2. Чемерис Г. Поняття графічної компетентності майбутнього бакалавра з комп'ютерних наук у вітчизняних та закордонних дослідженнях. *Молодь і ринок,* 2018. №5 (160). С. 129–133.

3. Коляса П. І. Формування графічної компетентності майбутніх інженерів-педагогів засобами цифрових технологій. Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 015 Професійна освіта. Тернопільський національний педагогічний університет імені Володимира Гнатюка, 2022. 223 с.

4. Горобець С. М. Методичні підходи щодо навчання комп'ютерній графіці студентів ВНЗ засобами інформаційно-комунікаційних технологій. *Вісник Житомирського державного університету імені Івана Франка,* 2018. Вип. 1(92). С. 75–79.

УДК 004.41

Гуменюк Станіслав,

асистент кафедри омп'ютерних наук та інформаційних технологій

Житомирський державний університет імені Івана Франка

ЕВОЛЮЦІЯ АРХІТЕКТУРНИХ ПАТЕРНІВ У РОЗРОБЦІ ANDROID-ДОДАТКІВ

У сучасному світі мобільних технологій, де кожен день з'являються нові інструменти та підходи до розробки, важливість зрозумілого та ефективного архітектурного патерну не може бути переоцінена. Еволюція архітектурних патернів у розробці Android-додатків стала відображенням змін у потребах користувачів, технічних можливостях обладнання та вимогах до швидкості розробки та масштабованості проєктів. Від простих MVC (Model-View-Controller) (Рис. 1) до більш складних та гнучких підходів, таких як MVVM (Model-View-ViewModel) та MVI (Model-View-Intent), архітектурні патерни

продовжують еволюціонувати, пропонуючи розробникам кращі інструменти для створення високопродуктивних, легких у тестуванні та підтримці мобільних додатків. Ця динаміка змін відкриває нові горизонти для досліджень та інновацій, роблячи архітектуру додатків ключовим компонентом у вирішенні сучасних викликів мобільної розробки. [5]

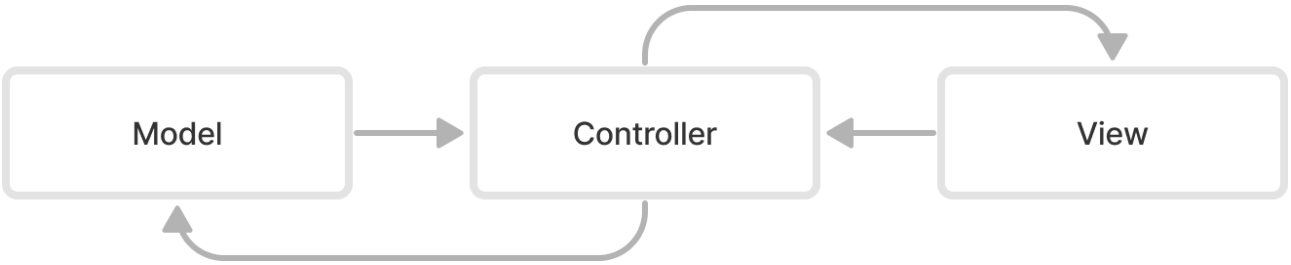

Рис. 1. Принцип взаємодії MVC

Розглянемо один із перших архітектурних патернів, ідея якого була взята за основу для майбутніх його видозмін та створення нових варіантів його використання. Йдеться про патерн MVC – Model-View-Controller. [1]

Під Model слід розуміти ту частину програми, яка містить бізнес-логіку та дані, які в ній використовуються. Тобто, це можуть бути різні запити на сервер (Request) та отримані від нього відповіді (Response), або дані, які зберігаються локально, та операції над ними. Також, сюди входять бізнес-правила та бізнес-логіка, що уособлюють в собі опис функціонала програмного забезпечення.

View – це візуальна частина програмного забезпечення, з якою користувач має можливість взаємодіяти. Вона містить відображення контенту, який залежить від даних, які загалом містяться в Model, а також, отримання інформації про всі можливі користувацькі жести, такі як click, long click, double click, swipe left & right, scrolling, drag & drop, і т.д.

Позаяк View, а саме те, що відображається для користувача, в цілому залежить від Model, то чому не зв’язати їх між собою? І спочатку, так і було, і це займало доволі мало часу для реалізації даної взаємодії. Але постають інші питання, які демонструють доволі негативну сторону такого підходу, як: Чи

можливо підтримувати такий підхід паралельно із розвитком проєкту? Чи можна змінити View без втручання у Model, або навпаки?

Саме тому виникає потреба у ще одній частині, як Controller, який виступає медіатором між ними. Controller бере на себе відповідальність у реагуванні на певні жести та дії користувача з View, і в результаті дії на них виконує ту, чи іншу бізнес-логіку Model, а також у зворотному порядку реагує на зміни в даних та очікує певний результат від операцій над ними, щоб доставити їх у View, і користувач матиме змогу бачити актуальний стан речей.

