war, lexical analysis, information-psychological operation, javascript, Node.js, React.js

ЗМІСТ

Вступ

В сучасному світі комунікації, зокрема офіційні, виконуються за допомогою соціальних мереж. Важливість соціальних мереж тяжко виразити в повній мірі. Більшість людей отримує новини і інформацію саме звідти, а не з офіційних новин. Через це виникає можливість використовувати їх задля зміни народного настрою з приводу, або публічної бачення ситуації. І найсильніший інструмент у цій справі — боти. З розвитком штучного інтелекту комунікації поступово автоматизуються: так звані боти - це поширене явище. Виникає потреба протидійти цьому впливу, і тим більше в сьогоднішних реаліях. Українське суспільство опинилось в умовах війни - в тому числі і інформаційної. Супротивник використовує соціальні мережі для ведення інформаційно-психологічних спецоперацій (ІПСО).

Особливо важка ситуація склалась в англомовному просторі, бо українські медіа там представлені слабо. А українські користувачі традиційно проводять час в україномовному або російськомовному просторі. Тому ІПСО супротивника в англомовному просторі проходять майже непоміченими українським суспільством. І це привід до опасіння, бо іноземна інформаційна сфера це дуже важлива частина війни. Якщо світ змінить своє ставлення до російськлї агресії і зменшить або зовсім зупинить допомогу це дуже сильно послабить можливості нашої армії.

Розроблювана в даній роботі система покликана автоматизувати частину роботи, тим самим допомогти людям, що працюють на інформаційному фронті, та залишити їм більше часу на творчу роботу. Також система ще збільшить швидкість і точність знаходження і надасть можливість аналізу нових елементів ІПСО. Для реалізації головної мети буде використовуватись алгоритми лексичного аналізу.

Розробка програми відбуватиметься за допомогою мови Javascript та її розширення Node.js. Також під час розробки будуть використані бібліотеки Gramjs та TypeORM, та система баз даних PostgreSQL.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної області

Предметною областю даної дипломної роботи є виявлення нових інформаційно психологічних операцій. Автоматизоване виявлення може полегшити дефіцит людських спеціалістів або допомагати наявним спеціалістам працювати ефективніше, значно скорочуючи час реакцій з хвилин на секунди. Також шляхом полегшення повсякденних завдань, може звільнити додатковий час на творчу роботу, що автоматизувати на цей час можливості нема. На даний момент найголовніші проблеми цієї області є нестача спеціалістів працюючих в англомовній середі та недостатня швидкість реакції. Ці проблеми покликана вирішити автоматизована система виявлення елементів інформаційно-психологічних операцій.

Для подальшого дослідження цієї теми потрібно ввести ключові поняття, а саме: інформаційно-психологічні операції, соціальні мережі та їх дослідження, а також методів автоматизованих розпізнавання та класифікації тексту, таких як лексичний аналіз.

Інформаційно-психологічна операція (ІПСО) — сплановані дії з передачі конкретної інформації та індикаторів до іноземних аудиторій, щоб вплинути на почуття, мотиви, критичне мислення і, зрештою, на діяльність цільових груп[1]. Створені для конфлікту чи мирного часу, ІПСО поєднують різні аспекти соціальних досліджень, психологічних теорій, інформаційної комунікації та дипломатії. Зазвичай їхня основна мета - це вплив на пізнання та поведінку аудиторії, щоб зробити її сприятливою для цілей та інтересів відправника ІПСО.

Незважаючи на те, що термін «психологічна операція» є відносно новим, це поняття має початок глибоко в історії. Від завоювання Олександра Великого до розповідей про Чингісхана, і навіть у творах Сунь Цзи психологічні тактики можна простежити аж до стародавніх часів. Психологічна війна і тактика з тих пір змінилася разом із розвитком комунікаційних технологій і прогресом в розумінні людської психіки. Поширення використання ІПСО вважається

ключовим елементом війни, очевидно через його внесок у перемогу союзників у Першій світовій війні[2]. Нині дипломатія стала найпопулярнішим методом боротьби зі злочинністю. Психологічні операції надають можливість боротися зі злочинністю та вирішувати конфлікти, не ставлячи додаткових людей під удар. Однак, щоб ІПСО була життєздатним варіантом, вона спочатку має бути зрозумілою і виправданою у своїй ефективності.

Соціальна мережа, — соціальна структура, утворена індивідами або організаціями. Вона відбиває розмаїті зв'язки між ними через різноманітні соціальні відносини, починаючи з випадкових знайомств і закінчуючи тісними родинними зв'язками. Розглянемо поширеність соціальних мереж:

- Соціальні мережі є найпопулярнішою діяльністю в Інтернеті,
- 91% дорослих онлайн регулярно користуються соціальними мережами,
- Facebook, YouTube і Twitter займають перше, третє та десяте місце серед найбільш відвідуваних сайтів в інтернеті.

Але навіть ці статистичні дані не дають повної інформації про вплив, який мають соціальні медіа. Користувачі проводять більше 20% свого часу онлайн на сайтах соціальних мереж. Лише у Facebook рівень проникнення на світовий ринок перевищує 12, а у Північній Америці сягає 50%. Ці показники швидко зростають, лише Facebook отримав 170 мільйонів нових користувачів між першим кварталом 2011 року та першим кварталом 2012 року, що на 25% більше. Використання Facebook на мобільних пристроях зростає ще швидше – 67% на рік. Обсяг інформації, яку можна побачити протягом одного дня, дає ще більш вражаюче свідчення величезного впливу соціальних мереж. Майже один мільярд активних користувачів Facebook спільно проводять приблизно 20 000 років онлайн щодня. За ті самі двадцять чотири години YouTube має понад 4 мільярди переглядів, що становить 500 років відео (поширене серед 800 мільйонів унікальних користувачів), а 140 мільйонів активних користувачів X(в минулому Twitter) надсилають понад 340 мільйонів постів. Важливо, що це не просто пасивне використання соціальних мереж.

Аналіз своїх відео, проведений YouTube, показує, що 100 мільйонів людей щотижня здійснюють певні «соціальні дії» (ставлячи «подобається», «не подобається», коментуючи тощо). Ці дії подвоїлися протягом двох років. Тепер Facebook інтегрує соціальні дії у свою онлайн-рекламу, наприклад, дозволяючи користувачам бачити, чи поставили їхні друзі лайки або проголосували за продукти, що рекламуються. Подібним чином хештеги в X (а тепер і на інших платформах соціальних медіа) дали користувачам ще один швидкий і простий спосіб висловити свої вподобання, антипатії, інтереси та занепокоєння, і вони створюють додаткові можливості (або проблеми) для зацікавлених осіб.

Аналіз соціальних мереж — дослідження соціальних мереж, що розглядає соціальні стосунки в термінах теорії мереж. До цих термінів належать поняття *вузла* (зображує окремого учасника в межах мережі) та *зв'язку* (зображує такі відношення між індивідами, як дружба, спорідненість, положення в організації, інтимні стосунки тощо)[3]. Аналіз соціальних медіа складається з триетапного процесу: видобування, розуміння та презентація[4].

Етап видобування включає отримання відповідних даних соціальних медіа шляхом моніторингу або «прослуховування» різних джерел соціальних мереж, архівування відповідних даних і вилучення відповідної інформації. Не всі зібрані дані будуть корисними. На етапі розуміння вибираються релевантні дані для моделювання, видаляються дані низької якості та використовуються різні вдосконалені методи аналізу даних для аналізу збережених даних і отримання на основі них розуміння. Етап презентації має на меті змістовне відображення результатів Етапу розуміння. Між цими етапами є певне збігання. Наприклад, етап розуміння створює моделі, які можуть допомогти на етапі видобутку. Так само візуальна аналітика підтримує людські судження, які доповнюють етап розуміння, а також допомагають на етапі презентації.

Ці етапи проводяться в безперервному, ітеративному процесі, а не строго лінійно. Якщо моделі на стадії розуміння не в змозі виявити корисні закономірності, їх можна вдосконалити шляхом збору додаткових даних, щоб

збільшити їх прогностичну силу. Подібним чином, якщо представлені результати нецікаві або мають низьку прогностичну силу, може знадобитися повернутися до етапів розуміння або захоплення, щоб скоригувати дані або налаштувати параметри, що використовуються в аналітиці. Система, що підтримує аналітику соціальних медіа, може пройти кілька ітерацій, перш ніж стане справді корисною.

Розробники постійно працюють над створенням нових методів виявлення ботів і протидії дезінформації в соцмережах. Сучасні методи[5] виглядають так:

1. Для виявлення ботів застосовується перевірка контексту. Наприклад, якщо опубліковані фото або повідомлення не відповідають типовій поведінці користувача, вони можуть вважатися потенційними ботами. Ще один спосіб — алгоритми перевірки тексту. Вони здатні визначити стиль письма та лексичну схожість із відомими джерелами для пошуку копій.
2. Відстеження нетипових подій та різких змін у соціальній мережі. Наприклад, якщо певний допис або фотографія поширюються занадто швидко чи мають надмірно високу взаємодію проти інших подібних повідомлень, то це може бути ознакою праці ботів.
3. Використання машинного навчання для автоматизації процесу виявлення фейків у соціальних мережах. Тренування на великому обсязі даних допомагає визначити типові патерни, які вказують на те, що певний контент дезінформує користувачів. Офіційно визнані боти можуть використовуватись як приклади, щоб навчити штучний інтелект виявляти подібний контент у майбутньому.
4. Аналізуються десятки критеріїв і параметрів: частота публікації постів на добу, співвідношення рівня активності у вихідні та будні тощо. Ретельно розглядаються теми та слова, на які реагують акаунти, та інші особливості поведінки в мережі. Таким чином вдається відстежити, який саме контент генерується та розповсюджується спеціально створеними програмами.

5. Застосування алгоритмів визначення рівня редагування фото або відео. Крім того, враховуються й технічні дані, навіть показники акселерометра смартфона. Якщо кут нахилу гаджета на кожному фото не змінюється, то ймовірніше зображення створені програмою, а не людиною. Також виявлення фейків можливе з використанням аналізаторів голосу та тексту. Вони сигналізують про підозрілу інформацію на відео або аудіозаписах.

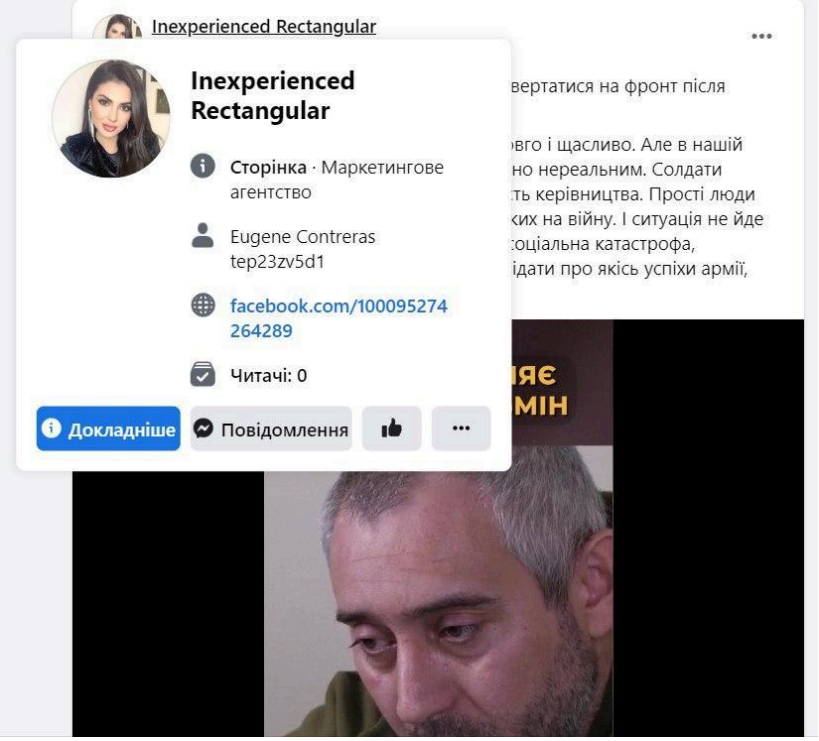


Рисунок 1.1 Прикдад бота

Окрім виявлення ботів зі сторони соціальних мереж, їх можуть виявляти і користувачі. Зазвичай соціальні мережі закликають користувачів до обережного відношення до непідтвердженої інформації і надають поради щодо виявлення ботів або дезінформації[6]. Наприклад:

1. Перевірте зображення. Щоб виявити бота, достатньо придивитися до деталей: тіней, пропорцій та різкості різних елементів. Якщо в кадр потрапили номери авто, назви вулиць, вивіски магазинів, за ними легко

визначити країну та місто, де було зроблено знімок. Для швидкого виявлення фейків перевіряйте унікальність фото за допомоги Google Image Search або TinEye. Ці інструменти дозволяють порівняти знайдені зображення, вказують дату їх появи в інтернеті.

2. Шукайте в дописах мовні маркери дезінформації. Це слова та фрази, що виражають лише припущення автора. Отже, інформація може бути неправдивою. Найрозповсюдженіші маркери: «кажуть», «подейкують», «заведено вважати», «стало відомо», «анонімне джерело повідомляє» тощо.
3. Ознайомтеся з профілем. Фейковий акаунт відрізняється неповними особистими даними (дивне ім'я, не вказана дата народження чи вигадане місце роботи). Також нереальні користувачі мають мало друзів або в них взагалі відсутні фоловери, доступ до дописів обмежений, мало вподобайок і коментарів.

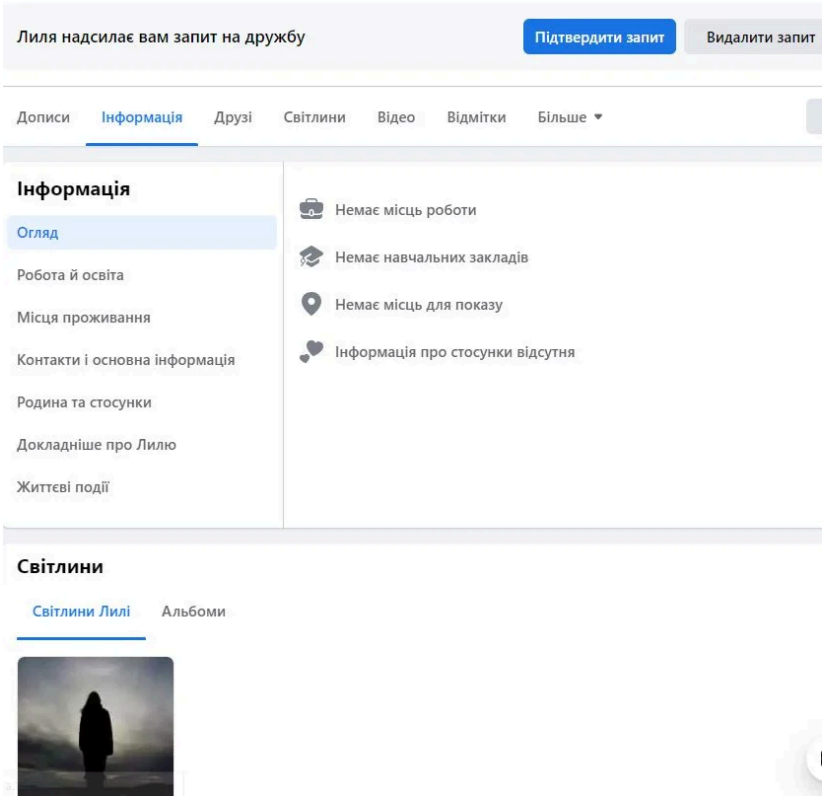


Рисунок 1.2 Приклад пустого акаунту

1.2 Аналіз об’єкта дослідження

За останні декілька років боти в соціальних мережах зробили великий крок вперед у технічному забезпеченні. Вони використовуються різними особами для дуже різних цілей[7]. Боти потрібні як для бізнесу: вони дозволяють наростити ефективність маркетингових інструментів, залучити аудиторію; так і у політиці: вони допомагають просунути ті чи інші ідеї в інформаційному середовищі. І, звісно, боти масово використовує в українських та англомовних інформаційних полях російський агресор для поширення неправдивої інформації, вигідних повідомлень, розпалювання ворожнечі тощо.

Зокрема агресор використовує ботоферми. Вони використовують програмне забезпечення – так звані “бот-комбайни”[3,4] – які дозволяють керувати десятками та сотнями таких акаунтів . Це настільки продумана технологія, що роботи-адміністратори соціальних мереж не завжди можуть відрізнити фейки від реальних користувачів. Ботоферми працюють таким чином. По-перше – отримання профілів користувачів, зареєстрованих в соціальних мережах пів року тому (це підвищує довіру до акаунта у роботів соцмережі). Коштують вони від 5 до 15 доларів за тисячу. Далі купується проксі-лист на всю тисячу свіжокуплених ботів: одночасна публікація тисячі коментарів з однієї адреси моментально фіксується адмінами соціальної мережі і такі профілі одразу блокують. Саме тому кожен бот має свою адресу. Наступне – закупівля спеціалізованого бот-комбайну для адміністрування. Це досить дороговартісна річ, від якості програмного забезпечення залежить успішність роботи цілої ферми. Далі – “прогрів” ботів. Це досить кропітка робота, що полягає в тому, аби впродовж певного часу стрічки цих профілів заповнювались різноманітним контентом – імітація живої людини. Рівень автоматизації тут такий, що боти пишуть один одному в месенджер, відповідають, приймаються в друзі, шлють вітання з Днем народження. Раніше виявити бота було досить легко – просто погугливши його фото, бо для великої кількості профілів світлини елементарно крали або з тих же соціальних мереж, або з бірж

фотозображень. З виникненням штучного інтелекту можна нагенерувати тисячі якісних фото людей, яких насправді не існує. Для цікавого і різноманітного контенту навіть наймають копірайтерів і фотографів. І знову тут штучний інтелект: нагенерувати цікавих коротеньких статей – справа години. Далі бот-комбайн розподілить ці статті між профілями, аби вони не були абсолютно ідентичними – вмикається генерація синонімів, і за декілька хвилин на тисячах профілів з'являються цікаві рецепти, історії з життя, лайфхаки, котики. Головне завдання такого процесу – максимально втягнути користувачів в слідкування даних певних спільнот, сторінок. А далі вже починається те, для чого ці боти були створені. Саме через велику складність розпізнання ботів до початку їх участі в інформаційно-психологічних операціях та велику кількість таких ботів, що можуть діяти майже одночасно, швидкість розпізнання та реакції становить дуже велику важливість.

Для пришвидшення та полегшення реакції на нові інформаційно-психологічні операції важливо не просто швидко помітити початок та позначити всіх учасникі як ботів, а також позначити оригінальні джерела. Також важливою частиною аналізу є перебіг ІПСО в часі.

1.3 Опис предмету дослідження

Лексичний аналіз (або лексичний розбір) — це процес перетворення послідовності символів в послідовність токенів (груп символів що відповідають певним шаблонам), та визначення їх типів. Програма, чи функція що виконує лексичний аналіз, називається лексичним аналізатором, токенізатором чи сканером. Лексичний аналізатор формує першу фазу аналізу тексту. Аналіз зазвичай відбувається за один прохід.

Лексичний аналіз можна розділити на два етапи: *сканування*, яке сегментує вхідний рядок у синтаксичні одиниці, які називаються *лексемами*, та класифікує їх у класи токенів; та *обчислення*, яке перетворює лексеми в оброблені значення.

Лексичні аналізатори, як правило, досить прості, причому більша частина складності відкладена на фази семантичного аналізу[8], та можуть бути згенеровані генераторами лексичних аналізаторів. Однак, лексичні аналізатори можуть іноді мати деяку складність, таку як обробка структури фрази, щоб полегшити вхідні дані, та спростити семантичний аналіз, та можуть бути написані частково, або повністю, вручну, або для підтримки більшої кількості функцій, або для більшої продуктивності. Лексичний аналізатор є скінченним автоматом, перехід в певні стани якого викликає функції, які зазвичай повертають тип лексеми, і саму лексему (токен).

Токен— це рядок літералів, впорядкованих відповідно до правил (наприклад, IDENTIFIER, NUMBER, COMMA). Процес утворення позначок з вихідного потоку називається **видобуванням позначок** і розбирач впорядковує їх за відповідними типами. Лексичний аналізатор зазвичай нічого не робить з об'єднаннями позначок, це завдання припадає на синтаксичний аналізатор. Наприклад, типовий сканер розпізнає дужки як позначки, але саме синтаксичний аналізатор перевіряє відповідність '(' і ')' дужок.

Припустімо наступне речення :

Швидка лисиця перескакнула собаку.

Переводиться в таку таблицю:

Lexeme	**Token type**
швидка	Прикметник
лисиця	Іменник
перескакнула	Дієслово
собаку	Іменник
.	Крапка

Лексичний аналізатор (створений автоматично приладдям або вручну) читає потік символів, визначає лексеми в потоці, і впорядковує їх як позначки.

Це називається «видобуванням позначок». Якщо сканер натрапляє на непридатну позначку, він звітує про помилку.

Наступним кроком іде синтаксичний розбір. Звідти, витрактовані дані можуть бути завантажені в структури даних для загального використання, інтерпретації, компіляції.

Різні письмові мови часто містять набір правил, лексичну граматику, що визначає лексичний синтаксис. Лексичний синтаксис зазвичай є регулярною мовою, граматичні правила складаються з регулярних виразів; вони визначають набір можливих послідовностей символів (лексеми) токена. Лексичний аналізатор розпізнає рядки, та для кожного рядка, знайденого лексичною програмою, вирішує як обробити, найчастіше просто створюється токен.

Дві важливі загальні лексичні категорії — це пробіл і коментарі. Вони також визначаються в граматиці і обробляються лексичним аналізатором, але можуть бути відкинуті (не виробляючи жодних токенів) та вважатися незначними, не більше, ніж розділення двох токенів.

Токенізація — це процес розмежування та, можливо, класифікації секцій рядка вхідних символів. Потім отримані токени передаються іншій формі обробки. Процес можна розглядати як підзадачу аналізу вхідних даних.

Наприклад, у рядку:

Швидка лисиця перескакнула через собаку.

Ці токени можуть бути представлені в XML:

<sentence> <word>Швидка</word> <word>лисиця</word> <word>перескакнула</word> <word>через</word> <word>собаку</word> </sentence>

Або як s-вираз:

(sentence (word Швидка) (word лисиця) (word перескакнула) (word через) (word собаку))

Коли клас токенів представляє більше ніж одну лексему, лексичний аналізатор часто зберігає достатню кількість інформації, щоб відтворити оригінальну лексему, щоб її можна було використовувати у семантичному аналізі. Програми чи функції для подальшого аналізу (надалі синтаксичний аналізатор) зазвичай отримують цю інформацію від лексичного аналізатора та зберігають її в абстрактному синтаксичному дереві. Це необхідно для того, щоб уникнути втрати інформації у випадку чисел та ідентифікаторів.

