DOI https://doi.org/10.15589/znp2022.1(488).9
УДК 004.4

IMPROVING THE ARCHITECTURE OF THE EXISTING SCHEDULE MANAGEMENT SOFTWARE PROGRAM OF ZHYTOMYR POLYTECHNIC STATE UNIVERSITY

ВДОСКОНАЛЕННЯ АРХІТЕКТУРИ НАЯВНОГО ПРОГРАМНОГО КОМПЛЕКСУ УПРАВЛІННЯ РОЗКЛАДОМ ДЕРЖАВНОГО УНІВЕРСИТЕТУ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

Andrii V. Morozov
morozov@ztu.edu.ua
ORCID: 0000-0003-3167-0683

Tetiana A. Vakaliuk
tetianavakaliuk@gmail.com
ORCID: 0000-0001-6825-4697

Dmytro D. Plechystyi
only.one.trogvar@gmail.com
ORCID: 0000-0002-4803-159X

Denys M. Zosimovych
unsinedz@gmail.com
ORCID: 0000-0002-1533-3882

А. В. Морозов,
канд. техн. наук, доцент

Т. А. Вакалюк,
докт. пед. наук, професор

Д. Д. Плечистий,
канд. техн. наук, доцент

Д. М. Зосімович,
здобувач

Zhytomyr Polytechnic State University, Zhytomyr
Державний університет «Житомирська політехніка», м. Житомир

Abstract. The *purpose* is to improve the architecture of the existing program complex of the schedule management of the Zhytomyr Polytechnic State University to minimize the further influence of external factors on it.
Method. The object of research is software architectures at the system and application level. The subject of research is external factors and the result of their influence on the formation and changes in software architectures. The methods of mobile technologies of cross-platform application development using the approaches of object-oriented programming, system analysis, comparative analysis, experimental analysis, and analysis of scientific and scientific-practical publications were used in the work.
Results. In this study, the architectural improvement of the existing software management schedule of Zhytomyr Polytechnic was implemented to minimize the impact of external factors on its architecture. Currently, software engineering is developing rapidly, and new research and experience in designing system and software architectures. The tools and approaches to building flexible and long-term software architectures implemented in this work can set a precedent for their further use in software development.
Scientific novelty. In the course of the work, some external factors that have an impact on the architecture of software products were identified, and the extent of this impact was investigated. They have led to the leveling of some development efforts and the re-implementation and redesign of some components of the system.
Practical importance. As a result, a mobile application for viewing the schedule was developed, which is a re-implementation of the previous version. As a result, the architecture was changed, the interface was improved, and new functionality was added. The implemented software product is intended for use by students and staff of Zhytomyr Polytechnic.
After minimizing the cost of deploying and maintaining a mobile application, you can move on to planning for its publication in the public domain after properly conducting closed and open testing.
Key words: architecture; software product; schedule management system; improve.

Анотація. *Мета* – вдосконалити архітектуру наявного програмного комплексу управління розкладом Державного університету «Житомирська політехніка» для мінімізації подальшого впливу зовнішніх чинників на неї.

Методика. Об'єкт дослідження – архітектури програмного забезпечення на рівні системи та застосунків. *Предмет дослідження* – зовнішні фактори та результат їхнього впливу на формування та зміни в архітектурах програмного забезпечення. У роботі було використано методи мобільних технологій розробки крос-платформних додатків із застосуванням підходів об'єктно-орієнтованого програмування, системного аналізу, порівняльного аналізу, експериментального аналізу, аналізу наукових та науково-практичних публікацій.
Результати. У цьому дослідженні було реалізовано архітектурне вдосконалення наявного програмного комплексу управління розкладом Житомирської політехніки для мінімізації впливу зовнішніх чинників на її архітектуру.
На поточний момент програмна інженерія розвивається стрімкими темпами, з'являються нові дослідження і досвід проєктування системних і програмних архітектур. Засоби і підходи до побудови гнучких і довготривалих програмних архітектур, що реалізовані в цій роботі, можуть стати прецедентом для подальшого їх використання для розробки програмних систем.
Наукова новизна. У ході роботи було виявлено деякі зовнішні чинники, які мають вплив на архітектури програмних продуктів, а також було досліджено масштаби зазначеного впливу. Вони призвели до нівелювання певних затрачених на розробку зусиль і повторної реалізації й перепроєктування деяких компонентів системи.
Практична значимість. У результаті було розроблено мобільний додаток для перегляду розкладу, який є реімплементацією попередньої версії. Внаслідок чого було змінено архітектуру, покращено інтерфейс, а також додано новий функціонал. Реалізований програмний продукт спрямований на використання студентами та персоналом Житомирської політехніки.
Мінімізувавши кошти на розгортання та підтримку мобільного застосунку, можна перейти до планування щодо його публікації у відкритий доступ після належно проведеного закритого та відкритого тестувань.
Ключові слова: архітектура; програмний продукт; система управління розкладом; вдосконалення.

