Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти – другий (магістерський)

Дослідження методів автогенерації програмного коду
(тема)

Виконав: студент 2 курсу, групи ІПЗм-18-3___
_____Паламар В.О._____
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-наукової програми
_Інженерія програмного забезпечення

Керівник _____доц. каф. ПІ Каук В.І._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2020 р.

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Освітньо-наукова програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри
_____
(підпис)
«_____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Паламарю Вячеславу Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів автогенерації програмного коду
_____
_____

затверджена наказом по університету від "27" березня 2020 р № 473

2. Термін подання студентом роботи до екзаменаційної комісії 20 травня 2020 р.
3. Вихідні дані до роботи *засоби автоматичної генерації програмного коду, пояснювальна записка. Використовувати ОС Windows, інтегроване середовище розробки*
4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі, опис існуючих інструментів автоматичної генерації програмного коду, використовувані методи, опис розробленої програмної системи, аналіз можливих застосувань*

5 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доц. каф. ПІ Каук В.І.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка *
1.	Аналіз предметної галузі	10 березня 2020 р.	
2.	Огляд існуючих методів	16 березня 2020 р.	
3.	Методи автоматичної генерації програмного коду	23 березня 2020 р.	
4.	Підготовка пояснювальної записки	30 березня 2020 р.	
5.	Спецчастина	6 квітня 2020 р.	
6.	Підготовка презентації та доповіді	4 травня 2020 р.	
7.	Попередній захист	7 травня 2020 р.	
8.	Нормоконтроль, рецензування	11 травня 2020 р.	
9.	Занесення диплома в електронний архів	15 травня 2020 р.	
10.	Допуск до захисту у зав. кафедри	17 травня 2020 р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання _____ _____2020 р.

Студент _____
(підпис)

Керівник роботи _____доц. каф. ПІ Каук В.І._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Атестаційна робота магістра містить: 58 с., 8 рис., 5 табл., 11 джерел.

АВТОГЕНЕРАЦІЯ ПРОГРАМНОГО КОДУ, КОМПІЛЯТОР, КОНСОЛЬНЕ ЗАСТОСУВАННЯ, NODE.JS, REST, ПРЕДМЕТНО-ОРІЄНТОВАНЕ ПРОЄКТУВАННЯ, ПОВЕДІНКОВО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, АСПЕКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ.

Метою роботи є дослідження методів автогенерації програмного коду, а також проєктування автоматичного генератору коду, який не мав би недоліків існуючих аналогів. Методи розробки базуються на інструментах розробки консольних та веб-застосувань на платформі Node.js, протоколу передачі даних HTTP, фреймворку для створення консольних та веб-застосувань React та його похідну, Ink.

В результаті було розглянуто методи автогенерації програмного коду та спроєктовано план розробки та архітектуру програмного забезпечення для автоматичної генерації коду.

PROGRAM CODE AUTOGENERATION, COMPILER, CONSOLE APPLICATION, NODE.JS, REST, DOMAIN DRIVEN DESIGN, BEHAVIOR DRIVEN DEVELOPMENT, ASPECT-ORIENTED PROGRAMMING.

The goal of the research is the methods of program code auto generation and design of the automatic code generator which would omit the drawbacks of existing analogues.

Development methods are based on the console and web app development tools based on Node.js platform, data transfer protocol HTTP, console and web app framework React and its derivative Ink. As the result, methods of the automatic code generation were researched and the design and architecture were implemented.

ЗМІСТ

ВСТУП

Автоматизація процесів вже давно стала частиною нашого життя. Ще у часи Давньої Греції автоматизація мала місце в житті людей. У ті часи автоматом («самодіючий» з грецької) називали пристрої, які могли самостійно, без видимого втручання людини, виконувати деякі прості дії. Таким пристроєм були, наприклад, двері до вівтаря, які відкривалися автоматично під час церемоній розведення жертовного вогню. Історія має багато інших прикладів автоматизації, такі як прядильні фабрики, млини, годинники, заводи Генрі Форда – цей список можна продовжувати нескінченно [1]. На сьогоднішній день тренд автоматизації лише посилюється. Керування виробничими процесами на підприємствах, державні послуги, транспорт, комунальні підприємства, розробка програмного забезпечення – автоматизація присутня в тих чи інших аспектах в кожній з вищевказаних галузей. Автоматизація процесів – друга найважливіша річ як для бізнесменів, так і для студентів [2]. Перекладання обов’язків та задач на машину, що було покликано до безперервної праці на благо людства, неймовірно допомагає позбутися багатьох відволікаючих факторів протягом дня.

У розрізі даної роботи найкращим визначенням автоматизації є наступне: автоматизація – це технологія, завдяки якій процес або процедура виконується з мінімальним втручанням людини [3]. Отже, перший головний факт – втручання людини все ж необхідне, головна задача автоматизації – це зробити дане втручання простим та мінімальним. Іншими словами, автоматизація – це не заміна, це абстракція над пристроєм, системою, процесом, дією тощо.

На сьогоднішній день можна сміливо стверджувати, що однією із найголовніших областей застосування програмного забезпечення є автоматизація тих чи інших процесів. Замовлення таксі, доставка їжі, ресторанний бізнес, логістика, державні послуги – ще кілька років тому для цього треба було кудись їхати чи комусь дзвонити. Зараз же усім цим можна керувати та користуватися через спеціалізоване програмне забезпечення. І масштаби таких

систем дійсно приголомшують. Наприклад, на кінець 2018-го року в США відсоток користування такими сервісами як Uber та Lyft дорівнював 72.5%, у той час як для класичного таксі це значення було приблизно 5% [4]. У даному випадку перемога автоматизації є очевидною – замість того, щоб тримати операторів, як б приймали замовлення та повідомляли про це замовлення водіїв, роботу оператора було автоматизовано.

Отже, як було вказано вище, автоматизація – одна з найголовніших областей застосування програмного забезпечення. Більш того, існує багато інструментів для автоматизації розробки програмного забезпечення, адже потреба у цьому існує, бо багато хто хоче створити систему, яка б конкурувала з іншою популярною системою – по-перше – і навіть у самому процесі розробки існує багато рутинних задач – по-друге. Тим не менш, питання, яким чином можна найкраще організувати автоматичну генерацію коду, і як при цьому зробити так, щоб це стало не заміною, а доповненням, помічником у роботі розробника програмного забезпечення – досі залишаються відкритими.

Метою даної роботи є дослідження існуючих засобів для автоматичної генерації коду, знаходження їх сильних та слабких сторін, та розробка альтернативних варіантів генерації, які б не мали недоліків аналогів.

Проведений аналіз предметної галузі буде використаний у подальшому моделюванні системи автоматичної генерації коду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА АКТУАЛІЗАЦІЯ РІШЕНЬ

1.1 Аналіз предметної галузі

Написання коду передбачає декілька (найчастіше складних) кроків – розбиття процесу чи явища на чіткі, послідовні інструкції, управління буферами вводу та виводу, керування пам’яттю тощо. Із розвитком індустрії процес написання коду полегшувався – створювалися мови програмування із автоматичним управлінням пам’яттю (також відомі як мови програмування із збірником сміття), для типових задач (робота із файлами певних форматів, веб-сервери, математичні розрахунки) створювалися бібліотеки та фреймворки, для полегшення моделювання створювалися об’єктно-орієнтовані мови програмування тощо. Тенденція посилення абстракцій при розробці програмного забезпечення можна побачити і сьогодні.

Розробник програмного забезпечення може цього не знати, але кодогенерація вже є частиною процесу написання коду. Одна з властивостей теорії обчислюваності – повнота по Тьюрингу – полягає в тому, що програма може написати іншу програму. Це цікава ідея, яка не так оцінена, як того заслуговує, хоча і зустрічається досить часто. Більш того, кодогенерація – один з найвагоміших процесів, який виконується компілятором. Кодогенерація є частиною процесу компіляції, за яку відповідає спеціальна частина компілятора – кодогенератор. Обов’язок кодогенератора – взяти синтаксично правильну програму та перетворити її в послідовність інструкцій, які будуть виконуватися машиною. Існування механізму кодогенерації безпосередньо пов’язане із автоматичною генерацією кода та дає нам зрозуміти, що той код, що ми пишемо – це абстракція над справжніми інструкціями, які будуть виконуватися машиною. Отже, абстракція над безпосередньо кодом також має право на життя. Інструмент автоматичної генерації коду не обов’язково має бути прив’язаним до тієї чи іншої мови програмування, ми маємо розглядати це як кінцевий продукт автоматичного кодогенератора. Схожий підхід можна побачити в мові програмування Java – код

компілюється у так званий байт-код, який виконується віртуальною машиною Java. Але таких машин існує декілька, кожна з них має ті чи інші переваги та недоліки. Тому, в ідеальному випадку, автоматичний генератор коду мав би перетворювати якісь інструкції у вигляді чи то графіків та схем, чи то тексту англійською мовою, у спеціальну «проміжну мову», для якої можна розробити спеціальні адаптери для будь-якої іншої мови програмування.