Токени ідентифікуються на основі конкретних правил лексичного аналізатора. Деякі методи, які використовуються для ідентифікації токенів, містять: регулярні вирази, спеціальні послідовності символів, які називаються флагом, спеціальні розділювальні символи, що називаються роздільниками та явно визначені за словником. Спеціальні символи, включаючи символи пунктуації, зазвичай використовуються лексичними аналізаторами для ідентифікації токенів через їх природне використання у письмових мовах .

Токени часто класифікуються за вмістом символів або за контекстом у потоці даних. Категорії визначаються правилами лексичного аналізатора. Вони часто включають елементи граматики мови, що використовується в потоці даних. Письмові мови зазвичай класифікують токени як іменники, дієслова, прикметники або знаки пунктуації. Категорії використовуються для подальшої обробки токенів або парсером, або іншими функціями у програмі.

Взагалі, лексичний аналізатор нічого не робить з комбінаціями токенів, ця задача залишається синтаксичному аналізатору. Наприклад, типовий лексичний аналізатор розпізнає дужки як токени, але нічого не робить для того, щоб кожна «(» відповідала «)».

Коли лексичний аналізатор подає токени синтаксичному аналізатору, він зазвичай представляє їх як перелічений список числових представлень. Наприклад, «Іменник» представлений як 0, «Прикметник» як 1, «Дієслово» як 2, і т. д.

Перший етап, зазвичай називаємий сканером, зазвичай базується на скінченному автоматі. В ньому закодована інформація про можливі послідовності символів, які можуть міститися в будь-якому з токенів, які він обробляє (окремі екземпляри цих послідовностей символів називаються лексемами). Наприклад, цілочисельний токен може містити будь-яку послідовність цифрових символів. У багатьох випадках перший символ, що не є пробілом, може бути використаний для виведення наступного токена, далі наступні вхідні символи обробляються один за одним до досягнення символу, який не знаходиться в наборі символів, прийнятих для цього токена.

Оцінювач

Лексема, однак, є лише рядком символів, відомих як певний тип (наприклад, строковий літерал, послідовність букв). Для того, щоб побудувати токен, лексичний аналізатор потребує другий етап, — оцінювання, який проходить по всім символам лексеми, щоб отримати значення. Тип лексеми в поєднанні з його значенням — це те, що правильно становить токен, який може бути наданий синтаксичному аналізатору. Деякі токени, такі як дужки, насправді не мають значень, і тому функція оцінювання для них може нічого не повертати: потрібний лише тип. Аналогічно, іноді оцінювачі можуть повністю приховати лексему від синтаксичного аналізатора, що є корисним для пробілів і коментарів. Оцінювачі для ідентифікаторів, як правило, прості , але можуть містити деяке згортання. Оцінювачі цілочисельних літералів можуть передавати рядок (відкладаючи оцінювання до фази семантичного аналізу), або можуть виконувати оцінювання власноруч, яке може бути залучено для різних основ або чисел з рухомою комою. Для простого літералу у лапках оцінювач має

видалити лише лапки, але оцінювач для екранованого строкового літералу містить лексичний аналізатор, який скасовує екранування послідовності.

Наприклад, у вихідному коді комп'ютерної програми, рядок може бути перетворений у наступний лексичний потік токенів; пробіли відхиляються, а спеціальні символи не мають значення:

IDENTIFIER net_worth_future EQUALS OPEN_PARENTHESIS IDENTIFIER assets MINUS IDENTIFIER liabilities CLOSE_PARENTHESIS SEMICOLON

Створення лексичного аналізатора вручну можливо, якщо список токенів невеликий, але в цілому, лексичні аналізатори генеруються за допомогою автоматизованих інструментів. Ці інструменти зазвичай приймають регулярні вирази, які описують токени, дозволені у вхідному потоці. Кожен регулярний вираз пов'язаний з правилом виводу у лексичній граматиці писемної мови, яка оцінює лексеми, що відповідають регулярному виразу.

Регулярні вирази компактно зображують шаблони, за якими можуть слідувати символи в лексемах. Наприклад, для англійської, токен IDENTIFIER може бути будь-яким англійським алфавітним символом або символом підкреслення, за яким слідує будь-яке число екземплярів ASCII символів та/або підкреслення. Це може бути компактно представлено рядком [a-zA-Z_][a-zA-Z_0-9]*. Це означає «будь-який символ a-z, A-Z або _, за яким слідує 0 або більше a-z, A-Z, _ або 0-9».

1.4 Аналіз актуальності

У той час як спеціалісти регулярно розпізнають та протидіять ІПСО, реакція зазвичай відбувається занадто повільно. В більшості випадків реакція відбувається вже після того як ІПСО привернуло до себе велику кількість уваги і почало виконувати свою функцію. Пришвидшена швидкість реакції надасть змогу мінімізувати наслідки ІПСО. Також використання автоматизованої системи для виконання рутинних завдань, звільнить час в спеціалістів для

творчої роботи, що являє собою велику частину протидії дезінформації. Наші вороги постійно розвивають та вдосконалюють засоби інформаційної війни, і збільшують кількість ІПСО. Для протидії знадобиться більше часу в спеціалістів, зокрема часу на творчу роботу, .

Також спеціалісти працюють якнайбільше в укро- та російськомовних сферах, тоді як агресор активно працює і в іноземних. Відсутність українських поглядів в англомовній сфері надає ворогу велику перевагу в інформаційній війні. Найбільший приклад російських тролей в англомовній сфері є їхня участь в спробах вплинути на вибори в Америці в 2016 році. Україна покладається на допомогу від іноземних країн, в яких більша кількість комунікацій відбувається на англійській мові. Враховуючи це, перебіг інформаційної війни в англомовній сфері має дуже велике значення. Автоматизована система допоможе активніше приймати в ній участь.

Окрім свого прямого застосування автоматизована система надає дані для подальшого аналізу перебігу ІПСО. Аналіз зібраних даних може надати можливість ефективніше розпізнавати ботів, що дуже важливо оскільки боти та ботоферми завжди розвиваються і стають складніше. Також автоматизована система може полегшити навчання нових спеціалістів. Інформаційна війна не сповільнюється, а навпаки прискорюється, особливо з вдосконаленням штучного інтелекту. Попит на спеціалістів і навантаження на вже існуючих лише зросте, і автоматизована система дасть можливість протидіяти цим трендам.

1.5 Стан вже існуючих рішень

Сучасні методи протидії ботам як зі сторони власників соціальних мереж так і зі сторони користувачів недосконалі. До того ж кількість ботів в соціальних мережах зростає без видимого кінця. У 2021 році мережа Фейсбук заблокувала і видалила 1 300 000 000 фейкових акаунтів: це при тому, що наразі мережа має близько 3 000 000 000 активних користувачів. Інстаграм виявив 95 000 000

ботів, Твіттер – 20 000 000, при загальній кількості 1 200 000 000 активних користувачів та 1 300 000 000 відповідно. Також з розвитком штучного інтелекту засоби які працювали в минулому, наприклад перевірка фотографій перестануть працювати, якщо це вже не відбулося. Штучний інтелект дає можливість генерувати сотні фотографій, які не можливо відрізнити від людей на око і не існують ніде в інтернеті. Якщо мова йде конкретно про ІПСО то можна сказати що реакція на них занадто повільна і не завжди можлива через недостатню кількість спеціалістів та більшу швидкість роботи у автоматизованих систем (ботів) ніж у людей. Особливо важка ситуація склалась в англомовному просторі, бо українські медіа там представлені слабо. А українські користувачі традиційно проводять час в україномовному або російськомовному просторі. Тому ІПСО супротивника в англомовному просторі проходять майже непоміченими українським суспільством.

1.6 Визначення цілей дослідження та постановка задачі

Метою досліджень в роботі є створення автоматизованої системи виявлення елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики, задля зменшення часу на розпізнання та реакцію, та надавання даних для подальшого аналізу.

Реалізація мети здійснюється вирішенням наступних основних завдань:

- створення набору базової лексики, а також список акаунтів, що вже визначені як боти і підлягають моніторингу;

- створення автоматизованої системи, що буде використовувати розширений лексичний аналіз для розпізнання початку нових ІПСО, після чього створить картку ІПСО і занесе в неї зібрані дані;

- створення інтерфейсної частини, забезпечення подачі картки ІПСО кінцевому користовачу.

Об’єктом дослідження роботи є системи виявленя елементів інформаційно-психологічної операції.

Предметом дослідження є створення автоматизованих систем розпізнання та збирання даних за допомогою розширеного аналізу лексики.

Система є розпізнавателем ІПСО.

Зовнішнім середовищем є нові пости акаунтів, що вже відомі як боти та знаходяться під моніторингом, та зібрані дані, що заносяться в картку ІПСО.

Незважаючи на те, що в даному напрямку вже іде робота зі сторони спеціалістів, тема ще має багато напрямків для розвитку – зокрема збільшення щвидкості реакції. При подальшій еволюції системи з’явиться можливість не тільки помічати початок нових ІПСО та збирати дані, а й навчати використовувати ці дані для подальшого аналізу інших акаунтів та розпізнання ботів.

2. АЛГОРИТМІЧНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Область видимості

Однією з найголовніших ідей практично всіх мов програмування є можливість зберігати значення змінних, а потім отримувати або змінювати ці значення. Правила додавання змінних до нашої програми так, щоб потім можна було їх знайти та отримати їхнє значення, повинні визначатися чіткими межами[9]. Ці межі визначають, які змінні будуть нам доступні у певних місцях програми, та процес пошуку змінних за їх ідентифікатором (ім'ям). Якщо дати більш технічне визначення, воно звучить так: Область видимості змінних чи просто “Область видимості” (англ. variable scope чи просто scope) — це область програми, у межах якої встановлено зв'язок між деякою змінною та її ідентифікатором (іменем), яким можна отримати значення цієї змінної[8]. За межами області видимості той самий ідентифікатор може бути пов'язаний з іншою змінною, або бути вільним (взагалі не пов'язаним з жодною зі змінних).

Для розуміння цієї концепції можна навести таку аналогію. Уявімо людину, якій було призначено ділову зустріч у деякому бізнес-центрі, де вона раніше не була. Він заходить у потрібний кабінет і тому що там ще нікого немає, він вирішує дізнатися час, для цього йому потрібний годинник. Ця людина починає шукати годинник, виявляє настільний годинник і впізнає за ним час. У цій аналогії: область видимості (scope) – це кабінет; ім'я змінної, через яку відбувається пошук потрібного предмета, це “годинник” (позначимо як clock); у цій галузі видимості, з цим ідентифікатором (“годинник”) асоціюється настільний годинник.

Тут варто згадати, що змінні, це свого роду контейнери (області пам'яті) для зберігання необхідних значень. Саме настільний годинник у рамках кабінету виступає деяким контейнером для зберігання часу. значення цієї змінної позначимо як "час на настільний годинник".

Тепер припустимо, що людина в кабінеті не знайшла абсолютно ніякого годинника, тобто в цій галузі видимості немає ніяких змінних (ні настільних

годинників, ні будь-яких інших), які були б пов'язані з ідентифікатором "годинник" (clock). І в такому випадку, в JavaScript запит отримання змінної з ім'ям clock викликає помилку: Uncaught ReferenceError: clock is not defined. Яка повідомляє про те, що в цій галузі видимості змінна, пов'язана з ідентифікатором clock, не визначена.

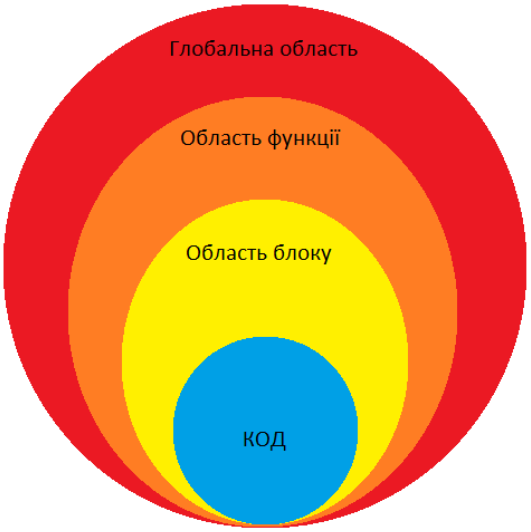


Рисунок 2.1 Приклад області видимості

Розвинемо аналогію далі і уявімо, що хоч годин у кабінеті і немає, ця людина знає, що в холі будівлі є настінний годинник. Він знає про це, оскільки проходячи до кабінету, наголосив на їх наявності. І тому він може отримати потрібне значення (час) зі свого оточення - "визирнути" з кабінету і подивитися час на настінному годиннику. У цій аналогії будівля - це ще одна область видимості, яка також є оточенням для області видимості "кабінет". У даній будівлі ідентифікатор "годинник" (clock) пов'язаний зі змінною "настінний годинник у холі".

Ми можемо продовжити цю аналогію. Уявимо, що і в будівлі немає годинника, але вони є на вулиці. Тому, навіть перебуваючи в кабінеті, він може

звернутися до свого оточення — визирнути на вулицю та подивитися час на вуличному годиннику.

У JavaScript області видимості обмежуються функціями (функціональна область видимості) або блоками інструкцій (блокова область видимості).
Також необхідно зазначити, що змінні, оголошені всередині функції, недоступні зовні, тобто навіть якщо годинник є в кабінеті, а людина знаходиться в будівлі або на вулиці, де немає жодного годинника, то звернеться до годинника в кабінеті вона не зможе. Більше того, він у принципі не має жодного уявлення про вміст кабінету, доки туди не зайде.

Якщо нам необхідно, щоб змінна була доступна звідусіль, її необхідно оголосити поза всіма функціями, тобто глобально. Глобальна область видимості - це та, яка не має зовнішньої області видимості (зовнішнього оточення). Вона є відправною точкою нашої програми і зовнішньою областю видимості для всіх інших, які в неї вкладені. У нашій аналогії, нехай це буде всесвіт, де є годинник, який можна побачити з будь-якої точки.

Змінні називаються глобальними, якщо вони доступні будь-якої точки програми. Тобто ті, які оголошені в глобальній, зовнішній області видимості. Змінні, які оголошені всередині функції та недоступні зовні, називаються локальними змінними. Область видимості, обмежена функцією чи блоком, це локальна область видимості. Глобальна область видимості не вкладена в жодні інші області і знаходиться поза всіма функціями, будучи батьківською для всіх інших вкладених у неї областей видимості.

У випадку, якщо та сама змінна оголошена в декількох вкладених областях видимості, її значення буде братися з поточної області видимості, де вона запитується. Якщо в поточній області змінна не визначена, як функції street, то значення візьметься з найближчого оточення (батьківської області видимості), в якому визначена необхідна змінна. У даному випадку, для області видимості street, значення clock буде братися з глобальної області видимості.

Це називається “затінення змінних”, коли поточна змінна приховує значення змінної оголошеної батьківської області видимості. Взагалі, щоб уникнути плутанини при командній розробці та неясності в тому, в яких змінних зберігається, краще не застосовувати затінення і використовувати різні імена змінних. Поки що ми розбирали локальні області, які були обмежені функціями. Але область видимості може бути обмежена блоком інструкцій.

Оточення області видимості — це доступна з поточної області видимості структура даних, в якій зберігаються зв'язки між ідентифікаторами та змінними з усіх зовнішніх областей видимості. Є дві переважаючі моделі того, як працює/влаштована область видимості: лексична (використовується в більшості мов програмування) і динамічна.

2.2 Лексична область видимості

Лексичний аналіз і є та концепція, де грунтується розуміння, що таке лексична область видимості і звідки походить її назва. Лексична область видимості - це область видимості, яка визначена під час аналізу лексеми. Іншими словами, лексична область видимості формується виходячи з того, де змінні, функції та інструкції розміщені в коді. Спочатку викликається функція wrapper, яка спочатку виведе значення змінної value і потім викличе функцію showValue. В області видимості функції wrapper змінна value "затіняє" змінну, оголошену в глобальній області видимості і тому функція wrapper виведе - "Value from wrapper: 3". Далі викликається функція showValue, яка також відображає значення змінної value, тільки в цій функції воно є числом 2. Оскільки лексичні області видимості (зв'язку змінних зі своїми ідентифікаторами) формуються на етапі лексичного розбору, а не на етапі виконання програми, то змінна value функції showValue буде взято з батьківської (для showValue) області видимості, де її значення — 2. Тому незважаючи на те, що showValue викликається з функції wrapper, де оголошено свою змінну value, функція showValue виведе значення тієї змінної, яка була зафіксована під час лексичного аналізу, тобто число 2.

Динамічна область видимості визначається не так на етапі написання коду, як і лексичної моделі, але в етапі виконання програми. Якби у JavaScript була динамічна область видимості, то при виконанні кожної функції з попереднього прикладу в консолі виводилося б цифра 3. Функція wrapper виводитиме 3, оскільки змінна value визначена в поточній області видимості, де їй присвоєно значення 3. Тепер розберемо чому при динамічній області видимості функції showValue значення змінної value теж буде 3. У цьому випадку, не знайшовши змінну value в локальній області видимості функції showValue, лексичний аналізатор замість підняття по ланцюжку навколишніх лексичних областей видимості, буде підніматися вгору по стеку викликів функцій , щоб знайти звідки showValue() була викликана. Оскільки showValue() викликалася з wrapper(), він шукатиме змінну саме там — в області видимості wrapper(), і виявивши її, встановить зв'язок між цією змінною в області видимості wrapper() та ідентифікатором, який запитується в showValue.

Насправді у JavaScript немає динамічної області видимості, але уявлення про її принципи роботи стане в нагоді[8]. Ключове порівняння: лексична область видимості визначається часом написання коду, тоді як динамічна область видимості визначається під час виконання програми. Лексичну область видимості цікавить, де функцію було оголошено, а динамічну — звідки була викликана функція. Лексичне оточення - це певна структура, яка використовується для визначення зв'язку Ідентифікаторів (імен) з конкретними змінними та функціями на основі вкладеності (ланцюжка) лексичних областей видимості. Змінні починають своє існування тоді, коли виконання програми досягає їхньої області видимості. І тоді цим змінним потрібне місце для зберігання, щоб згодом до них можна було звернутися навіть із вкладених областей видимості. У специфікації JavaScript структура даних, яка забезпечує такий простір для зберігання змінних у пам'яті, а також надає механізм їх пошуку та можливість отримати до них доступ з внутрішніх областей, називається лексичним оточенням. Воно зіставляє ідентифікатори (імена) зі

змінними і функціями вже у межах однієї області видимості, а цілих ланцюжків вкладених один одного областей. Його структура дуже нагадує структуру об'єктів JavaScript.

Є дві моделі взаємодії та роботи зі змінними: Лексичне оточення (пов'язане з ланцюжком областей видимості) та Динамічне (пов'язане зі стеком контекстів виконання). Так як проектна робота виконується на Javascript розглядатиметься пристрій Лексичного оточення. Суть лексичних областей видимості в тому, що навіть на етапі виконання програми вони зберігають зв'язок зі своїми зовнішніми/батьківськими областями, які були визначені на етапі лексичного розбору (тобто сформовані виходячи з того, де змінні, функції та інструкцій спочатку були написані у коді). За рахунок збереження такого зв'язку формується цілий ланцюжок областей видимості, кожна з яких знає свою батьківську область видимості.

Як це реалізується в JavaScript? Кожна функція, яка є своєю чергою окремою областю видимості, на етапі ініціалізації, запам'ятовує свою батьківську область видимості, де вона міститься. Це відбувається за рахунок того, що кожна функція має внутрішню властивість [[Environment]] яка зберігає в собі посилання на зовнішню область видимості (ця властивість недоступна нам із самої програми і використовується JavaScript-движком). Якщо розглядати внутрішній пристрій Лексичного оточення у межах специфікації ES9. То воно складається із Запису Оточення та посилання на зовнішнє лексичне оточення. Зазвичай Лексичне Оточення асоціюється з певними синтаксичними структурами JavaScript, такими як оголошення функцій або блоками інструкцій. Щоразу коли обробляється такий код, наприклад викликається функція, то цієї нової області видимості створюється своє Лексичне оточення. Для цього оточення формується:

1. Запис Оточення environment record, що містить у собі зв'язки ідентифікаторів зі змінними, які створені в області видимості цього Лексичного оточення. Також

вона містить і іншу необхідну інформацію, наприклад значення ключового слова цього, про яке буде розказано в іншій частині.

2. Посилання outer, яке вказує на зовнішнє/батьківське оточення для цієї області видимості. Саме в це поле потрапляє значення внутрішньої властивості Environment, яке зберігає в собі посилання на батьківську область видимості. І тому завжди існує ланцюжок Лексичних оточень. Вона починається з поточного (виконується в даний момент) Лексичного оточення, продовжується зовнішніми оточеннями, і закінчується глобальним Лексичним оточенням, у якого поле outer одно null.

Відразу варто сказати, що Лексичні оточення є виключно описовою концепцією пристрою та роботи програми на JavaScript. Воно не має будь-якої реалізації в коді JavaScript і в програмі немає прямого доступу до нього та можливості маніпулювати їм безпосередньо. Але розуміння цієї концепції дає розуміння про структуру коду, значення цікавих для нас змінних і доступність тих чи інших даних у певній точці програми. Тут важливо зауважити, що на рядку інтерпретатор спочатку намагається знайти потрібну змінну в поточному записі оточення (спочатку y, потім x), а потім, не виявивши її в поточному записі оточення, шукає в зовнішньому оточенні. Змінну y буде знайдено у батьківському оточенні fooEnvironment. А, щоб знайти змінну x, інтерпретатор піде ланцюжком далі, до самого глобального оточення GlobalEnvironment.
Такий порядок пошуку можливий завдяки тому, що посилання на зовнішній об'єкт змінних зберігається в полі outer, яке в свою чергу встановлюється з внутрішньої властивості функції - Environment. Ці властивості закриті від прямого доступу, але знання про них дуже важливе для розуміння того, як працює JavaScript. Якщо у поточному оточенні потрібної змінної немає, завдяки існуючому у кожному лексичному оточенні полю outer, де міститься посилання батьківське оточення, пошук триває до того часу, поки змінна не виявиться у якомусь із зовнішніх оточень. У випадку, якщо інтерпретатор дійшов

ланцюжком до глобального оточення, у якого поле outer дорівнює null, і при цьому і там не знайшов необхідну змінну (тобто змінна не була оголошена ніде в коді), тоді виникне помилка ReferenceError: nameOfVariable is not defined.