ПОСТАНОВКА ЗАВДАННЯ

Проблема збільшення складності робіт з розробки програмних продуктів і, як наслідок, коштів на їх розробку зі збільшенням кількості функціоналу, який вже реалізовано, є відомою, очевидною та прогнозованою на сьогодні.

Є велика кількість інженерних практик, більшість з яких була виведена емпіричним шляхом, впровадження яких дозволяє з деякою ймовірністю попередити надмірну величину кошторису у разі «нашарування» функціоналу на наявний програмний продукт. Їхня ефективність була перевірена та рекомендована інженерною спільнотою, що зумовлюється кількістю статей та конференцій, на яких розробники ПЗ діляться досвідом і рекомендаціями з впровадження певних практик. Такі практики найчастіше стосуються саме проєктування моделі системи, а також її архітектури.

Архітектура програмного забезпечення – спосіб організації програмної системи, який включає всі її компоненти, види взаємодії між ними, середовище, в якому така система виконуватиме роботу, а також принципи, використані для її побудови [1].

Архітектури програмного забезпечення є досить різноманітними та з різним рівнем гнучкості до мутацій, а також можуть стосуватися різних рівнів реалізації, таких як: архітектура окремого функціоналу застосунку, додатку загалом або всього системного рішення. У цій роботі ми досліджуватимемо архітектури рівня систем та застосунків.

АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ І ПУБЛІКАЦІЙ

У своїй праці Оуен Вудс та Нік Розанскі досліджували шляхи моделювання архітектур, які відображають та тримають у балансі потреби стейкхолдерів програмних продуктів [20]. Спираючись на зазначене дослідження, Оуен Вудс описав поширені архітектурні помилки та рекомендації щодо їх уникнення, а саме [21]: нефункціональні помилки є більш складними; частий фокус на більш локальні проблеми, приділяючи менше часу аналізу ширшого контексту; правильний системний дизайн дозволяє легко нашаровувати новий функціонал; оцінка важливості функціоналу шляхом виміру кількості прибутку з точки зору окупності інвестицій (ROI, return of investment); системи та їх архітектури будуються для стейкхолдерів.

Байрон Дж. Вільямс та Джефрі К. Карвер частково проаналізували проблематику підтримки програмних продуктів, які мають застарілу або невідповідну вимогам архітектуру [22]. У науковій роботі Дзіяндзюна Дзяо та Хондзі Яня було запропоновано підхід до оцінки впливу архітектурних змін на систему [23].

ВІДОКРЕМЛЕННЯ НЕВИРІШЕНИХ РАНІШЕ ЧАСТИН ЗАГАЛЬНОЇ ПРОБЛЕМИ

Проте саме вдосконалення архітектури наявного програмного комплексу управління розкладом залишається питанням невирішеним.

Мета дослідження – вдосконалити архітектуру наявного програмного комплексу управління розкладом Державного університету «Житомирська полі-

техніка» для мінімізації подальшого впливу зовнішніх чинників на неї.

МЕТОДИ, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об'єкт дослідження – архітектури програмного забезпечення на рівні системи та застосунків. *Предмет дослідження* – зовнішні фактори та результат їхнього впливу на формування та зміни в архітектурах програмного забезпечення.

У роботі було використано методи мобільних технологій розробки крос-платформних додатків із застосуванням підходів об'єктно-орієнтованого програмування, системного аналізу, порівняльного аналізу, експериментального аналізу, аналізу наукових та науково-практичних публікацій.

ОСНОВНИЙ МАТЕРІАЛ

Протягом попередніх досліджень було проведено заходи щодо змін в архітектурі програмного комплексу. В результаті реалізована архітектура системи виглядає таким чином (рис. 1).

