Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**РОЗШИРЕННЯ ІНТЕГРОВАНОГО СЕРЕДОВИЩА РОЗРОБКИ ДЛЯ АВТОГЕНЕРАЦІЇ КОДУ**»

Виконав: студент 4-го року навчання,

Освітньої програми «Інженерія Програмного Забезпечення», 121

Архипчук Богдан Олександрович

Керівник: Франків О.О.

Старший викладач

Рецензент

____

Кваліфікаційна робота захищена

з оцінкою

____

Секретар ЕК

____

«____» ___________________ 2024 р.

Київ – 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ

УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики технологій факультету інформатики

ЗАТВЕРДЖУЮ

Викладач кафедри інформатики ,

канд. фіз-мат. наук, доц. ____________ Гороховський С.С.

(підпис)

„____”__________________________2024р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Архипчуку Богдану Олександровичу

факультету інформатики 4 курсу бакалаврської програми

ТЕМА: Розширення інтегрованого середовища розробки для автогенерації коду

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Вступ

Кодогенерація

Обґрунтування підходу до рішення проблеми

Імплементація та аналіз результатів

Висновки

Джерела

Дата видачі „___” _________ 2024 р. Керівник ___________

(підпис)

Завдання отримав___________

(підпис)

ЗМІСТ

АНОТАЦІЯ 7

ВСТУП.. 8

РОЗДІЛ 1 Кодогенерація 10

Основні принципи та характеристики макросів у мові програмування Swift. 10

Проблематика та виклики в імплементації макросів 11

Аналіз альтернативних методів генерації коду та розробки макросів............ 12

Аналіз принципу роботи розширень Xcode 15

Впровадження та аналіз API XcodeKit для розширень.................................... 17

Детальний розгляд Swift Syntax як інструменту для маніпуляції кодом 19

Роль Swift Syntax у процесах автоматизованої генерації коду 21

Висновки розділу.. 23

РОЗДІЛ 2 Обґрунтування підходу до рішення проблеми 27

Обґрунтування вибору макросів у розробці 27

Аналіз вибраних макросів для оптимізації генерації коду.............................. 28

Застосування патернів Swift Syntax для аналізу коду...................................... 35

Висновки розділу.. 36

РОЗДІЛ 3 Імплементація та аналіз результатів 38

Аналіз вихідних проблем та методів їх вирішення.. 38

Структурне проектування програмного забезпечення 39

Реалізація ключових методів у проєкті 40

Процес дизайну та імплементації користувацького інтерфейсу 42

Аналіз отриманих результатів... 44

ВИСНОВКИ 46

ДЖЕРЕЛА 48

Додаток А

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	жовтень			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	жовтень – листопад			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	листопад			
4.	Постановка експерименту, аналіз отриманих результатів наукового дослідження	листопад – березень			
5.	Проміжний контроль виконання роботи	лютий			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	січень – березень			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)				
	Розділ 2 (аналітично-дослідницька частина)				
	Розділ 3 (проектно-рекомендаційна частина)				
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	квітень – початок травня			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	середина травня			

9.	Подання на зовнішню рецензію	середина травня			
10.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративного матеріалу	**до ____ травня**			
11.	Попередній захист кваліфікаційної роботи на засіданні кафедри	**до ____ травня**			
12.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами	**до ____ травня**			
13.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено «___» _________________ 20___ р.

Науковий керівник ___________________ (ПІБ)

Виконавець кваліфікаційної роботи Архипчук Богдан Олександрович (ПІБ)

АНОТАЦІЯ

Дана наукова робота присвячена дослідженню сучасних методів автоматизації генерації коду в програмуванні, особливо з використанням інтегрованих розширень середовища розробки. В роботі розглядаються проблеми та можливості, пов'язані з генерацією коду в умовах зростаючих вимог до ефективності та швидкості розробки програмного забезпечення.

Особливу увагу приділено аналізу інструментів генерації коду та їх інтеграції в середовище Xcode, а також дослідженню мови програмування Swift. У роботі наголошується на необхідності скорочення рутинної роботи розробників за рахунок автоматизації, що може значно підвищити продуктивність і знизити фізичне та емоційне навантаження на програмістів.

Результати дослідження демонструють потенціал нових технологій для розширення можливостей генерації коду, що відкриває шлях для розробки інноваційних рішень у сфері програмування.

ВСТУП

У світлі стрімкого розвитку технологій та постійного зростання вимог до швидкості та якості програмного забезпечення, проблема ефективного використання часу та ресурсів стає все більш актуальною. Величезні обсяги коду, які пишуться щодня, вимагають не тільки значних часових витрат, а й зусиль, що можуть призвести до фізичного та морального виснаження розробників. Така ситуація акцентує важливість пошуку нових підходів для оптимізації розробки, зокрема через механізми автоматичної генерації коду та використання можливостей штучного інтелекту.

Сучасні технології надають потужні інструменти для цього, але їх інтеграція в щоденні практики розробки часто наштовхується на перепони: складність впровадження, страх перед новими технологіями та необхідність глибокого навчання. Помітно, що навіть з наявністю інтегрованих інструментів від провідних технологічних компаній, багато розробників продовжують використовувати традиційні методи, що веде до повторення рутинних завдань.

Метою цієї роботи є розробка та реалізація нового інструменту автогенерації коду, що інтегрується в середовище розробки Xcode та відкриває шляхи для подальших інновацій в автоматизації процесів розробки. Це завдання включає в себе:

- Аналіз наявних інструментів для генерації коду, з особливим акцентом на їх інтеграцію в середовища розробки.
- Вивчення доступних методів інтеграції в середовище Xcode.
- Дослідження специфіки роботи з синтаксисом мови Swift, яка є основною мовою програмування у середовищі Xcode.
- Реалізація прототипу розширення для Xcode, що автоматизує генерацію коду, та аналіз отриманих результатів для оцінювання можливостей подальшого розвитку та вдосконалення інструменту.

Значимість даної роботи полягає не тільки в розробці нових технічних рішень, але й у впливі на загальну продуктивність процесу створення програмного забезпечення, скороченні часу реалізації проектів і, відповідно, зниженні вартості розробки. Автоматизація рутинних процесів здатна звільнити значні ресурси, які можуть бути перенаправленні на розв'язання більш складних і інноваційних задач, що, в свою чергу, може призвести до створення якісно нових продуктів та послуг.

РОЗДІЛ 1 Кодогенерація

Основні принципи та характеристики макросів у мові програмування Swift

З появою можливостей використання макросів у мові програмування Swift у вересні 2023 року, відкриваються нові горизонти для розробників, спрямовані на оптимізацію та ефективність коду. Макроси, як інструмент автоматизації і зменшення дублювання коду, не є новиною для багатьох мов, таких як C# або Java, де вони інтегровані через фреймворки та бібліотеки. Однак, для Swift, який вже відомий своєю чистотою та високою продуктивністю, введення макросів дозволяє досягнути ще більшої гнучкості та масштабованості.

Макроси в Swift працюють шляхом трансформації вихідного коду на етапі компіляції, що дозволяє виконувати складні маніпуляції з кодом без необхідності в ручному дублюванні логіки [9]. Основна концепція полягає у використанні двох основних типів макросів:

@freestanding: макроси, які не залежать від контексту інших об'єктів у коді.

@attached: макроси, які інтегруються із вже існуючими деклараціями і збагачують їх функціональність.

На відміну від макросистем в таких мовах як Rust або Lisp, де макроси можуть маніпулювати синтаксисом і структурою коду на дуже низькому рівні, Swift намагається зберегти баланс між гнучкістю макросів та строгістю типізації та безпечним управлінням пам'яттю, що є характерним для цієї мови.

Для реалізації макросів в Swift використовується Swift Syntax – це бібліотека, яка надає детальне дерево розбору всього коду, дозволяючи макросам аналізувати та модифікувати код на лету під час компіляції. Це включає розширення можливостей коду через створення шаблонів, інтеграцію з існуючим кодом та оптимізацію виконання.

Завдяки макросам, розробники Swift можуть створювати більш виразні та мінімалістичні API, зменшувати повторення коду і підвищувати швидкість

розробки без жертвування продуктивністю. Макроси можуть знайти застосування в областях, де важлива швидкість і гнучкість розробки, таких як веб-розробка, розробка мобільних додатків, а також у створенні широкомасштабних систем, де необхідно уникнути дублювання коду та забезпечити високу продуктивність.

В рамках цієї роботи було прийнято рішення сфокусуватись більше на @attached макросах, адже цілю є створення розширення до інтегрованого середовища розробки, яке буде генерувати код базуючись на вже існуючому функціоналі.

Проблематика та виклики в імплементації макросів

Незважаючи на те, що вже існує спосіб кодогенерації, який начебто мав би закрити потреби в багатьох випадках де відбувається велика кількість повторюваного коду, це не так. Процес написання самих макросів є досить важким і має високий поріг входження задля початку використання. Це пов’язано як і зі складною структурою синтаксису мов програмування так і технологіями, які відрізняються від звичних для розробників інструментів.