Рисунок 2.2 — схематичне представлення Лексичної області видимості

Наприкінці виконання функції відбувається повернення на те місце, де вона була викликана і виконання коду триває далі. При цьому зазвичай після виконання функції Запис Оточення зі змінними видаляється та пам'ять очищається.

2.3 Проектування системи

Надсистема, що досліджується – комплекс інструментів для виявлення елементів ІПСО. Сюди входять: база даних з інформацією про підозрювані акаунти та вже відомі елементи ІПСО, функція, котра зчитує текст постів цих акаунтів, самі пости та акаунти, лексичний аналізатор, що регресує текст на лексеми та класифікує їх на токени та функція, що порівнює отримані лексеми з базою даних та іншими постами(рис. 2.3). Сама система – це лексичний алгоритм, який аналізує текст на повторення вже відомих елементів ІПСО, або початок нових та надсилає отримані дані спеціалістам по боротьбі з інформаційними операціями супротивника при виявленні.

Зовнішньо система залежить від функції, що зчитує тексту, бо лексичний алгоритм не зможе правильно виконувати аналіз якщо отримає текст з помилками.

Внутрішньо система складається з таких компонентів, як: лексичний аналізатор, функція виявлення елементів ІПСО, база вже відомих елементів,

підозрюваних акаунтів та нововиявлених елементів та пристрої виводу вводу, що надсилають дані спеціалістам.

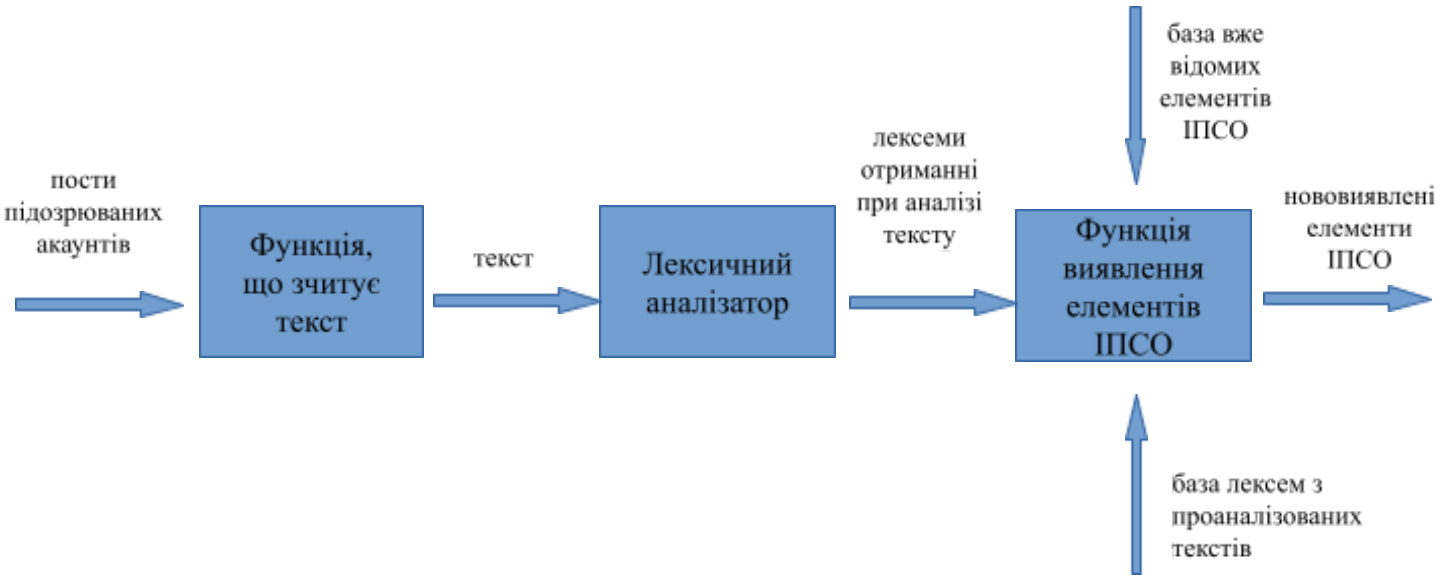


Рисунок 2.3. Структура надсистеми

Лексичний аналізатор користується регулярними визначеннями для аналізу та класифікації лексем на різні токени.

Регулярне визначення можна виразити даним регулярним виразом:

letter = [*a* − *zA* − *Z*],

де «a-z» – діапазон, що означає усі символи латинського алфавіту нижнього регістру;

«A-Z» – верхнього регістру.

Регулярне визначення може використовуватися в регулярному виразі для будь-якого елементу в рамках тієї ж лексичної граматики:

letter = [a − zA − Z]

digit = [0 − 9]

*identifier = (letter | _) (letter | digit | _) *,*

де *letter* – означає будь який символ нижнього та верхнього регістру латинського алфавіту;

digit – цифру від 0 до 9;

identifier – валідна назва змінної.

Тобто правильний ідентифікатор повинен починатися з великої або малої літери латинського алфавіту або зі знаку нижнього підкреслювання, за яким слідують символи, цифри або знаки нижнього підкреслювання у кількості від 0 до безкінечності. Як видно, в регулярному визначенні ідентифікатор використовує визначення символу та цифри в своєму правилі для того, щоб визначити що ідентифікатор це будь який рядок, що починається з символу або нижнього підкреслення за яким слідує символ, цифра або знак нижнього підкреслення, яке можна використати від нуля до безкінечної кількості разів.

Основними сутностями предметної галузі в області лексичного аналізу тексту задля відстежування ботів або фейкових акаунтів є оцінюваний текст (T), набір лексем які є складовими частинами тексту (L), лексичний аналізатор, на основі якого реалізовується лексичний алгоритм (LA), лексичний алгоритм визначення реальності акаунтів (LCA), лексичний синтаксис (LS), та список слів, які пов’язані з акаунтами, що вже приймають участь у ІПСО (WL).

Лексичний алгоритм можна виразити наступною формулою:

$$LCA = < T, L, LA, WL, LS >,$$

де T– текст, що підлягає оцінюванню,

L– набір лексем, які є складовою будь-якого тексту,

LA– лексичний аналізатор,

WL– список слів, які пов’язані з акаунтами, що вже приймають участь у ІПСО,

LS – лексичний синтаксис,

В свою чергу, даний для оцінювання текст (T), можна описати наступною формулою:

$$T = < L >,$$

де L – це набір лексем, які є основою та складовою частиною будь-якого тексту і які будуть розпізнаватися лексичним аналізатором.

Лексичний аналізатор (LA), в свою чергу можна описати набором таких

сутностей:

$$LA = < TA, LS, WL >,$$

де ТА – набір токенів, отриманих з розпізнаних лексем;

LS – лексичний синтаксис, тобто набір частин мови в залежності від синтаксису англійської мови, що визначають синтаксис токенів;

WL – список слів, які пов’язані з акаунтами, що вже приймають участь у ІПСО. Регулярні визначення можна описати всіма можливими метасимволами та алфавітами.

В свою чергу, лексичний синтаксис можна описати як набір класів слів:

$$LS = < C >,$$

де С – набір частин мови, які описуються регулярними виразами, які задаються для даної мови і можуть бути описані як набір всіх можливих правил:

$$C = < noun, identifirer, verb, preposition, conjuction, adverb, pronoun, adjective, determiner, interjection, punctuation mark >,$$

де кожному з правил буде описаний регулярний вираз;

RD – регулярні визначення, що визначають можливу структуру валідного токену. Регулярні визначення можна описати всіма можливими метасимволами та алфавітами.

2.4. Алгоритмічне забезпечення програми

Для того щоб розпізнати токен описаний регулярним визначення, регулярний вираз в визначенні часто трансформується в КАСи – скінченні автомати станів (FSMs – Finite State Machines). Результуючий КАС має скінченний набір станів, що включає початковий стан та набір можливих станів[10]. КАС переходить від одного стану до іншого шляхом «поїдання» одного з символів або елементів регулярного виразу. Після переходу з початкового стану в один з приймаючих станів виходить дійсний рядок, описаний регулярним виразом. Наприклад, регулярний вираз «a | b» (a чи b) може бути представлений у вигляді наступного КАСу (рис. 2.4).

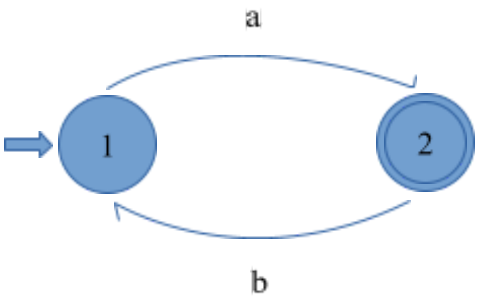


Рисунок 2.4 – Скінченний автомат станів для регулярного виразу «a | b»

Представлений вище КАС має два стани помічених номерами 1 і 2. Стрілка що вказує на 1 і приходить нізвідки означає, що 1 – це початковий стан, а внутрішнє коло в стані 2 означає, що 2 – це приймаючий стан цього КАСу.

Недетермінований скінчений автомат – це п’ятірка:

$$M = < Q, \Sigma, \delta, q0, F>$$

, де - Q = {q0, q1, .. , qn-1} – скінчена множина станів автомата;

- Σ = {a1, a2, .., am} – скінчена множина вхідних символів (вхідний алфавіт);

- q0 ∈ Q – початковий стан автомата;

- δ – відображення множини Q*Σ в множину P(Q). Відображення ❤ як правило називають функцією переходів;

- F ⊆ Q – множина заключних станів. Елементи з F називають заключними або фінальними станами.

Якщо M – скінчений автомат, то пара (q, w) ∈ Q*Σ* називається конфігурацією автомата M. Оскільки скінчений автомат – це дискретний пристрій, він працює по тактам. Такт скінченого автомата M задається бінарним відношенням |= , яке визначається на конфігураціях: (q1,aw) |= (q2,w) , якщо δ(q1, a) містить q2 та для всіх w ∈ Σ*.

Скінчений автомат M розпізнає (допускає) ланцюжок w, якщо (q0, w) |=* (q, ε) для деякого q ∈ F, де |=* - рефлексивно-транзитивне замикання бінарного відношення |= . Мова, яку допускає автомат M розпізнає автомат M:

$$L(M)=\{ w \mid w \in \Sigma^* \text{ та } (q0, w) |=^* (q, \varepsilon) , q \in F \}$$

При визначенні скінченого автомата M, використовують декілька способів визначення функції δ, наприклад:

- це табличне визначення δ;
- діаграма проходів скінченого автомата.

Табличне визначення функції δ - це таблиця M(qi,aj), де aj ∈ Σ, qi ∈Q, тобто M(qi,aj) = { qk | , qk ∈ δ(qi,aj) }.

Діаграма переходів скінченого автомата M - це невпорядкований граф G(V, P), де V – множина вершин графа, а P – множина орієнтованих дуг, причому з вершини qi у вершину qj веде дуга позначена ak , коли qj∈δ(qi,ak). На діаграмі переходів скінченого автомата це позначається так:

Рисунок 2.5 — Діаграма переходів скінченого автомата

Скінчений автомат M називається детермінованим, якщо δ(qi, ak) містить не більше одного стану для любого qi ∈ Q та ak ∈ Σ. Для довільного недетермінованого скінченого автомата M можна побудувати еквівалентний йому детермінований скінчений автомат M1, такий що L(M) = L(M1).

В подальшому при програмуванні скінчених автоматів важливо мати справу з так званими "мінімальними автоматами". Мінімальним для даного скінченого автомата візьмемо еквівалентний йому автомат з мінімальною кількістю станів. Навіть при поверхневому аналізі діаграми переходів наведеного скінченого автомата видно, що вершини q3, q4 та q5 є "зайвими", тобто при їх вилученні новий автомат буде еквівалентний початковому.

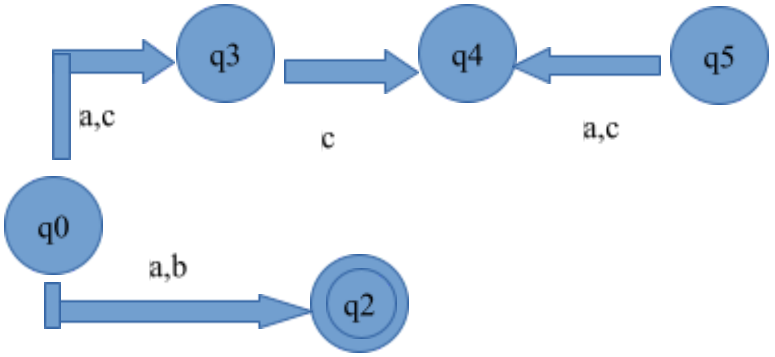


Рисунок 2.6 — Діаграма переходів із «зайвими» вершинами

З наведеного вище прикладу видно, що для отриманого детермінованого скінченого автомата можна запропонувати еквівалентний йому автомат з меншою кількістю станів, тобто мінімізувати скінчений автомат. Очевидно що серед зайвих станів цього автомата є недосяжні та тупикові стани.

Стан q скінченого автомата M називається недосяжним, якщо на діаграмі переходів скінченого автомата не існує шляху з q0 в q. Для знахождения недосяжних станів спочатку спробуємо побудувати множину досяжних станів. Якщо Qm - множина досяжних станів скінченого автомата M, то Q\Qm - множина недосяжних станів. Побудуємо послідовність множин Q0, Q1, Q2, … таким чином, що:

1. Q0 = { q0 }.
2. Q1 = S0 ∪ q | q ∈ δ (q0, a), для всіх a ∈ Σ }.
3. Qi = Si-1 ∪ q | q ∈ δ (qj, a), qj ∈Qi-1 та для всіх a ∈ Σ }.

…….

n. Qn = Qn+1 = ……..

Очевидно, що кількість кроків 1, 2… скінчена, тому що послідовність Qi монотонна (Q0 ⊆ Q1 ⊆ Q2 ⊆ ….) та обмежена зверху - |Qm| <= |Q|. Тоді Qm – множина досяжних станів скінченого автомата, а Q\Qm – множина недосяжних станів. Вилучимо з діаграми переходів скінченого автомата M недосяжні вершини. В новому (мінімізованому) автоматі функція δ визначається лише для досяжних станів. Побудований нами скінчений автомат з меншою кількістю станів буде еквівалентний початковому.

Стан q скінченого автомата M називається тупиковим, якщо на діаграмі переходів скінченого автомата не існує шляху з q в F. Для пошуку тупикових станів початку спробуємо знайти нетупикові стани. Якщо Sm - множина нетупикових станів, то Q\Sm - множина тупикових станів. Побудуємо послідовність множин S0, S1, S2, … таким чином, що:

1. $S0 = \{ q \mid q \in F \}$.
2. $S1 = S0 \cup \{ q \mid \delta(q, a) \cap S0 \neq \varnothing , a \in \Sigma \}$.
3. $Si = Si\text{-}1 \cup \{ q \mid \delta(q, a) \cap Si\text{-}1 \neq \varnothing , a \in \Sigma \}$.

…….

n. Sn = Sn+1 = ……..

Очевидно, що кількість кроків 1, 2… скінчена, тому що послідовність Si монотонна ($S0 \subseteq S1 \subseteq S2 \subseteq \ldots.$) та обмежена зверху - |Sm| <= |Q|. Тоді Sm – множина нетупикових станів скінченого автомата, а Q\Sm – множина тупикових станів. В новому (мінімізованому) автоматі функція δ визначається лише для нетупикових станів. В КАСі зображеному на рис. 2.6 множина недосяжних станів - {q5}, а множина тупикових станів - {q3, q4}. Таким чином, для вище наведеного скінченого автомата еквівалентний йому автомат з меншою кількістю станів буде:

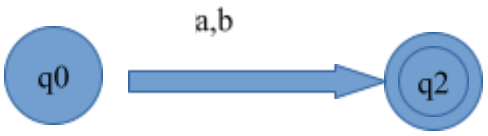


Рисунок 2.7 — діаграма переходів мінімального автомату

Автомат, у котрого відсутні недосяжні та тупикові стани, піддається подальшій мінімізації шляхом “склеювання” еквівалентних станів.

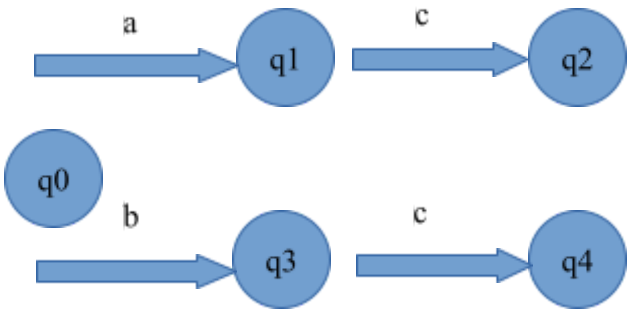


Рисунок 2.8 — діаграма переходів автомату у якого відсутні недосяжні та тупикові стани

Очевидно, що для наведеного вище скінченого автомата можна побудувати еквівалентний йому скінчений автомат з меншою кількістю станів:

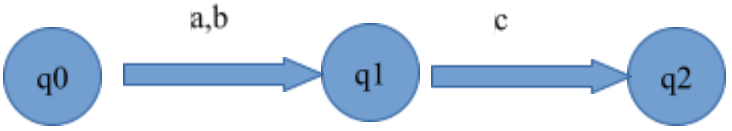


Рисунок 2.9 — діаграма переходів «склеєного» автомату

Ми досягли бажаного нам результату шляхом “склеювання” двох станів q1, q3. та q2, q4. Два стани q1 та q2 скінченого автомата M називаються еквівалентними, якщо множини слів, які розпізнає автомат, починаючи з q1 та q2, співпадають. Алгоритм побудови мінімального скінченого автомата виглядає так:

1. Побудувати скінчений автомат без тупикових станів.
2. Побудувати скінчений автомат без недосяжних станів.
3. Знайти множини еквівалентних станів та побудувати найменший (мінімальний) автомат.

Граматику G можна виразити так:

$$G=<N, \Sigma, P, S>$$

,де: N – скінчена множина - допоміжний алфавіт (нетермінали);

Σ – скінчена множина – основний алфавіт (термінали);

P – скінчена множина правил виду $\alpha \to \beta$, $\alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)$.

S – виділений нетермінал (аксіома).

В залежності від структури правил граматики діляться на чотири типи (класифікація граматик по Хомському):

- Тип 0: граматики загального виду, коли правила не мають обмежень, тобто $\alpha \to \beta$, $\alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)$.
- Тип 1: граматики, що не укорочуються, коли обмеження на правила мінімальні, а саме:

$\alpha \to \beta$, $\alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^*$, $|\alpha| \leq |\beta|$

- Тип 2: контекстно-вільні граматики, коли правила в схемі P мають вигляд:

$Ai \to \beta$, $Ai \in N$, $\beta \in (N \cup \Sigma)^*$.

- Тип 3: скінченоавтоматні граматики, коли правила в схемі P мають вигляд:

$Ai \to wAj$, $Ai, Aj \in N$, $w \in \Sigma^*$;

$Ai \to w$, $w \in \Sigma^*$;

$Ai \to wAj$, $Ai, Aj \in N$, $w \in \Sigma^*$.

В класі скінченоавтоматних граматик виділимо так звані праволінійні граматики – це граматики, які в схемі P мають правила виду :

$Ai \to wAj$, $Ai, Aj \in N$, $w \in \Sigma^*$;

$Ai \to w$, $w \in \Sigma^*$;

Клас праволінійних граматик співпадає з класом граматик типу 3. Клас мов, що породжуються праволінійними граматиками, співпадає з класом мов, які розпізнаються скінченими автоматами.

Нехай Σ - скінчений алфавіт. Регулярна множина в алфавіті Σ визначається рекурсивно:

1. $\varnothing$ - пуста множина – це регулярна множина в алфавіті Σ;
2. $\{\varepsilon\}$ – пусте слово – регулярна множина в алфавіті Σ;
3. $\{a\}$ – однолітерна множина – регулярна множина в алфавіті Σ;
4. Якщо P та Q – регулярні множини, то такими є наступні множини:

$P \cup Q$ (операція об’єднання);

P * Q (операція конкатенації);

{P} = {ε} ∪ P ∪ P*P ∪ ……. (операція ітерації).

5. Ніякі інші множини, окрім побудованих на основі правил 1-4 не є регулярними множинами. Таким чином, регулярні множини можна побудувати з базових елементів правил 1-3 шляхом скінченого застосування операцій об'єднання, конкатенації та ітерації.

Регулярні вирази позначають регулярні множини таким чином, що:

1. 0 позначає регулярну множину ∅;

2. ε позначає регулярну множину {ε};

3. a позначає регулярну множину {a};

4. Якщо p та q позначають відповідно регулярні множини P та Q, то p + q позначає регулярну множину P ∪ Q;ир . p позначає регулярну множину P * Q; p* позначає регулярну множину {P}.

5. Ніякі інші вирази, окрім побудованих на основі правил 1-4 не є регулярними виразами.

Формальне визначення класу лексем можна виконати одним з вже описаних раніше способів:

- За допомогою праволінійних граматик;
- За допомогою скінчених автоматів;
- За допомогою регулярних виразів;
- Перерахувати лексеми даного класу як скінчену множину елементів.

Перші три способи визначення класів лексем за своєю потужністю еквівалентні. Якщо деякий клас лексем мови програмування скінчена множина, то одним з тривіальних способів визначення лексем цього класу є їх перерахування. Наприклад, клас punctuation marks можна визначити як скінчену множину P0 = { «.» , «?» , «!» , «:" , ….}

Сформулюємо фундаментальне твердження теорії граматик та автоматів: класи мови, які розпізнаються скінченими автоматами, збігаються з класами мови, які були визначені праволінійними граматиками та регулярними виразами

та навпаки. Відмітимо, що аналіз досвіду використання перерахованих засобів визначення класів лексем показує, що скінчені автомати знайшли широке використання при розробці лексичних аналізаторів для конкретних мов програмування, а регулярні вирази та праволінійні граматики широко використовуються в системах автоматизації побудови мовних процесорів як засоби високого рівня денотативності опису класів лексем.

Програму, яка розпізнає множину лексем, можна реалізувати двома способами:

- перший спосіб: створимо програмний модуль, роботою котрого керує таблиця управління скінченого автомата M(qi, aj), яка визначається в програмі явно;
- другий спосіб: розробимо програмний модуль з управлінням за номером поточного стану скінченого автомата та поточною вхідною літерою.