Рис. 1. Архітектура поточної системи

За сховище даних відповідає безсерверне рішення Firestore. Воно є DBAAS сервісом, що надається Google Firebase. Також провайдером надається бібліотека, яка дозволяє інтегрувати такий сервіс з мобільним додатком Flutter, що використовується у системі.

На поточний момент є проблема інтеграції нового сховища в мобільному додатку через те, що попереднє рішення для доступу до сховища використовувало власний API, що є несумісним з поточним. Отже, є необхідність проведення змін у мобільному додатку.

Розглянемо програмну архітектуру мобільного додатку. Відповідно до структури файлів можна зробити висновок, що рішення має багатошарову архітектуру, а саме складається з таких рівнів, як:

– презентаційний – містить програмні елементи, які відповідають за відображення інтерфейсу програми;

– доменний – містить основні інтерфейси та бізнес-логіку додатка, не містить реалізацій усіх оголошених інтерфейсів;

– рівень даних – містить реалізації деяких інтерфейсів та логіку, пов'язану з доступом до даних, у тому числі реалізацію доступу до сховищ.

Деякі ресурси та класи досить загального призначення містяться в каталогах *resources* та *core* відповідно та можуть використовуватися на кожному з рівнів.

Багатошарова архітектура – один з найбільш поширених підходів до компонування програмного коду, в якому модулі або компоненти зі спільними функціональностями організуються у вигляді горизонтальних шарів, кожен з яких виконує певну роль в усій системі та застосовується в операційних діях шару вищого порядку [2].

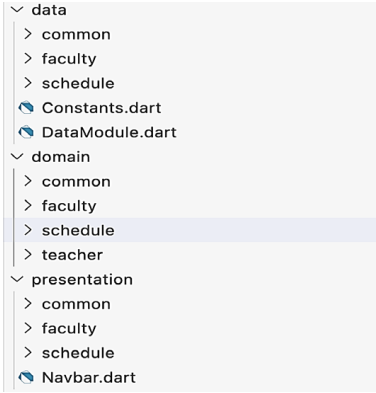


Рис. 2. Структура шарів системи

Розглянемо структуру шарів додатка детальніше (рис. 2). Можемо помітити чітке розділення компонентів у рамках кожного з шарів на категорії, наприклад факультет та розклад. Це свідчить про групування відповідних класів у рамках деякого обмеженого контексту (bounded context). Також присутні деякі компоненти загального направлення (директорії *common*).

Обмежений контекст (bounded context) – центральний шаблон концепції дизайну, що базується на предметній галузі (англ. *Domain Driven Design, DDD*). Він пропонує спосіб розв'язання проблеми моделювання частин великої системи шляхом розділення їх на обмежені контексти та явного визначення інтерфейсів комунікації між ними [3].

Domain Driven Design – підхід до розробки програмного забезпечення, який зосереджує розробку на програмуванні предметної моделі, що має цілісне розуміння процесів та правил домену [3].

Серед менш важливих проблем можна помітити однакові за сигнатурами інтерфейси, існування яких є необов'язковим через те, що вони повторюють сигнатури один одного.

Також у додатку відсутній єдиний підхід до конфігурації компонентів, про що свідчить наявність різних файлів констант.

Для реалізації патерну інверсії залежностей (англ. *Invesrion of control, IOC*) було написано декілька модулів з шаблонним (boilerplate) кодом, таких як *DataModule.dart*, *App.dart*. Серед шаблонів було реалізовано такі патерни проєктування: Singleton, Delayed Initialization.

Інверсія залежностей (Inversion of Control) – архітектурний принцип, що застосовується для досягнення меншої зв'язності різних об'єктно- орієнтованих компонентів системи. Для цього розділяється відпові-

дальність за створення та використання компонентів усередині системи [4].

Одинак (Singleton) – патерн проєктування, що дозволяє створювати лише один об'єкт певного типу у системі [5].

Відкладена ініціалізація (Delayed Initialization) – підхід, що дозволяє розділити логіку створення та ініціалізації об'єкта. Це дає можливість виконувати синхронну або асинхронну ініціалізацію лише в той момент, коли це необхідно.

У поточній реалізації компонентів інтерфейсу присутні проблеми компонування, повторення однотипного коду та неоднозначності поведінки у разі деяких сценаріїв.

З іншого боку, в додатку наявні такі рішення, як патерн «Репозиторій». Локальна інтерпретація використовує іменування «Провайдер», проте це не змінює суті класу. Використання такого патерну дозволяє змінити реалізацію сховища з мінімальними зусиллями та без прихованих наслідків для пов'язаних компонентів.