Для прикладу візьмемо звичайну авторизацію в мобільному додатку. Уявімо екран, який містить у собі два поля для електронної пошти та паролю, а також кнопку “Sign In”, яка є неактивною, тобто користувач не може з нею взаємодіяти. Користувач заповнює відповідні поля, і після цього кнопка стає доступною. При натисканні на неї надсилається запит на сервер, для того, щоб авторизувати користувача. Залежно від результату даного запиту, буде відображено відповідне повідомлення на екрані. В майбутньому будемо використовувати саме даний приклад для пояснення інших патернів.

Тепер розглянемо як даний процес відбувається у MVC. Із прикладу, виокремимо функціонал з авторизацією до Model, та назвімо його SignInUseCase, який буде приймати в себе значення email та password, а повертати результат нашого запиту Result, який може мати два значення: Success - у випадку успішної авторизації, та Error із текстом помилки – у випадку невдачі при авторизації.

View міститиме відображення всіх UI елементів на даному екрані, і власне взаємодію з ними. Умовно EmailEditText та PasswordEditText для позначення полів, та SignInButton для кнопки авторизації. Також View буде містити обробку взаємодії користувача із ними, як OnTextChanged – реагування на введення тексту користувачем у поле та отримання поточного його значення, та OnClick – при натисканні на кнопку. Додатково View буде містити метод ChangeButtonEnabled для зміни доступності кнопки, тобто чи зможе користувач з нею взаємодіяти, чи ні.

В цьому випадку Controller містить у собі посилання на Model та View. Тобто Controller точно знає з ким та як він має змогу взаємодіяти. Отже, View очікує коли користувач почне вводити інформацію в одне з наявних полів. Після того, як користувач закінчив введення в одному із полів, View отримує відповідний Event зі значенням. В даному патерні, View також знає про Controller, тому при тригері відповідного Event дане значення передається до нього, яке він запам’ятовує, і паралельно перевіряє чи всі поля заповнені, або передає цю роботу відповідній бізнес-логіці у Model. Після заповнення всіх полів Controller викликає відповідний метод View, щоб змінити статус кнопки на доступну. Після натискання на кнопку View отримує відповідний Event про цю подію, і водночас викликає відповідний метод Controller для авторизації користувача. Controller звертається до Model та передає їй всі необхідні дані, та очікує на результат. Після отримання результату, Controller викликає конкретні методи View зміни його статусу.

Основними недоліками MVC є складність замінювати одні частини на інші. Тобто, якщо замінити одну View, на іншу, то вона напряму вплине на реалізацію Controller. До переваг варто віднести те, що бізнес-логіка та інтерфейс користувача є незалежними один від одного, тому якщо зміниться, наприклад формат отримання даних із JSON в XML у Model, то ці зміни ніяким чином не вплинуть на пряму на View, але можуть вплинути на Controller. Також перевагою є те, що даний патерн доволі просто реалізується, але погано масштабується та тестується.

Наступником у розвитку MVC стає архітектурний патерн, який набув широкого використання на практиці є MVP – Model-View-Presenter (Рис. 2). На зміну звичного нам Controller приходить Presenter.