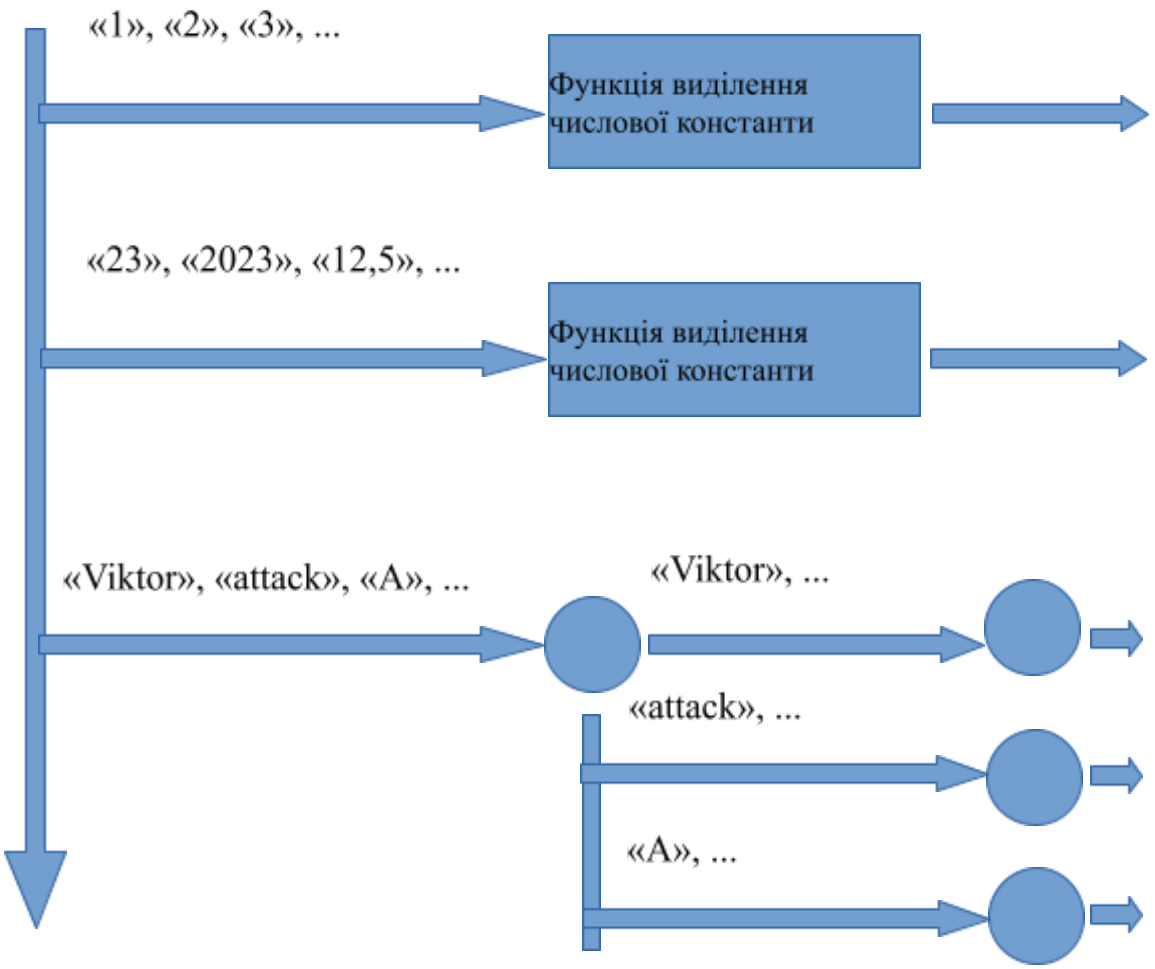
Спочатку множину лексем розіб'ємо на наступні класи:

- літерні константи («Victor», «attack», «A», ...)
- цілочислові константи («1», «2», «3», ...)
- дійсні константи («23», «2023», «12,5», ...)
- роздільники лексем (« », «.», «,», ...)

Побудуємо діаграму переходів об'єднаного скінченого автомата для лексичного аналізатора.(рис.2.10)

Після виділення нової лексеми програмний модуль переходить у початковий стан. Перед звертанням до функції, яка виділяє лексему з вхідного тексту, необхідно виконати функцію ungetch(fp). Окрім виділення лексем з вхідного тексту програми лексичний аналізатор повинен виконувати функцію локалізації лексичних помилок. Функція локалізації лексичної помилки повинна "пропустити" фрагмент тексту вхідної програми мінімальної довжини так, щоб подальша робота була продовжена. Очевидно, що таку умову задовольняють літери, по яких лексичний аналізатор переходить з початкового стану в інші. З діаграми переходів видно, що це – літери з множини { « », «.», «,», ...}.

« », «.», «,», ...



.....

.....

.....

Рисунок 2.10 - діаграма переходів об'єднаного скінченого автомата для лексичного аналізатора

3. ПРОЕКТНІ РІШЕННЯ. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис проектних рішень з програмного забезпечення

JavaScript (JS) — це полегшена інтерпретована мова програмування з першокласними функціями. Хоча він найбільш відомий як мова сценаріїв для веб-сторінок, багато середовищ без браузерів також використовують його, наприклад Node.js, Apache CouchDB і Adobe Acrobat. JavaScript — це однопотокова, динамічна мова на основі прототипу, яка підтримує об’єктно-орієнтований, імперативний і декларативний стилі (наприклад, функціональне програмування). Динамічні можливості JavaScript включають конструкцію об’єктів під час виконання, списки змінних параметрів, змінні функцій, динамічне створення сценаріїв, самоаналіз об’єктів і відновлення вихідного коду. Стандартами для JavaScript є специфікація мови ECMAScript (ECMA-262) і специфікація API інтернаціоналізації ECMAScript (ECMA-402).

Node.js — це міжплатформене середовище виконання JavaScript, яке дозволяє розробникам створювати серверні та мережеві програми за допомогою JavaScript[11]. Менеджер пакетів вузлів (англійською Node Packet Manager, або npm) — це менеджер пакетів, який завантажується та додається разом із Node.js. Його клієнт командного рядка (англійською CLI) можна використовувати для завантаження, налаштування та створення пакетів для використання в проектах Node.js[12]. Завантажені пакунки можна імпортувати за допомогою ES imports і CommonJS require() без включення каталогу залежностей node_modules/, до якого вони завантажуються, оскільки Node.js розпізнає пакти без відносного чи абсолютного шляху, указаного під час імпорту. Node.js запускає двигун V8 JavaScript та ядро Google Chrome, поза браузером. Це дозволяє Node.js бути дуже продуктивним. Програма Node.js працює в одному процесі без створення нового потоку для кожного запиту.

Node.js надає набір асинхронних примітивів вводу-виводу у своїй стандартній бібліотеці, які запобігають блокуванню коду JavaScript, і загалом бібліотеки в

Node.js написані з використанням неблокуючих парадигм, що робить поведінку блокування скоріше винятком, ніж нормою. Коли Node.js виконує операцію введення-виведення, як-от читання з мережі, доступ до бази даних або файлової системи, замість блокування потоку та витрачання циклів процесору на очікування, Node.js відновить операції, коли повернеться відповідь. Це дозволяє Node.js обробляти тисячі одночасних з'єднань з одним сервером, не вносячи тягаря керування паралельністю потоків, що може бути значним джерелом помилок.

Node.js надає унікальну перевагу, оскільки розробники інтерфейсів, які пишуть JavaScript для браузера, тепер можуть писати код на стороні сервера на додаток до коду на стороні клієнта без необхідності вивчати абсолютно іншу мову[13]. У Node.js нові стандарти ECMAScript можна використовувати без проблем, оскільки не потрібно чекати, поки всі користувачі оновлять свої веб-переглядачі – розробник сам вирішує, яку версію ECMAScript використовувати, змінивши версію Node.js. Також можна ввімкнути певні експериментальні функції, запустивши Node.js із прапорцями.

Як асинхронне середовище виконання JavaScript, кероване подіями, Node.js розроблено для створення масштабованих мережевих програм. Це на відміну від сьогоднішньої більш поширеної моделі паралелізму, основаному на потоках. Мережа на основі потоків відносно неефективна і дуже складна у використанні. Крім того, користувачі Node.js не хвилюються про блокування процесу, оскільки блокувань немає. Майже жодна функція в Node.js безпосередньо не виконує введення-виведення, тому процес ніколи не блокується, за винятком випадків, коли введення-виведення виконується за допомогою синхронних методів стандартної бібліотеки Node.js. Оскільки ніщо не блокується, масштабовані системи дуже легко розробляти на Node.js.

Node.js подібний за дизайном до таких систем, як Ruby's Event Machine і Python's Twisted. Node.js розвиває модель подій трохи далі. Він представляє цикл подій як конструкцію часу виконання, а не як бібліотеку[14]. В інших

системах завжди існує блокуючий виклик для запуску циклу подій. Як правило, поведінка визначається через зворотні виклики на початку сценарію, а в кінці сервер запускається через блокуючий виклик, наприклад EventMachine::run(). У Node.js немає такого виклику запуску циклу подій. Node.js просто входить у цикл подій після виконання вхідного сценарію. Node.js виходить із циклу подій, коли більше немає зворотних викликів для виконання. Така поведінка схожа на JavaScript браузера — цикл подій прихований від користувача.

HTTP є найважливішою частиною у Node.js, розробленому з урахуванням потокового передавання та низької затримки. Це робить Node.js дуже придатним для створення веб-бібліотеки або фреймворку. Оскільки Node.js розроблено без потоків, це означає, що можна скористатися перевагами кількох ядер у своєму середовищі. Дочірні процеси можуть бути породжені за допомогою API child_process.fork(). На цьому ж інтерфейсі побудовано кластерний модуль, який дозволяє спільно використовувати сокети між процесами, щоб забезпечити балансування навантаження на свої ядра.

Для нашої програми ми також використовуємо бібліотеку GramJS. **GramJS** - це клієнт Telegram[15], написаний на JavaScript для Node.js і браузерів. Його ядро базується на Telethon, а Telethon - це бібліотека для розробки програм для телеграму на Python.

PostgreSQL — це потужна об’єктно-реляційна база даних з відкритим вихідним кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, які безпечно зберігають і масштабують найскладніші дані[16]. Ідея PostgreSQL вперше з’явилась аж в 1986 року в рамках проекту POSTGRES в Каліфорнійському університеті в Берклі та має понад 35 років активного розвитку на основній платформі. PostgreSQL заслужив міцну репутацію завдяки своїй перевіреній архітектурі, надійності, цілісності даних, надійному набору функцій, розширюваності та відданості спільноти з відкритим кодом, яка стоїть за програмним забезпеченням, для постійного надання продуктивних та інноваційних рішень. PostgreSQL працює на всіх

основних операційних системах, сумісний з ACID з 2001 року та має потужні додаткові модулі, такі як популярний розширювач геопросторових баз даних PostGIS[17]. Не дивно, що PostgreSQL стала реляційною базою даних з відкритим вихідним кодом для багатьох людей і організацій.

PostgreSQL має багато функцій, які допомагають розробникам створювати додатки, адміністраторам — захищати цілісність даних і створювати відмовостійке середовище, а також допомагають керувати своїми даними, незалежно від того, наскільки великий чи малий набір даних. Окрім того, що PostgreSQL є безкоштовним і відкритим вихідним кодом, він дуже розширюваний[18]. Наприклад, можна визначати власні типи даних, створювати власні функції, навіть писати код з різних мов програмування без перекомпіляції бази даних. PostgreSQL намагається відповідати стандарту SQL, якщо така відповідність не суперечить традиційним функціям або може призвести до невдалих архітектурних рішень. Багато функцій, необхідних стандарту SQL, підтримуються, хоча іноді з дещо відмінним синтаксисом або функціями. Починаючи з випуску версії 16 у вересні 2023 року, PostgreSQL відповідає принаймні 170 із 179 обов’язкових функцій для відповідності SQL:2023 Core[19].

Чому саме PostgreSQL? PostgreSQL надає різноманітні функції для полегшення роботи з базами даних. Ось невичерпний список функцій PostgreSQL:

- Можливість утримувати такі типи даних як: integer, numeric, string, boolean, JSON/JSONB, XML, точка, коло, композитний, і т.д.;
- Утримує цілісність даних за допомогою таких параметрів: UNIQUE, NOT NULL, Первинні ключі , Іноземні ключі , Обмеження виключення , Явні блокування, Рекомендаційні блокування;
- Забезпечує одночасність та продуктивність за допомогою: індексування, складного планувальника, оптимізатора запитів, сканування лише індексу, статистики з кількома стовпцями , транзакцій, усіх рівнів ізоляції

транзакцій, визначені в стандарті SQL, компіляції виразів «точно вчасно» (англійською “just in time” або JIT);

- Забезпечує надійність та можливість аварійного відновлення за допомогою: журналування з випереджальним записом (англійською Write-ahead logging або WAL) , реплікації, відновлення на момент часу (аншлійською Point-in-time-recovery або PITR), активних режими очікування , табличних просторів;
- Гарантує безпеку за допомогою: надійної системи контролю доступу , безпеки на рівні стовпців і рядків , багатофакторної аутентифікації з сертифікатами та додатковим методом;
- Надає такі можливості розширення: збережені функції та процедури , підтримує процедурні мови (наприклад PL/pgSQL), інші мови, доступні через розширення (наприклад Java), конструктори SQL/JSON і вирази шляху, іноземні оболонки даних, настроюваний інтерфейс зберігання для таблиць;
- Надає такі можливості для інтернаціоналізації, текстового пошуку: підтримка міжнародних наборів символів, наприклад через порівняння ICU, сортування без урахування регістру та наголосу , повнотекстовий пошук.

Крім того, PostgreSQL дуже розширюваний: багато функцій, наприклад індекси, мають визначені API, щоб їх можна було створювати за допомогою PostgreSQL задля вирішення різних проблем. За десятиліття використання було доведено, що PostgreSQL має високу масштабованість щодо величезної кількості даних, якими можна керувати, так і щодо найменших одночасно працюючих користувачів. У виробничих середовищах є активні кластери PostgreSQL, які керують багатьма терабайтами даних, і спеціалізовані системи, які керують петабайтами.

TypeORM — це ORM(Object-relational mapping tool), який може працювати на платформах NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo та Electron, а також використовувати з TypeScript і JavaScript. Його мета полягає в тому, щоб завжди підтримувати найновіші функції JavaScript і надавати додаткові функції, які допомагають розробляти будь-які програми, які використовують бази даних - від невеликих програм з кількома таблицями до великих корпоративних програм з кількома базами даних. TypeORM підтримує як шаблони Active Record, так і Data Mapper, на відміну від усіх інших ORM JavaScript, які існують на даний момент. Це означає, що з його допомогою можна писати високоякісні, слабко пов’язані, масштабовані та підтримувані програми найбільш продуктивним способом. На TypeORM сильно впливають інші ORM, такі як Hibernate, Doctrine та Entity Framework. Особливості TypeORM:

- Підтримує як DataMapper, так і ActiveRecord.
- Сутності та стовпці.
- Специфічні для бази даних типи стовпців.
- Менеджер сутності.
- Репозиторії та спеціальні сховища.
- Чиста об'єктно-реляційна модель.
- Підтримує кілька шаблонів успадкування.
- Міграції та автоматична генерація міграцій.
- Об'єднання з'єднань. тиражування.
- Використання кількох екземплярів бази даних.
- Робота з декількома типами баз даних.
- Запити між базами даних і схемами.
- Елегантний синтаксис, гнучкий і потужний QueryBuilder.
- Ліве і внутрішнє з'єднання.
- Правильна розбивка сторінок для запитів із використанням об’єднань.

3.2 Розробка компонентів програмного забезпечення

3.2.1 База даних та підготовка вхідних даних

База даних буде створена за допомогою PostgreSQL, та розроблювана програма буде під’єднуватися та працювати з нею за допомогою TypeORM. В базі даних буде три таблиці: channel, ipso and post. Таблиця channel складається з id, імені каналу, коментаря з описом каналу та базовою бібліотекою слів пов’язаною з каналом. Таблиця ipso складається з id каналу, id постів та нові лексичні токени, які є елементами IIPCO. Таблиця post включає в себе id каналу, текст посту та лексичні токени отримані після деконструкції посту. Таблиця channel буде вже заповненою, оскільки вважаємо, що нам вже відомі назви каналів та бібліотека слів, які там зазвичай використовуються. Таблиці post та ipso пусті, та будуть заповнюватися під час перебігу програми.

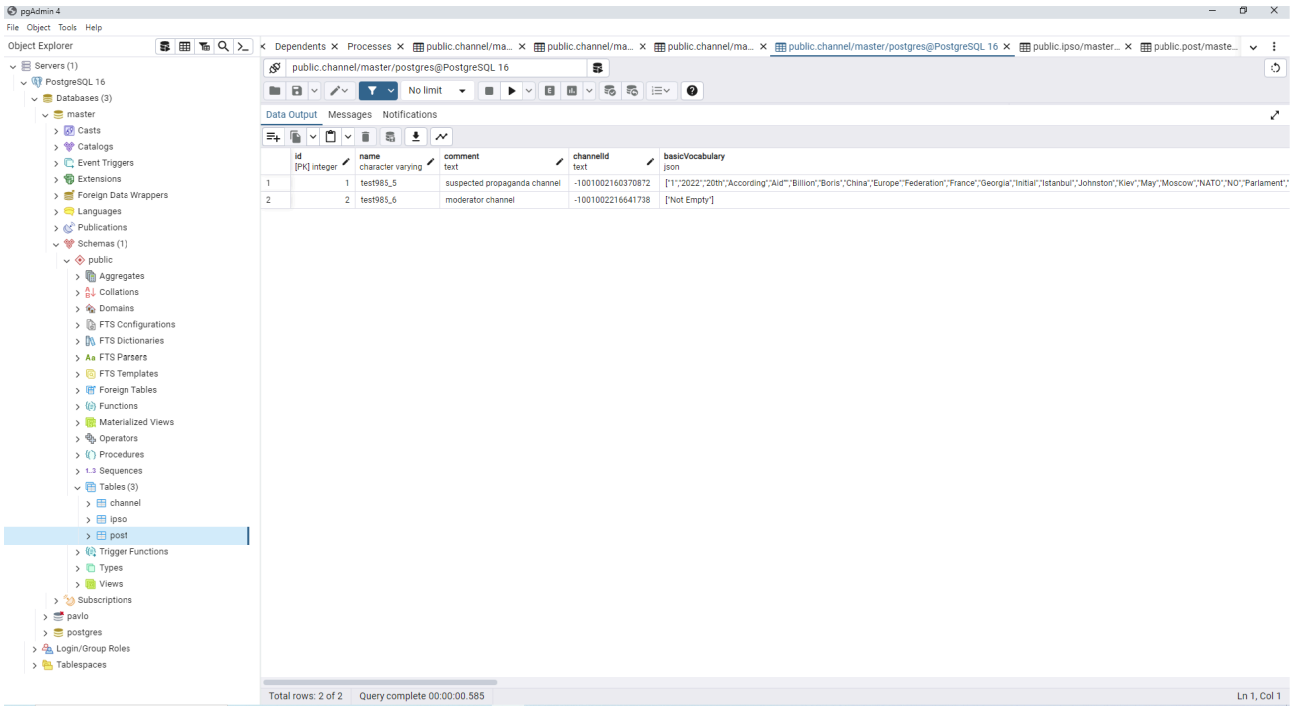


Рисунок 3.1 — Приклад таблиці з інформацією про канали в базі даних

Розроблювана програма працює з телеграм каналами, тому підготовка вхідних даних включає в себе наповнення тестового телеграм каналу повідомленнями.

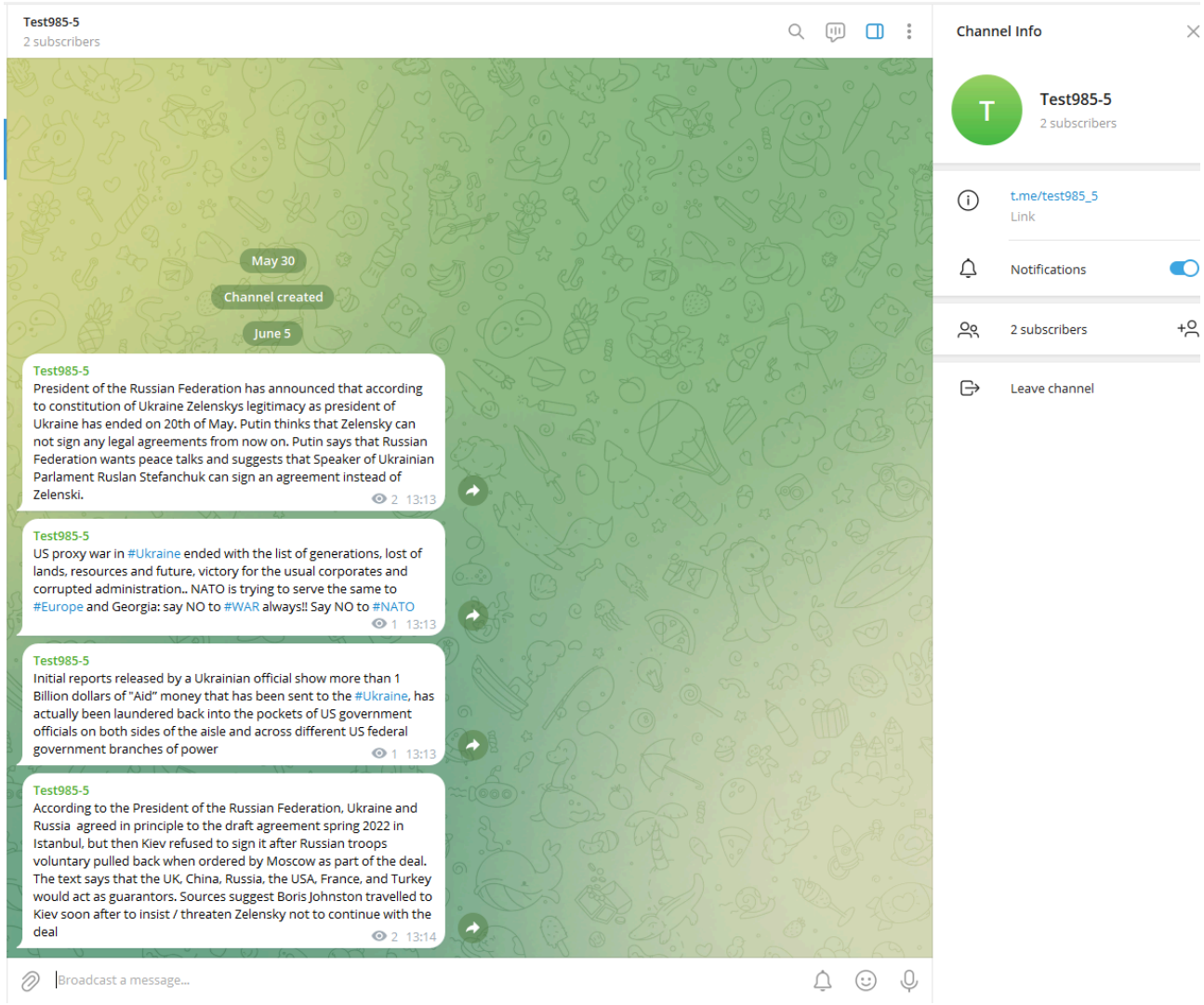


Рисунок 3.2 — Приклад тестового каналу з повідомленнями, які підготовлені до видобування та аналізу

3.2.2 Опис класів

Розроблена програма має три класи:

- dbConnection, що відповідає за підключення до бази даних Postgresql та різні маніпуляції з нею за допомогою TypeORM;
- telegramChannel, що відповідає за підключення до телеграму, видобутку з нього постів для аналізу та надсилання повідомлень про підозрювані ІПСО спеціалістам, за допомогою бібліотеки Gramjs;

- Lexical, що відповідає саме за лексичний аналіз та прийняття рішення щодо виявлення нових елементів ІПСО.