Репозиторій – патерн проєктування, що пропонує абстракцію над рівнем доступу до даних. Таким чином, система звертатиметься саме до неї для всіх пов'язаних операцій зі сховищем. Використання такого патерна допомагає досягти меншої зв'язності компонентів та реалізовувати доменні об'єкти без прив'язки до способу збереження даних [6].

Одним з головних недоліків поточної шарованої архітектури мобільного додатка також є те, що вона доволі монолітна. Це означає, що зі збільшенням кількості функціоналу розробка нового займатиме більше часу та може принести більше регресії під час тестування.

Як було визначено на основі проведеного аналізу, поточна архітектура мобільного додатка є досить недосконалою. Нова архітектура повинна відповідати таким вимогам: слідувати останнім тенденціям та принципам побудови; бути гнучкою до змін та розширення; бути не надто затратною щодо зусиль та часу для побудови; враховувати недоліки попередніх архітектур.

Технології для реалізації нової моделі повинні бути частково збережені. Кореневим фреймворком мобільного додатка має залишатися Flutter. Мова для розробки – Dart. Кінцевими платформами повинні залишатися Android та iOS. Для розробки плагінів для кінцевих платформ можливе використання мов Kotlin та Swift.

Flutter – фреймворк з відкритим вихідним кодом від компанії Google для побудови крос-платформних додатків, які є нативно скомпільованими, а також мають чудовий сучасний інтерфейс [7].

Dart – мова програмування, оптимізована для побудови швидких клієнтів для будь-яких платформ. Серед головних переваг виступають оптимізація побудови професійних інтерфейсів, продуктивна розробка зручним набором утиліт, а також велика швидкість виконання на всіх платформах, включаючи Arm, x64, x86 [8].

Попередньо мобільний застосунок було побудовано з використанням крос-платформного фреймворку, проте опубліковано було лише під платформу Android через недостачу платформних засобів для iOS розробки. Проте деякий час минув з моменту первинної розробки системи, натепер існують можливості побудови і публікації додатка під обидві мобільні платформи.

Як можна визначити з проведеного порівняльного аналізу, основною перевагою крос-платформних фреймворків є можливість дистрибуції застосунку з однієї бази коду.

Програмний фреймворк – абстракція, в якій програмне забезпечення, що надає загальну функціональність, може бути селективно змінено додатковим користувацьким кодом, який адаптує його під кінцеві вимоги розробки. Програмні фреймворки моделюються для досягнення універсальності та можливості повторного використання, щоб надати функціональний інструментарій та фундамент для побудови великих систем [9].

Одним з найважливіших показників виконання такої роботи вважаємо кількість зусиль, що були на неї затрачені. Ми не можемо дозволити собі розробку двох мобільних додатків під кожну з платформ, які були зазначені у вимогах. Через те, що сучасні крос-платформні фреймворки надають досить високу швидкодію вихідних програм, ми можемо частково пожертвувати незначною різницею в ній. Отже, вибір крос-платформних фреймворків є обґрунтованим.

З іншого боку, важливими є властивості ресурсів, які використовуються для побудови системи. Команда розробки складається з трьох людей, які займають такі ролі:

– архітектор системи, автор цього дослідження, який відповідає за проєктування архітектури всього програмного комплексу, а також його окремих компонентів; до обов'язків також входять проведення регулярного перегляду коду (code review), планування та коригування процесу розробки, постановка вимог, контроль за професійним ростом та навчанням членів команди розробки;

– інженер-програміст, роль, яку виконують два стажери, згідно з якою вони виконують фактичну реалізацію програмної системи відповідно до вимог і архітектурних рішень, прийнятих архітектором програмного комплексу.

Перегляд коду (code review) – підхід до розробки програмного забезпечення, згідно з яким код, написаний розробником, систематично переглядається та аналізується з боку його колег. Така практика дозволяє підвищувати професійний рівень членів

команди розробки шляхом регулярного обміну досвідом. Також покращується якість коду та переглядається коректність підходів та алгоритмів, використаних у реалізації деякого функціоналу [10].

Важливим є цільовий напрям розвитку стажерів, яким є крос-платформна розробка мобільних додатків Flutter мовою Dart. Попередньо стажери отримали деякий досвід розробки у цій сфері. Отже, виходячи з особливості наявних ресурсів, резонним є продовження реімплементації мобільного застосунку з використанням фреймворку Flutter та мови програмування Dart.