Мета автоматичного генератора коду – конструювання (низького рівня) програмного забезпечення із специфікацій (високого рівня) [5]. Таким чином, автоматичний генератор коду має працювати наступним чином:

– зчитати вхідну специфікацію;

– перевірити правильність специфікації;

– зібрати фрагменти коду (провести «компіляцію» специфікації);

– провести оптимізацію (якщо можливо);

– згенерувати реалізацію на вихідній мові для відповідної платформи.

Різниця між автоматичним генератором кода та звичайним компілятором полягає в тому, що при використанні звичайного компілятору прірва між тим, що написано, та тим, що буде виконуватися (синтаксичне дерево – машинний код), не така велика, як при використанні автоматичного кодогенератора (модель – машинний код) [5]. Окрім того, навіть якщо подолати цю проблему, існує ще одна.

Вона полягає в тому, чи можемо ми довіряти такому кодогенератору. Якщо ми маємо на увазі кодогенератор як частину компілятора, то так – ці інструменти вже перевірені часом та, що більш важливо, вони мають трансформувати лише обмежений набір даних або символів, тобто трансформувати абстрактне синтаксичне дерево (яке має обмежений набір понять) в інструкцію для машин. Автоматична генерація коду в цьому плані – більш складний та масштабний процес, адже набір наших даних необмежений. Ми можемо задати уявному автоматичному генератору коду задачу будь-якої складності – від генерування функції множення чисел до драйверу для системи управління бази даних. Іншими словами, ми можемо гарантувати правильність згенерованого коду компілятора,

але жоден існуючий автоматичний генератор коду не може цього гарантувати (крім тих, які розроблялися під чітку задачу – створення таблиць бази даних, простого REST-сервера, математичних формул тощо).

На жаль, але автоматичний кодогенератор ніяк не відноситься до існуючих кодогенераторів в компіляторах. Як було зазначено вище, вони ідейно схожі, але практична реалізація буде зовсім інша, адже автоматична генерація коду – це не про компілятори, це про перетворення високорівневої специфікації на низькорівневий програмний код.

Виявляється, що існує підхід, який допомагав би наблизити специфікацію, написаною мовою людей, до програмного коду, написаною на мові програмування. Більш того, їх існує декілька.

Розпочати слід з найбільш відомого – предметно-орієнтованого проєктування. Цей підхід спрямований на моделювання комплексного об’єктно-орієнтованого ПЗ. Його переваги полягають у тому, основа увага концентрується на предметній області, а програмні моделі створюються таким чином, щоб відображати глибоке розуміння предметної області. Вперше цей термін було запроваджено Еріком Евансом в його книзі, яка має таку ж назву що і у даного підходу [6].

Для того, щоб краще розуміти даний підхід, необхідно розуміти основні визначення, які запроваджуються даним підходом.

Першим таким визначенням є домен. Домен – це предметна область, що буде використовуватися програмістом під час розробки програмного забезпечення. Для кращого описання моделі використовуються моделі, загальна мова та контекст.

Модель – це абстракція або система абстракцій, метою якої є описання окремих аспектів домену.

Загальна мова – це така мова, що будується навколо моделей домену. Вона використовується і програмістами, і експертами предметної галузі.

І, нарешті, контекст – це середовище, в якому чітко визначені значення предметів та дій. Суттю предметно-орієнтованого проектування є конкретне

визначення контекстів і обмеження моделювання в їх рамках. Треба точно знати контекст, в якому буде використовуватися модель.

На сьогоднішній день цей підхід є найбільш підходящим для вирішення нашої задачі – перетворення високорівневої специфікації на низькорівневий код. Але не лише цей підхід може допомогти досягти цієї мети.

Іншим підходом, який слід розглянути в розрізі даного дослідження – це аспектно-орієнтоване програмування.

Аспектно-орієнтоване програмування (АОП) – це парадигма, яка дозволяє відокремити наскрізну функціональність. Наскрізна функціональність - це така функціональність, яку не можна віднести до конкретного шару або рівня. Мета АОП – відокремити технічні проблеми (логування, транзакції, бази даних) з доменної моделі та полегшити розробку та реалізацію моделей предметної області, сконцентрованих перш за все на логіці бізнес-домену [7].

В даній парадигмі під аспектом розуміють модуль, що реалізує наскрізну функціональність. Для того, щоб змінити поведінку іншого коду, аспекти застосовують так звані поради в точках з’єднання, визначених зрізом.

Порада – це додаткова логіка. Це той код, що може бути виконаний до, після чи замість точки з’єднання.

Точка з’єднання – це точка в програмі, де рекомендується застосувати пораду. Під точкою в програмі можна розуміти створення об’єкта, виклик змінної тощо. Точки з’єднання можна об’єднати в зрізи, які, у свою чергу, визначають, наскільки дана точка з’єднання підходить до поради.

Завдяки аспектно-орієнтованому підходу можна розглядати програму як набір класів або модулів, кожен з яких виражає ту чи іншу особливість функціонування системи.

Аспектно-орієнтований підхід можна застосувати і для автоматичного генератора коду. Це дозволило би розробнику програмного забезпечення зосередитися на бізнес-логіці, а генератор коду піклуватися про технічні, не пов’язані з бізнес-логікою моменти, які все ж необхідні.

І останній підхід, який також слід мати на увазі під час створення автоматичного генератора коду – це розробка, керована поведінкою (BDD). Основною ідеєю даної методології є суміщення чисто технічних інтересів та інтересів бізнесу під час розробки, дозволяючи тим самим керуючому персоналу та програмістам розмовляти однією мовою. Для спілкування між цими групами використовується предметно-орієнтована мова, основу якої складають конструкції із природної мови, зрозумілі не фахівцям, які зазвичай виражають поведінку програмного продукту та очікувані результати.

Тим не менш, сам по собі даний підхід не зовсім підходить для використання в автоматичному генераторі коду, оскільки він оснований на розробці, керованій тестами, а тому концентрується лише на передумовах та на очікуваних результатах. Але під задачу автоматичного генератора коду ця методологія цілком підпадає – вона наближає високорівневу специфікацію та низькорівневу програмну реалізацію один до одного.

1.2 Аналіз аналогів

Існує безліч засобів автоматичної генерації коду. Для порівняння та аналізу аналогів будуть використовуватися наступні критерії:

– зрозумілість не технічним спеціалістам;

– підтримка декількох мов програмування;

– універсальність (можливість описання тих чи інших частин специфікації).

Після аналізу деякої кількості аналогів буде наведено таблицю, яка підсумує цей аналіз. Таким чином будуть чітко окреслені обмеження існуючих інструментів для автоматичної генерації програмного коду, які необхідно подолати для створення кращого рішення.

Слід розпочати з низькорівневих прикладів та поступово перейти до більш абстрактних інструментів.

Одним із прикладів генерації коду є так звана source-to-source трансформація. Її мета полягає у трансформації програми з однієї мови на іншу. Прикладом такої програми є transform tools.

Transform tools – це веб-ресурс для трансформації об’єктів з однієї мови в іншу. Даний ресурс надає можливість легко перетворити JSON-об’єкт в TypeScript інтерфейс, в структуру мови програмування Go або команду створення таблиці в MySQL.

Недоліком даного інструмента є те, що він може виконувати трансформацію лише для схожих за структурою об’єктів. Окрім того, даний сервіс підтримує лише об’єкти, тобто не існує можливості перетворити метод, написаний на одній мові, в метод, написаній на іншій. Також слід зазначити, що скоріш за все даний інструмент не буде зрозумілим для великої кількості не технічних фахівців, адже він розрахований перш за все на розробників програмного забезпечення та розуміння тих чи інших мов програмування. Із інтерфейсом даного сервісу можна ознайомитися на рисунку 1.1.