Дослідження Eldad Yechiam [14] розглядає цікаву проблематику, яка тісно пов'язана зі складнощами впровадження макросів. У дослідженні аналізується низка чинників, що відштовхують користувачів від активного використання макросів, навіть серед досвідчених програмістів. Це особливо важливо, враховуючи, що макроси могли б значно спростити і автоматизувати рутинні задачі.

Основні труднощі полягають у тому, що спочатку користувачі стикаються з високим порогом навчання та інтуїтивної складності використання макросів, що спричиняє високий відсоток помилок і невдач під час виконання завдань. Такі негативні досвіди знижують мотивацію до подальшого використання макросів, натомість користувачі вибирають більш прості та зрозумілі альтернативи.

Втім, Єхіам також вказує на потенційні переваги використання макросів та пропонує декілька принципів тренування, що можуть сприяти підвищенню зацікавленості та зручності використання макросів. Серед цих принципів:

1. **Використання сценаріїв із реального використання під час тренувань**, щоб показати переваги використання макросів у порівнянні з іншими методами.
2. **Надання плану "запасного варіанту"**, який дозволяє користувачам використовувати макроси у спрощеній формі для забезпечення адекватної продуктивності.
3. **Демонстрація відносної переваги**, яка має на меті підкреслити, що навіть простіші версії макросів є ефективнішими за інші засоби при виконанні монотонних задач.
4. **Зосередження на негайному, а не на віддаленому успіху**, що допомагає учасникам тренувань відчути переваги використання макросів безпосередньо під час навчання.

Ці підходи можуть значно покращити тренувальний процес і зменшити опір при впровадженні макросів, тим самим забезпечуючи їх більш активне використання.

Виходячи з висновків цього дослідження є очевидним, що створення розширення, яке могло б автоматизувати процес написання самих макросів і дозволило б легке їх використання мало б не аби яку цінність серед спільноти розробників та зменшило б той самий поріг входження.

Аналіз альтернативних методів генерації коду та розробки макросів

Перед тим як розробляти свою версію автоматизації для створення макросів, було важливо дослідити вже існуючі рішення, або те як інші дослідники намагались спростити весь процес кодогенерації.

Однією з цікавих робіт з якою я зіштовхнувся під час дослідження було дослідження Ballantyne M., King A. та Felleisen M. “Macros for domain-specific languages. Proceedings of the ACM on programming languages” [1]. Інноваційна архітектура пропонує багатоцільову мову, спеціально розроблену для спрощення створення макросів у різних середовищах програмування. Ця архітектура використовує концепцію специфічних мов доменів (DSL), які успадковують можливості макросів від своїх базових мов, покращуючи розширюваність та інтеграцію DSL у наявні кодові бази.

Основний принцип цього підходу полягає в тому, щоб спростити роботу компілятора, зосередившись на невеликій базовій мові, дозволяючи DSL керувати більш специфічними функціоналами та розширеннями синтаксису. Це не лише оптимізує процес розробки, а й значно знижує складність управління кількома мовами та системами макросів.

Основні особливості та переваги:

Спадкоємність системи синтаксису та розширення - Спадкоємність критичних компонентів, таких як система синтаксису та можливості розширення від базової мови, дозволяє DSL безшовно інтегруватися та підтримувати консистенцію в різних середовищах програмування. Ця модель спадкоємності сприяє створенню чистішого та більш утримуваного коду.

Гнучкі та міцні компілятори DSL - Компілятори DSL розроблені навколо фіксованої базової мови, яка може бути надійно розширена за допомогою макросів. Ця гнучкість дозволяє розробникам адаптувати мову до своїх конкретних потреб без змін основної архітектури.

Повноваження через розширюваність - Розробники отримують можливість самостійно розширювати можливості мови, інтегруючи функції, які спочатку не передбачалися дизайнерами мови. Це особливо корисно у спеціалізованих галузях, де вимоги можуть швидко змінюватися або бути високо специфічними.

Впровадження багатоцільової мови для макросів обіцяє знизити криву навчання для розробників, сприяючи вищому рівню їхнього прийняття. Більше того, така мова могла б стати основою для розширення інтегрованого середовища розробки (IDE), спрямованого на автоматизацію генерації макросів на основі наявних функціональностей, тим самим покращуючи продуктивність розробників і якість програмного забезпечення.

Альтернативним рішенням, яке в сучасному світі є досить актуальним в багатьох сферах це спробувати навчити свою власну модель штучного інтелекту. Саме з такою думкою я зіткнувся з ще одною цікавою роботою Chen M. “Evaluating large language models trained on code” [5]. Вона розглядає застосування моделей, керованих штучних інтелектів (ШІ), таких як Codex, навчених на обширних репозиторіях коду, для генерації фрагментів коду з описів на природній мові або докстрінгів. Хоча ці моделі демонструють значний потенціал у автоматизації генерації коду, вони також підкреслюють значні виклики, особливо в досягненні високої точності та надійності.

Оцінка та аналіз результативності:

Рамки оцінювання - Дослідження вводить метрику **pass@k**, яка оцінює функціональну коректність, а не просто синтаксичну точність. Цей підхід важливий, оскільки він відображає метод оцінки коду розробниками, заснований на функціональності, а не на формі.

Аналіз результативності - Незважаючи на обіцянки, показані моделями ШІ в автоматизації завдань кодування, поширені помилки та часті невідповідності у пропозиціях коду виявляють значні перешкоди. Ці моделі часто зіштовхуються зі складними інструкціями і не можуть послідовно генерувати функціонально правильний код без помилок.

Моделі генерації коду, керовані ШІ, можуть прискорити процеси розробки і зменшити навантаження повсякденних завдань кодування. Вони мають

особливі переваги для складання початкових фрагментів коду, які розробники можуть уточнювати і інтегрувати у більші проекти.

Враховуючи потенціал значних помилок, покладання на ШІ у критичних застосуваннях повинно підходити з обережністю. Наслідки помилок у таких контекстах можуть бути особливо серйозними, що підкреслює необхідність ретельного тестування та валідації коду, згенерованого ШІ.

Інтеграція алгоритмів ШІ для генерації коду в середовища розробки могла б радикально змінити практики розробки програмного забезпечення, пропонуючи як нові можливості, так і виклики. Хоча технологія має трансформаційний потенціал, її впровадження має бути ретельно керованим, щоб мінімізувати ризики та повністю використовувати її можливості для покращення робочих процесів розробки.

Як висновок з цих двох проаналізованих робіт можна точно сказати, що використання ШІ є занадто ризикованим, адже складність потрібного результату та коду експоненційно збільшують відсоток помилкового рішення, а також потребує дуже великої кількості валідації. Створення свого власного синтаксису для написання макросів в різних мовах програмування звучить як цікаве рішення і дає більше розуміння в те, як варто уніфікувати майбутнє розширення, можливо не для різних мов, але точно для різних ситуацій та зробити його більш відкритим до покращень у майбутньому

Аналіз принципу роботи розширень Xcode

Xcode Extensions - це інструменти, призначені для розширення можливостей Xcode, інтегрованого середовища розробки від Apple. Представлені в Xcode 8, ці розширення знаменують собою зрушення в підході Apple до більш контрольованої екосистеми, яка надає пріоритет безпеці та стабільності, а не широкій кастомізації. На відміну від попередньої відкритої моделі, яка дозволяла стороннім плагінам мати значний доступ до базової

системи, поточна структура обмежує роботу розширень у середовищі «пісочниці», значно обмежуючи їхню сферу застосування та доступ до системи.

Розширення Xcode в першу чергу дозволяють маніпулювати текстом у редакторі коду. Вони активуються через меню Xcode і працюють виключно з поточним відкритим файлом. Такий сфокусований підхід дозволяє розробникам виконувати такі завдання, як

1. Форматування коду: Автоматичне форматування коду відповідно до попередньо визначених стилів.
2. Рефакторинг: Спрощення та покращення існуючого коду без зміни його поведінки.
3. Вставка фрагментів: Швидке додавання часто використовуваних або попередньо визначених фрагментів коду в поточний документ.

Ці можливості, хоча і обмежені, можуть значно прискорити виконання рутинних завдань і підвищити точність кодування. Розширення Xcode можна поширювати двома основними способами:

1. Mac App Store: Розширення, що розповсюджуються через App Store, проходять перевірку Apple, яка гарантує безпеку та сумісність, але також може затримувати оновлення та обмежувати гнучкість розповсюдження.
2. Незалежне розповсюдження: Дозволяє розробникам більше контролювати цикли випуску та оновлення розширень. Однак цей метод вимагає від користувачів ручного встановлення розширень, часто з використанням налаштувань безпеки macOS, що може стати бар'єром для менш технічно підкованих користувачів.

Обидва методи розповсюдження мають свої переваги та проблеми, і вибір між ними залежить від цілей розробника та потреб цільової аудиторії. Незважаючи на свої обмеження, розширення Xcode можуть бути потужним інструментом в арсеналі розробника. Практичні застосування включають в себе автоматизація рутинних завдань кодування може значно скоротити час розробки і допомогти

зосередитися на вирішенні більш складних проблем та підвищення якості коду через стандартизування форматування коду та застосовуючи рекомендації щодо стилів, що допомагає підтримувати узгодженість коду, особливо в командній роботі.