Процес вибору первинної архітектури є досить відповідальним. Через те, що він виконується повторно, ми можемо брати до уваги помилки попередньої архітектури.

Розпочнемо з вибору фундаментального архітектурного патерна, якому слідуватимуть усі компоненти додатка. На сьогодні в мобільній розробці використовуються такі шаблони управління станом: Redux; MVP (Model-View-Presenter); MVVM (Model-View-ViewModel); BLOC (Business Logic Component). Розглянемо переваги і недоліки кожного з них.

Redux – архітектурний патерн для управління та оновлення стану застосунку, який використовує для цього об'єкти «дій» (actions). Він диктує використання централізованого сховища для стану, який використовується по всьому додатку, з правилами, які регулюють зміну стану в передбачуваному вигляді [11].

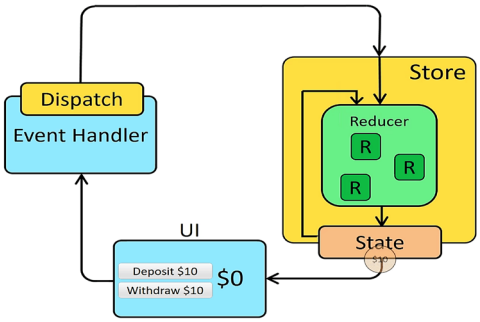


Рис. 3. Архітектура схема прикладу реалізації Redux [11]

На рис. 3 наведено приклад реалізації патерна Redux для управління простим користувацьким інтерфейсом. Схема складається з трьох компонентів: сховища стану (store); користувацького інтерфейсу (UI); диспетчера подій (dispatch).

Сховище виконує роль єдиного джерела правди для всього додатка, а також містить правила для зміни стану внаслідок обробки подій (reducers). Інтерфейс відповідає за відображення стану, а також за ініціацію подій, спричинених внаслідок дій користувача. Обов'язком диспетчера є абстрагування ініціатора дій (actions) від обробника (reducer). Також диспетчер може керувати об'єднанням дій у бетчі, які можуть бути оброблені як одне ціле, таким чином змінюючи стан атомарно для декількох дій.

Одним з головних недоліків патерна Redux є той факт, що він першочергово був розроблений для мови JavaScript. Також він вимагає іммутабельності об'єктів стану, що досить легко реалізовується у разі необхідності часткової зміни стану засобами JavaScript, проте є досить нетривіальною задачею для мови Dart через її статичну типізацію та відсутність spread-операторів для об'єктів.

Наступним розглянемо архітектурний шаблон Model-View-Presenter (рис. 4). MVP – архітектурний патерн, який розв'язує деякі проблеми патерна MVC (Model-View-Controller), що виникають за умов специфіки клієнтських застосунків, а саме [12]:

– більшість бізнесової логіки знаходиться в контролері. Протягом життя додатка розмір файлу значно збільшується, а його підтримка стає складнішою;

– через те, що користувацький інтерфейс та механізми доступу до даних є тісно пов'язаними, шари Контролеру та Представлення потрапляють у спільний компонент;

– автоматичне тестування різних шарів додатка стає досить складним через те, що більшість компонентів, які вимагають тестування, залежать від програмних інтерфейсів платформи.

Шаблон MVP вирішує зазначені проблеми й надає зручний спосіб структурування коду проєкту. Причина широкомасштабного використання такого патерна полягає в тому, що він дає модулярність, можливість тестування та більшу чистоту і підтримуваність вихідного коду. На схемі (рис. 4) зображено основні компоненти шаблону MVP.

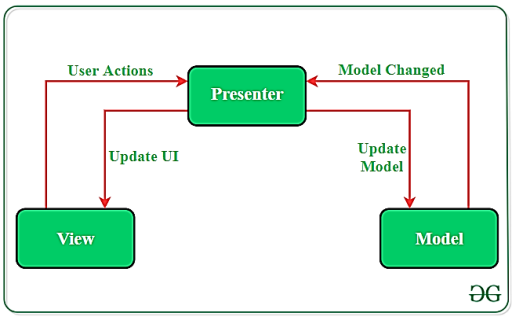


Рис. 4. Архітектурний патерн Model-View-Presenter [12]

Модель (Model) – шар для збереження даних. Він відповідає за обробку предметної логіки (бізнес-правил реального світу) та комунікацію зі сховищем даних і мережевими інтерфейсами [13].