Database має такі методи:

Назва	Параметри	Опис
initConnection		Ініціює з’єднання з базою даних Postgresql
channelInfo	channelName, channelComment, channelId, channelVocabulary	Додає інформацію про канал в таблицю каналів
writePost	postText, deconstructedPost	Додає інформацію про пост в таблицю постів
getChannel	channelComment	Повертає назву каналу з таблиці каналів за допомогою коментарю
getChannelVocabulary	channelComment	Повертаю бібліотеку слів каналу з таблиці каналів за допомогою коментарю
writeIpso	newWords	Додає інформацію про нові елементи ІПСО в таблицю ІПСО
clearDatabase		Очищує всі бази даних для підготовки тесту

Таблиця 3.1 Методи класу Database

Database має такі параметри:

Назва	Тип	Опис

channelName	string	Назва каналу
channelComment	string	Коментарій про канал
channelId	string	Id телеграм каналу
channelVocabulary	JSON	Бібліотека слів каналу
postText	string	Текст посту
deconstructedPost	JSON	Текст посту розкладений на лексичні токени
newWords	JSON	Результат лексичного аналізу

Таблиця 3.2 Параметри класу **Database**

TelegramConnection має такі методи:

Назва	Параметри	Опис
initConnection		Ініціалізує під’єднання до телеграм акаунту
getPost	ChannelName, messageId	Повертає текст постів за допомогою імені каналу та id постів в каналі
clear		Зачищає всі канали для підготовки тесту
sendPost	post, channelName	Посилає повідомлення текст якого зазначено в канал, чиє ім’я зазначено
getMessageNumber	channelName	Повертає array id постів

Таблиця 3.3 Методи класу TelegramConnection

TelegramConnection має такі параметри:

Назва	Тип	Опис
channelName	string	Назва телеграм каналу
messageId	array	Id постів в каналі
post	string	Текст посту

Таблиця 3.4 Параметри класу TelegramConnection

Lexical має такі методи:

Назва	Параметри	Опис
initBasicVocabulary	lexArray	Додає масив слів до бібліотеки
includeWord	word	Перевіряє чи є слово в масиві
addToBasicVocabulary	word	Додає слово до бібліотеки
compareLexArray	lexArray	Перевіряє масив і повертає масив слів, яких немає в бібліотеці
Deconstruct	text	Розбирає текст на лексичні токени

Таблиця 3.3 Методи класу Lexical

Lexical має такі параметри: lexArray — це деякий масив слів, word – це деяке слово, text – це деякий текст.

Діаграма класів системи, що розроблюється подана на рис. 3.3.

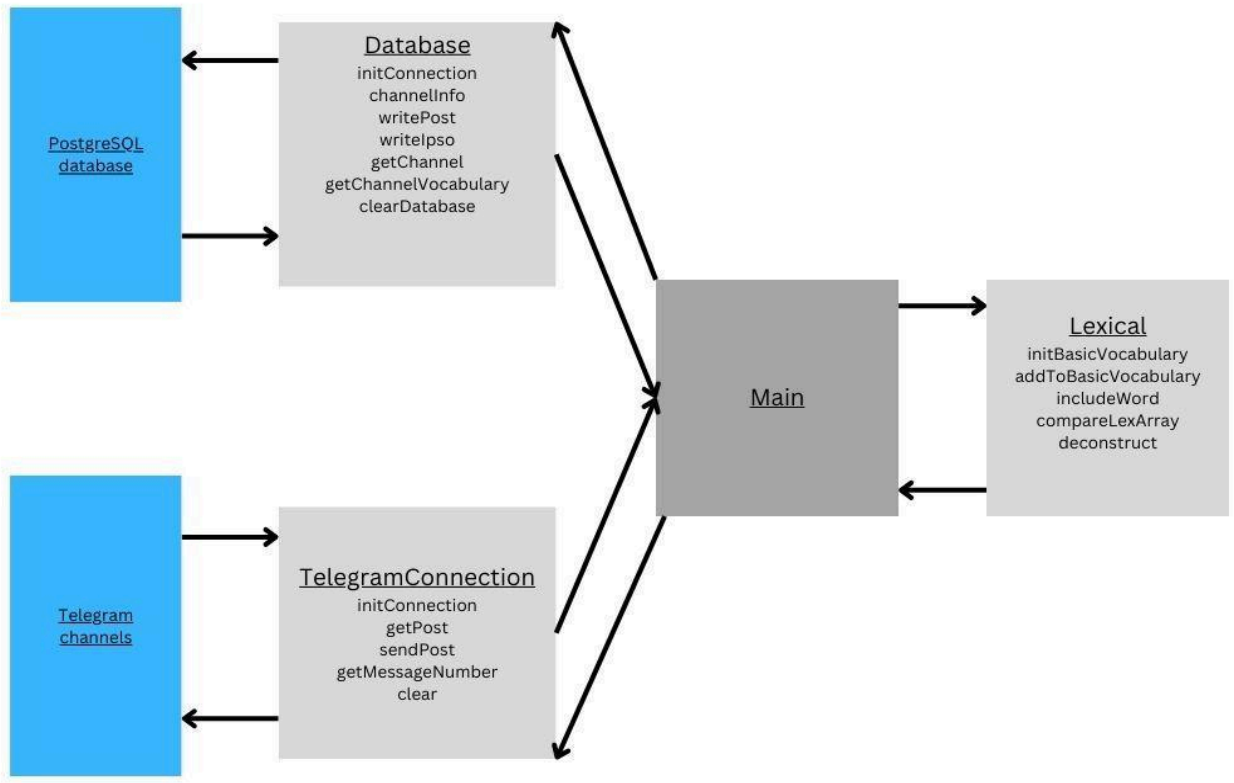


Рисунок 3.3. Діаграма класів системи

Завдяки такій організації класів дуже зручно працювати і з телеграм каналами, і з базою даних з будь-якої частини користувацького застосування. В даній роботі вся робота з готовою програмою проводиться у консольному застосуванні. Тестовий приклад наведено у наступному підрозділі.

3.3 Опис тестового прикладу

З самого початку програма підключається до бази даних та телеграм акаунту. Після цього вона бере інформацію про підозрювані канали з бази даних, та підключається до них. В нашому випадку програма прочитає повідомлення в підозрюваному каналі і після лексичного аналізу покладе результат у базу даних. Якщо лексичний аналіз виявить нові елементи ІПСО, то програма покладе інформацію про них в таблицю ІПСО та надішле повідомлення в окремий телеграм канал.

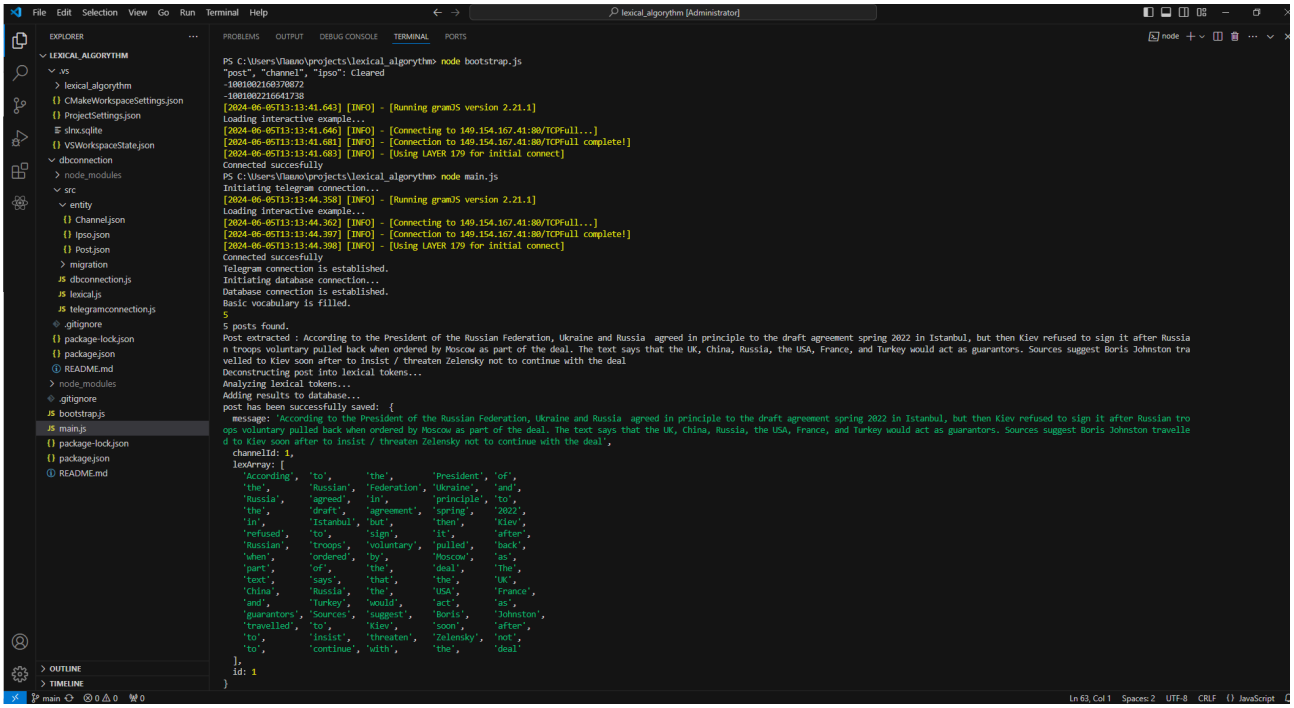


Рисунок 3.8 Тестовий приклад (частина 1)

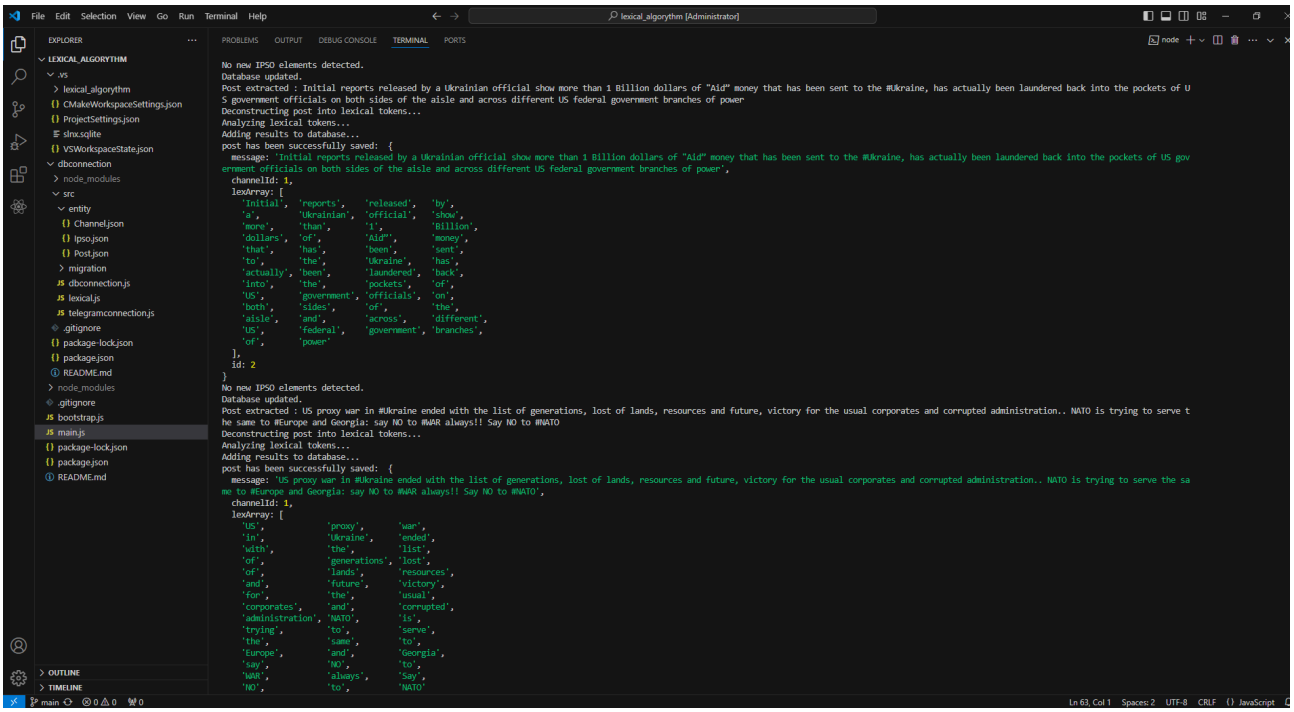


Рисунок 3.9 Тестовий приклад (частина 2)

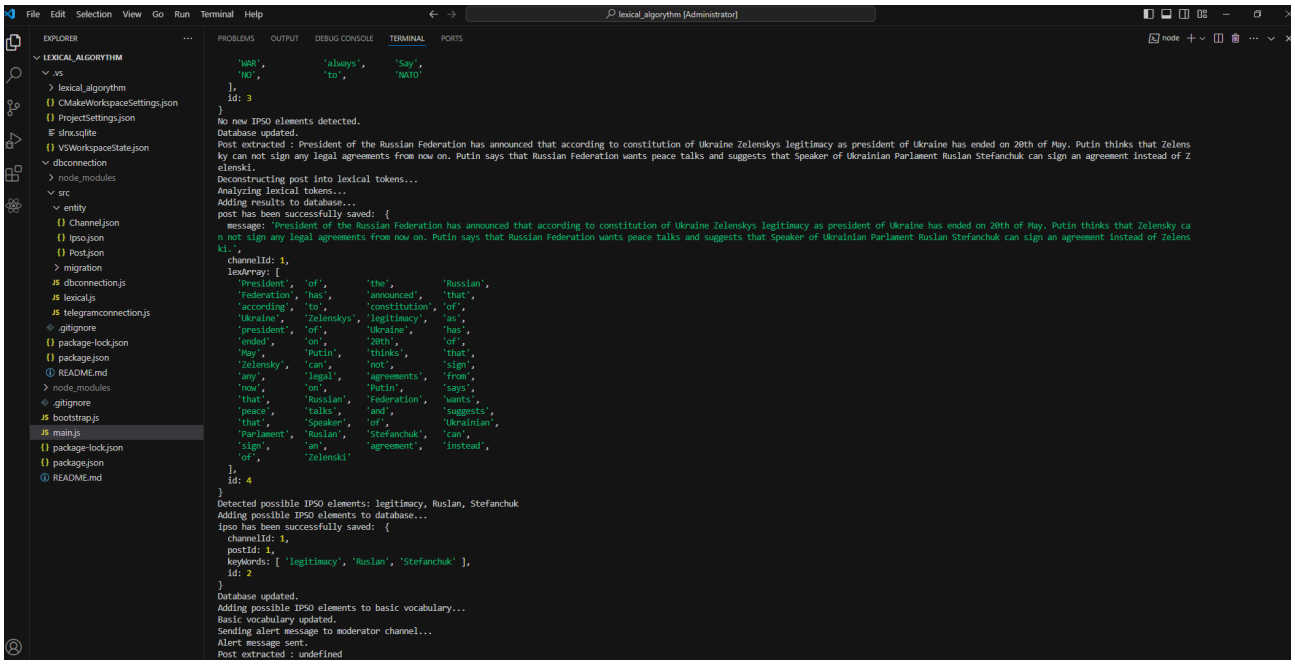


Рисунок 3.10 Тестовий приклад (частина 3)

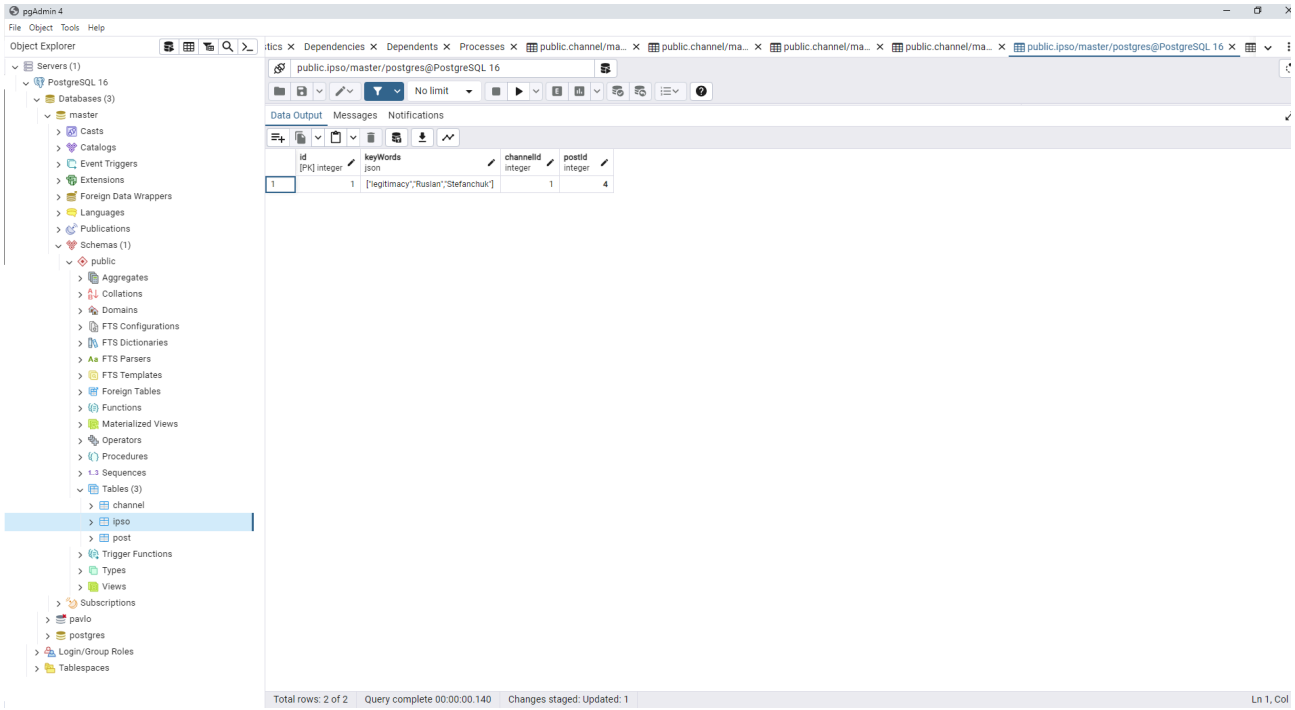


Рисунок 3.4 Приклад таблиці ІПСО

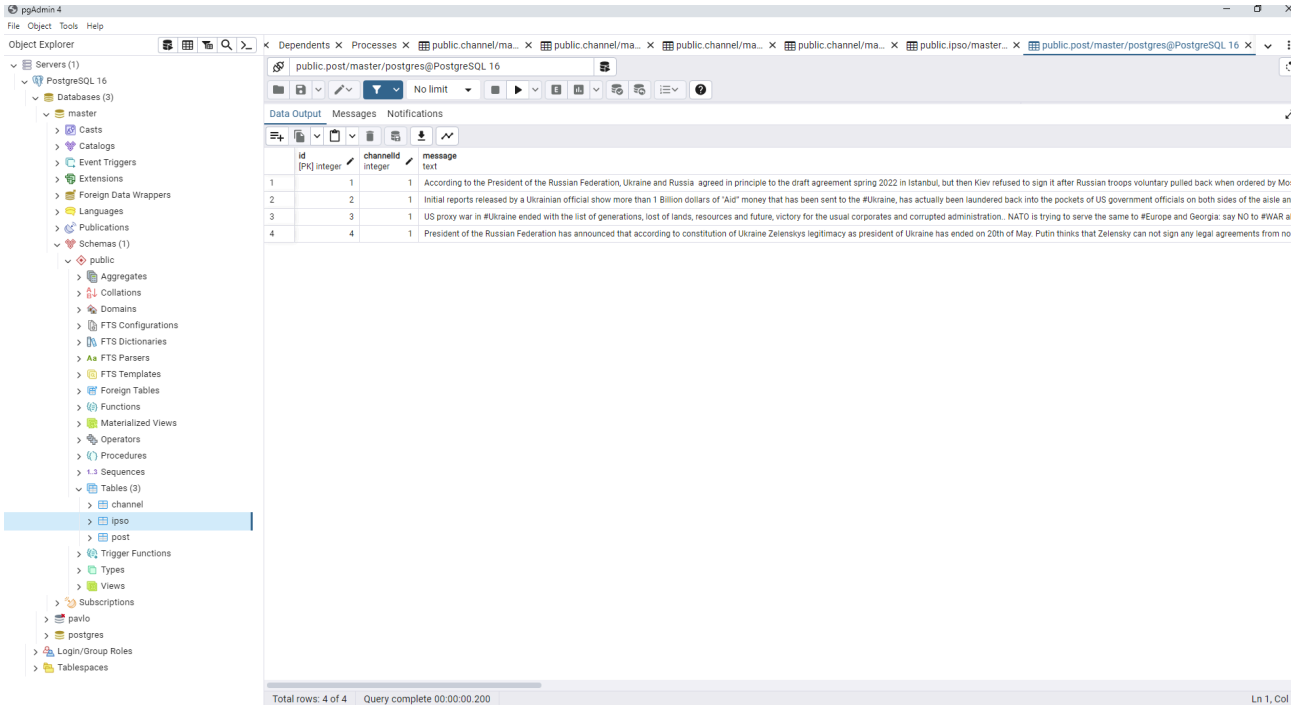


Рисунок 3.6 Приклад таблиці постів

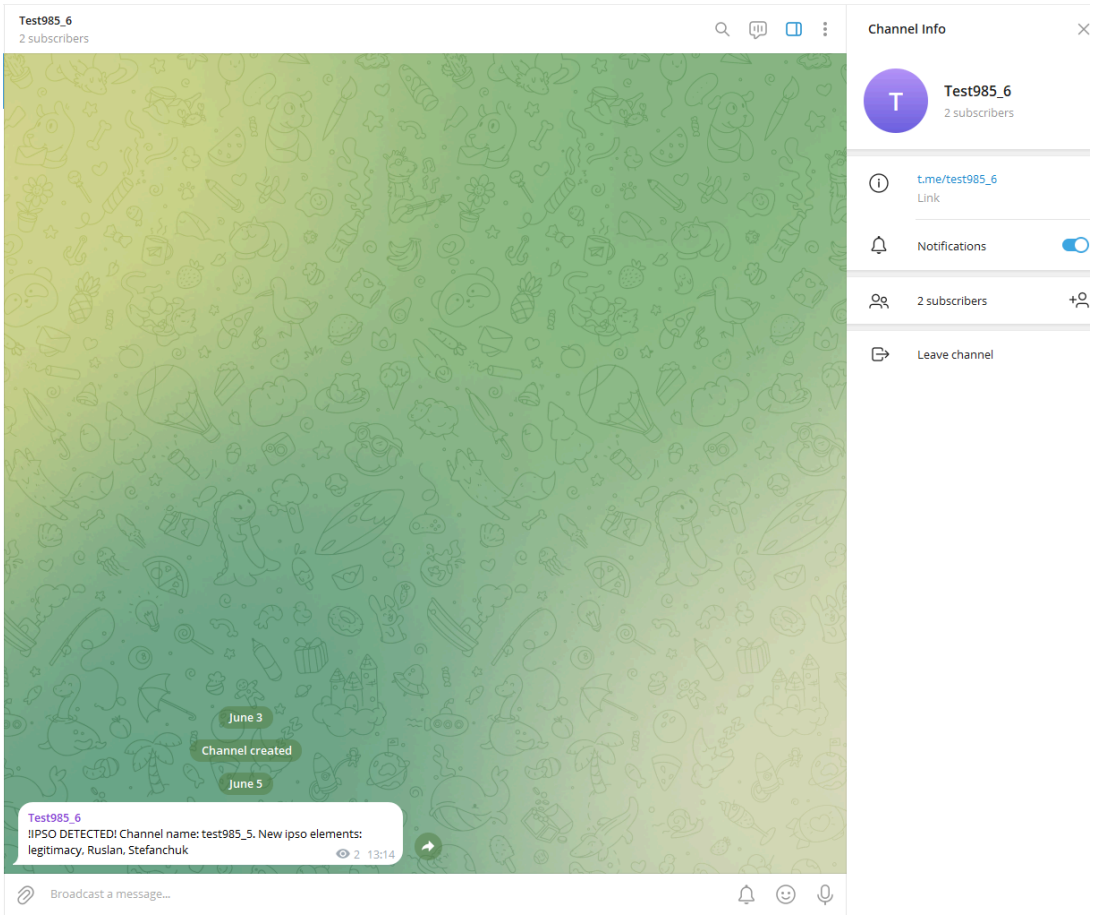


Рисунок 3.7 Приклад відправленого повідомлення про виявлення нових елементів ІПСО.