Налаштування під конкретні потреби: Розробники можуть створювати власні розширення, які відповідають унікальним потребам їхніх проектів, наприклад, генерувати шаблонний код для повторюваних шаблонів модулів або специфічних архітектурних фреймворків.

Особливо просунуте використання розширень Xcode - це автогенерація коду на основі даних, введених користувачем. Це передбачає отримання вхідних даних від розробника, їх обробку у межах можливостей розширення та створення або зміну коду відповідно до них. Хоч це і обмежено контекстом поточного файлу, такі розширення можуть значно прискорити процес кодування, автоматизуючи повторювані або складні шаблони коду.

На закінчення, хоча розширення Xcode пропонують вужчу сферу застосування у порівнянні з відкритими системами плагінів минулого, вони все ще забезпечують значну цінність, підвищуючи продуктивність розробників та підтримуючи високий рівень безпеки та стабільності. Розуміння того, як ефективно розробляти, поширювати та використовувати ці інструменти, є ключем до максимізації їхніх переваг у середовищі Xcode.

Впровадження та аналіз API XcodeKit для розширень

XcodeKit є невід'ємною частиною розробки розширень редактора коду Xcode. Він надає API, який розробники використовують для маніпулювання текстом коду, пропонуючи такі функції, як форматування коду, вставка шаблонних блоків коду та виконання базового рефакторингу. Однак, дизайн XcodeKit та його реалізація в середовищі пісочниці підкреслюють делікатний

баланс, який Apple прагне підтримувати між функціональністю та безпекою системи.

Обмеження XcodeKit:

XcodeKit обмежує дії з розширеннями виключно файлом, який наразі відкрито у редакторі. Таке обмеження області видимості суттєво обмежує можливості розширення виконувати більш цілісний аналіз проекту або перетворення, які охоплюють декілька файлів або весь проект. Така ізоляція доступу до файлів є насамперед засобом безпеки, призначеним для запобігання потенційно шкідливій взаємодії розширень із системними файлами або іншими компонентами проекту, яка може скомпрометувати середовище користувача.:

Ще одним суттєвим обмеженням є відсутність у XcodeKit доступу до ширшого контексту проекту та файлової системи. API не надає методів для аналізу залежностей між модулями або загальної структури проекту. Це особливо обмежує можливості розширень, призначених для оптимізації кодових баз шляхом розуміння їх структури та взаємозалежностей. Розробникам часто доводиться вдаватися до зовнішніх інструментів або складних обхідних шляхів, що може ускладнити процес розробки та зменшити безшовну інтеграцію розширень з Xcode.

Розширення Xcode, що працюють у пісочниці, ізольовані від решти системи, що хоча і знижує ризики безпеки, запобігаючи доступу розширень до критично важливих системних ресурсів, але також обмежує функціональність, яку ці розширення можуть досягти. Наприклад, операції, які вимагають доступу до мережі або модифікації системних файлів, не можуть бути виконані лише за допомогою XcodeKit.

Скуте середовище, створене цими обмеженнями, вимагає від розробників прийняття творчих стратегій для обходу цих бар'єрів без шкоди для політики безпеки, що застосовується у пісочниці. Ці методи можуть включати використання вторинних скриптів або служб, які працюють за межами Xcode,

але взаємодіють з розширеннями за допомогою затверджених методів обміну даними. Крім того, розуміння цих обмежень є важливим для розробки розширень, які є одночасно корисними та сумісними з суворими заходами безпеки Apple.

Незважаючи на ці обмеження, майбутній розвиток XcodeKit та його API має потенціал для створення більш складних інструментів для розширень Xcode. Apple може розширити API, щоб безпечно включати більше інформації по всьому проекту, зберігаючи при цьому цілісність середовища пісочниці. Покращення могли б включати контрольований доступ до ширшого обсягу файлів проекту або безпечні методи аналізу структури проекту без витоку конфіденційних даних.

На завершення, хоча XcodeKit надає надійну платформу для розробки розширень, його поточні можливості та обмеження обмежують обсяг того, чого розробники можуть реально досягти. Глибше розуміння його операційних обмежень і потенційної еволюції має вирішальне значення для розробників, які прагнуть оптимізувати використання цього інструменту в жорстко регульованому середовищі. Оскільки Apple продовжує розвивати XcodeKit, залишається значна можливість для розширення цих API, щоб пристосувати їх до більш потужних та інтегрованих інструментів розробки.

Детальний розгляд Swift Syntax як інструменту для маніпуляції кодом

Swift Syntax - це потужна бібліотека Swift, призначена для розбору, аналізу, генерації та програмного перетворення вихідного коду Swift. Вона вийшла з основного репозиторію мови Swift у серпні 2017 року як незалежний проект, заснований на libSyntax. Таке відокремлення підкреслює його спеціалізовану роль у роботі з синтаксисом Swift-коду, на відміну від семантичного аналізу чи перевірки типів, якими керують інші компоненти компілятора Swift.

Основна корисність Swift Syntax полягає у його здатності працювати безпосередньо з абстрактним синтаксичним деревом (AST), яке генерується компілятором Swift під час початкового синтаксичного аналізу вихідного коду. АСТ являє собою структуровану, ієрархічну інтерпретацію коду, яка є більш абстрактною, ніж сире текстове представлення. Маніпулюючи AST, Swift Syntax полегшує тип редагування вихідного коду, відомий як «структуроване редагування», який виходить за рамки простої маніпуляції з текстом, дозволяючи здійснювати більш складні перетворення, такі як заміна ідентифікаторів, коригування викликів методів або комплексне переформатування відповідно до визначених правил.

Дизайн Swift Syntax не передбачає розуміння семантики чи інформації про тип, що є свідомим обмеженням. Він працює суто на синтаксичному рівні, на відміну від таких інструментів, як SourceKit, які надають багатшу контекстну інформацію про код. Такий фокус робить Swift Syntax особливо придатним для завдань, які вимагають прямих і заснованих на правилах маніпуляцій з самою структурою коду, таких як форматування і лінтування. Ці завдання є важливими для підтримки якості та узгодженості коду, особливо у великомасштабних середовищах розробки програмного забезпечення.

Цікавим аспектом Swift Syntax є його інтеграція в ширші інструментальні екосистеми, такі як Swift Migrator, який допомагає розробникам оновлювати кодові бази Swift між версіями мови. Крім того, його постійний розвиток і впровадження як в Apple, так і за її межами, вказує на його важливу роль у середовищі програмування на Swift.

Робота Swift Syntax фундаментально пов'язана з процесом компілятора Swift. Компілятор перетворює Swift-код у виконуваний машинний код у кілька етапів, починаючи з синтаксичного аналізу коду для генерації AST. Swift Syntax взаємодіє з цим AST, дозволяючи досліджувати та перетворювати Swift-код без його компіляції у машинний код. Ця можливість дозволяє розробникам писати

інструменти та скрипти, які можуть програмно змінювати код Swift, забезпечуючи автоматизацію та точність, яких бракує ручним методам.

Отже, Swift Syntax - це спеціалізований інструмент, який покращує екосистему Swift, надаючи надійні, програмно доступні методології для маніпуляцій з вихідним кодом. Його здатність розбирати і трансформувати код на синтаксичному рівні - без заглиблення в більш глибокий семантичний аналіз - робить його цінним активом для розробників, які зосереджені на якості коду, міграції та трансформації завдань. Траєкторія його розвитку та інтеграція з ключовими інструментальними платформами підкреслюють його важливість і потенціал для ширшого впровадження в майбутньому.

Роль Swift Syntax у процесах автоматизованої генерації коду

Swift Syntax розбирає вихідний код Swift на абстрактне синтаксичне дерево (AST), яке представляє ієрархічну структуру коду. Це дерево розбиває код на ряд вузлів, кожен з яких представляє різні синтаксичні елементи, такі як декларації, оператори та вирази. Маніпулюючи цими вузлами, розробники можуть програмно змінювати вихідний код, що є набагато надійнішим і менш схильним до помилок методом, ніж текстове редагування.

Основна перевага використання Swift Syntax для генерації коду полягає в його точності та безпеці. На відміну від текстової генерації коду, яка схильна до помилок через свою залежність від маніпуляцій з рядками, Swift Syntax гарантує, що всі перетворення відповідають синтаксичним правилам мови Swift, таким чином запобігаючи виникненню синтаксичних помилок. Крім того, оскільки Swift Syntax відображає останні синтаксичні зміни з кожним випуском компілятора Swift, він природно адаптується до еволюції мови, що робить його особливо придатним для перспективних інструментів, які генерують або маніпулюють кодом Swift.