Відображення (View) – шар користувацького інтерфейсу. Він надає візуалізацію даних та слідкує за діями користувача для того, щоб сповістити Ведучого [13].

Ведучий (Presenter) – запитує дані в моделі та виконує інтерфейсну логіку для того, щоб зробити вибір, що показати. Він управляє станом Відображен-

ня та виконує обробку на основі його сповіщень про користувацькі дії [13].

Недоліки патерну MVP полягають у його надмірній формалізації [13]. Наслідками є такі незручності:

– відображення тісно пов'язане з ведучим. Таким чином, зміни в одному з компонентів впливатимуть на інший та вимагатимуть змін у ньому;

– у міру збільшення складності користувацького інтерфейсу, а саме різноманітності дій у ньому, зростає і розмір контрактів, що робить архітектуру доволі масивною;

– відсутність реактивних підходів вимагає додаткової реалізації стандартних операцій (boilerplate) для оновлення Представлення.

У попередній реалізації мобільного застосунку було використано архітектурний шаблон MVVM. Розглянемо його детальніше.

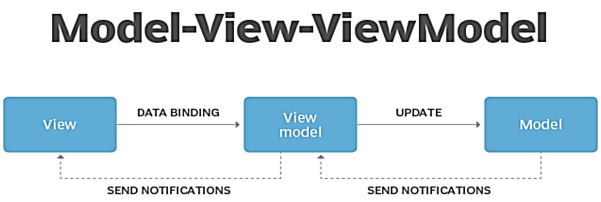


Рис. 5. Архітектурний патерн Model-View-ViewModel [14]

MVVM – архітектурний патерн, який структурує програмний додаток шляхом розділення бізнес-логіки та елементів користувацького інтерфейсу [14]. На схемі (рис. 5) зображено основні компоненти шаблону MVVM.

Представлення (View) – колекція видимих елементів, які отримують вхідні дії від користувача. Вона включає користувацький інтерфейс, анімації та текст. Вміст Представлення не може напряму змінювати те, що відображається [14].

Модель представлення (View model) знаходиться між шарами Представлення та Моделі. Тут знаходяться елементи керування Представленням. Водночас з'єднання елементів користувацького інтерфейсу з елементами керування Моделі представлення виконується шляхом байндингу [14].

Модель (Model) містить логіку програми, яку викликає Модель представлення після отримання вхідних даних від користувача через Представлення [14]. Варто зазначити, що досвід використання такого патерна був досить позитивний, адже на той час він повністю задовольняв вимоги до нього.

Досить схожим за своєю ідеологією розділення бізнес-логіки від логіки інтерфейсу є шаблон BLOC.

BLOC (Business Logic Component) – архітектурний патерн, який працює за принципом отримання подій та генерації станів. Для нього неважливим є джерело події. Однією з переваг такого шаблону над MVVM є повне відділення бізнес-логіки від представлення. В умовах сучасної мобільної розробки під різні за властивостями дисплеї та платформи мобільних пристроїв наявність відділеної від користувацького інтерфейсу логіки дає гарну можливість повторного використання компонентів коду [15].

BLOC та MVVM були досить різними на момент появи першого. Проте ця різниця поступово зникала у міру варіативності реалізацій та трактування такого патерна. Нині головною відмінністю таких архітектурних шаблонів є те, що BLOC є більш абстрактним.

Також важливим фактом є те, що BLOC було представлено та рекомендовано компанією Google, яка відповідає за розробку фреймворку Flutter. А отже, причини його використання є досить вагомими.

У результаті, проаналізувавши сучасні тренди архітектурних патернів для управління станом додатка, кореневим шаблоном для нового мобільного додатка було вибрано BLOC.

Однією з найбільших проблем попередньої архітектури була її монолітність. У результаті складність розробки значно зростала зі збільшенням кількості функціоналу, що реалізовувався у застосунку. Також такий підхід значно ускладнював процес автоматизованого тестування окремих частин.

Цього разу було запропоновано модульний підхід до організації компонентів функціоналу всередині проєкту. Майже кожен елемент функціоналу було розроблено як окремий пакет.