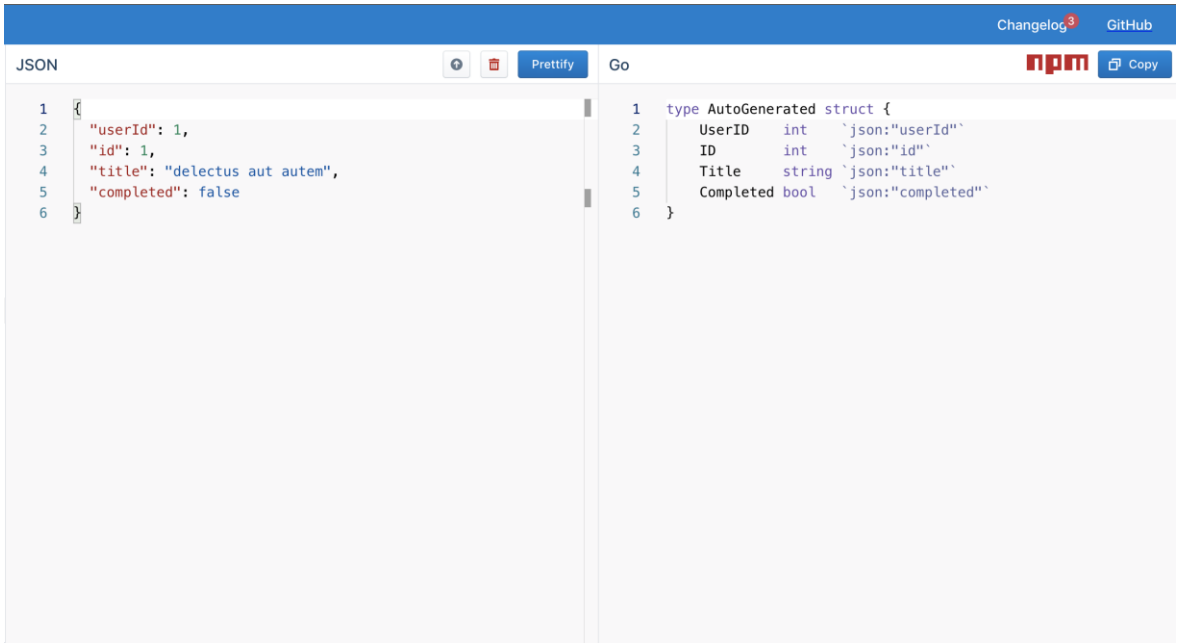

Рисунок 1.1 – Інтерфейс Transform tools

Вищевказаний інструмент не може вважатися повноцінним source-to-source компілятором, оскільки, як було зазначено вище, цей інструмент не розуміє усіх конструкцій підтримуваних мов. До повноцінних інструментів можна віднести такі компілятори та транспілери, як Babel, HipHop, ClojureScript, Haxe. Усі, окрім Haxe, можуть перетворювати лише з однієї мови в іншу, у той час як Haxe підтримує декілька вихідних мов. Тим не менш, дані бібліотеки не можна вважати повноцінними аналогами, оскільки вони все одно вимагають написання коду, хоча й абстрагують деякі конструкції мов.

Іншим яскравим прикладом кодогенерації є інструмент компілятора мови програмування Go під назвою go generate. Go generate дозволяє генерувати додаткові команди, які задаються через спеціальні шаблони, які розуміє Go. Це особливо зручно, якщо модуль має підтримувати параметризовані типи. Уявімо, що у нас є метод, який має підтримувати декілька типів (наприклад, число та строку). В мові, яка підтримує параметризовані типи, це не проблема, але в Go такої підтримки немає. Ця проблема вирішується завдяки go generate. Створивши спеціальний шаблон, ми можемо згенерувати один і той самий метод для всіх необхідних типів, замість того, щоб писати кожен метод самому. Окрім цього, даний підхід інколи використовується в Go для інверсії залежностей.

Цей інструмент вже трохи ближчий до автоматичної генерації коду, але він має декілька недоліків. По-перше, цей інструмент може працювати лише з Go, по-другу – він ніяк не наближає нас до нашої початкової мети – наблизити специфікацію та реалізацію один до одного. Безумовно, цей інструмент додає базову автоматизацію, але це зовсім не те, що ми шукаємо.

Розглянувши кілька прикладів, можна виділити першу категорію інструментів генерації коду – це категорія низькорівневих генераторів. Вони націлені на те, щоб або перетворити одну мову на іншу, або додати якісь можливості до самої мови програмування. Вони лише частково автоматизують генерацію коду, що непогано, але недостатньо у розрізі даної роботи. Окрім того, не технічні фахівці не розуміють, як користуватися цими інструментами, оскільки вони орієнтовані виключно на технічних спеціалістів.

Очевидно, що перша категорія кодогенераторів не підпадає під наші вимоги. Можливо, більш високорівневі інструменти більше підпадають під наші потреби.

На цьому етапі слід згадати, що задача автоматичного генератора коду – перетворити специфікацію на програму. Але специфікація може складатися з багатьох частин, тому слід розглянути деякі з них.

Частиною специфікації можуть бути UML та ER діаграми.

UML (Unified Modeling Language) – це, як видно з назви, уніфікована мова моделювання, яку використовують в об’єктно-орієнтованому програмуванню та проєктуванні. Вважається, що UML є невід’ємною частиною процесу розробки програмного забезпечення. Існує багато видів UML діаграм.

Прикладом інструменту автоматичної генерації коду, що використовує UML діаграми у якості вхідних даних, є Visual Paradigm. Цей інструмент дозволяє генерувати структуру класів з діаграми класів, послідовність інструкцій з діаграми послідовностей. Підтримує багато мов програмування. Інтерфейс даної програми можна побачити на рисунку 1.2.

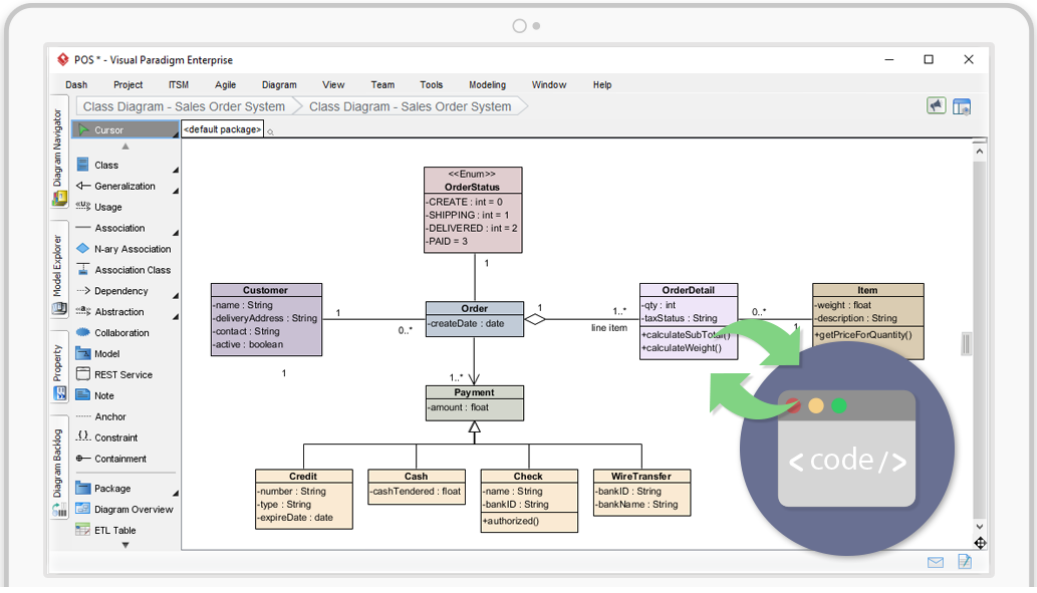

Рисунок 1.2 – Інтерфейс програми Visual Paradigm

ER модель, також відома як модель «сутність-зв’язок» - це така модель даних, що надає можливість описувати концептуальні схеми завдяки узагальненим конструкційним блокам. ER-модель використовується при високорівневому (концептуальному) проектуванні баз даних. Автоматична генерація коду з такої діаграми доволі популярна, адже така діаграма доволі точно описує взаємозв’язок сутностей. Більш того, створення таблиць в базі даних – задача доволі рутинна (такі діаграми найчастіше використовують для описання саме зв’язку таблиць в реляційних базах даних).

Прикладом інструмента, що дозволяє генерувати таблиці бази даних з ER-діаграми, є ERD+. Цей інструмент надає можливість створювати ER діаграми та реляційні схеми. Завдяки ньому можна описати не лише зв’язки, але й атрибути моделей предметної області. Після описання моделей, користувач має можливість експортувати діаграму у вигляді SQL запитів для створення таблиць бази даних, і ці запити будуть враховувати вищезазначені зв’язки та атрибути. Інтерфейс програми можна побачити на рисунку 2.3. Це веб-застосування, тому нічого не треба встановлювати на власний ПК, вистачить лише Інтернет-браузера.