На рисунках можна побачити, що програма видобуває повідомлення із каналу, розбирає його на лексичні токени, аналізує їх та кладе отриману інформацію в таблиці в базі даних.

В таблицях пости мають параметр channelId що відповідає id каналу, з якого взятий пост в таблиці каналів. А елементи ІПСО мають також параметр postId, що відповідає значенню id поста в якому лексичний аналіз виявив нові елементи ІПСО. При відравленні повідомлення програма також вказує назву каналу, в якому були помічені нові елементи ІПСО.

4.ЕРГОНОМІКА ІНТЕРФЕЙСУ КОРИСТУВАЧА

4.1 Вимоги до програмного забезпечення та основні підходи до його проектування з погляду користувача.

Сьогодні технічне оснащення сучасних робочих місць потребує суцільного освоєння комп'ютерних програм як основних інструментів праці. Взаємодія з комп'ютером стає невід'ємною частиною їх роботи. У зв'язку з цим тема ергономіки в проектуванні програмного забезпечення і в, зокрема, інтерфейсів є актуальною і затребуваною. З точки зору ергономіки, цілі, які потрібно досягти в результаті проектування і розробки програмних продуктів наступні:

- Ефективність: за допомогою цього програмного продукту користувач може вирішити завдання і впоратися з проблемами, які перед ним стоять, з мінімальним вкладенням ресурсів.
- Продуктивність, результативність: за найменший можливий час досягати найбільшого можливого результату.
- Задоволеність працею користувача: відсутність негативних емоцій від використання програмного забезпечення.

Досягнення зазначених цілей значною мірою залежить від якості запланованої архітектури програмного забезпечення. Під час розробки архітектури програмного забезпечення виконується його модульно-ієрархічна побудова. Модуль – це замкнута програма, яку можна викликати з будь-якого іншого модуля в програмі і можна компілювати окремо від усіх інших модулів[20]. Переваги створення модульних програм пояснюються наступними факторами:

- програмні модулі, як правило, вирішують одне функціональне завдання, використовують на вході і на виході невелику кількість даних. Внутрішні змінні модуля не пов'язані з внутрішніми змінними інших модулів. Тому окремі модулі можуть створюватися і регламентуватиме різними розробниками незалежно один від одного;

- модульні програми легко підтримувати, читати і вдосконалювати. Виправлення окремого модуля викликає мінімальні зміни в інших модулях, пов'язаних з ним по управлінню та інформації;
- модульні програми мають підвищену певність та швидкість розробки, так як існує можливість розподілу робіт по створенню модулів різної складності і важливості між програмістами різної кваліфікації;
- можна створювати бібліотеки найбільш уживаних підпрограм, які потім можна використовувати в якості комплектуючих частин при розробці інших додатків;
- процедура завантаження всієї програми в оперативну пам'ять спрощується при використанні оверлейного методу;
- виникає багато природних контрольних точок для спостереження за просуванням проекту з управління і за інформацією.

Для розробки архітектури застосовується атрибутний метод проектування (Attribute-Driven Design, ADD) – це методика визначення програмної архітектури, в якій процес декомпозиції ґрунтується на передбачуваних атрибутах якості продукту. Це рекурсивний процес декомпозиції, на кожному з етапів якого відбувається відбір тактик і архітектурних зразків, які відповідають тим чи іншим сценаріями якості, а також розподіл функціональності, спрямоване на конкретизацію типів модулів даного зразка. Результатом застосування ADD є перші кілька рівнів подання декомпозиції модулів архітектури, а також всі інші пов'язані з ними уявлення. Не варто, втім, думати, що після ADD[21] стають відомі всі деталі уявлень, - система описується як набір контейнерів функціональності і існуючих між ними взаємозв'язків. Етапи ADD:

1. Вибір модуля для декомпозиції. Як правило, в якості вихідного модуля береться система в цілому. Всі необхідні вхідні дані (обмеження, функціональні вимоги і вимоги до якості) для нього повинні бути відомі.

2. Уточнення модуля в кілька етапів:

а) вибір архітектурних мотивів з набору конкретних сценаріїв реалізації якості і функціональних вимог. На цьому етапі визначаються найбільш важливі в контексті проведення декомпозиції моменти;

б) вибір архітектурного зразка, відповідного архітектурним мотивів. Створення (або вибір) зразка, тактики якого дозволяють реалізувати ці мотиви. Виявлення дочірніх модулів, необхідних для реалізації цих тактик;

в) конкретизація модулів, розподіл функціональності з елементів Use Case, складання кількох вистав;

г) визначення інтерфейсів дочірніх модулів. Декомпозиція має своїм результатом нові модулі, а також обмеження на типи взаємодії між ними. Для кожного з модулів ці відомості слід зафіксувати в документації по інтерфейсу;

д) перевірка та уточнення елементів Use Case в сценаріях якості і накладення, виходячи з них, обмежень на дочірні вузли. На цьому етапі ми перевіряємо, чи всі фактори врахували, а також, в розрахунку на подальшу декомпозицію або реалізацію, готуємо дочірні модулі.

3. Перераховані вище етапи слід виконати щодо всього, що потребує подальшої декомпозиції модулів. Для оцінки розробленої архітектури застосовують метод аналізу компромісних архітектурних рішень (Architecture Tradeoff Analysis Method, ATAM) – комплексна універсальна методика оцінки програмної архітектури. Відповідно до назви цей метод виявляє ступінь реалізації в архітектурі тих чи інших завдань за якістю, а також (виходячи з припущення про те, що будь-яке архітектурне рішення впливає відразу на кілька завдань за якістю) механізм їх взаємодії - іншими словами, їх взаємозамінність. Метод аналізу вартості та ефективності (Cost Benefit Analysis Method, CBAM) базується на методі ATAM і забезпечує моделювання витрат і вигод, пов'язаних з прийняттям архітектурно-проектних рішень, і сприяє їх оптимізації. Методом CBAM оцінюються технологічні та економічні фактори, а також самі архітектурні рішення.

Архітектура програмного забезпечення (англ. Software architecture, ПЗ) – це структура програми або обчислювальної системи, яка включає програмні компоненти, видимі зовні властивості цих компонентів, а також відносини між ними. Цей термін стосується також документування архітектури програмного забезпечення. Документування архітектури ПЗ спрощує процес комунікації між зацікавленими особами (англ. Stakeholders), дозволяє зафіксувати прийняті на ранніх етапах проектування рішення про високо рівневі дизайні системи і дозволяє використовувати компоненти цього дизайну і шаблони повторно в інших проектах. Архітектура ПЗ є реалізацією не функціональних вимог до системи, в той час як проектування ПЗ є реалізацією функціональних вимог.

Архітектурний вигляд складається з 2 компонентів:

• елементи;

• відносини між елементами.

Структурні компоненти архітектури програмного комплексу та зв’язки між ними значною мірою визначаються характером задач для яких він призначений та вимогами майбутнього користувача або замовника.

Архітектурні види можна поділити на 3 основні типи:

1. Модульні види (англ. Module views) – показують систему як структуру з різних програмних блоків.

2. Компоненти-і-конектори (англ. Component-and-connector views) - показують систему як структуру з паралельно запущених елементів (компонентів) і способів їх взаємодії (конекторів).

3. Розміщення (англ. Allocation views) – показує розміщення елементів системи в зовнішніх середовищах.

Приклади модульних видів :

• Декомпозиція (англ. Decomposition view) – складається з модулів в контексті відношення «є підмодулем».

• Використання (англ. Uses view) – складається з модулів в контексті відношення «використовує» (тобто один модуль використовує сервіси іншого модуля)

• Вид рівнів (англ. Layered view) – показує структуру, в якій пов'язані за функціональністю модулі об'єднані в групи (рівні)

• Вид класів / узагальнень (англ. Class / generalization view) – складається з класів, які пов'язані через відношення «успадковується від» і «є екземпляром».

Приклади видів компонентів-і-конекторів:

• Процесний вид (англ. Process view) – складається з процесів, з'єднаних операціями комунікації, синхронізації і / або виключення.

• Паралельний вид (англ. Concurrency view) – складається з компонентів і конекторів, де конектори представляють собою «логічні потоки»

• Вид обміну даними (англ. Shared-data (repository) view) – складається з компонентів і конекторів, які створюють, зберігають і отримують постійні дані.

• Вид клієнт-сервер (англ. Client-server view) – складається з взаємодіючих клієнтів і серверів, а також конекторів між ними (наприклад, протоколів і спільних повідомлень)

Приклади видів розміщення:

• Розгортання (англ. Deployment view) – складається з програмних елементів, їх розміщення на фізичних носіях і комунікаційних елементів

• Впровадження (англ. Implementation view) – складається з програмних елементів і їх відповідності файловим структурам в різних середовищах (розробницького, інтеграційної та т.д.)

• Розподіл роботи (англ. Work assignment view) – складається із модулів і опису того, хто відповідальний за впровадження кожного зних.

Хоча було розроблено кілька мов для опису архітектури програмного забезпечення, в даний момент немає згоди з приводу того, який набір видів повинен бути прийнятий в якості еталону. Як стандарт «для моделювання програмних систем (і не тільки)» була

створена мова UML де модульна структура представляється діаграмою пакетів.

Людино-машинний інтерфейс забезпечує зв'язок між користувачем і комп'ютером – він дозволяє досягати поставлених цілей, успішно знаходити рішення поставленого завдання. Взаємодія – обмін діями і реакціями на ці дії між комп'ютером і користувачем. Мета створення ергономічного інтерфейсу полягає в тому, щоб відобразити інформацію настільки ефективно наскільки це можливо для людського сприйняття і структурувати відображення на дисплеї таким чином, щоб привернути увагу до найбільш важливих одиниць інформації. Основна ж мета полягає в тому, щоб мінімізувати загальну інформацію на екрані і представити тільки те, що є необхідним для користувача.

Існує ряд основних принципів побудови інтерфейсів:

• Принцип угруповання – згідно з цим правилом, екран програми повинен бути розбитий на ясно окреслені блоки елементів, може бути, навіть з заголовком для кожного блоку.

• Гаманець Міллера – ємність пам'яті обмежена сім'ю цифрами. Необхідне відповідне групувати суті в програмі (пункти меню, закладки, опції на цих закладках і т. П.) Бажано з урахуванням цього правила – тобто не більше семи в групі, в крайньому випадку – дев'яти.

• Бритва Оккама або KISS:

- будь-яке завдання має вирішуватися мінімальним числом дій;
- логіка цих дій повинна бути очевидною для користувача;
- руху курсору і навіть очей користувача повинні бути оптимізовані.

• Видимість відображає корисність – винести найважливішу інформацію і елементи управління на перший план і зробити їх доступними користувачеві, а менш важливу – перемістити, наприклад,

в меню. Для розробників та користувачів ефективний інтерфейс має свої переваги. Для розробників програмного забезпечення:

- покращене представлення продукту;
- підвищений попит на продукт;

- більш короткий час навчання;
- більш низькі вимоги до сервісу після продажу;
- тверда основа, на якій відбувається розвиток серійного виробництва продукту;
- зниження ризику помилкових дій і нещасних випадків;
- скорочення кількості документації.

Для користувачів:

- більш короткий час навчання;
- підвищена загальна застосовність навичок;
- зросла автономія при використанні системи;
- зниження кількості часу, необхідного для виконання завдання;
- зменшення кількості помилок;
- зростання задоволення від роботи.

Для розробки зручного інтерфейсу на етапі його проектування складається діаграма прецедентів – Use Case. Діаграма варіантів використання (сценаріїв поведінки, прецедентів) є вихідним концептуальним поданням системи в процесі її проектування і розробки. Дана діаграма складається з акторів, варіантів використання і відносин між ними. При побудові діаграми можуть використовуватися також загальні елементи нотації: примітки і механізми розширення.

Суть даної діаграми складається в наступному: проектована система представляється у вигляді безлічі акторів, що взаємодіють з системою за допомогою так званих варіантів використання. При цьому актором (дійовою особою) називається будь-який об'єкт, суб'єкт або система, що взаємодіє з моделюється системою ззовні. У свою чергу варіант використання – це специфікація сервісів (функцій), які система надає актору. Іншими словами, кожен варіант використання визначає деякий набір дій, що здійснюються системою при взаємодії з актором. При цьому в моделі ніяк не відбивається те,

яким чином буде реалізовано цей набір дій. У структурному підході аналогом діаграми варіантів використання є діаграми IDEF0 і DFD, варіантів використання - роботи (IDEF0) і процеси (DFD), акторів - зовнішні сутності (DFD).

4.2 Параметри, які необхідно враховувати при розробці інтерфейсу користувача.

4.2.1. Ергономічні цілі і показники якості програмного продукту

Програмний додаток розробляється для забезпечення роботи користувача, тобто для того щоб він за допомогою комп'ютерної програми швидше і якісніше вирішував свої виробничі завдання. З точки зору ергономіки, найважливіше в програмі – створити такий призначений для користувача інтерфейс, який зробить роботу ефективною і продуктивною, а також забезпечить задоволеність користувача від роботи з програмою. Ефективність роботи означає забезпечення точності, функціональної повноти і завершеності при виконанні виробничих завдань на робочому місці користувача. Створення ПІ має бути націлене на показники ефективності: точність роботи, функціональна повнота та завершеність роботи.

Послідовність дій і набір інструментальних засобів користувача в ПІ повинні бути підпорядковані технологічним процесам виконання виробничого завдання. Не треба боятися складності системи, треба уникати такого інтерфейсу, який не відповідає алгоритму вирішення користувальницьких задач.

4.2.2. Основні характеристики, що враховуються при розробці ПІ користувача.

Необхідно ретельно продумати і усвідомити сценарій взаємодії програми з користувачем, привівши його до оптимальної (щодо розглянутих показників) системі виконання завдань, і реалізувати ПІ відповідно до цієї системою. Для того, щоб розібратися в технології вирішення завдань користувача, розробнику необхідно дослідити діяльність користувача та з'ясувати пов'язані з цим моменти (наприклад Які типові операції використовує користувач при

вирішенні задачі? Чи Яка інформація необхідна користувачеві для вирішення завдання?).

Продуктивність роботи відображає обсяг витрачених ресурсів при виконанні завдання, як обчислювальних, так і психофізіологічних. Дизайн ПІ повинен забезпечувати мінімізацію зусиль користувача при виконанні роботи і приводити до: скорочення тривалості операцій, зменшення часу навігації і вибору команди, підвищенню загальної продуктивності користувача, збільшення тривалості стійкої роботи користувача і т.д.

Скорочення невиробничих витрат і зусиль користувача – важлива складова якості програмного забезпечення. Для оцінки продуктивності використовуються відповідні показники, що перевіряються фахівцями з ергономіки в процесі usability тестування робочого прототипу. Формування таких показників відбувається в процесі визначення вимог до ПІ (наприклад На які операції користувач витрачає найбільше часу?)

4.2.3. Вимоги до зручності і комфортності інтерфейсу

Задоволеність користувача від роботи тісно пов'язана з комфортністю його взаємодії з додатком, і сприяє збереженню професійних кадрів на підприємстві Замовника за рахунок привабливості роботи на даному робочому місці. Вимоги до зручності і комфортності інтерфейсу зростають зі збільшенням складності робіт і відповідальності користувача за кінцевий результат. Висока задоволеність від роботи досягається в разі:

- Прозорою для користувача навігації і цільової орієнтації в програмі.
- Ясності й чіткості розуміння користувачем текстів і значення ікон.
- Швидкості навчання при роботі з програмою.
- Наявності допоміжних засобів підтримки користувача.

При розробці зручного та ефективного інтерфейсу враховується профіль користувача – набір дозволених підказок, операцій і класів даних, доступ до яких необхідний користувачам для виконання певних робочих обов'язків. Прийнято розрізняти три типи користувачів: добре підготовлених, середнього

рівня, та новачків.

Для оцінки необхідного рівня зручності інтерфейсу також використовуються спеціальні опитувальники, формуляри, чек-листи, однак до даної роботи краще залучати фахівців з ергономіки. Зручний інтерфейс допомагає користувачеві впоратися з втомою і напругою при роботі в умовах високої відповідальності за результат.

4.2.4. Проблеми розробки прототипу інтерфейсу

1. Облік особливостей пристроїв введення / виводу інформації, використовуваних користувачем, наприклад:

- розмір екрану монітора;
- розширення екрану;
- палітра кольорів;
- характеристики звуку і відеокарти;
- вид миші;
- тип клавіатури;
- необхідність додаткового обладнання.

2. Специфіка інтерактивних елементів, пов'язана з вибором платформи, стандартних бібліотек:

- програмна організація введення / виведення інформації;
- зміна і створення нових елементів форм (контролів);
- придбання нестандартних бібліотек у інших фірм.

3. Вибір технології та методів ведення діалогу програми з користувачем:

- ступінь активності користувача при взаємодії (автоматичний режим або перехоплення управління програмою на себе, забезпечення доступу до всіх засобів інтерфейсу незалежно від дій користувача);
- ступінь врахування ситуації (контекстні підказки, меню подальших подій або об'єктів, запам'ятовування типових шляхів діалогу);
- відповідність очікуванням користувача (прогноз, попередня обробка, предформатування);

- стійкість, терпимість до помилок користувача шляхом виправлення типових помилок;

- дублювання вручну окремих функцій системи і додаткові контрольні процедури роботи окремих режимів;

- настройка ПІ на різний рівень підготовки користувача (образність або метафоричність предметної області на противагу скорочень і гарячих клавішах);

- ступінь адаптивності ПІ під переваги користувача (зміна способу і порядку відображення, перекомпонування екрану, вибір окремих характеристик (стилю) та ін.);

- настройка ПІ на специфіку завдання (новий формат даних, зміна набору об'єктів, доповнення атрибутів об'єктів).

4. Розміщення інформації і керуючих елементів в поле екрану, в вікні. При композиції екрану необхідно враховувати обмежені розміри простору екрану, в зв'язку з чим виникає задача оптимального розташування максимально можливого обсягу інформації шляхом:

- логічної ув'язкою даних в залежності від алгоритму роботи користувача, а не орієнтацією на структуру і послідовність фізичних таблиць даних;

- визначення рівня "детальності - узагальненості" виведення інформації (знаходження компромісу між бажанням вивести багато записів одночасно і / або відразу побачити детальну інформацію по кожній з них);

- виділення важливої інформації на екрані;

- чіткого визначення основних і допоміжних блоків інформації;

- визначення статичних полів на екрані, а також полів, де інформація періодично змінюється;

- уникнення перекриваються вікон на екрані;

- застосування принципів гармонії при компонуванні екрану (симетрія, балансу мас, дотримання пропорцій, поєднання кольорів).

5. Формування зворотного зв'язку між користувачем і додатком:

- показ актуального стану системи, режиму роботи системи (автономного, штатного, захищеного і ін.) і режиму взаємодії (наприклад, відображення, редагування або пошук даних);

- виведення окремих, важливих для робочої операції даних і показників;

- відображення дій користувача (натискання клавіш, запуск процесу, динаміка виконання процесу, отримання очікуваного і іншого результату);

- ясність і інформативність повідомлень системи. При роботі з автоматизованими слідкуючими системами у користувача може спостерігатись втрата уваги, тому, у випадку настання особливих ситуацій доцільно використовувати звукові сигнали (наприклад, у системах стеження за температурою у зоні реактора). Для розмежування вхідної та вихідної інформації можна використовувати різні кольори або яскравість (особливо це доцільно коли швидкість виконання дії менше 2 с.). Коли виконання операції вимагає часу більш ніж 20 с., у користувача може з’явитись відчуття, що програма не працює, у таких випадках доцільно виводити на екран проміжну інформацію або відзначати робочий стан біжучим рядком.