Swift Sourcery [15] використовує Swift Syntax для розширення можливостей генерації коду. Використовуючи Swift Syntax як основу синтаксичного аналізу, Sourcery може ефективно і точно інтерпретувати та трансформувати Swift-код. Ця інтеграція дозволяє Sourcery пропонувати такі функції, як автоматична генерація макетів, реалізація таких протоколів, як AutoMockable, і генерація повторюваних шаблонів коду на основі визначених користувачем шаблонів.

Наприклад, розглянемо задачу створення макетів об'єктів для тестування. Традиційно це вимагає ручного написання великого коду, який імітує поведінку протоколів або класів, що тестуються. Однак зі Swift Sourcery розробники просто анотують протокол за допомогою AutoMockable, а Sourcery автоматично генерує всю необхідну інфраструктуру для імітації, використовуючи Swift Syntax для розбору і генерації відповідних вузлів AST. Це не тільки економить час, але й забезпечує узгодженість всіх імітацій.

Пряма маніпуляція Swift Syntax з AST зменшує накладні витрати, пов'язані з генерацією та підтримкою великих обсягів коду. Sourcery може генерувати складні структури коду за лічені секунди - завдання, яке в іншому випадку зайняло б години при ручному виконанні. Абстрагуючись від процесу генерації коду, Swift Syntax гарантує, що весь згенерований код відповідає одному формату та структурі, мінімізуючи розбіжності та потенційні людські помилки у коді, написаному вручну. З розвитком Swift, Swift Syntax оновлюється, щоб відображати нові синтаксичні структури та патерни. Це гарантує, що Sourcery залишається сумісним з останніми розробками мови Swift, тим самим захищаючи згенерований код від майбутніх змін у мові. Оскільки Swift Syntax працює в межах синтаксичних правил мови Swift, код, згенерований Sourcery, з меншою ймовірністю містить синтаксичні помилки, що покращує загальну якість коду і скорочує час налагодження.

Отже, використання Swift Syntax у Swift Sourcery ілюструє практичні переваги інструментів синтаксичної маніпуляції у сучасній розробці

програмного забезпечення. Він не тільки спрощує завдання розробника, автоматизуючи повторювані завдання кодування, але й підвищує надійність і ремонтопридатність кодової бази. Оскільки ціль цієї роботи мати можливість легко маніпулювати перетвореннями вихідного коду очевидно що Swift Syntax є основою для створення власного інтегрованого розширення для автогенерації коду.

Висновки розділу

Вивчення можливостей макросів у мові програмування Swift відкриває перед розробниками нові перспективи для оптимізації та автоматизації процесів кодування. Впровадження макросів у Swift дозволяє значно зменшити дублювання коду та підвищити гнучкість розробки. На особливу увагу заслуговують макроси @attached, які інтегруються з існуючими деклараціями, збагачуючи їх функціональність і сприяючи створенню виразних API.

Однак реалізація макросів стикається з певними труднощами, включаючи високий поріг входу для розробників через складність синтаксису і технологічні проблеми. Дослідження Eldad Yechiam [14] вказує на низку факторів, які ускладнюють широке впровадження макросів навіть серед досвідчених програмістів. Пропонуються конкретні принципи навчання, які можуть сприяти кращому розумінню та ефективному використанню макросів, у тому числі за допомогою реальних сценаріїв і надання запасних планів для спрощення процесу.

Поряд з класичними методами генерації коду розглядається можливість використання штучного інтелекту, як показано в дослідженні Chen M. [5] Хоча такі системи показують багатообіцяючі результати в автоматизації написання коду, вони також створюють значні проблеми в точності та надійності. Основними перевагами моделей штучного інтелекту є здатність швидко

генерувати початкові фрагменти коду, але їх використання в критично важливих додатках вимагає обережності через ризик значних помилок.

Ці дослідження підкреслюють необхідність розробки розширення, яке могло б спростити процес створення макросів, а також зменшити кількість повторюваного коду. Розширення Xcode, хоча і пропонують обмежені можливості в порівнянні з системами плагінів з відкритим вихідним кодом минулого, все ще відіграють значну роль у спрощенні процесів розробки. Вони надають корисні інструменти для автоматизації, форматування коду, рефакторингу та вставки шаблонів, що допомагає розробникам економити час і зосередитися на більш складних завданнях.

Незважаючи на обмеження в інтерактивності файлів і доступі до контексту проекту, XcodeKit забезпечує міцну основу для створення надійних розширень. Це ставить завдання не тільки використовувати доступні функції, але й шукати творчі способи обійти обмеження, використовуючи зовнішні інструменти та сервіси для досягнення ширших цілей розробки.

У майбутньому розвиток API XcodeKit може змінити ландшафт розширень, пропонуючи більш гнучкі та потужні інструменти для можливостей Xcode. Очікується, що Apple продовжить розширювати ці можливості, створюючи більш комплексні та інтегровані рішення, які підвищать загальну ефективність і продуктивність процесів розробки. Нарешті, глибоке розуміння технічних характеристик і обмежень XcodeKit є ключем до ефективного використання розширення для оптимізації вашої роботи. Таке розуміння дозволяє не тільки максимізувати поточні можливості, але й адаптуватися до майбутніх змін, які відбуваються в рамках цієї технології.

Використання синтаксису Swift при розробці розширення Xcode для автоматичної генерації макросів на основі даних, введених користувачем, має кілька суттєвих переваг. Використовуючи Swift Syntax, розширення може ефективно аналізувати та маніпулювати вихідним кодом Swift на синтаксичному

рівні, забезпечуючи надійну основу для автоматизації точних і безпечних перетворень коду. Здатність Swift Syntax безпосередньо взаємодіяти з абстрактним деревом синтаксису (AST) є особливо корисною для цієї програми. AST надає детальне і структуроване представлення коду, що дозволяє виконувати складні маніпуляції, які виходять за рамки простого редагування тексту. Ця можливість має вирішальне значення для розширення Xcode, призначеного для створення макросів, оскільки вона гарантує, що перетворення є синтаксично правильними і відповідають суворим правилам безпеки типів і синтаксису Swift.

Автоматизуючи ці перетворення, розширення може допомогти прискорити цикли розробки, зменшити кількість помилок ручного кодування та підвищити ефективність. Крім того, Swift Syntax підтримує структуроване редагування, що дозволяє розширенню реалізовувати складні перетворення коду на основі даних, введених користувачем. Це може включати вставку попередньо визначених фрагментів коду, модифікацію існуючого коду для включення макрофункцій або динамічне налаштування коду відповідно до конкретних вимог розробника. Структурована природа такого редагування гарантує, що весь код, згенерований розширенням, зберігає високий рівень читабельності, що є критично важливим для великомасштабної розробки програмного забезпечення.

Ще однією ключовою перевагою використання Swift Syntax в цьому контексті є те, що він йде в ногу з постійним розвитком мови Swift. Оскільки Swift продовжує розвиватися, Swift Syntax отримує оновлення, які відображають зміни в синтаксисі та особливостях мови. Ця адаптивність гарантує, що розширення Xcode залишається функціональним і актуальним навіть після виходу нових версій Swift, тим самим захищаючи інструмент від майбутніх мовних змін.

Таким чином, використання синтаксису Swift для розробки розширення Xcode для автоматичної генерації макросів на основі вводу користувача пропонує потужний підхід до підвищення продуктивності розробників та якості

коду. Пряма взаємодія з AST, підтримка структурованого редагування та адаптивність до еволюції мови роблять синтаксис Swift ідеальним вибором для створення складних, надійних та перспективних інструментів розробки в середовищі Xcode. Ця інтеграція не тільки спрощує процес розробки, але й збагачує можливості Xcode, надаючи розробникам передові інструменти для автоматизації та оптимізації робочих процесів кодування.

РОЗДІЛ 2 Обґрунтування підходу до рішення проблеми

Обґрунтування вибору макросів у розробці

При розробці цього проекту стратегічний вибір на користь макросів зумовлений їхньою високою гнучкістю та ефективністю в автоматизації повторюваних завдань. Це важливий аспект сучасних процесів розробки програмного забезпечення, де метою є максимізація ефективності та мінімізація помилок. Вибираючи макроси, ми вирішуємо поширену проблему рутинного коду, який може бути трудомістким і схильним до помилок при написанні вручну. Макроси полегшують автоматичну генерацію складних блоків коду з простих шаблонів, значно скорочуючи ручні зусилля і підвищуючи загальну продуктивність.

Однією з основних проблем при розробці програмного забезпечення є необхідність ефективного управління декларативними структурами коду, що часто призводить до дублювання коду і ручних виправлень, які забирають значний час розробки. Тут макроси слугують потужним інструментом. Вони автоматизують заміну цих структур і дозволяють генерувати класи і методи на основі метаданих, динамічно адаптуючись до контексту виконання. Ця можливість дозволяє розробникам переключити свою увагу з повторюваного управління кодом на більш творчі та стратегічні аспекти розвитку проекту, тим самим оптимізуючи використання ресурсів і підвищуючи інноваційний потенціал проекту.