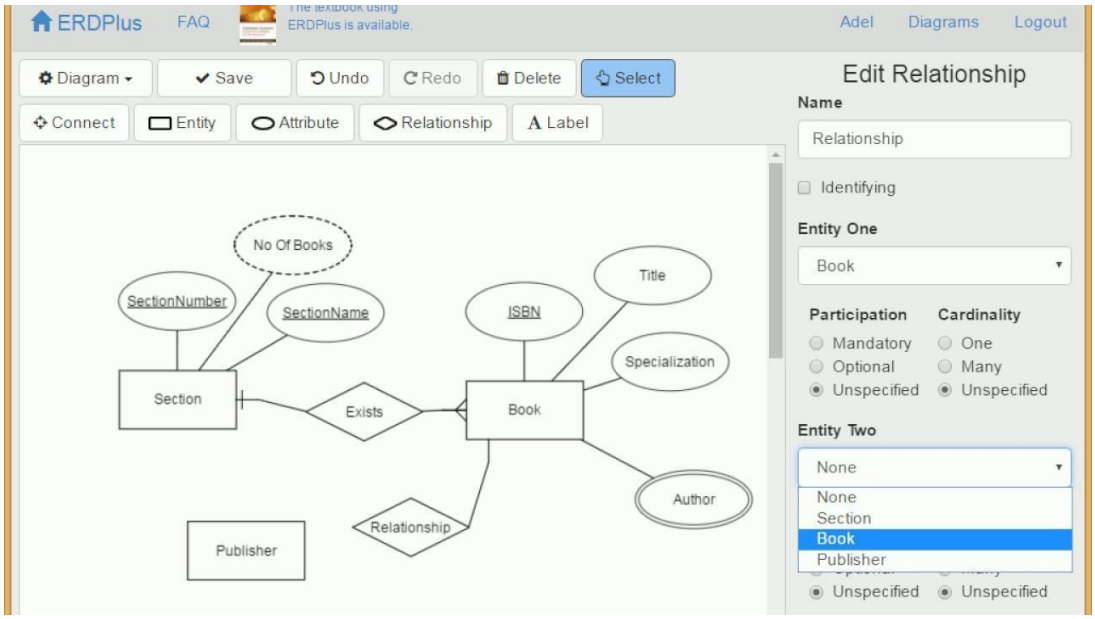

Рисунок 1.3 – Інтерфейс веб-застосування ERD+

Ці приклади надають нам можливість ще одну категорію автоматичних генераторів коду – графічні. В якості специфікації програми виступає та чи інша діаграма відношень, а на виході ми отримуємо, у більшості випадків, структуру об’єктів, класів, таблиць бази даних тощо. Здавалося б, що цього більш ніж достатньо. Але це не так.

Дані інструменти чудово себе проявляють в том, щоб описати як об’єкти пов’язані один з одним. Але Ерік Еванс, автор предметно-орієнтованого проєктування, стверджує, що цього недостатньо [6]. Справа в тому, що дані інструменти чудово підходять для описання того, що є що та що із чим пов’язано. Але ці інструменти зовсім або майже зовсім не відображають, як і що працює. Майже всі реальні задачі та предметні області, які розробник програмного забезпечення хоче відобразити в коді – це не лише набір сутностей та зв’язок між ними. Так, ці інструменти беруть цю задачу на себе. Але задача побудови правил, обмежень, взагалі бізнес логіки все ще залишається на програмісті. Безумовно, такі інструменти забезпечують ідентичність доменних та програмних моделей, але вони ніяк не можуть гарантувати те, що бізнес процеси та програмні процеси також будуть ідентичні. Тим не менш, дані інструменти можна вважати автоматичними генераторами коду, хоча й обмеженими у своїх можливостях.

Також існують інструменти, що може описати не зв’язки, а процеси та дії, такі інструменти засновані на BDD підході, який було описано раніше. Справа в тому, що BDD використовує спеціальний стиль написання сценаріїв під назвою Gherkin. Gherkin має набір ключових слів, який наведено в наступній таблиці. В 2007 році Деном Нортом було запропоновано шаблон для специфікації, який отримав популярність і врешті решт став відомий як мова Gherkin. [8]

Ця нотація зрозуміла як технічним, так і не технічним спеціалістам, що цілком підпадає під вимоги автоматичного генератора, що має буде реалізовано в ході виконання цієї роботи.

Ключові слова цієї специфікації наведено у таблиці 1.1.

Таблиця 1.1 – Ключові слова нотації Gherkin

Ключове слово англійською	**Україномовна адаптація**	**Опис**
Story (Feature)	Історія	Специфікація починається з цього слова, після якого через двокрапку в умовній формі пишуть назву історії.
As a	Як (у ролі)	Роль тієї персони в бізнес-моделі, яка зацікавлена в даній функціональності.
In order to	Для того, щоб досягти	Коротко про те, які цілі ставить перед собою персона.
I want to	Я хочу, щоб	Коротко про кінцевий результат.
Scenario	Сценарій	Кожен сценарій однієї історії починається з цього слова, після якого через двокрапку в умовній формі пишуть мету сценарію.
Given	Дано	Початкова умова. Якщо початкових умов декілька, то кожна нова умова додається з нового рядка за допомогою ключового слова And.
When	Коли (щось трапляється)	Подія, яка ініціює даний сценарій, Якщо подію не можна розкрити через одне речення, то усі наступні деталі розкриваються через ключові слова And та But.
But	Але	Допоміжне ключове слово, аналог заперечення.

Існує декілька інструментів, які використовують Gherkin при генерації коду. Їх недолік полягає в тому, що всі вони спрямовані на тести і не на реальні

застосування та програмні засоби. Окрім того, автоматизація в них все одно не повна, адже такі інструменти задають лише структуру тестів, у той час як сам тест необхідно писати програмісту. Тим не менш, такі інструменти можна винести в третю категорії – автоматична генерація сценаріїв для тестів.

Слід нагадати, що мета даного дослідження – знайти універсальний інструмент, який мав би можливість розуміти таку специфікацію, яке описує не тільки зв'язки чи тільки процеси, але й все разом. Проаналізувавши аналоги, можна прийти до висновку, що універсального інструменту не існує, всі існуючі генератори так чи інакше зосереджуються на одній області. Хочеться розробити такий інструмент та таку форму специфікації, щоб можна були описувати і тестові сценарії, і безпосередньо код, і зв'язки сутностей тощо. Мета даної роботи – дослідити та розробити такий інструмент, який підходив би для генерації швидких прототипів, коли початкова специфікація ще досить мала і її легко виразити, використовуючи правила DDD, BDD та AOD.

Ці парадигми об'єднує те, що вони запроваджують певний словник термінів, що допоможе зробити вхідну специфікацію більш структурованою. Таким чином, це полегшує задачу, адже хоча й специфікація може мати будь-який зміст, при суворій структурі ми можемо працювати із такою специфікацією. Це так само, як і при звичайному написанні програми – ми можемо виразити майже будь-яку думку через код, але мі підкорюємося певним правилам мов програмування, тобто використовуємо певну структуру та конструкції при написанні програми.

Таку специфікацію необхідно чітко описати, щоб структура була зрозуміла кожному. Слід зазначити, що це не складна задача, адже вищезазначені підходи описані та досліджені в багатьох публікаціях. Ці підходи (окрім аспектно-орієнтованого програмування) спрямовані саме на те, щоб у технічних та нетехнічних спеціалістів була спільна мова.

Підсумки аналізу існуючих інструментів для автоматичної генерації програмного коду наведено в таблиці 1.2.

Таблиця 1.2 – Порівняння існуючих аналогів

Назва	Зрозумілість	Підтримка мов програмування	Універсальність
Transform Tools	Низька	Висока	Низька
Go generate	Низька	Низька	Низька
Visual Paradigm	Середня	Висока	Низька
ERD+	Середня	Низька	Низька

Як бачимо, великим недоліком існуючих інструментів є недостатня зрозумілість не технічним фахівцям та майже повністю відсутня універсальність.

1.3 Постановка задачі

Дана робота має за мету створення розширеної мови специфікації, яка б об’єднувала в собі нотацію DDD, BDD та AOD, та автоматичного генератора коду, який би розумів цю специфікацію. Дане рішення має виділятися на фоні конкурентів високорівневістю та відсутністю прив’язаності до певного способу використання (генерація таблиць бази даних, бібліотеки тестування тощо).

У ході дослідження було поставлено наступні задачі:

а) провести аналіз предметної області, виявити проблемні місця та актуалізувати рішення;

б) сформувати вимоги до програмної системи;

в) спроєктувати архітектуру програмного забезпечення;

г) змоделювати необхідні UML-діаграми;

д) розпланувати чітку організацію роботи;

е) описати вхідну специфікацію для автоматичного генератора коду;

ж) реалізувати автоматичний генератор коду.