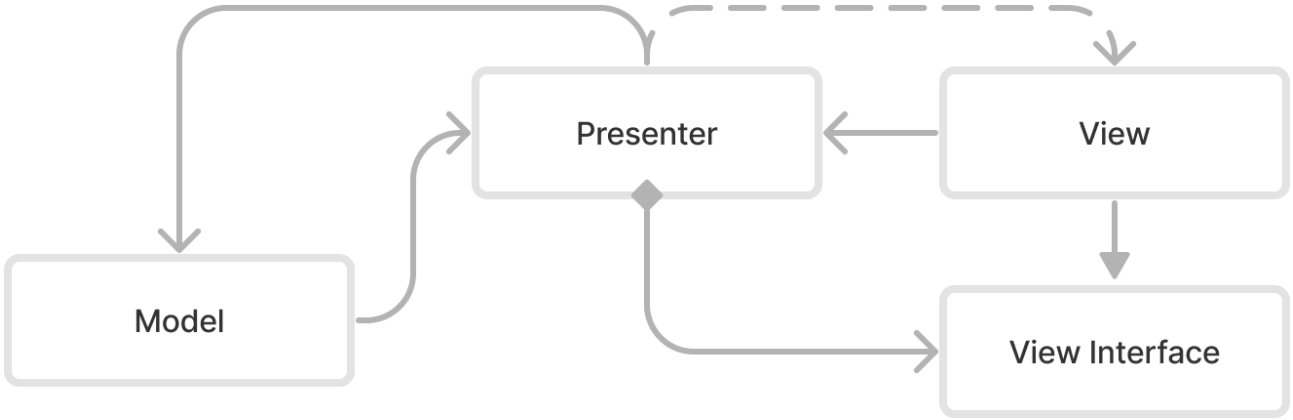

Рис. 2. Принцип взаємодії MVP

Взаємодія між Model та Presenter без змін, але зв'язок View із Presenter змінився. Якщо раніше View чітко мало інформацію про Controller, та навпаки Controller мав інформацію про View, то тепер ми оперуємо з абстракціями. Тісний зв'язок замінюється на контракт, який обидві частини зобов'язуються виконувати. Це нам дає змогу з легкістю, замінювати одне View на інше, при цьому ніяким чином не зачіпати Presenter при цій заміні, і навпаки. У MVC якщо виникала потреба замінити View на іншу, це напряму зачіпало Controller. Недоліком MVP, як і MVC, саме у мобільній розробці на Android, є те що медіатори в обох випадках не можуть пережити та відповідно реагувати на життєвий цикл мобільного застосунку, чи окремого екрану або його фрагменту. В даних випадках це все лягає повністю на плечі розробників, і вони самостійно створюють власну реалізацію даного процесу. Доволі часто даний підхід був пов'язаний саме з мовою програмування Java та бібліотекою для реактивного програмування RxJava, також є варіант, який більш спрямований на систему Android, а саме RxAndroid. До того ж серед переваг, є масштабованість та тестування, але із мінусів, це створення та реалізація відповідних контрактів. [3]

Далі на етапі еволюції, що і залишається актуальним і на даний час, є патерн MVVM – Model-View-ViewModel (Рис. 3).

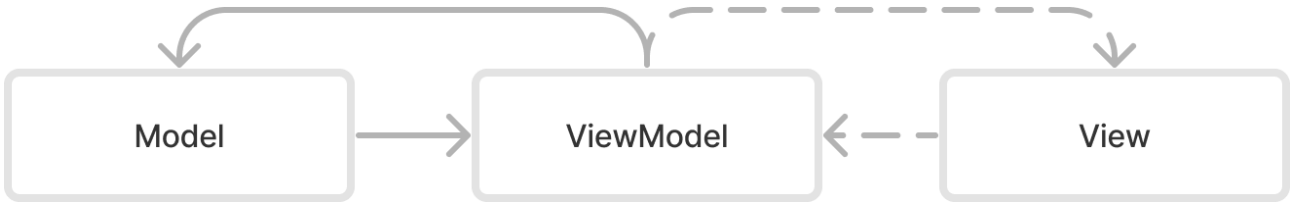

Рис. 3. Принцип взаємодії MVVM

Якщо прибрати тісний зв'язок між Controller та View у MVC, або прибрати його між Presenter та View, а також видозмінити відповідний контракт між ними, і зробити його одностороннім, отримаємо варіант реалізації ViewModel. ViewModel взагалі нічого не знає про візуальну частину за яку відповідає View, але View знає про ViewModel. Вона надає різні потоки даних на які View підписується, та виступає у ролі слухача. При отриманні нових значень, View відповідно реагує на них, тобто оновлює свій контент. А позаяк вона знає про ViewModel, то може відправляти їй UI Events, які пов'язані із взаємодією користувача із UI частиною View. Перевагою саме ViewModel є її незалежність від View, та уміння переживати життєвий цикл екрана, що дає змогу, наприклад, при перевороті екрана відновити його стан і відразу показати актуальну інформацію. Основним недоліком може бути те, що потоків даних на які підписується View, може бути безліч, якщо це комплексний екран, і UI для одних значень, може суперечити іншим, і тоді може з'являтися неоднозначна та неочевидна поведінка при взаємодії користувача із View.

Вже із розвитком нового підходу до верстки екранів в мобільній розробці на Android Jetpack Compose, розвинувся і новий підхід, новий патерн як MVI – Model-View-Intent (Рис. 4).

В основі даного патерну покладена ідея того, що екран, та всі його UI елементи, повинні відображатися відповідно одному чіткому стану. Якщо при неправильному використанні MVVM, можлива така ситуація, що екран може буде в декількох станах одночасно, то в даному підході ні. Замість декількох різних потоків даних, формується один загальний, який відповідає за поточний стан екрану, або його фрагменту. В цілому підхід із ViewModel стосовно

життєвого циклу мобільного застосунку залишився, тому і MVI має цю перевагу. [2]