Flutter CLI дає можливість створення декількох типів проєктів. На рис. 6 зображено види шаблонів, на основі яких можна згенерувати проєкт.

```
-t, --template=<type>        Specify the type of project to create.

          [app]              (default) Generate a Flutter application.
          [module]           Generate a project to add a Flutter module to an
                             existing Android or iOS application.
          [package]          Generate a shareable Flutter project containing
                             modular Dart code.
          [plugin]           Generate a shareable Flutter project containing an
                             API in Dart code with a platform-specific
                             implementation for Android, for iOS code, or for
                             both.
          [skeleton]         Generate a List View / Detail View Flutter
                             application that follows community best practices.
```

Рис. 6. Шаблони проєктів Flutter

Однією з найбільш важливих проблем для вирішення на початкових стадіях є введення та вирішення залежностей.

IOC (Inversion of Control, інверсія управління) – архітектурний принцип, що диктує використання інверсії елементів управління для досягнення низької зв'язаності об'єктно-орієнтованих структур [16].

Побудова користувацького інтерфейсу додатка відбувається декларативно. У результаті описується дерево компонентів, віджетів, кожен з яких має деяку інформацію про батьківські на момент його побудови. Отже, кожен віджет будується в деякому контексті батьківських віджетів.

Описаний механізм дозволяє нам розміщувати деякі сконфігуровані сервіси у вершині дерева, таким чином вони будуть доступні всім підвершинам.

На рис. 7 зображено частину дерева віджетів реалізованого додатка. Компоненти “Dependencies” та “Startup actions” описують відповідно набір наявних сервісів та деякі дії, які необхідно виконати або почати заздалегідь.

Для реалізації IOC у рамках розробленого мобільного застосунку було використано пакет “provider”. Така бібліотека дозволяє описувати спосіб резолюції сервісу певного типу декількома способами. В додатку було використано саме фабричний метод. Віджет “Dependencies” використовується для налаштування провайдерів сервісів. Провайдери створюються на різних рівнях. Це спричинено тим, що провайдери нижчого рівня залежать від провайдерів більш високого.

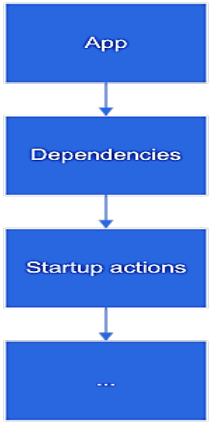


Рис. 7. Дерево віджетів

Прикладом провайдера, що має залежність від іншого, слугує провайдер “UpdateCheckBloc”, чия реалізація залежить від компонента обробника помилок “ErrorBloc” (рис. 8).

```
UpdateCheckBloc getUpdateCheckBloc(BuildContext context) => UpdateCheckBloc(
    errorSink: context.read<ErrorBloc>().errorSink,
    versionsRepository: FirestoreRepository()); // UpdateCheckBloc
```

Рис. 8. Код створення об’єкта UpdateCheckBloc

Для клієнтських додатків доволі важливими є засоби для моніторингу їхньої роботи, а також збір телеметрії щодо роботи деяких внутрішніх процесів.

Беручи до уваги той факт, що, на відміну від серверних систем, мобільні додатки встановлюються на кожний пристрій окремо, зміни коду та публікація його нової версії не будуть оновлені одночасно для всіх клієнтів. А отже, важливо планувати розгортання змін та виправлень до функціоналу заздалегідь.

Найвищим пріоритетом телеметрії мобільного додатка ми вважаємо збір даних про помилки. Таким чином, ми матимемо можливість зрозуміти, що пішло не так на пристроях деяких користувачів, а також виправляти програмні помилки ще до того, як до нас звертатимуться за підтримкою.

Для моніторингу помилок використовувався сервіс Firebase Crashlytics.

Firebase Crashlytics – інструмент інформування про програмні помилки в режимі реального часу, який допомагає пріоритезувати та виправляти найбільш поширені збої, базуючись на рівні впливу на реальних користувачів [17]. Зазначений сервіс має власний пакет для інтеграції з додатками Flutter, який і було використано у розробці. На рис. 9 наведено фрагмент коду, який обертає логіку запуску застосунку у власну зону перехоплення помилок. Обробником виступає метод пакета “firebase_crashlytics”.

```
Future<void> main() async {
  await initAsync();
  runZonedGuarded(() {
    runApp(
      Phoenix(
        child: EasyLocalization(
            supportedLocales: [Locale('uk'), Locale('en')],
            path: "assets/translations",
            fallbackLocale: Locale('uk'),
            child: App()), // EasyLocalization
      ), // Phoenix
    );
  }, FirebaseCrashlytics.instance.recordError);
}
```

Рис. 9. Створення зони контролю програмних помилок

Різниця між виконанням коду всередині зонованого контексту та звичайного try-catch механізму полягає в тому, що зона дозволяє перехоплювати асинхронні помилки, які не було оброблено шляхом синхронізації з поточним середовищем виконання.