6. Проектування панелей меню та інструментів (toolbars) і вибір пунктів в них:

- логічна і смислова угруповання пунктів;

- фіксована позиція панелей на екрані;

- обмеження на ширину списку виборів і кроків (глибини) меню;

- використання звичних назв, широко поширених ікон-піктограм, традиційних ікон-символів і акуратне введення скорочень;

- розміщення найбільш часто використовуваних пунктів (зазвичай на початку списку).

7. Розробка засобів орієнтації і навігації:

- легкість визначення свого місцезнаходження і вказівка напрямку курсування;

- зручний перехід від узагальненого погляду до конкретних деталей (варіювання ступеня деталізації аналізованих об'єктів);

- швидкий пошук в списку або таблиці;

- вказівка на додатково існуючу інформацію і спосіб її отримання;
- використання коштів перегортання і прокрутки.

8. Створення форм для введення даних:

- використання одного або декількох механізмів введення в рамках режиму (клавіатура, миша, штрих-декодер, світлове перо, ін.);
- визначення способів введення даних (таблиці, списки, проста форма, меню та ін.);
- мінімізація обсягу введення;
- виділення редагованих обов'язкових і необов'язкових, а також нередагуємих полів;
- використання механізмів швидкого введення (за замовчуванням, скорочення, з продовженням та ін.);
- Виділення введеної або відредагованого інформації.

4.2.5. Принципи реалізації призначеного для користувача інтерфейсу

Стильова гнучкість – можливість використовувати різні інтерфейси з одним і тим же додатком, на практиці реалізується у вигляді набору "skins", для web-інтерфейсів – за допомогою таблиці стилів, в тому числі можливість у виборі користувачем власних установок ПІ (колір, ікони, підказки тощо .)[22]. Спільне нарощування функціональності – можливість розвивати додаток без руйнування (тобто залишаючись в рамках) існуючого інтерфейсу.

Масштабованість – можливість легко налаштовувати і розширювати як інтерфейс, так і сам додаток при збільшенні числа користувачів, робочих місць, обсягу і характеристик даних.

Адаптивність до дій користувача – додаток повинен допускати можливість введення даних і команд безліччю різних способів (клавіатура, миша, інші пристрої) і багато варіативність доступу до прикладних функцій (ікони, «гарячі клавіші», меню ...), крім того програма повинна враховувати

можливість переходу і повернення від вікна до вікна, від режиму до режиму, і правильно обробляти такі ситуації.

Незалежність в ресурсах – для створення призначеного для користувача інтерфейсу повинні надаватися окремі ресурси, спрямовані на зберігання і обробку даних, необхідних для підтримки користувача (словники, контекстно-залежні списки, набори даних за замовчуванням або за останнім запитом, історії запитів та ін.)

Можливість перенесення – при переході на іншу апаратну (програмну) платформу, повинен здійснюється автоматично перенесення і призначеного для користувача інтерфейсу, і кінцевого додатки.

4.3 Вимоги до процесів інтерфейсу та проектування і реалізація його компонентів.

4.3.1. Вимоги до вводу / виведення даних

Як було зазначено раніше, основне призначення користувацького інтерфейсу – це процедури вводу / виведення інформації яка може бути у графічному вигляді або у вигляді даних (рис. 4.1.). Для успішного та ефективного виконання цих процедур розробнику необхідно чітко визначити і правильно описати у програмі основні характеристики даних. Наступним кроком є визначення способу вводу даних та як буде виконуватись виведення результатів.

Також необхідно передбачити та визначити всі необхідні повідомлення для ведення успішного діалогу користувача з програмним додатком. Це можуть бути повідомлення: про стан пристроїв; про стан програми; про стан виконання певних операцій; про помилки виконання операцій; підказки і т.д.

Для ефективного виконання завдань, для яких призначена програма, доцільно передбачити перевірку на коректність інформації, що вводиться. Тому процес вводу доцільно організувати у вигляді циклічної структури що повторюєзапит на введення інформації, отримання її та перевірки поки не отримає правильну інформацію[20].

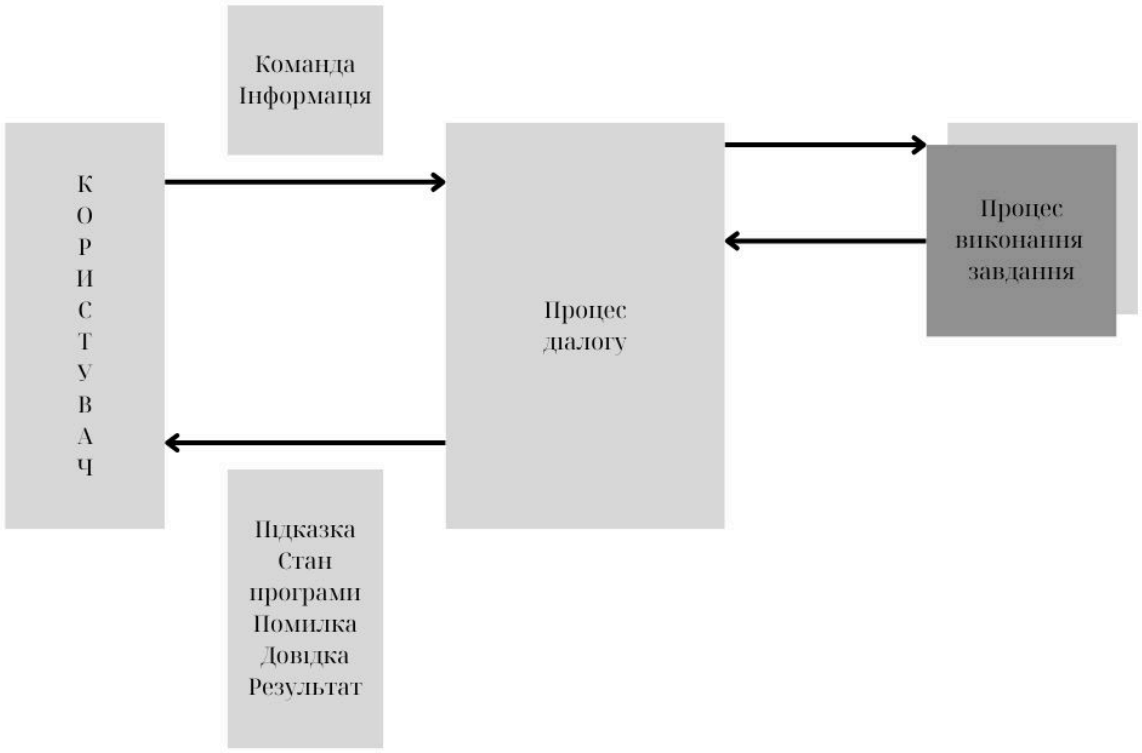


Рис. 4.1. Процес вводу/виводу та типи повідомлень

Інтерфейс має свій синтаксис і семантику. Синтаксис включає в себе типи компонентів, такі як текстовий, значок, кнопка і т. д., а зручність використання підсумовує семантику призначеного для користувача інтерфейсу. Якість призначеного для користувача інтерфейсу характеризується його зовнішнім виглядом (синтаксис) і зручністю використання (семантика). Виділяють наступні основні компоненти на основі яких будується інтерфейс користувача: екранні форми або вікна; електронні таблиці та стандартні бланки; діалог типу «питання-відповідь»; меню та команди.

При проектуванні дизайну інтерфейсу необхідно з’ясувати зміст, розташування, формат, колір і яскравість та подальшу інформацію повідомлень. Розробку графічного інтерфейсу користувача можна поділити на дві задачі: дизайн зовнішнього вигляду та програмну реалізацію.

4.3.2. Екранні форми або вікна

Вікно або екранна форма – це прямокутна частина екрану або весь екран, через яку користувач спостерігає за окремими аспектами своєї взаємодії з програмним додатком. Екранні форми можна класифікувати по ряду ознак.

1. За характером зв'язку з таблицями розрізняють зв'язані і незв'язані екранні форми.

2.За кількістю використовуваних таблиць виділяють одно табличну і багато табличні форми.

3.За характером підпорядкування окремих частин багато табличні форми класифікуються як прості, ієрархічні і синхронізовані.

4. По виконанню функцій розрізняють форми введення, виведення, керуючі, змішані.

5. За розподілом даних по екранах (сторінок) форми діляться на одно сторінкові і багатосторінкові.

6. За способом реалізації екранні форми можуть бути спливаючими і не спливаючими.

7. За формою подання інформації екранні форми можуть містити символьну інформацію, ділову графіку, інформацію, представлену в мультимедійній формі.

Програми з повною реалізацією віконного інтерфейсу окремо працюють з окремими підзадачами в різних вікнах. Така програма може одночасно відкривати / працювати з декількома документами, розміщуючи їх в окремі субвікна (наприклад, багатовіконний редактор з документом в кожному вікні). Організацію цих субвікон в подібних програмах реалізують декількома способами: одновіконний режим (SDI), багатовіконний режим (MDI, TDI), псевдобагатовіконний режим (PMDI)[22].

Також одновіконний режим може підтримувати систему фреймів, при якій загальне вікно розбито на кілька функціонально незалежних областей, фреймів (кватирок)(рис. 4.2.).

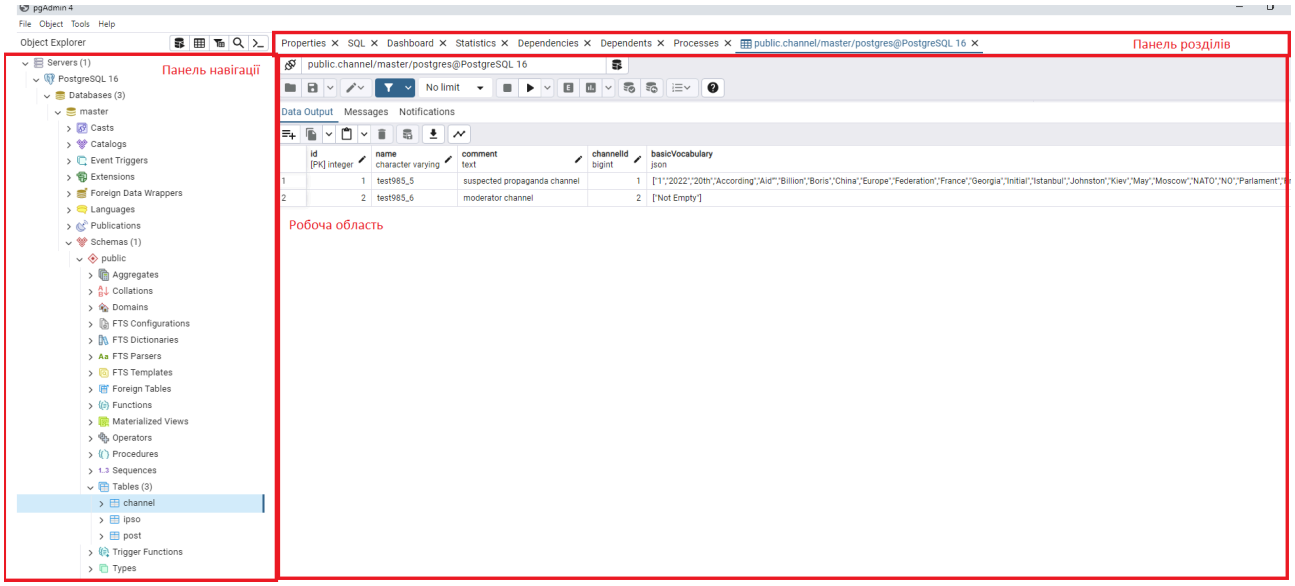


Рис. 4.2. Приклад розбиття екрану на зони

Діалогове вікно – в графічному інтерфейсі є спеціальним елементом інтерфейсу, призначеним для виведення інформації та (або) отримання відповіді від користувача. Здійснює двосторонній «діалог» між користувачем і ПК. Серед діалогових вікон можна виділити головні та підпорядковані вікна. Діалогові вікна бувають модальними і немодальними, в залежності від блокування або можливості взаємодії користувача з додатком (або системою в цілому) до отримання відповіді від нього. Модальне вікно (modal) блокує роботу батьківського вікна. Батьківське вікно повертає активність лише після закриття модального вікна.

Додатками прийнято називати прикладні програми. Кожна програма має головне вікно. В ході роботи з додатком можуть відкриватися додаткові підлеглі вікна. Спливаюча форма розташовується поверх інших відкритих форм, навіть якщо активною є інша форма. Спливаюча форма може бути немодального або модальної. Якщо спливаюча форма – модальна, користувач має можливість отримати доступ до інших об'єктів і командам меню, поки форма відкрита. Якщо спливаюча форма є немодального, не можна отримати доступ до будь-яких інших об'єктів або командам меню, поки форма відкрита. Користувач повинен виконати будь-яку дію, щоб фокус був переключений на іншу форму (або вікно).

4.3.3. Розробка таблиць та стандартних бланків (шаблонів)

Часто в програмних додатках необхідно реалізувати процедури, пов’язані з вводом / виведенням інформації, яка представляється шаблонами або паттернами у вигляді таблиць та стандартними бланками. З метою ефективного та ергономічного використання додатків в реалізація таких документів повинна відповідати їх прийнятому вигляду та наповненню.

Для конструювання стандартних документів можна застосовувати компоненти бібліотеки графічного інтерфейсу представляючи їх у вигляді окремих вікон або об’єднуючи у фрейми (frame). Для дизайну шаблону необхідно чітко визначити: яка інформація повинна бути представлена, метод та місце її представлення. Також необхідно розробити сценарій управління шаблоном (заповнення полів, відображення і т.д.) та розробити бібліотеку його програмної реалізації.

4.4 Проектування і реалізація компонентів інтерфейсу.

Для командного інтерфейсу характерно взаємодія користувача з ЕОМ за допомогою командного рядка, в якому вводяться команди певного формату, а потім передаються до виконання. Командний інтерфейс реалізований у вигляді пакетної технології та технології командного рядка.

Робота з командним інтерфейсом полягала в наступному:

- користувач вводив команду за допомогою послідовності символів (командного рядка);
- комп'ютер зіставляв команду, що надійшла, з наявним в його пам'яті набором команд;
- виконується дія, яка відповідає заданій команді.

До переваг інтерфейсу командного рядка відносять:

- низькі вимоги до апаратних засобів – мінімальним набором для роботи є клавіатура і символьне пристрій виведення або термінал;
- висока ступінь уніфікації – взаємодія забезпечується через введення і виведення символів, який часто реалізується через файловий ввід-висновок;

- широка можливість інтеграції програм – за допомогою використання командного інтерпретатора і перенаправлення вводу-виводу.

Недоліками командного інтерфейсу вважають:

- погану наочність інтерфейсу – необхідно пам'ятати команди або використовувати довідник;
- обмежені можливості виведення інформації – відсутня графіка.

Багато командних мов підтримує макроси, які розширюють функціональні можливості діалогу, скорочуючи при цьому кількість команд, що вводяться. Макрос містить кілька командних рядків. При зверненні до нього в процесі діалогу окремі рядки команд макросу виконуються оди за одним, як якщо б вони вводилися з клавіатури.

Для кращого розуміння реалізації та роботи інтерфейсу,як для програміста так і користувача, створюється схема навігації у вигляді графу або граф, який називається мережею переходів. Призначення схеми навігації – висловити основні шляхи призначеного для користувача інтерфейсу в системі. Це основні шляхи через екрани системи, але не обов'язково всі можливі шляхи. Її можна представити як карту доріг призначеного для користувача інтерфейсу системи.

Схема навігації служить в якості фону і зв'язку між окремими розкадровки. Розкадровки описують навігацію переміщень користувача за елементами призначеного для користувача інтерфейсу для виконання системних функцій, а схема навігації визначає допустимі шляху навігації. Схема навігації висловлює структуру користувальницького інтерфейсу системи, а розкадровка – динаміку. Схема навігації полегшує оцінку того, скільки "клацань" повинен зробити користувач, щоб досягти певного екрану або певної дії. Для схеми навігації можна використовувати різні уявлення.

Обрані представлення і всі рішення по налаштуванню слід задокументувати, вони відносяться до проекту рекомендацій. При розробці

схематичного представлення структури інтерфейсу необхідно дотримуватись певних правил при виборі позначень:

- зміст повинен відображати модель представлень з точки зору користувача;
- необхідно використовувати мову аудиторії для якої призначений додаток;
- необхідні не просто позначення, а системні послідовні позначення.

Створення схеми починається з розробки загальної структури системи («вид з висоти пташиного польоту»), тобто необхідно виділити окремі функціональні блоки і визначити, як саме ці блоки зв'язуються між собою. Під окремим функціональним блоком будемо розуміти функцію / групу функцій, пов'язаних з призначенням або області застосування в разі програми і групу функцій / фрагментів інформаційного наповнення. Проектування загальної структури складається з двох паралельно відбуваються: виділення незалежних блоків і визначення зв'язку між ними.

Для виділення незалежних блоків важко дати які- небудь конкретні рекомендації, оскільки дуже багато що залежить від проектованої системи. Проте, можна с упевненістю рекомендувати уникати приміщення в один блок більше трьох функцій, оскільки кожен блок в системі буде укладено в окремий екран або групу керуючих елементів. Перевантажувати інтерфейс небезпечно. Результатом цієї роботи має бути список блоків з необхідними поясненнями.

Існує три основних види зв'язку між блоками. Це логічний зв'язок, зв'язок за поданням користувачів і процесуальний зв'язок. Логічний зв'язок визначає взаємодію між фрагментами системи з точки зору розробника (суперкористувача). Користувачі мають свою думку про систему, і ця думка теж є важливим видом зв'язку. Нарешті, процесуальний зв'язок описує нехай не цілком логічну, але природню для наявного процесу взаємодію: наприклад, логіка безпосередньо не командує людям спочатку приготувати обід, а потім з'їсти його, але зазвичай виходить саме так. Всі три типи взаємозв'язку повинні бути заздалегідь передбачені при конструюванні системи.

ВИСНОВКИ

В результаті виконання атестаційної роботи було розроблено програму для виявлення нових елементів ІПСО за допомогою розширеного лексичного аналізу, бази даних на PostgreSQL та телеграм акаунту. Виконано обґрунтування актуальності проблеми і визначена основна мета – зменшення часу на розпізнання та реакцію, та надавання даних для подальшого аналізу.

Також було проведено структурно-функціональний аналіз процесу виконання лексичного аналізу, де визначені основні завдання, структурні елементи та ресурси. Отримані результати подані у вигляді теоретико-множинного опису.

Отже, в результаті виконання атестаційної випускної роботи було проведено:

- аналіз предметної області;
- комплексне проектування системи;
- розробка програмного забезпечення, як було зазначено вище;
- ергономічні дослідження в області інформаційних технологій.

На основі розглянутого дослідження, зроблено висновок, що задачу виявлення нових елементів ІПСО можна оптимізувати, а також збільшити точність та зменшити кількість витраченого часу задля покращення результату.

Список використаних джерел

[1]Col Frank L Goldstein, Col Benjamin F Findley, Psychological Operations - Principles and Case Studies , 2012

[2]Zhooriyati Sehu Mohamad, The Application of Psychological Operation (PSYOP): A Case Study on The Siege of Sauk. 2022 [Електроний ресурс] // – URL:

https://www.researchgate.net/publication/362361574_The_Application_of_Psychological_Operation_PSYOP_A_Case_Study_on_The_Siege_of_Sauk

[3]Raghav Bali, Dipanjan Sarkar, Tushar Sharma ,Learning Social Media Analytics with R: Transform data from social media platforms into actionable business insights, 2017

[4]Weiguo Fan, Michael D. Gordon, The Power Of Social Media Analytics, 2014

[Електроний ресурс] // – URL: https://www.researchgate.net/publication/259148570_The_Power_of_Social_Media_Analytics

[5]Georgios Kambourakis, Marios Anagnostopoulos, Weizhi Meng, Peng Zhou, Botnets: Architectures, Countermeasures, and Challenges (Series in Security, Privacy and Trust) 1st Edition, 2019

[6]Heli Tiirma-Klaar, Jan Gassen, Elmar Gerhards-Padilla, Peter Martini, Botnets (SpringerBriefs in Cybersecurity), 2013

[7]Niels Provos, Thorsten Holz, Virtual Honeypots: From Botnet Tracking to Intrusion Detection 1st Edition, 2007

[8]Patrick Hanks, Lexical Analysis: Norms and Exploitations, 2013

[9]Donald E. Knuth, The Art of Computer Programming, Vols. 1-3, 1998

[10]Sanju Pillai, ”An exploration of lexical analysis” [Електроний ресурс] // – URL: **https://www.researchgate.net/publication/311251505_An_exploration_on_lexical_analysis**

[11]Node.js – About [Електроний ресурс] // – URL: **https://nodejs.org/en/about**

[12]Sebastian Springer, Node.js: The Comprehensive Guide to Server-Side JavaScript Programming (Rheinwerk Computing), 2022

[13]Mario Casciaro, Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques 3rd ed., 2020

[14] Thomas Hunter II, Distributed Systems with Node.js: Building Enterprise-Ready Backend Services 1st Edition, 2020

[15] Nicolas Modrzyk, Building Telegram Bots: Develop Bots in 12 Programming Languages using the Telegram Bot API 1st ed., 2018

[16]PostgreSQL – About [Електроний ресурс] // – URL: https://www.postgresql.org/about/

[17] Abraham Silberschatz, Operating System Concepts, 2018

[18] Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova, PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries 1st ed., 2021

[19] Jan L. Harrington, Relational Database Design and Implementation: Clearly Explained 4th Edition, 2016

[20]Barry Tillman, Peggy Tillman, Rhonda Renee Rose, Wesley E. Woodson, Human Factors and Ergonomics Design Handbook, 3rd Edition, 2016

[21]Philip Kortum, Usability Assessment: How to Measure the Usability of Products, Services, and Systems (Users' Guides to Human Factors and Ergonomics Methods), 2016

[22] Shahnawaz Mohsin, Imtiaz Ali Khan, Mohhamed Ali, Ergonomic Design of CIM System Interface: Ergonomics in Computer Integrated Manufacturing Paperback, 2019

Додаток А

Шаблон таблиці Канал

```
{
  "name": "Channel",
  "columns": {
    "id": {
      "primary": true,
      "type": "int",
      "generated": true
    },
    "name": {
      "type": "varchar"
    },
    "comment": {
      "type": "text"
    },
    "channelId":  {
      "type": "text"
    },
    "basicVocabulary": {
      "type": "json"
    }
  }
}
```

Шаблон таблиці Ipso

```
{
  "name": "Ipso",
  "columns": {
    "id": {
      "primary": true,
      "type": "int",
      "generated": true
    },
    "keyWords": {
      "type": "json"
    },
    "channelId": {
      "type":"int"
    },
    "postId": {
      "type": "int"
    }
  }
}
```