У ході дослідження було проведено аналіз предметної області, сформульовано основну мету та визначено задачі, які необхідно виконати для успішного завершення роботи.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Загальні вимоги

Для проектування програмної системи необхідно сформулювати певні вимоги такі як:

– перелік функціоналу, що повинна містити система;

– архітектура програмної системи;

– частини, з яких вона буде складатися;

– технології, які слід застосувати, для досягнення найкращого результату.

2.2 Функціональні вимоги

Отже, необхідно розробити формат специфікації, який поєднував би терміни DDD, BDD та AOD, та автоматичний генератор коду, що розумів би цю специфікацію та міг би перетворювати її в ту чи іншу мову програмування. Початкова версія генератора має буде реалізована у вигляді консольного додатку, який очікує файл специфікації у якості вхідного параметра.

Для реалізації даної задачі необхідно виконати наступне:

а) описати формат специфікації, яку має розуміти автоматичний генератор коду;

б) створити валідатор, який би перевіряв принаймні синтаксичну правильність специфікації;

в) створити парсер, який міг би розбивати специфікацію на окремі блоки;

г) створити адаптер для однієї з популярних мов програмування (в даній роботі було обрано TypeScript).

2.3 Нефункціональні вимоги

Окрім перелічених вище функціональних вимог, слід розглянути нефункціональні вимоги, а саме вимоги до мов програмування, фреймворків та інструментів:

а) автоматичний генератор коду буде розроблено із використанням мови програмування TypeScript та платформи Node.js;

б) клієнт матиме вигляд консольного застосування, який буде надавати можливість генерувати артефакти специфікації та безпосередньо програмний код;

в) специфікація передбачає вигляд, близький до Gherkin нотації, поширену термінами із AOP (аспект, порада, зріз) та DDD (репозиторій, сервіс, модель).

Також до нефункціональних вимог можна віднести те, що генератор має працювати швидко та мати можливість повідомляти про етапи процесу обробки специфікації. Крім того, дане застосування має підтримувати сучасні настільні операційні системи (Windows, Linux, macOS).

3 АРХІТЕКТУРА ТО ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проєктування ПЗ

В ході аналізу вимог важливим етапом їх формування було відображення станів, взаємодій даних тощо у явному вигляді. В цьому нам допомогли UML-діаграми. Для проектування програмної системи було використано засіб проектування draw.io та спроєктовано наступні діаграми UML:

а) Діаграма послідовностей відображає взаємодії об’єктів впорядкованих за часом. На ній показано у вигляді вертикальних ліній різні процеси або об’єкти, що існують водночас. Надіслані повідомлення зображуються у вигляді горизонтальних ліній в порядку відправлення.

Діаграма відображає лише ті об’єкти, які безпосередньо беруть участь у взаємодії, при цьому ніякі статичні зв’язки з іншими об’єктами не вказуються. Для описання візьмемо процес роботи автоматичного генератора коду. Користувач буде передавати файл(и) специфікації до генератора, після чого генератор буде робити перевірку синтаксичної коректності специфікації. Після цього перевірену специфікацію буде передано далі, де вона буде передано безпосередньо генератору, який, у свою чергу, поверне код у тому форматі, що було вказано користувачем (чи то синтаксичне дерево, чи то кінцевий код на обраній мові програмування). Якщо користувач обирає мову програмування, яка ще не підтримується, то генератор поверне помилку та закінчить процес генерації. Діаграма послідовностей зображена на рисунку 3.1.

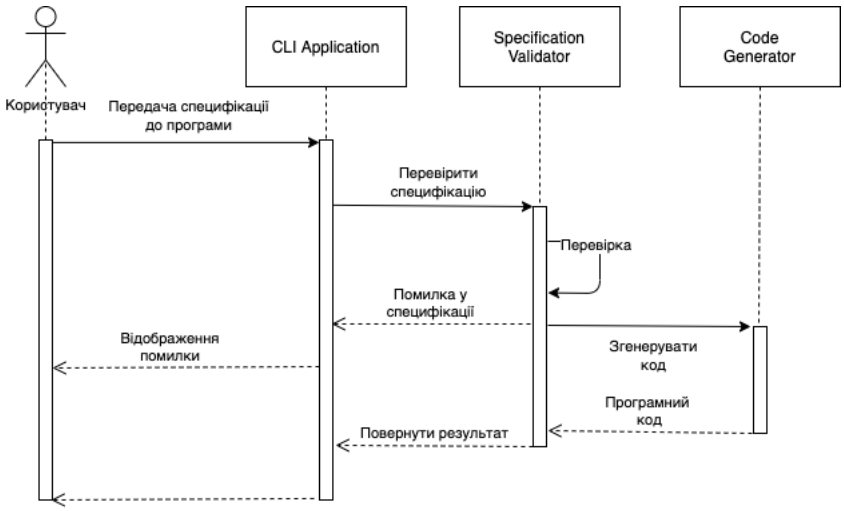

Рисунок 3.1 – Діаграма послідовностей

б) діаграма кооперації використовується для того, щоб специфікувати особливості реалізації окремих найбільш значущих операцій в системі. На ній відображені взаємодії між частинами композиційної структури або ролями кооперації. В відмінності від діаграм послідовності, на діаграмі комунікації явно вказуються відносини між об'єктами, а час як окреме вимірювання не використовується (застосовуються порядкові номери викликів). Завдяки їй стає можливим графічне представлення не тільки послідовність взаємодії, але й усіх структурних відносини між об'єктами, які беруть участь у цій взаємодії.

Представимо систему, що спроєктована, систему у вигляді діаграми кооперації. Діаграма зображена на рисунку 3.2.

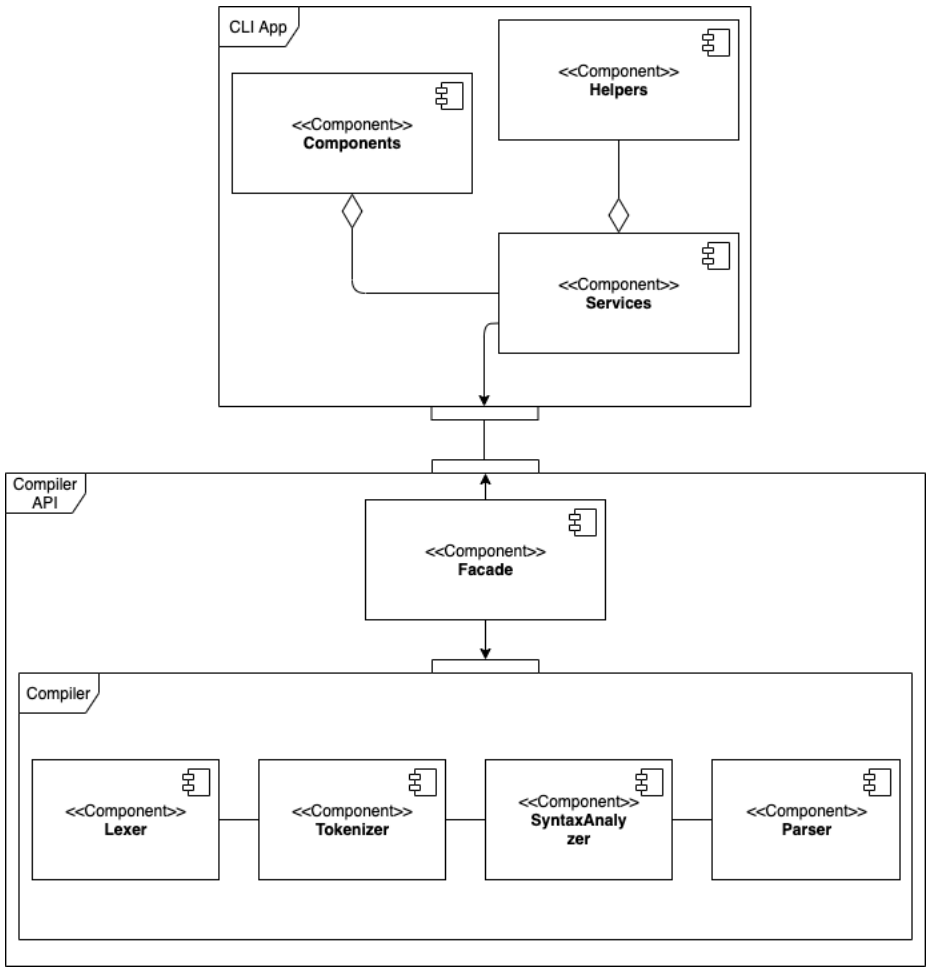

Рисунок 3.2 – Діаграма кооперації

в) Наступною є діаграма класів. Ця діаграма демонструє відношення статичних, декларативних елементів моделі. Такий тип діаграм може застосовуватися як при розробці нової системи, так і при зворотному проектуванні. Дана діаграма є останнім кроком у фазі проектування і першою у фазі розробки. На ній відображені основні моделі, використані при розробці програмної системи.