Крім того, макроси особливо корисні у великомасштабних проектах завдяки своїй здатності створювати код, який залишається незалежним від жорстко закодованих значень. Ця характеристика має вирішальне значення для підтримки високої гнучкості та масштабованості проекту, дозволяючи створювати модульні рішення, які легко інтегруються в різні частини проекту без необхідності значного переписування або глибокого перегляду коду. Це не

тільки прискорює розробку, але й гарантує, що кодова база залишається адаптованою до майбутніх змін або вдосконалень.

Ще однією значною причиною для використання макросів є зниження вхідного бар'єру для розробників при роботі зі складними технологічними рішеннями. Високий рівень абстракції та автоматизації, який забезпечують макроси, дозволяє розробникам використовувати розширені можливості мови програмування, не заглиблюючись у складнощі їхньої реалізації. Такий підхід спрощує процес розробки і скорочує час, необхідний для навчання та інтеграції нових членів команди, тим самим прискорюючи загальне просування проекту.

З точки зору якості коду, макроси відіграють важливу роль у мінімізації помилок. Автоматизація, яку вони забезпечують, гарантує послідовне застосування стандартів кодування у всіх частинах проекту, тим самим зменшуючи ймовірність помилок, пов'язаних з людським фактором. Ця узгодженість безпосередньо пов'язана з якістю кінцевого продукту, оскільки допомагає підтримувати високі стандарти цілісності та надійності коду.

Тому використання макросів - це не просто технічний вибір, а стратегічне рішення, яке значно підвищує продуктивність розробки, покращує масштабованість і гнучкість проектів, а також зменшує час, що витрачається на ручні операції з кодом.

Аналіз вибраних макросів для оптимізації генерації коду

MethodMacroCreator служить для створення макросів, які розширюють декларації в коді з допомогою певного методу, вибраного користувачем. Цей клас використовується для генерації **MemberMacro**.

Метод MacroCreator починає свою роботу з отримання основних параметрів: вхідного коду (inputCode), назви модуля (moduleName) і назви макросу (macroName). Ці параметри є основоположними для процесу створення макросу, визначаючи специфіку того, як метод буде трансформовано і застосовано у ширшому масштабі проекту.

Спочатку я перевіряю повноту і правильність цих вхідних даних. Я перевіряю, що inputCode не порожній, і переконуюся, що moduleName і macroName є дійсними. Цей крок є важливим, щоб уникнути ускладнень під час процесу розширення макросів і переконатися, що згенерований код базується на повних і надійних даних.

Після перевірки вхідних даних я обробляю inputCode за допомогою структури DeclSyntax, щоб визначити його структуру і тип. MethodMacroCreator спеціально розроблений для обробки оголошень функцій, перетворюючи їх на розширені версії, які можна динамічно інтегрувати в кодову базу. Якщо вхідні дані не є функцією, я видаю помилку, щоб виділити цей несподіваний тип вхідних даних.

Ядро трансформації передбачає створення макросу, який інкапсулює оригінальний код функції і плавно інтегрує його в потрібний модуль. Для кожного допустимого входу я створюю оголошення макросу, яке включає повну реалізацію функції, як зазначено у вхідному коді inputCode.

Цей макрос структурований всередині MemberMacro, що дозволяє приєднати перетворений метод до відповідного контексту в додатку. Роль макросу полягає в тому, щоб динамічно вставляти наданий код функції всюди, де застосовується макрос, розширюючи функціональність цільового об'єкта коду без необхідності ручного переписування або значних модифікацій.

Макрос MemberMacro у Swift - це тип макросу, призначений для динамічного додавання нових членів, таких як методи або властивості, до існуючих оголошень, таких як класи, структури або переліки даних, до яких він приєднаний. Ця можливість дозволяє нам розширювати функціональність цих типів, не змінюючи безпосередньо вихідний код, що особливо корисно для додавання спільних функцій для декількох типів або для розширення типів, наданих сторонніми бібліотеками.

Основна функціональність MemberMacro полягає у здатності доповнювати оголошення додатковими елементами коду залежно від контексту його використання. Це означає, що макрос може адаптувати свій вивід, генеруючи користувацькі члени на основі оточуючого коду, визначених користувачем атрибутів або інших макросів, застосованих до тієї самої декларації.

MemberMacro особливо корисний у поточній роботі, де повторювані функції потрібно додати до кількох типів, при розширенні типів із зовнішніх бібліотек без модифікації коду, або коли потрібно зменшити кількість шаблонного коду шляхом динамічної генерації на основі атрибутів декларацій. Ця функціональність сприяє створенню більш модульної та підтримуваної архітектури коду в межах кодової бази Swift, підвищуючи гнучкість та можливість повторного використання коду.

VariableMacroCreator працює як більш комплексний інструмент для створення **PeerMacro**, який базується на контексті введеного користувачем коду.

Цей інструмент є невід'ємною частиною мого підходу до автоматизації та спрощення процесу генерації динамічних структур коду. Він відіграє вирішальну роль в управлінні та генерації коду на основі визначених користувачем вхідних та вихідних специфікацій, гарантуючи, що процес розробки не тільки спрощується, але й стає менш схильним до помилок.

Під час реалізації VariableMacroCreator я зосередився на створенні макросів, які перетворюють оголошення змінних на більш складні, функціонально багаті структури. Процес починається з прийняття декількох параметрів: inputCode, outputCode, moduleName і macroName. Ці параметри формують фундаментальні дані, з яких ліпиться процес генерації макросів.

Спочатку я перевіряю цілісність і наявність цих вхідних даних за допомогою VariableMacroCreator. Він перевіряє, що inputCode не є порожнім, і аналогічно перевіряє outputCode, moduleName і macroName. Ці перевірки необхідні для запобігання помилкам під час розширення макросу і для того, щоб

гарантувати, що кожна частина генерації коду базується на повних і достовірних даних.

Після перевірки вхідних даних я обробляю inputCode і outputCode за допомогою структури DeclSyntax. Це включає синтаксичний аналіз коду, щоб зрозуміти його структуру і тип. Коли вхідний код визначає змінну, а вихідний код - функцію, я заглиблююсь у створення змістовного перетворення.

Суть цього перетворення полягає в ретельному аналізі структури змінної та функції. Якщо на вході змінна, а на виході функція, я визначаю ключові компоненти, такі як ім'я змінної, тип і будь-яку потенційну кореляцію з параметрами функції. Цей ретельний аналіз дозволяє мені розумно вставити або змінити код, щоб узгодити змінну з контекстом роботи функції.

Наприклад, я перевіряю, чи функція посилається на змінну безпосередньо, і відповідно коригую код згенерованого методу. Я динамічно коригую назву методу та параметри, щоб відобразити вплив змінної, гарантуючи, що нова функція ідеально відповідає призначенню змінної. Цей процес не є статичним, а адаптується на основі конкретних зв'язків і посилань, знайдених у коді.

Крім того, я включив обробку помилок у VariableMacroCreator, щоб керувати ситуаціями, коли може виникнути невідповідність типів або інші невідповідності, які можуть порушити процес генерації коду. Я додаю специфічні помилки, щоб спрямувати виправлення цих проблем, гарантуючи, що результуючий макрос буде точним і надійним.

Фінальним етапом цього процесу є створення структурованого оголошення макросу. Я обгортаю новостворений код у макроструктуру, визначаючи його в рамках PeerMacro, який можна приєднати до вихідного об'єкта коду. Цей макрос слугує розширенням початкового оголошення, доповнюючи його новими функціональними можливостями, визначеними перетвореним кодом.

PeerMacro у Swift - це ще один тип макросів, призначений для підвищення модульності та повторного використання коду шляхом введення «однорангових» декларацій поряд з існуючою декларацією, до якої приєднано макрос. Це означає, що коли застосовується PeerMacro, він може генерувати додаткові структури коду, такі як функції, властивості або навіть нові типи, які існують на тому ж ієрархічному рівні, що й оригінальна декларація. Це особливо корисно для додавання функціональності, яка доповнює початкове оголошення, але яку краще підтримувати як окремий об'єкт, зберігаючи розподіл завдань і сприяючи більш чистому та організованому коду.

PeerMacro працює у певному контексті, аналізуючи та перетворюючи навколишній код на основі атрибутів, визначених у вузлі AttributeSyntax, який описує макрос. Цей макрос оцінюється у визначеному контексті розширення, що дозволяє йому динамічно адаптуватися до умов, в яких він застосовується, потенційно змінюючи свій результат залежно від оточення або інших зовнішніх факторів.

Можливість вводити однорангові декларації є важливою для створеного в цій роботі VariableMacroCreator, де додаткова функціональність повинна бути логічно згрупована з існуючим кодом, але не повинна порушувати або ускладнювати внутрішню роботу оригінального класу або структури. Це також дозволяє чіткіше розмежовувати базову функціональність типу та розширення або модифікації, що вносяться макросами, роблячи код легшим для розуміння та супроводу.

По суті, PeerMacro надає можливість для розширення функціональності базуючись повністю на контекст введеним нашим користувачем, де ми задаємо, що саме в результаті поверне макрос.