Шаблон таблиці Пост

```
{
```

```
  "name": "Post",
  "columns": {
    "id": {
      "primary": true,
      "type": "int",
      "generated": true
    },
    "channelId": {
      "type":"int"
    },
    "message": {
      "type": "text"
    },
    "lexArray": {
      "type": "json"
    }
  }
}
```

Клас Database

```
const typeorm = require("typeorm");
const { DataSource } = require("typeorm");

const EntitySchema = typeorm.EntitySchema;

class Database{
  static dbConnection

  async initConnection(){
    const AppDataSource = new DataSource({
      "type": "postgres",
      "host": "localhost",
      "port": 5432,
      "username": "postgres",
      "password": "Netput1nazad",
      "database": "master",
      "synchronize": true,
      "logging": false,
      entities: [
        new EntitySchema(require("./entity/Post.json")),
        new EntitySchema(require("./entity/Channel.json")),
        new EntitySchema(require("./entity/Ipso.json"))
      ]
    })
    this.dbConnection = await AppDataSource.initialize();
  }

  async
channelInfo(channelName,channelComment,ichannelId,channelVocabulary){
```

```
    const channelRepository = this.dbConnection.getRepository("Channel");
    const channel = {
      name:channelName,
      comment:channelComment,
      channelId:ichannelId,
      basicVocabulary:channelVocabulary
    };
    await channelRepository.save(channel) ;
  }

  async writePost(postText,deconstructedPost){
    const postRepository = this.dbConnection.getRepository("Post");
    const post = {
      message: postText,
      channelId: 1,
      lexArray:deconstructedPost
     };
    const savedPost = await postRepository.save(post);
    console.log("post has been successfully saved: ", savedPost);
  }

  async getChannel(channelComment){
    const channelRepository = await
this.dbConnection.getRepository("Channel");
    const channel = await
channelRepository.findOneBy({comment:channelComment});
    return channel.name;
  }

  async getChannelVocabulary(channelComment){
    const channelRepository = await
this.dbConnection.getRepository("Channel");
    const channel = await
channelRepository.findOneBy({comment:channelComment});
    return channel.basicVocabulary;
  }

  async writeIpso(newWords){
    const ipsoRepository = this.dbConnection.getRepository("Ipso");
    const ipso = {
      channelId: 1,
      postId:1,
      keyWords:newWords
     };
    ipsoRepository.save(ipso);
    const savedIpso = await ipsoRepository.save(ipso);
    console.log("ipso has been successfully saved: ", savedIpso);
  }
```

```
  async clearDatabase() {
    try {
     const entities = this.dbConnection.entityMetadatas;
     const tableNames = entities.map((entity) =>
`"${entity.tableName}"`).join(", ");

     await this.dbConnection.query(`TRUNCATE ${tableNames} RESTART IDENTITY
CASCADE;`);
     console.log(tableNames+": Cleared");
    } catch (error) {
     throw new Error(`ERROR: Cleaning test database: ${error}`);
    }
   }

}

module.exports = {
  Database
 }
```

Клас Lexical

```
class Lexical{
  static basicVocabulary

  constructor (){
    this.basicVocabulary = [];
  }

  initBasicVocabulary(lexArray){
    lexArray.forEach(element =>{
        this.addToBasicVocabulary(element)
    })
  }

  includeWord(word){
    return this.basicVocabulary.includes(word);
  }

  addToBasicVocabulary(word){
    if (!this.includeWord(word)){
      this.basicVocabulary.push(word);
    }
  }

  async compareLexArray(lexArray){
    const newWords = [];
    await lexArray.forEach(element => {
     if (!this.includeWord(element)) {
```

```
        newWords.push(element);
      }
    });
    if (newWords.length > 0){
      return newWords;
    } else {
      return undefined;
    }
  }

  async deconstruct(text){
    const resultText = await
text.replace(/[`~!@#$%^&*()_|+\-=?;:'",.<>{}\[\]\\\/]/gi, ' ');
    const lexArray = await resultText.split("
").filter(function(element){return element});
    return lexArray;
  }

}
module.exports = {
  Lexical
}
```

Клас TelegramConnection

```
const { Api, TelegramClient } = require("telegram");
const { StringSession } = require("telegram/sessions");
const readline = require('readline');

class TelegramConnection{
  static client
  static sessionString

async initConnection() {
  const apiId = 24909780;
  const apiHash = '7492c52fddda8ce989f20db7a49d64c3';
  const stringSession = new
StringSession("1AgAOMTQ5LjE1NC4xNjcuNDEBu6I3Zlxudb2ygMQKZuo6Ahr1HmBbJKUsENFX
VXk4s+5rFYZlZesNscQaiSENwKqalTcLozgKqv52mb4YwKDbC9/Xydo5Agmz/nBUrWa4VGkvfXPD
69jO+9PtcBZxhkyswez8rPbanWHFVk9khzpNNO4G5SfBsz/oENHxyDu9FlxESQmnMxs5SWd0Cbpq
DbAEE4LBIUwlQPwRTF/b5pR81YezJAJpg+qu9MVYJQpGCvQ0RGHKDcdOXrDPr/8OrdRiqudhZ6jO
EnWR8U4sy+IHF5D5IzDvZNdOYJMwuhZTpFI4SQ59OgMuSYi9yVxsOtrKY9AFfnxSZAiDd5amEF4Q
cTQ=NjcuNDEBuxagXLLzvj9ZafRWP9vw4tuK9qQO+/GuZdAk8b60zhD55orMZUSmfd37niDzFY5x
lGYM6fiANMLp7xtZK8AQjT1v9+y1qWdcU2eeMH1iHf3UqxNAjW7PoSQ/TPyCO759o0kxSDeYIsJg
qcuFvMb2UsaMTCkJvtGvb1P21sQOBe1Q+e4b6C5zxpe3u8p5ZEx7UY0Hn8k+DclBrkUvPnhgkdxs
lo1rt6pMVxVVqhWl56QwfYn7GS/kZuTno6/WeAIXCROQuLbQBDbi209LXJNwmwBNkjVKjsKqw5NL
h4xwGVu/LjbUW7NAMYtmCMOK7r7XLOPlc/owciRwxjvt7HtwhuQ=");
  const phoneNumber = '+380968659112';
  const accountPassword = '47329';

  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
```

```
  });

  this.client = new TelegramClient(stringSession, apiId, apiHash, {
    connectionRetries: 5,
  });

  console.log("Loading interactive example...");
  await this.client.start({
    phoneNumber: phoneNumber,
    password: accountPassword,
    phoneCode: async () =>
    new Promise((resolve) =>
      rl.question("Please enter the code you received: ", resolve)
    ),
    onError: (err) => console.log(err),
  });
  console.log("Connected succesfully");
  this.sessionString = this.client.session.save();
  this.client.setLogLevel("none");
}

async getPost(channelName,id) {
  await this.client.connect();
  const message = await this.client.getMessages(channelName);
  const text = message[id].message;
  return text;
}

async clear(){
  await this.client.connect();
  const messages = await this.client.getMessages('test985_5')
  if(messages.length > 0){
    const messageIds = messages.map(element => element.id);
    await this.client.deleteMessages(
      'test985_5',
      messageIds,
      {revoke: true}
    )
  }
  const messages1 = await this.client.getMessages('test985_6')
  if(messages1.length > 0){
    const messageIds1 = messages1.map(element => element.id);
    await this.client.deleteMessages(
      'test985_6',
      messageIds1,
      {revoke: true}
    )
  }
```

```
}

async sendPost(post, channelName){
  await this.client.connect();
  await this.client.invoke(
    new Api.messages.SendMessage({
      peer: channelName,
      message: post,
      noWebpage: true,
      noforwards: true,
    })
    );
}

async getMessageNumber(channelName){
  await this.client.connect();
  const message = await this.client.getMessages(channelName);
  const number = message.length;
  console.log(number);
  return number;
}

}

module.exports = {
  TelegramConnection
}
```

Bootstrap

```
const {Database} = require("./dbconnection/src/dbconnection");
const { TelegramConnection } =
require("./dbconnection/src/telegramconnection");

const database = new Database;
const telegramChannel = new TelegramConnection;

  (async() => {
    await database.initConnection();
    await database.clearDatabase();
    await database.channelInfo('test985_5',"suspected propaganda
channel","-1001002160370872",["1","2022","20th","According","Aid”","Billion"
,"Boris","China","Europe","Federation","France","Georgia","Initial","Istanbu
l","Johnston","Kiev","May","Moscow","NATO","NO","Parlament","President","Put
in","Russia","Russian","Say","Sources","Speaker","The","Turkey","UK","US","U
SA","Ukraine","Ukrainian","WAR","Zelenski","Zelensky","Zelenskys","a","accor
ding","across","act","actually","administration","after","agreed","agreement
","agreements","aisle","always","an","and","announced","any","as","back","be
en","both","branches","but","by","can","constitution","continue","corporates
","corrupted","deal","different","dollars","draft","ended","federal","for","
from","future","generations","government","guarantors","has","in","insist","
instead","into","is","it","lands","laundered","legal","list","lost","money",
"more","not","now","of","official","officials","on","ordered","part","peace"
,"pockets","power","president","principle","proxy","pulled","refused","relea
```

```
sed","reports","resources","same","say","says","sent","serve","show","sides"
,"sign","soon","spring","suggest","suggests","talks","text","than","that","t
he","then","thinks","threaten","to","travelled","troops","trying","usual","v
ictory","voluntary","wants","war","when","with","would"]);
    await database.channelInfo('test985_6',"moderator
channel","-1001002216641738",["Not Empty"]);
    await telegramChannel.initConnection();
    await telegramChannel.clear();
    await telegramChannel.sendPost("President of the Russian Federation has
announced that according to constitution of Ukraine Zelenskys legitimacy as
president of Ukraine has ended on 20th of May. Putin thinks that Zelensky
can not sign any legal agreements from now on. Putin says that Russian
Federation wants peace talks and suggests that Speaker of Ukrainian
Parlament Ruslan Stefanchuk can sign an agreement instead of
Zelenski.","test985_5");
    await telegramChannel.sendPost("US proxy war in #Ukraine ended with the
list of generations, lost of lands, resources and future, victory for the
usual corporates and corrupted administration.. NATO is trying to serve the
same to #Europe and Georgia: say NO to #WAR always!! Say NO to
#NATO","test985_5");
    await telegramChannel.sendPost('Initial reports released by a Ukrainian
official show more than 1 Billion dollars of "Aid" money that has been sent
to the #Ukraine, has actually been laundered back into the pockets of US
government officials on both sides of the aisle and across different US
federal government branches of power','test985_5");
    await telegramChannel.sendPost('According to the President of the
Russian Federation, Ukraine and Russia  agreed in principle to the draft
agreement spring 2022 in Istanbul, but then Kiev refused to sign it after
Russian troops voluntary pulled back when ordered by Moscow as part of the
deal. The text says that the UK, China, Russia, the USA, France, and Turkey
would act as guarantors. Sources suggest Boris Johnston travelled to Kiev
soon after to insist / threaten Zelensky not to continue with the
deal',"test985_5");
  })()
```

Main

```
const {Database} = require("./dbconnection/src/dbconnection");
const { TelegramConnection } =
require("./dbconnection/src/telegramconnection");
const {Lexical} = require ("./dbconnection/src/lexical");

const database = new Database;
const telegramChannel = new TelegramConnection;
const lexical = new Lexical;

(async() => {

console.log("Initiating telegram connection...");
await telegramChannel.initConnection();
console.log("Telegram connection is established.");

console.log("Initiating database connection...");
await database.initConnection();
console.log("Database connection is established.");
```

```
const channelName = await database.getChannel("suspected propaganda
channel");
const basicVocabulary = await database.getChannelVocabulary("suspected
propaganda channel");
await lexical.initBasicVocabulary(basicVocabulary);
console.log("Basic vocabulary is filled.");

const length = await telegramChannel.getMessageNumber(channelName);
console.log(length + " posts found.");

for(let i = 0; i < length; i++){
 const post = await telegramChannel.getPost(channelName, i);
 console.log("Post extracted : " + post);

 if(post ){
  console.log("Deconstructing post into lexical tokens...");
  const deconstructedPost = await lexical.deconstruct(post);

  console.log("Analyzing lexical tokens...");
  const newWords = await lexical.compareLexArray(deconstructedPost);

  console.log("Adding results to database...");
  await database.writePost(post, deconstructedPost);

  if(!newWords){
   console.log("No new IPSO elements detected.");
   console.log("Database updated.");
  } else{
   console.log("Detected possible IPSO elements: " + newWords.join(", "));
   console.log("Adding possible IPSO elements to database...");
   await database.writeIpso(newWords);
   console.log("Database updated.");

   console.log("Adding possible IPSO elements to basic vocabulary...");
   await lexical.initBasicVocabulary(newWords);
   console.log("Basic vocabulary updated.");

   console.log("Sending alert message to moderator channel...");
   const channelName1 = await database.getChannel("moderator channel");
   await telegramChannel.sendPost("!IPSO DETECTED! Channel name: " +
channelName + ". New ipso elements: " + newWords.join(", "), channelName1);
   console.log("Alert message sent.")
  }
 }
}

})()
```

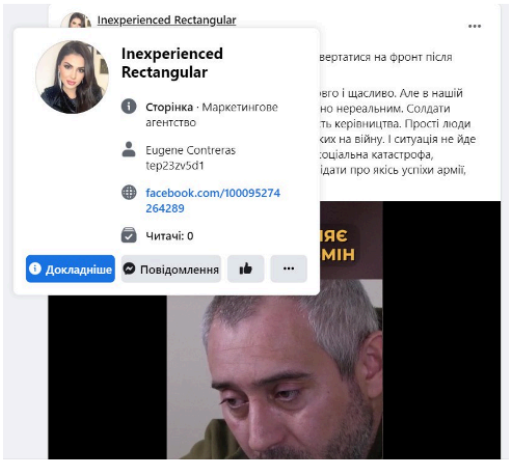
Додаток Б

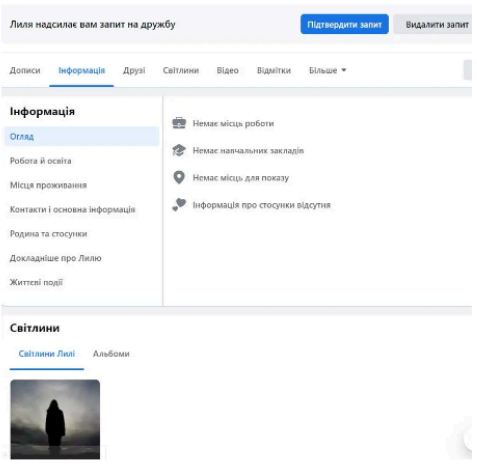
Автоматизована система виявленя елементів інформаційно-психологічної операції за допомогою розширеного аналізу лексики

Виконав:
Студент 4-го курсу, групи КН-20-2
Дородний П.Г.

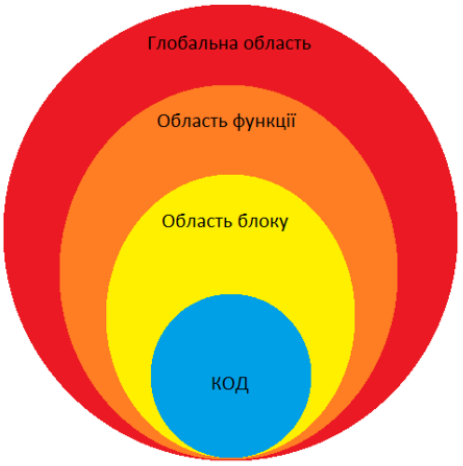
Керівник:
д.т.н., проф. Гончаренко Т.А.
д.т.н., проф. Горда О.В.

Приклади бота та пустого акаунту



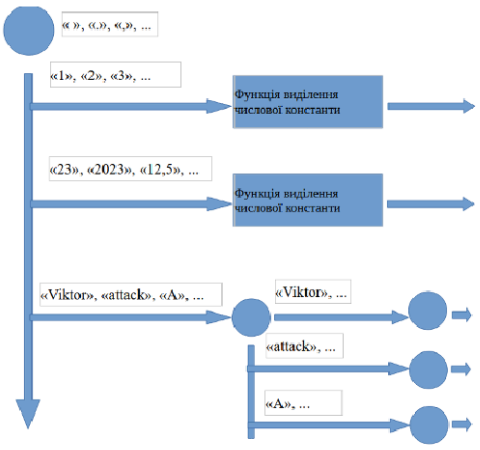


Область видимості

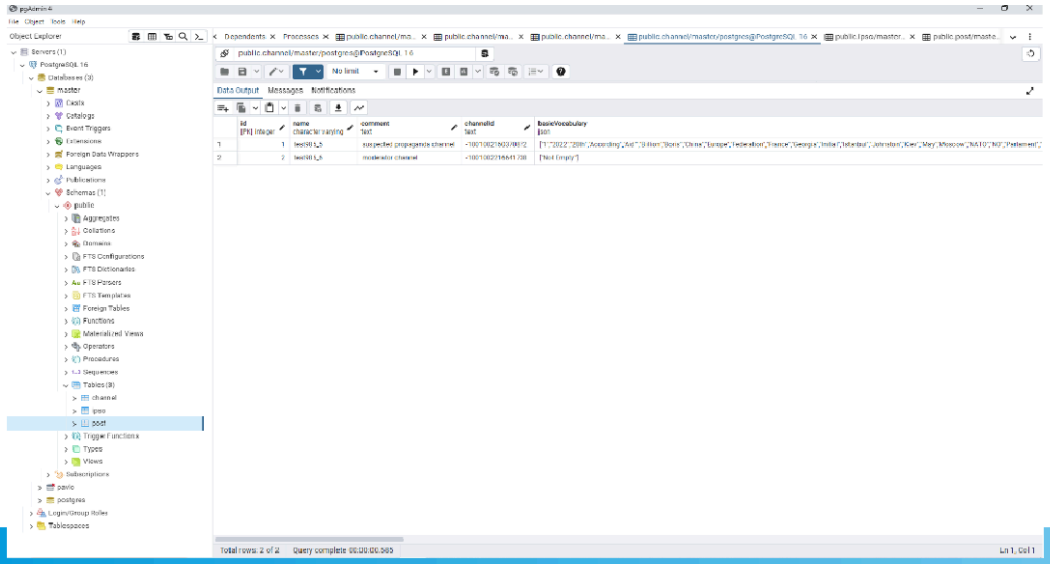


Лексична область видимості

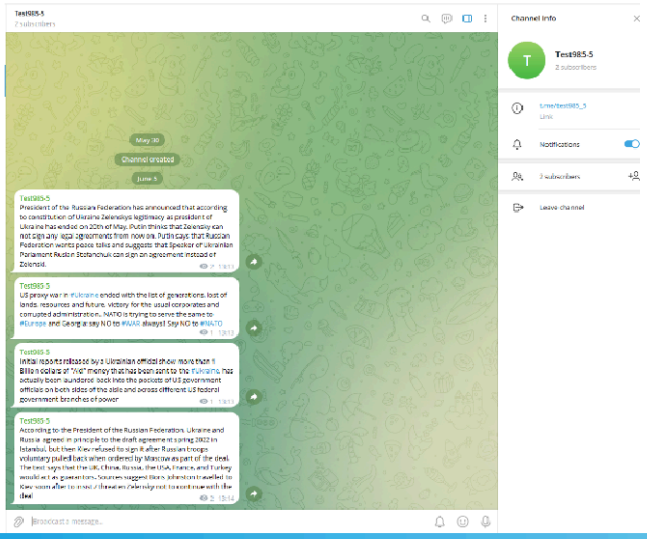
Кінцевий Автомат Станів системи



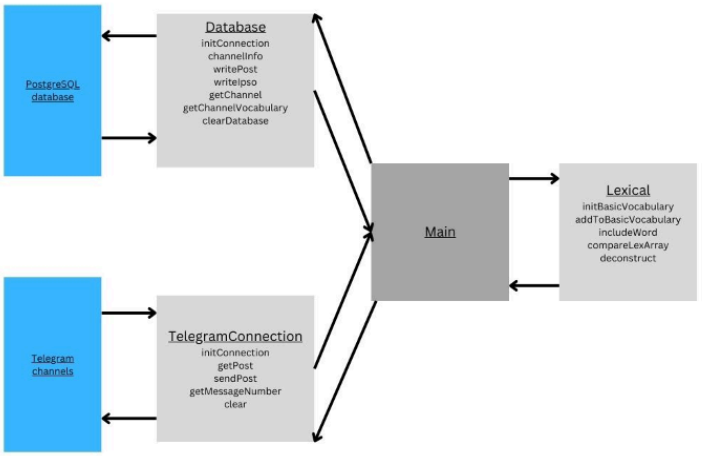
Приклад таблиці з інформацією



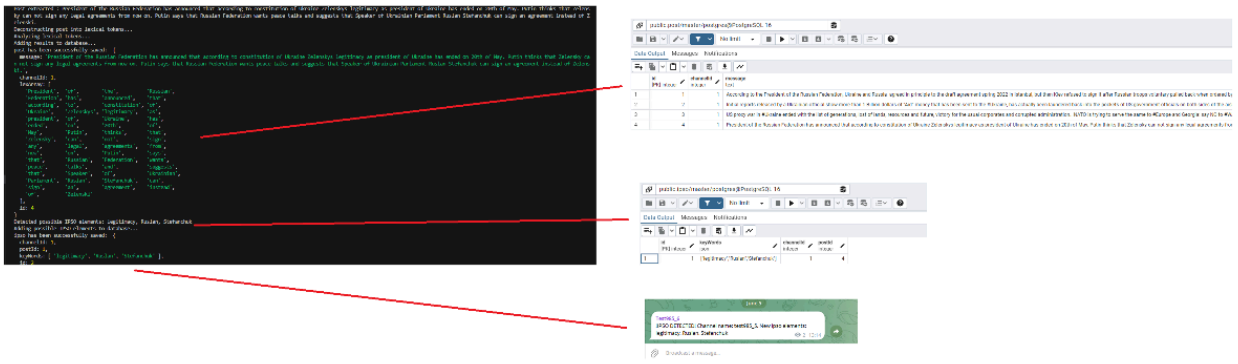
Тестовий канал з підготовленими повідомленнями



Діаграма класів системи



Тестовий приклад



ВИСНОВКИ

В результаті виконання атестаційної роботи було розроблено програму для виявлення нових елементів ІПСО за допомогою розширеного лексичного аналізу, бази даних на PostgreSQL та телеграм акаунту. Виконано обґрунтування актуальності проблеми і визначена основна мета – зменшення часу на розпізнання та реакцію, та надавання даних для подальшого аналізу.

Також було проведено структурно-функціональний аналіз процесу виконання лексичного аналізу, де визначені основні завдання, структурні елементи та ресурси. Отримані результати подані у вигляді теоретико-множинного опису.