Оскільки ідею роботи генератора коду було надано у попередній діаграмі, необхідно розглянути інший важливий аспект даної системи – це генерація артефактів для створення специфікації. Клас, який буде керувати створенням артефакту – це ArtifactGenerator. Його обов’язок – створити певний артефакт специфікації, виконавши ті чи інші кроки. Для цього буде використаний патерн

«Абстрактна фабрика» [9]. Іншими словами, у нас є клас AbstractArtifactFactory, і при необхідності створення якогось певного артефакту (сценарію, аспекту, моделі), ми створимо ScenarioFactory, AspectFactory та ModelFactory відповідно. Таким чином, ми не будемо прив'язані до певної реалізації, а лише до «шаблону». Дана діаграма зображена на рисунку 3.3.

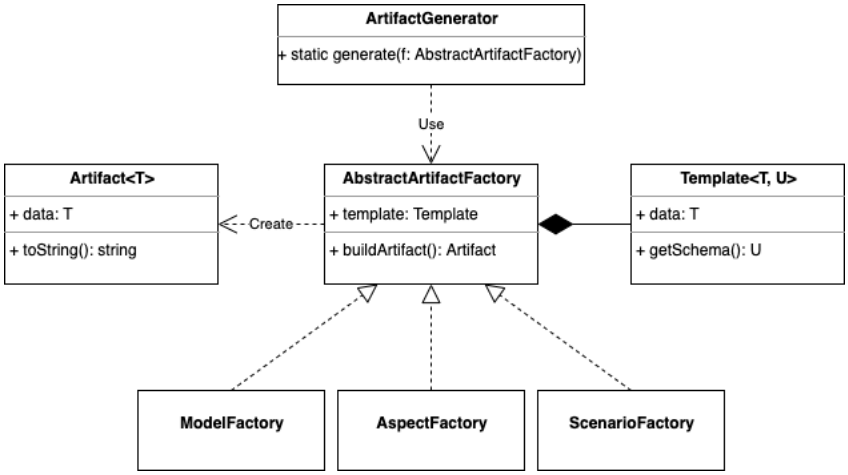

Рисунок 3.3 – Діаграма класів

Користувач матиме дві основні дії при користуванні даною системою, а саме:

– генерація артефактів, що допоможуть створити правильну специфікацію;

– передача створеної специфікації автоматичному генератору коду;

– отримання програмного коду, заснованого на специфікації.

Через те, що застосунок буде керуватися виключно через командний рядок, він не зовсім підходить не технічним фахівцям. Але, беручи до уваги гнучкість запропонованої архітектури, створити веб-застосування буде не такою складною задачею. Тим не менш, на даному етапі було прийняте рішення використати саме консольний рядок, оскільки дана робота більше зосереджена на автоматичній генерації коду, а не на досвіді користувача.

Іншою перевагою вищеописаної архітектури є те, що вона дозволяє легко створювати нові артефакти специфікації за потребою. У майбутньому можна

буде розширити API системи та надавати можливість користувачам створювати власні артефакти, реалізувавши певний програмний інтерфейс.

Слід зазначити, що в даних діаграмах не було зазначено, як саме проміжний код автоматичного генератора буде перетворюватися на код справжньої мови програмування. Планується, що буде розроблено адаптер, який матиме можливість перетворю синтаксичне дерево проміжної мови на проміжне дерево інших мов програмування. Це буде працювати ніби source-to-source компілятор, тому було прийнято рішення не відображати цей процес у діаграмах, адже він доволі розповсюджений. [10]

3.2 Проєктування архітектури ПЗ

На базі вищезазначених вимог було побудовано схему архітектури системи.

Програму реалізовано за схемою клієнт-сервер, де у якості клієнта виступає застосування командного рядка, а в якості сервера – фасад навколо компілятора нашого автоматичного генератора коду. Головна перевага даної архітектури – можливість створення новий видів клієнтських застосувань у майбутньому (десктопні, мобільні, веб застосування).

З консольного застосування надходить запит до фасаду компілятора, який реалізовано у вигляді Node.js сервера. Запит містить у собі специфікацію для програмного коду. Окрім того, сервер матиме необхідні методи для створення артефактів специфікації.

На рисунку 3.4 зображено схему архітектури системи.

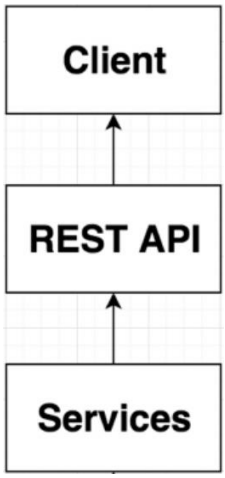

Рисунок 3.4 – Архітектура системи

Бізнес-логіка даної системи розподілена між сервісами, оскільки це дозволяє розподілити функціонал на незалежні блоки, які мають можливість працювати між собою.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

Для розробки консольного застосування для генерації артефактів специфікації було використано платформу NodeJS.

NodeJS – платформа для виконання мережевих застосувань, написаних на мові програмування JavaScript. Дану платформу було обрано перш за все через те, що вона підтримує поняття потоків, які допомагають легко працювати з операціями вводу та виводу. Оскільки планується створювати артефакти специфікації на основі тих чи інших шаблонів, то даний інструмент допоможе перетворити шаблонний текст на реальний текст специфікації (описання моделей, сценаріїв тощо).

Перед тим як описувати шаблони, слід зазначити, що для їх написання була використана мова шаблонів Handlebars. В більшості випадків дану мову використовують для створення HTML-сторінок, але вона підтримує й інші мови програмування.

Таким чином, шаблон для описання value-об’єктів мав би наступний вигляд (ця ж структура використовується під час виклику методу *getSchema()*):

```
describe {{type}} {{name}}:
      {{#each values}}
            {{this.name}} is {{this.type}};
      {{/each}}
end
```

В даному випадку type – це тип описуваного об’єкта, name – це назва об’єкта, а values - це список атрибутів об’єкта, що містять ім’я та тип атрибуту.

Завдяки шаблонам ми маємо можливість описувати майже будь-яку структуру, що в свою чергу надає можливість розширювати функціонал автоматичного генератора коду в майбутньому.

Класс фабрики для генерації моделей має наступний вигляд:

```
class ModelFactory<T, U> {
      constructor(template: Template) {
```

```
            this.template = template;
        }

        buildArtifact(value: T): U {
            this.template.getSchema(value);
        }
}
```

Уявімо, що ми хочемо описати модель нотатки. Передавши необхідні дані до методу *buldArtifact*, отримаємо наступний текст, який може використовуватися як не технічними фахівцями, так і в якості вхідних даних для автоматичного генератора коду. Текст наведено далі:

```
describe value Note:
    - title is string;
    - text is string;
end
```

Тип *value* відповідає типу з такою ж назвою в предметно-орієнтованому проєктуванню, тобто це така сутність, що має певний набір атрибутів. Завдяки шаблонам та фабрикам ми отримаємо вхідні дані для автоматичного генератора, але, оскільки текст артефактів максимально наближено до звичайного тексту, артефакти специфікації можна описувати власноруч. Таким чином можна описати багато інших об’єктів з реального життя – місця в літаку, гральні картки, ролі тощо.

Отже, вхідні дані для автоматичного генератора коду отримано. Більш того, ці дані є більш-менш структуровані, що є необхідною умовою для нашого генератора.

Наступним етапом є перетворення об’єктів, описаних в специфікації, на об’єкти кінцевих мов програмування. З точки зору користувача все, що необхідно зробити для того, щоб отримати кінцевий код, необхідно виконати наступну команду:

```
acg gen -path=/path/to/spec -target=desired_lang
```

Користувач вказує шлях до специфікації (згенерованої або написаної власноруч) та бажану мову програмування. Використовуючи вищевказаний приклад із нотаткою, отримаємо наступні результати:

Таблиця 4.1 – Згенерований код для різних мов програмування

Мова	Код

Мова специфікації	`describe value Note:`
	`  - title is string;`
	`  - text is string;`
	`end`

TypeScript	`class Note {title: string;text: string;}`
JavaScript	`class Note {title;text;}`
Golang	`type Note struct {title string; text string;}`
SQL	`CREATE TABLE Note (title VARCHAR(255), text VARCHAR(255))`

Слід зазначити, що враховуючи той факт, що частина мов має динамічну типізацію, то й типи зі специфікації не будуть відображені в кінцевому коді. В майбутньому можна додати інформацію про типи, використовуючи коментарі.