VariableMacroCreator є відображенням стратегічного підходу до використання макросів для динамічного розширення та покращення можливостей коду. Він заповнює прогалину між статичними деклараціями та

динамічними функціями, дозволяючи створювати адаптивні, ефективні та стійкі до помилок програмні компоненти. Цей інструмент є прикладом мого прагнення використовувати передові методи програмування для покращення загальної якості та гнучкості кодової бази мого проекту.

ViewCellMacroCreator є прикладом використання **DeclarationMacro**, що інкапсулює стандартні елементи інтерфейсу, такі як **UIView**, в макроси, що автоматично генерують класи для компонентів таблиці **UITableViewCell**.

Цей інструмент є центральним у моїй методології автоматизації створення динамічних та ефективних елементів інтерфейсу користувача, забезпечуючи послідовний процес розробки з мінімізацією помилок.

Макрокод `ViewCellMacroCreator` призначений для перетворення оголошень класів вводу в повні підкласи UITableViewCell, які інкапсулюють оригінальну функціональність представлення. Я починаю процес з прийняття основних параметрів: `inputCode`, `moduleName` та `macroName`. Вони формують базові дані, з яких розгортається генерація макросів.

Спочатку я перевіряю цілісність цих вхідних даних. Дуже важливо, щоб `inputCode` не був порожнім, так само як і `moduleName` та `macroName` мають бути дійсними. Ці перевірки запобігають помилкам під час розширення макросу і гарантують, що кожна частина генерації коду ґрунтується на повних і достовірних даних.

Після перевірки вхідних даних я обробляю `inputCode` за допомогою структури `DeclSyntax`. Цей крок включає синтаксичний аналіз вхідних даних для визначення їх структури та типу. Макрокод `ViewCellMacroCreator` спеціально створено для обробки оголошень класів, які мають перетворитися на підкласи UITableViewCell. Якщо вхідні дані не є класом, я видаю помилку, щоб вказати на несподівану природу оголошення.

Суть перетворення полягає у визначенні початкового класу подання з `inputCode` і створенні відповідного підкласу UITableViewCell навколо нього. Я

використовую регулярний вираз для вилучення імені класу, який успадковується від UIView або відповідає протоколу, що передбачає UIView. Це витягнуте ім'я потім стає основою для нового класу комірки.

Для кожного правильного введення я генерую макродекларацію, яка включає повну реалізацію підкласу UITableViewCell з назвою `<OriginalClassName>Cell`. Цей підклас містить екземпляр оригінального подання і налаштовує його розташування у комірці. Згенерований код містить методи для ініціалізації комірки, налаштування її компонування та забезпечення правильного розміщення подання у поданні вмісту комірки.

Макрос, згенерований цим інструментом, загортає цей код у структуру `DeclarationMacro`, яку потім можна застосувати для розширення функціональності початкового подання, забезпечивши багаторазову реалізацію комірки. Такий підхід не лише економить час, але й гарантує, що компоненти інтерфейсу користувача мають однаковий стиль і поводяться передбачувано в усьому додатку.

DeclarationMacro дозволяє генерувати нові декларації у кодовій базі, керуючись логікою розширення макросів, яка обробляє специфічні вхідні дані, надані в окремій декларації розширення макросів. Це дозволяє визначати багаторазові шаблони або патерни, які можуть бути реалізовані у вигляді конкретних ділянок коду, де це необхідно. Наприклад, макрос можна створити для автоматичного генерування класів моделі даних зі схеми JSON.

Макрос DeclarationMacro надзвичайно корисний в поточному сценарії, що вимагає генерації шаблонного коду. Інкапсулюючи логіку створення в макроси, ми можемо забезпечити узгодженість і дотримання шаблонів проектування, спростити обслуговування і зменшити кількість помилок, які виникають через ручне дублювання складних структур коду.

ViewCellMacroCreator відображає моє стратегічне використання макросів для динамічного розширення та покращення можливостей компонентів

інтерфейсу. Він заповнює прогалину між статичними деклараціями подання та їх динамічною інтеграцією у комірки UITableView, дозволяючи створювати адаптивні, ефективні та стійкі до помилок компоненти користувацького інтерфейсу.

Застосування патернів Swift Syntax для аналізу коду

MethodMacroCreator, VariableMacroCreator і ViewCellMacroCreator - це класи, які обробляють різні сценарії створення макросів. Кожен з них використовує синтаксис Swift для розбору і генерації коду Swift на основі певних шаблонів, які я опишу нижче:

Використання DeclSyntax - це загальний шаблон для всіх класів. DeclSyntax використовується для розбору вхідного рядка коду на структуроване дерево синтаксису. Це дерево представляє різні частини коду Swift, що дозволяє аналізувати та маніпулювати кодом. Цей підхід є фундаментальним для перевірки відповідності вхідного коду очікуваному типу оголошення Swift (наприклад, функції або змінної) і має вирішальне значення для забезпечення правильної поведінки згенерованого макросу.

Відповідність шаблону за допомогою захисних інструкцій - у Swift оператори захисту використовуються для дострокового завершення роботи, якщо не виконуються певні умови, що допомагає зберегти код чистим і менш вкладеним. Цей патерн широко використовується у наданих класах, щоб забезпечити виконання необхідних умов для створення макросів, таких як непорожній вхідний код або правильні імена модулів. Таке використання захисних операторів покращує обробку помилок, виявляючи їх на ранніх стадіях виконання.

Визначення та розширення макросів - кожен клас визначає макрос і вказує спосіб його розширення. Це передбачає створення рядка коду Swift, який представляє функціональність макросу. Використання багаторядкових рядкових

літералів (з використанням потрійних лапок) дозволяє вбудовувати складний Swift-код безпосередньо в рядки. Ці макроси зазвичай передбачають створення нових оголошень або модифікацію існуючих, залежно від призначення макросу, наприклад, додавання нових методів, змінних або навіть нових класів.

Регулярні вирази для вилучення даних - у ViewCellMacroCreator використовується регулярний вираз для вилучення назви класу з вхідного коду. Цей шаблон спеціально вибрано для точного визначення назв класів, які відповідають певному формату, що є важливим для забезпечення роботи макросу з правильним типом. Цей метод ефективний для синтаксичного аналізу та вилучення певних фрагментів даних з неструктурованого вхідного коду, що часто потрібно, коли вхідний код змінюється або має менш жорстку структуру.

Динамічна генерація коду з використанням контексту - в даній роботі використовується контекст синтаксису для динамічної генерації коду. Наприклад, перевіряються тип вузлів синтаксису (чи є вузол функцією або змінною) і використовуються деталі з цих вузлів (наприклад, імена або типи змінних) для побудови нового Swift-коду, який враховує контекст. Такий підхід дозволяє згенерованим макросам бути дуже адаптивними та пристосованими до конкретних потреб вхідного коду.

Загалом, ці патерни використовують потужні можливості синтаксичного аналізу та генерації коду Swift Syntax для створення гнучких, динамічних та контекстно-залежних макросів. Це робить код не тільки більш надійним і зручним для підтримки, але й підвищує його здатність ефективно обробляти широкий спектр сценаріїв програмування.

Висновки розділу

Автоматизуючи генерацію складного коду з простих шаблонів, макроси значно підвищують продуктивність, дозволяючи розробникам зосередитися на більш стратегічних елементах проектів. Така автоматизація не лише спрощує

процес розробки, але й забезпечує гнучкість та масштабованість програмного забезпечення, яке можна адаптувати до різних вимог проекту без значних модифікацій.

Крім того, використання макросів підвищує загальну якість коду. Оскільки макроси зводять до мінімуму ручне кодування, вони зменшують ймовірність людських помилок, що призводить до створення більш надійного та узгодженого програмного забезпечення. Єдине застосування стандартів кодування у всьому проекті ще більше посилює цю якість.

Крім того, макроси спрощують підтримку коду, дозволяючи створювати модульні рішення, які можна легко оновлювати або інтегрувати в різні частини проекту. Ця можливість зменшує складність, пов'язану з управлінням та оновленням кодових баз, що полегшує командам підтримку та розширення програмного забезпечення з часом.

По суті, стратегічне впровадження макросів не тільки оптимізує процес кодування, але й підвищує надійність і ремонтопридатність програмних систем, прокладаючи шлях до більш ефективної і безпомилкової розробки програмного забезпечення.

РОЗДІЛ 3 Імплементація та аналіз результатів

Аналіз вихідних проблем та методів їх вирішення

В процесі розробки розширення для Xcode за допомогою XcodeKit були визначені кастомні методи, які при використанні стають доступними в меню користувача. Таке розширення приєднується до основного додатку під MacOS, надаючи певні можливості з точки зору UI/UX. Однак навіть з таким інструментарієм виникли певні проблеми.

По-перше, постало питання інтеграції бібліотеки Swift Syntax. Оскільки фреймворк Xcode Extension має обмеження в пісочниці свого застосування і не може розглядатися як окремий репозиторій, який може містити посилання на певні бібліотеки, виникла необхідність створити копію існуючого пакету з усіма необхідними класами оголошень синтаксису, а також видалити всі залежності, які могли б перешкодити компіляції коду.