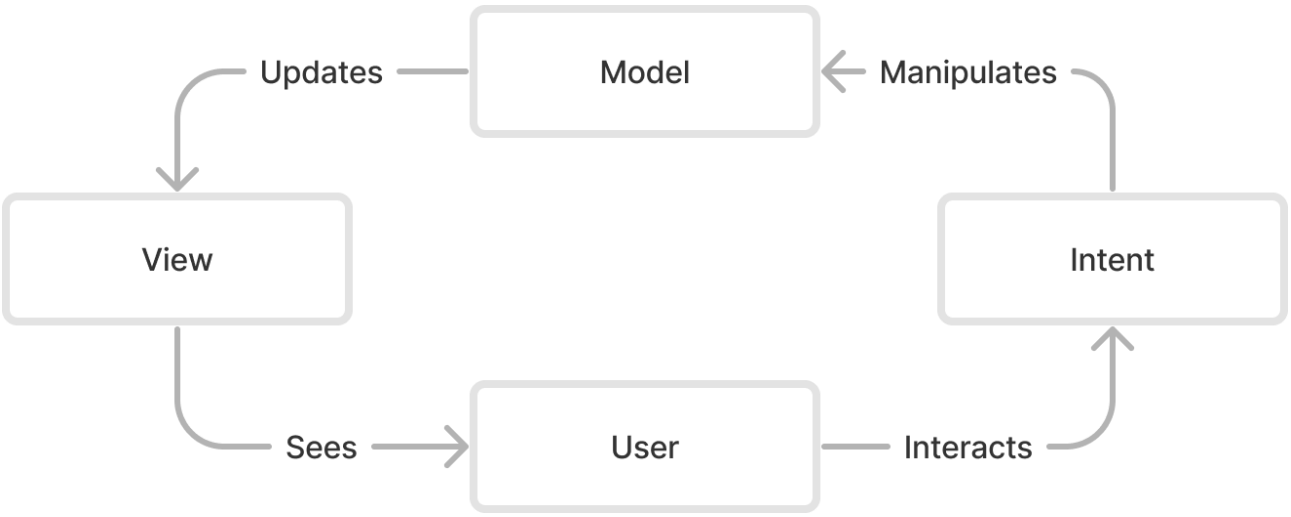

Рис. 4. Принцип взаємодії MVI

Варто відзначити, вимоги сучасної мобільної розробки та постійна зміна вимог користувачів та технологій забезпечують постійний розвиток та адаптацію архітектурних підходів. Від раннього прийняття MVC, який забезпечив основу для подальшого розвитку архітектур, до більш витончених та гнучких патернів, таких як MVP, MVVM, і навіть MVI. Як наслідок – з'явилася можливість розробляти застосунки, які є одночасно надійними, легкими у підтримці та високопродуктивними. Також, важливим аспектом є здатність архітектурних патернів витримувати тест часу, адаптуючись до нових викликів, таких як життєвий цикл застосвнку та інтеграція новітніх технологій. В цілому, еволюція архітектурних патернів у розробці Android-застосунків відображає зростання та дозрівання індустрії мобільних застосунків, вказуючи на важливість гнучкості, масштабованості та вдосконалення процесів розробки для задоволення постійно зростаючих очікувань користувачів.

Список використаних джерел та літератури

1. Alkan M. What is MVC?. *Medium*. URL: https://medium.com/@muhammetalkan/what-is-mvc-

30c6b92b7994#:~:text=It's%20an%20architectural%20pattern%20used,the%20parts%20makes%20it%20simpler (дата звернення: 29.04.2024).

2. Gazzah R. MVI Architecture With Android. *Medium*. URL: https://medium.com/swlh/mvi-architecture-with-android-fcde123e3c4a (дата звернення: 01.05.2024).

3. Shukla S. Decoding Android Architectures: MVVM, MVC, MVP, MVI. *Medium*. URL: https://shirsh94.medium.com/decoding-android-architectures-mvvm-mvc-mvp-mvi-6ee0ee313565 (дата звернення: 01.05.2024).

4. ViewModel overview | Android Developers URL: https://developer.android.com/topic/libraries/architecture/viewmodel (дата звернення: 03.05.2024).

5. Мартін Роберт. Чиста архітектура: Мистецтво створення програмного забезпечення. Видання друге. / пер. з англ. І. Бондар-Терещенко. Харків: Вид-во «Ранок» : Фабула, 2023. 368 с.

УДК542.61:546.32:549.641.23

Денисюк Роман,
кандидат хімічних наук, доцент, доцент кафедри хімії
Писаренко Сніжана,
доктор філософії з галузі знань «Хімічна та біоінженерія», асистент кафедри хімії
Камінський Олександр,
кандидат хімічних наук, доцент, доцент кафедри хімії
Житомирський державний університет імені Івана Франка

МОДИФІКАЦІЯ СТАНДАРТНИХ МЕТОДИК ВИЗНАЧЕННЯ СПОЛУК ТИТАНУ (IV) У ВОДНИХ РОЗЧИНАХ

На сьогоднішній день дуже поширеним є використання сполук титану у багатьох галузях, наприклад: косметична, фармацевтична, лако-фарбова та ін. З