Отже, даний генератор вміє створювати описані об’єкти, тобто вже містить у собі функціонал, що мають багато інших аналогів.

Але вміння генерувати класи чи структури не є достатнім для автоматичного створення повноцінних програм. Уявімо, що ми розробляємо застосування для нотаток. Наша програма вже знає про існування про безпосередньо об’єкт нотатки, але необхідно додати операції для створення, видалення та отримання нотаток. Для цього ми можемо використати сценарії, яки створено вищевказаним інструментом для створення артефактів специфікації.

Сценарії зазвичай описують мовою Gherkin (особливо при використанні поведінково-орієнтованого проєктування), і типова специфікація мала б наступний вигляд:

```
SCENARIO: User creates a *Note*
      GIVEN: User inputs the desired title and text
      WHEN: User submits the note information
      THEN: The note is created

SCENARIO: User deletes a *Note*
      GIVEN: User finds a note to delete
      WHEN: User presses the delete button
      THEN: The note is deleted
```

```
SCENARIO: User reads a *Note*
      GIVEN: User sees a list of notes
      WHEN: User selects note
      THEN: Note information is displayed
```

Хоча й специфікація виглядає доволі просто, даний автоматичний генератор коду не зможе створити усі необхідні частини коду та згенерувати повністю готову програму. Як було зазначено в початку цієї роботи, мета автоматизації – зменшити втручання людини в той чи інший процес, а не виключити дії людини взагалі. Тому виникає логічне питання, яке полягає в тому, що в даному випадку автоматичний генератор коду може взяти на себе і чи можемо ми змінити щось у тексті специфікації, щоб зробити її структуру більш чіткою, адже чи більш чітка структура – тим більше шансів правильно розпізнати ключові слова та згенерувати відповідний програмний код.

Перший варіант – можна використати більш коротку назву для сценаріїв, у нашому випадку це може бути *Note: Read, Note: Delete, Note: Create.* Це полегшує роботу для автоматичного генератора коду, але трохи втрачається наближеність до реальної мови. Інший варіант – поширити набір термінів Gherkin-нотації та Зберігати необхідну інформацію там. Було прийнято рішення використовувати саме цей варіант. Таким чином, сценарій для створення нотатки матиме наступний вигляд:

```
@(VALUE: Note, OPERATION: Create)
SCENARIO: User creates a *Note*
      GIVEN: User inputs the desired title and text
      WHEN: User submits the note information
      THEN: The note is created
```

Було додано два ключові слова – *VALUE* та *OPERATION*. Вони майже ніяк не впливають на текст специфікації, але додають ту саму необхідну інформацію, що так потрібна автоматичному генератору коду. Таким чином, шаблон для створення сценарію матиме наступний вигляд:

```
@(VALUE: {{valueName}}, OPERATION: {{operationName}})
SCENARIO: {{scenarioText}}
      GIVEN: {{givenText}}
      WHEN: {{whenText}}
```

```
THEN: {{thenText}}
```

Сценарії дозволять згенерувати сервіси, що будуть працювати із об’єктам, зазначеними в сценарії. Згенерований код матиме наступний вигляд:

Таблиця 4.2 – Згенерований код сервісів для різних мов програмування

Мова	Код

Мова специфікації	`@(VALUE: Note, OPERATION: Create)`
	`SCENARIO: User creates a *Note*`
	`GIVEN: User inputs the desired title and text`
	`WHEN: User submits the note information`
	`THEN: The note is created`

TypeScript	`class NoteService { create(note: Note){} }`
JavaScript	`class NoteService { create(note) {} }`

Golang	`type NoteService struct{}`
	
	`func (n *NoteService) create(note Note) {}`

SQL	`INSERT INTO Note VALUES()`

Слід зазначити, що в деяких випадках (а саме в випадку з SQL) генерується не код сервісу, а майже повна операція. Кінцевий код дуже залежить від того, як буде реалізовано адаптер мови специфікації до кінцевої мови програмування.

Тим не менш, реалізацію методів все одно має реалізувати безпосередньо розробник програмного забезпечення. В майбутньому можливо додавати якусю дуже базову реалізацію, але немає жодних гарантій що вона буде робити саме те що необхідно програмісту чи бізнесу. Але дана генерація надає одну дуже важливу перевагу – вона гарантує те, що всі методи та сценарії, які вказані в сценаріях, будуть реалізовані в програмному коді. Це досягається також завдяки валідатору, який може перевірити, чи реалізовано всі сценарії з специфікації в фінальному застосуванні.

Але існує не так багато застосувань, де існують лише операції створення та видалення. На даному етапі специфікація ніяк не відображає ти чи інші обмеження, що може мати система. Якщо розглядати застосування з нотатками, то треба описати певні правила для їх створення (довжина тексту, максимальна

кількість тощо). Окрім того, іноді буває, що необхідно описати певні додаткові дії, що не стосуються безпосередньо логіки застосування, але які все ж необхідні (вимірювання швидкості роботи методу, логування, валідація тощо). В розрізі даної роботи даний функціонал було вирішено реалізувати завдяки аспектно-орієнтованому програмуванню. Іншими словами, було вирішено, що у той час, як головні дії застосування будуть описані через сценарії, додаткові дії будуть описані через аспекти.

Слід зазначити, що аспекти (в більшості випадків) – це доволі технічні дії, які було б важко описати не технічним фахівцям. Навіть якщо, наприклад, валідацію можна описати звичайною людською мовою – їх буде дуже важко структурувати таким чином, що вони залишались наближеними до людської мови та мали чітку структуру. На жаль, в існуючій реалізації автоматичного генератора коду балансу досягнути не вдалося, тому було вирішено, що описання аспектів буде відповідальністю розробника програмного забезпечення.

Прикладом аспекту може бути перевірка того, що ми видаляємо лише ті нотатки, які існують. Оновлений пункт специфікації матиме наступний вигляд:

```
@(VALUE: Note, OPERATION: Delete)
SCENARIO: User creates a *Note*
      GIVEN: User finds a note to delete
      WHEN: User presses the delete button
      THEN: The note is deleted
ASPECTS:
      BEFORE: validate
```

Тут вказаний єдиний аспект сценарію, що означає, що перед безпосереднім виконанням сценарію необхідно виконати метод validate. Тут знов слід зазначити, що деталі реалізації будуть задані безпосередньо розробником, але елементи специфікації будуть гарантовано відображені в кінцевому коді.

Фінальні приклади коду наведені в таблиці 4.3

Таблиця 4.3 – Згенерований код аспектів для різних мов програмування

Мова	Код

Мова специфікації	`@(VALUE: Note, OPERATION: Delete)`
	`SCENARIO: User creates a *Note*`
	`GIVEN: User finds a note to delete`
	`WHEN: User presses the delete button`
	`THEN: The note is deleted`
	`ASPECTS:`
	`BEFORE: validate`

TypeScript	`class NoteService { delete(note: Note){this.validate(note);} validate(note: Note){} }`
JavaScript	`class NoteService { delete(note) {this.validate(note);} validate(note){} }`

Golang	`type NoteService struct{}`
	
	`func (n *NoteService) delete(note Note) {`
	`n.validate();`
	`}`
	`func (n *NoteService) validate(note Note) {}`

Таким чином, автоматичний генератор коду покриває всі необхідні пункти специфікації, а саме опис сутностей системи, їх основні та додаткові дії.

5 ВИКОРИСТАННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Перш за все слід зазначити, що ті підходи, що були використані при розробці автоматичного генератора коду (аспектно-орієнтоване та поведінково-орієнтоване програмування, предметно-орієнтоване проєктування) були розроблені не в навчальних закладах чи лабораторіях. Ці підходи були розроблені спеціалістами, які працюють в так званій enterprise розробці, тобто тими людьми, хто вирішує проблеми бізнесу. Дані підходи зарекомендували себе як гарні інженерні практики. Мета цих підходів – об'єднання та структуризація технічних та не-технічних фахівців. Але скільки людей – стільки і думок, тому не завжди всім учасникам процесу розробки вдається знайти консенсус. Дослідження та розробка автоматичного генератора коду була спрямована на подолання цієї проблеми.

З одного боку, даний генератор коду гарантує то, що вся наявна інформація в специфікації інформації матиме своє відображення в коді, тобто сутності та процеси існуючого бізнесу будуть відображені в програмному коді. З іншого, автоматичний генератор коду не має ніякої можливості генерувати безпосередньо реалізацію методів, відповідних до дії, зазначених в специфікації. Людська мова хоча й має певну структуру, але вона значно гнучкіша в порівнянні з будь-якою мовою програмування, а, як усім відомо, текстова репрезентація коду має певні ключові слова і проходить певні перетворення і врешті решт стає машинним кодом.