Ще однією проблемою була можливість запуску інтерфейсної частини розширення. Розширення Xcode не може безпосередньо відображати елементи інтерфейсу, тому користувачеві доступний лише невеликий набір системних вкладок в меню управління Xcode. Цього явно недостатньо для повної реалізації кінцевого бачення проекту, тому було потрібно альтернативне рішення. Спочатку ми розглядали можливість використання такого інструменту, як Apple Scripts - системні файли, які можуть керувати більшістю дій в MacOS, включаючи відкриття додатків. Однак обмежене середовище розширення Xcode не дозволило повністю реалізувати цей метод. Тому альтернативою стало використання Universal Links - специфічної для Apple технології, яка спрощує взаємодію між веб-контентом і нативними мобільними або десктопними додатками. Запроваджена у 2015 році, ця технологія дозволяє за однією URL-адресою спрямовувати користувача або до певного місця в додатку, або на відповідну веб-сторінку. Якщо додаток встановлено, натискання на універсальне посилання перенаправляє користувача безпосередньо до вказаного контенту;

якщо ні, то посилання перенаправляє на веб-сайт. За допомогою універсальних посилань можна перенаправити користувача зі стандартного меню Xcode на потрібний інтерфейс з введенням необхідних даних.

Нарешті, важливо було налаштувати комунікацію між двома частинами проекту: розширенням і основним контейнером MacOS-додатку, оскільки вони повинні передавати дані між собою. Це питання було вирішено за допомогою системи User Defaults - папки з файлами, яка існує в середовищі додатку або, як згадувалося раніше, пісочниці. Завдяки цьому можна отримувати прості дані у вигляді рядків, чисел та байтів.

Структурне проектування програмного забезпечення

Створення архітектури проєкту, яка передбачає майбутнє розширення та спрощує процес входження нових розробників, є ключовим аспектом ефективної реалізації програмного забезпечення. Враховуючи, що Swift є мовою з сильною протокольною орієнтацією, що сприяє використанню численних абстракцій, в проєкті було прийнято рішення розробити систему протоколів, яка дозволяє гнучко і логічно організовувати код для розширення Xcode.

Протокол Builder (див. Додаток Б)

Основою системи є протокол Builder, який визначає основні характеристики кожного макрос-генератора. Кожен будівельник макросів повинен включати назву, яка відображатиметься у випадаючому меню вибору, та унікальний ідентифікатор, який дозволить в основному додатку ідентифікувати, який саме метод було викликано. Це важливо, адже пряма комунікація між компонентами обмежена, і використання ідентифікатора допомагає управляти взаємодією між різними частинами системи.

Протокол MacrosCreator (див. Додаток В)

Для більш деталізованої роботи з макросами було створено протокол MacrosCreator, який визначає метод createMacro. Цей метод включає параметри, які необхідні для створення макросу: вхідний код користувача, бажаний вихідний код, назву модулю, де буде зберігатися макрос, та назву самого макросу. Це дозволяє генерувати макроси, які можуть бути використані для автоматизації задач у розробці програмного забезпечення, забезпечуючи високу адаптивність і масштабованість системи.

Використання цих протоколів забезпечує, що кожен новий компонент або макрос може бути легко інтегрований у існуючу систему без необхідності переписування або значних змін в коді. Такий підхід не тільки спрощує процес додавання нових функцій, але й знижує поріг входження для нових розробників, які можуть ефективно взаємодіяти з проєктом без необхідності глибокого занурення в усі аспекти платформи.

У підсумку, архітектура, заснована на протоколах у Swift, створює міцну основу для розвитку та масштабування проєкту, забезпечуючи необхідну гнучкість та можливість адаптації під змінні вимоги розробки програмного забезпечення.

Реалізація ключових методів у проєкті

У відповідь на різноманітність потреб користувачів та їхніх контекстуальних вимог, було вирішено зосередитись на реалізації декількох ключових типів макросів для розширення Xcode. Вся взаємодія з розширенням базується на аналізі та маніпуляції з вхідним кодом, який користувач виділяє, що забезпечує високий рівень адаптивності та персоналізації використання макросів.

Одним з перших реалізованих методів було створення @attached макросів, які дозволяють користувачам автоматично генерувати код, що розширює функціональність класів чи структур у Swift. Використовуючи клас DeclSyntax з

бібліотеки Swift Syntax, який дозволяє ідентифікувати всі декларації у коді, цей підхід відкриває можливість точно визначати і маніпулювати елементами коду для створення бажаної функціональності. На базі DeclSyntax розроблено MemberMacro, який застосовується для додавання нових декларацій всередині вже існуючих типів або розширень [2].

Далі, в рамках демонстрації гнучкості системи, було реалізовано ViewCellMacroCreator. Цей макрос-генератор приймає вхідний код користувача та створює новий клас UITableViewCell з візуалізацією компоненту, який був вказаний користувачем. Така імплементація дозволяє легко створювати користувацькі компоненти на базі вже існуючих елементів інтерфейсу, адаптуючи їх під поточні потреби розробника.

Після реалізації двох відносно простих макросів, було важливо показати, як можна використовувати можливості Swift Syntax для створення більш складних та адаптивних рішень, що відповідають конкретним потребам користувачів. Цьому служить розробка VariableMacroCreator — складнішого механізму для автоматичного генерування макросів на основі вхідного та очікуваного вихідного коду користувача.

VariableMacroCreator працює з кодом, який користувач надає: це може бути як вихідний код змінної, так і методу, який користувач бажає отримати для оновлення значення цієї змінної. На базі цих даних ми використовуємо можливості Swift Syntax для розбору вхідних параметрів на синтаксичні елементи, що дозволяє нам детально аналізувати та маніпулювати структурою коду.

Ключовим елементом процесу є аналіз сумісності типів та структурних відношень між вхідним та вихідним кодом. Ми перевіряємо, чи існує відповідність між змінною, вказаною користувачем, та методом, який має її оновлювати. Це означає, що ми визначаємо, чи ім'я змінної присутнє у назві методу, та чи типи даних в обох випадках є сумісними.

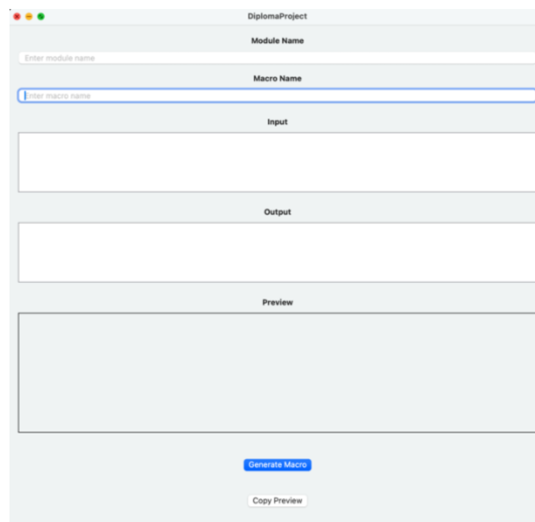
В результаті, коли з'являється взаємозв'язок між змінною та методом, наша система адаптує код таким чином, щоб він відповідав обраному контексту. Це включає в себе вибудовування результуючого макросу так, щоб він міг бути впроваджений безпосередньо в середовище, до якого він прикріплений. Наприклад, якщо ім'я змінної міститься у методі, ми забезпечуємо можливість отримання даних з цього методу без втручання закодованих значень.

На завершення, використання синтаксичного дерева Swift Syntax дозволяє нам не тільки аналізувати, але й оптимізувати процеси генерації коду, забезпечуючи точність та відповідність вимогам користувача. Завдяки глибокому розумінню структури коду, VariableMacroCreator здатен створювати високо адаптивні та функціонально насичені макроси, які ефективно вирішують конкретні задачі розробників.

Процес дизайну та імплементації користувацького інтерфейсу

Після успішного розроблення моделі для генерації макросів, наступним кроком у проекті стало створення інтуїтивно зрозумілого інтерфейсу користувача. Цей інтерфейс був розроблений з метою забезпечення зручності та ефективності у процесі створення макросів, дозволяючи користувачам легко взаємодіяти з інструментом без потреби в глибоких знаннях програмування.

Основний інтерфейс користувача (Мал. 3) включає декілька ключових полів для вводу, які є критично важливими для процесу створення макросу:



Мал. 3, основний інтерфейс користувача

Module Name - Це поле призначене для введення назви модулю, де буде імплементовано макрос. Важливість цього поля полягає в тому, що всі макроси виконуються через зовнішній фреймворк, створений користувачем, що дозволяє забезпечити модульність та легкість інтеграції з різними проектами.

Macro Name - Назва макросу, яку користувач хоче створити. Це поле важливе для ідентифікації та подальшого виклику макросу у коді.

Input - Вхідний код, який користувач вводить як базу для генерації макросу. Це може бути фрагмент коду, з якого макрос буде витягувати необхідні елементи для сворення нової функціональності.

Output - Поле для вводу очікуваного результату макросу. Це поле активне лише для **VariableMacroCreator**, де користувач може вказати вихідний код, що повинен бути сгенерований. У випадках інших макросів це поле залишається вимкненим, щоб уникнути плутанини.

Preview - Область, де користувач може побачити кінцевий результат генерації макросу. Це дозволяє перевірити і в разі потреби скоригувати код перед його імплементацією в проект.

Для створення макросу користувач повинен заповнити всі необхідні поля і натиснути кнопку "Generate Macro". Якщо якесь з полів залишено пустим, система виведе повідомлення про помилку, яке інформує, яке саме поле не заповненеТакий підхід допомагає уникнути помилок при ініціації макросу і забезпечує, що користувач введе всі необхідні дані правильно.

Розробка ефективного користувацького інтерфейсу для генерації макросів у Swift є ключовим елементом для забезпечення успішного впровадження цієї функціональності. Інтерфейс, який враховує інтуїтивність використання та зручність доступу до всіх необхідних функцій, збільшує продуктивність користувачів та сприяє більшій адаптації розширення у повсякденній розробці.

Аналіз отриманих результатів

У процесі розробки розширення для Xcode, яке спрямоване на генерацію макросів за допомогою Swift Syntax, було досягнуто значних результатів, які можуть бути аналізовані з різних аспектів: технічного виконання, користувацького досвіду та взаємодії з системою macOS.

Реалізація макросів була здійснена за допомогою набору протоколів, що визначають структуру та поведінку макросів у розширенні. Це забезпечило модульність та гнучкість системи, дозволяючи легко додавати нові типи макросів. Протоколи **Builder** та **MacrosCreator** сприяли створенню чіткої архітектури, що спрощує розуміння та розширення проекту майбутніми розробниками.

Інтерфейс, який був розроблений для розширення, забезпечує інтуїтивне і зручне середовище для користувачів. Він дозволяє користувачам легко вводити необхідні дані для генерації макросів, переглядати результати та виправляти помилки без потреби глибокого занурення в деталі реалізації коду. Ця простота використання значно підвищує продуктивність розробників і сприяє адаптації інструменту у повсякденній роботі.

Інтеграція з Xcode та macOS була однією з важливих задач проекту, яка потребувала особливої уваги через обмеження пісочниці та ізоляцію процесів. Використання Universal Links як рішення для навігації та взаємодії з користувацьким інтерфейсом дозволило обійти деякі з цих обмежень, надаючи гладку та ефективну взаємодію між розширенням та веб-компонентами. Це значно розширило можливості розширення, дозволяючи більш тісну інтеграцію з іншими додатками та сервісами на macOS.

Проект з розробки макросів для Xcode показав, що з допомогою Swift Syntax можна ефективно створювати високоадаптивні інструменти для розробників, що значно спрощують процеси кодування та знижують поріг входження для новачків. Інтеграція з macOS та використання специфічних для платформи рішень, таких як Universal Links, відкриває нові можливості для подальшого розвитку і адаптації інструментів. Впровадження цього інструменту не тільки покращує процес розробки, але й вносить важливий внесок у спільноту розробників, забезпечуючи їм потужний інструмент для оптимізації робочих процесів.

ВИСНОВКИ

У цій роботі досліджено розробку інтегрованого розширення середовища для автоматичної генерації коду в Xcode, зосереджуючись на використанні мови програмування Swift. Метою дослідження було підвищення ефективності роботи розробника шляхом мінімізації повторюваних завдань за рахунок автоматизації та використання сучасних інструментів і технологій.

Дослідження розпочалося з огляду існуючих методів генерації коду, включаючи макроси та інші автоматизовані інструменти, зокрема ті, що інтегрують машинне навчання для адаптації до різних мов програмування. Реалізація макросів у Swift, хоча і є складним завданням, представляє собою перспективний шлях для зменшення дублювання коду та підвищення продуктивності розробників.

Процес інтеграції нових інструментів в Xcode був ретельно вивчений. Незважаючи на потенційні перешкоди, такі як складність інтеграції та початковий опір новим технологіям, розроблений прототип розширення продемонстрував значний потенціал для зменшення робочого навантаження на ручне кодування.

Важливим компонентом дослідження було використання синтаксису Swift для полегшення процесу генерації коду. Цей інструмент виявився безцінним для ефективного та безпечного маніпулювання структурами коду в середовищі компілятора Swift.

Прототип розширення, розроблений під час цього дослідження, успішно автоматизував кілька аспектів генерації коду, продемонструвавши не лише доцільність, але й потенціал для розширення цих можливостей до ширшого застосування в розробці програмного забезпечення.

Практичні наслідки цієї дипломної роботи є далекосяжними. Зменшуючи час і зусилля, необхідні для рутинних завдань кодування, розробники можуть перенаправити свою увагу на більш складні та інноваційні завдання, тим самим

підвищуючи загальну продуктивність і творчий потенціал спільноти розробників програмного забезпечення. Крім того, розроблене розширення може слугувати основою для майбутніх досліджень і розробок у сфері автоматизованого кодування.

Інтеграція таких інструментів потенційно може трансформувати практику розробки програмного забезпечення, зробивши високоякісний код більш доступним і менш трудомістким у створенні. Крім того, оскільки машинне навчання та штучний інтелект продовжують розвиватися, їх інтеграція в інструменти автоматизованої генерації коду, подібні до тих, що досліджуються в цій дисертації, призведе до подальшої революції в цій галузі.

Подальша робота може бути спрямована на більш глибоку інтеграцію зі штучним інтелектом для розширення можливостей розроблених інструментів генерації коду. Розширення масштабу проекту для підтримки більшої кількості мов програмування та середовищ розробки також може забезпечити більші переваги та привабливість.

На завершення, слід зазначити, що дана дослідницька робота поклала основу для істотного прогресу у сфері розробки програмного забезпечення, акцентуючи увагу на автоматизації та оптимізації процесів генерації коду у контексті інтегрованих середовищ розробки. Без сумніву, подальший розвиток цих інструментів відіграватиме критичну роль у формуванні напрямків розвитку програмної інженерії у майбутньому.

ДЖЕРЕЛА

1. Ballantyne M., King A., Felleisen M. Macros for domain-specific languages. Proceedings of the ACM on programming languages. – 2020. – Vol. 4, OOPSLA. – С. 1–29. – Режим доступу до ресурсу: https://doi.org/10.1145/3428297

2. Bebek T. Swift macros. Medium. – Режим доступу до ресурсу: https://medium.com/@tahabebek/swift-macros-36417a8557a

3. Bulavin V. Xcode source editor extension tutorial: getting started. Yet Another Swift Blog. – Режим доступу до ресурсу: https://www.vadimbulavin.com/xcode-source-editor-extension-tutorial/

4. Chen K. Xcode extension 1/5 – starting guide. Medium. – Режим доступу до ресурсу: https://kowei-chen.medium.com/xcode-extension-1-5-starting-guide-519a95bdc865

5. Chen M. Evaluating large language models trained on code. – Режим доступу до ресурсу: https://arxiv.org/abs/2107.03374

6. Expand on swift macros - WWDC23 - videos - apple developer. Apple Developer. – Режим доступу до ресурсу: https://developer.apple.com/videos/play/wwdc2023/10167/

7. GitHub - apple/swift-syntax: a set of swift libraries for parsing, inspecting, generating, and transforming swift source code. GitHub. – Режим доступу до ресурсу: https://github.com/apple/swift-syntax/

8. How to automate swift boilerplate code with sourcery. ATOMIC ROBOT. – Режим доступу до ресурсу: https://atomicrobot.com/blog/sourcery/

9. Macros documentation. – Режим доступу до ресурсу: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/macros/

10. Mattt. SwiftSyntax. NSHipster. – Режим доступу до ресурсу: https://nshipster.com/swiftsyntax/

11. Smith Z. XcodeKit and xcode source editor extensions. NSHipster. – Режим доступу до ресурсу: https://nshipster.com/xcode-source-extensions/

12. SwiftLeeds. Working with XcodeKit by Aryaman Sharda - SwiftLeeds 2023, 2023. YouTube. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=hsX-b7lobF0

13. Write swift macros - WWDC23 - videos - apple developer. Apple Developer. – Режим доступу до ресурсу: https://developer.apple.com/videos/play/wwdc2023/10166

14. Yechiam E. Why are macros not used? A brief review and an approach for improving training. Computers & education. – 2006. – Vol. 46, no. 2. – С. 206–220. – Режим доступу до ресурсу: https://doi.org/10.1016/j.compedu.2004.08.009

15. Zabłocki K. GitHub - krzysztofzablocki/Sourcery: Meta-programming for Swift, stop writing boilerplate code. GitHub. – Режим доступу до ресурсу: https://github.com/krzysztofzablocki/Sourcery

Додаток Б

```
protocol Builder {

    var title: String { get }
    var triggerIdentifier: TriggerIdentifier { get }
}
```

Протокол для методів розширення

Додаток В

```
public protocol MacrosCreator {

    func createMacro(
        inputCode: String,
        outputCode: String?,
        moduleName: String,
        macroName: String
    ) throws -> String

}
```

Протокол для створення макросів