Можливо, автоматичний генератор коду можна було би поєднати із якимось інструментом машинного навчання, що міг би аналізувати людську мову та створювати певну класифікацію часто використовуваних слів (створити, видалити, зчитати). Але справа в тому, що таку реалізація буде вести себе непередбачувано, адже, як було зазначено раніше, перевірити правильність вихідного коду можна лише при наявності чітко структурованого початкового ходу (в нашому випадку початковий код – це структурована специфікація).

Машинне навчання не може гарантувати такого чіткого перетворення, але могла б (теоретично) замінити ті додаткові ключові слова, що було додано до Gherkin-нотації. Тим не менш, існуючий варіант, де користувачі вказують ці ключові слова власноруч, виглядає більш надійним.

На практиці даний інструмент допомагає чітко відображати специфікацію у коді. Більш того, таким чином можна перевіряти, чи не суперечить специфікація само собі. Як показує досвід, в деяких випадках дивні та неочевидні конструкції програмного коду можуть бути зумовлені зайвою складністю або неточністю специфікації. Використовуючи автоматичний генератор коду, можна швидко перевіряти, як існуюча специфікація “лягає” на програмний код.

Слід зазначити, що розробка програмного забезпечення є доволі творчим процесом, і він не завжди підлягає чітким правилам. При використанні автоматичного генератора коду, можливо, втрачається індивідуальність, але з іншого боку – це надає можливість учасникам процесу розробки програмного забезпечення не витрачати час на написання шаблонного коду, і витратити цей час на покращення специфікації, адже специфікація – це найцінніше знання кожного більш-менш великого проекту.

Буде чесним визнати, що на даному етапі цей інструмент здатний генерувати шаблонний код для доволі простих задач. Генератор коду дозволяє створювати код саме для бізнес-складової застосування, але інфраструктурна частина коду все ще залишається відповідальністю розробника, бо неможливо чітко описати, скажімо, HTTP-сервер, та – більш того – згенерувати відповідний код на бажаній мові програмування, бо кожна мова має не одну реалізацію для вирішення такої задачі. З іншого боку, вищезазначена архітектура дозволяє створювати адаптери, і при певних покращеннях, базові частини інфраструктурного коду також можна буде генерувати автоматично.

В наукових дослідженнях даний інструмент можна використовувати в ті моменти, коли фахівець хоче зосередитися на дослідженні та структуризації певних процесів та сутноей, і після цього швидко створити шаблонний код, яким потім буде доповнюватися безпосередньо реалізацією. Крім того, даний

інструмент можна використовувати в навчальному процесі, адже він допомагатиме навчити гарній звичці, яка полегшить життя розробнику прорамного забезпечення – спочатку думати, а вже потім писати програмний код.

На практиці головна мета цього інструмента – об’єднати знання, вимоги та код у єдине ціле. Він не робить це на всі 100%, але шар логіки – саме це і є об’єднаним знанням – він покриває.

Узагальнюючи, можна сказати, що автоматизація потрібна перш за все там, де чітко проглядаються ті чи інші шаблонні сутності та дії. В інших випадках – власноручна розробка виглядає більш привабливо.

ВИСНОВКИ

В процесі виконання атестаційної роботи було досліджено методи автогенерації програмного коду та створено план розробки та архітектуру застосування для автоматичної генерації програмного коду.

Було проведено аналіз існуючих аналогів, виділено певні категорії інструментів автогенерації коду, вивченні їх переваги та недоліки.

В ході даної роботи було проведено аналіз вимог до продукту, причому однією із вимог стало подолання найбільшого недоліку усіх існуючих аналогів, а саме зосередженість лише на певному аспекті розробки програмного забезпечення.

Застосування було спроєктовно таким чином, щоб у майбутньому ним могли користуватися не технічні фахівці. Це було досягнуто завдяки дослідженню існуючих методів, що наближають не технічних та технічних фахівців один до одного в плані єдності знань щодо програмної системи та предметної області.

Можна зробити висновок, що автоматична генерація коду можлива лише при чіткій структуризації вхідних даних, саме тому було описано структурні аспекти предметно-орієнтованого проєктування, розробки керовану поведінкою та аспектно-орієнтованого програмування. [11]

Окрім того, були обрані інструменти для реалізації застосування з автоматичної генерації програмного коду.

Також було розглянуто галузі застосування автоматизації поза розробки програмного забезпечення, це було зроблено для підтвердження тренду автоматизації праці в усіх аспектах життя людини.

Отримане програмне забезпечення не містить недоліки існуючих аналогів, адже воно є універсальним (тобто надає можливість надавати гнучку специфікацію в якості вхідних даних) та зрозумілим для не технічних спеціалістів, адже власники продукту, бізнес аналітики, проєктні менеджери вже

давно використовують ту нотацію, що була взята за основу для формату вхідного тексту автоматичного генератора програмного коду.

Даний генератор працює достатньо швидко. Це важливий критерій, адже якщо автоматичний генератор коду працював би повільно, то це було б перевагою того, щоб писати код самостійно. Так, наприклад, генерація коду зі специфікації простого застосування для нотаток займає 0.5-1 с.

Автоматичний генератор програмного коду робить саме те, що підпадає під визначення автоматизації – він зменшує втручання людини, але не виключає його. Як було зазначено раніше, даний генератор може створювати лише сутності та шаблони для процесів, але на даному етапі він не надає можливості створювати абсолютно повністю готове програмне забезпечення. Він робить інші важливі речі – надає можливість витрачати менше часу на шаблоний код та зрозумілість не технічним фахівцям. Даний генератор рекомендується використовувати тим командам з розробки програмного забезпечення, які ставлять відповідність між кодом та специфікацією на перше місце. Цей інструмент надає можливість розробнику концентруватися на безпосередньо бізнес-логіці, витрачаючи менше часу на шаблонний код. Також цей генератор чудово підходить для написання так званих доказів концепцій, тобто для перевірок ідей на ранніх етапах розробки програмного забезпечення. Це дозволить перевіряти базову специфікацію на відсутність суперечностей, що є дуже важливим кроком на початковому етапі розробки, бо помилки, допущені на перших етапах, дуже важко виправляти тоді, коли вже написано багато коду. В даному досліджені автоматичний генератор коду розгодався не як заміна розробника, а як помічник під час перших кроків написання програмного забезпечення, помічник, що об’єднує знання команди в єдине ціле, що надає дуже великі переваги для такої команди в майбутньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. История развития автоматизации: Материалы VI Международной студенческой научной конференции [«Студенческий научный форум»] , (Москва, 2014) / Российская Академия Естествознания, 2014. – 25 с.

2. Свейгарт Э. Автоматизация рутинных задач с помощью Python: практическое руководство для начинающих [Текст] / Э. Свейгарт. – М.: Вильямс, 2017. – 573 с.

3. Groover, Mikell. Fundamentals of Modern Manufacturing: Materials, Processes, and Systems [Текст]. / Mikell Groover. – Wiley, 2012. – 1128 с.

4. Бізнес Інсайдер [Електронний ресурс] / Business Insider. – Режим доступу: https://www.businessinsider.com/uber-lyft-are-gaining-even-more-market-share-over-taxis-and-rentals-2018-7 – 01.03.2020 – Назва з екрану.

5. Fischer, Bernd. Logical Foundations for Automated Code Generation / Bernd Fischer. // Estonian Summer School in Computer and Systems Science. – 2006. – №5. – С. 11 – 13.

6. Evans, Eric. Domain-Driven Design. Tackling Complexity in the Heart of Software [Текст]. / Eric Evans. – Addison-Wesley, 2004. – 529 с.

7. Groves, Matthew. AOP in .NET. Practical Aspect-Oriented Programming [Текст]. / Matthew D. Groves. – Manning, 2013. – 296 с.

8. Smart, John. BDD in Action. Behavior-Driven Development for the whole software lifecycle [Текст]. / John F. Smart. – Manning, 2014. – 384 с.

9. Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. [Текст]. –Addison-Wesley, 2015. – 368 с.

10. Reinhard Wilhelm, Helmut Seidl. Compiler Design: Virtual Machines. [Текст]. – Springer, 2011. – 187 с.

11. Паламар, Вячеслав. Дослідження методів автогенерації програмного коду / Вячеслав Паламар. // НАУКОВЕ ЗАБЕЗПЕЧЕННЯ ТЕХНОЛОГІЧНОГО ПРОГРЕСУ XXI СТОРІЧЧЯ. – 2020. – С. 58-62.