Більш класичним варіантом обробки помилок була ручна реєстрація обробників для помилок фреймворку та поточного ізолейту. Приклад такого підходу наведено на рис. 10.

```
if (kDebugMode) {
  await FirebaseCrashlytics.instance.setCrashlyticsCollectionEnabled(false);
} else {
  FlutterError.onError = FirebaseCrashlytics.instance.recordFlutterError;
  Isolate.current.addErrorListener(RawReceivePort((pair) async {
    final List<dynamic> errorAndStacktrace = pair;
    await FirebaseCrashlytics.instance.recordError(
      errorAndStacktrace.first,
      errorAndStacktrace.last,
    );
  }).sendPort);
}
```

Рис. 10. Обробка помилок фреймворку та поточного ізолейту

Ізолейт – середовище виконання коду мови Dart. Головна різниця між ізолейтом та класичним потоком полягає в тому, що перший повністю обмежує доступ до власної ділянки пам’яті.

На поточний момент було створено засади для логування додаткової телеметрії. Логування виконується шляхом виклику методу print(). Під час розробки такі виклики виводять у консоль передані дані.

Є декілька підходів до перевірки якості функціоналу. Серед них – автоматичні та ручні перевірки. У команді розробки було встановлено політику обов'язкового покриття бізнесового коду юніт-тестами.

Юніт-тестування – напрям автоматизованого тестування програмного коду, який рекомендує написання тестів для логічно ізольованих незалежних частин системи. Для більшості мов програмування це функції, методи або властивості [18].

Тести було розміщено у директоріях "test/" для кожного з проєктів Flutter, у тому числі в головному додатку та функціональних пакетах. Такий спосіб організацій файлів тестів диктується нам засобами командного інтерфейсу Flutter. У разі виклику команди "flutter test" утиліта виконує пошук фалів за патерном "test/**/*_test.dart". Результати виконання програмних тестів додатку наведено на рис. 11.

```
zosimovych@Denyss-MacBook-Pro src % sh ./run-tests.mac.sh
00:04 +2: All tests passed!
00:02 +2: All tests passed!
00:03 +3: All tests passed!
00:08 +9: All tests passed!
```

Рис. 11. Результат виконання програмних тестів

Ручне тестування передбачає перевірку функціоналу шляхом виконання деяких дій у побудованому та поставленому на пристрій застосунку. Щоразу перед злиттям Git гілки деякого функціоналу з основною обов'язковою вимогою було встановлення тестового додатка на пристрій або емулятор для перевірки інтерфейсу та поведінки програми.

В умовах розробки під 2 кінцеві платформи перевірка додатка на обох була б значною втратою часу для більшості випадків. Найчастіше ручна перевірка виконувалася на пристрої або емуляторі iOS. Тестування функціоналу на платформі Android відбувалося лише за крайньої необхідності, тобто за наявності специфічних платформних плагінів або публікації додатка в будь-якому з режимів тестування на майданчик Google Play.

Емулятор iOS найкраще підходить для тестування інтерфейсу додатка, адже встановлення на нього тестового додатка займає найменше часу з можливих опцій. Також він використовується і в момент розробки для макетування екранів (рис. 12). Великою зручністю є те, що зазначений емулятор входить у пакет інструментів для побудови iOS застосунків XCode і не вимагає окремого встановлення.

Обмеженням емулятора є неможливість запуску додатка в режимі профайлингу та відсутність можливості інтеграції із сервісами Apple, такими як: APNS, Apple Push Notifications Service – сервіс пуш нотифікацій; Apple ID – сервіс авторизація за допомогою аккаунта Apple.

Встановлення мобільного застосунку на фізичний пристрій може бути виконано декількома способами. Перший і найочевидніший – за допомогою фізичного кабелю з'єднання телефону з робочою станцією. Такий спосіб використовується для перевірки функціоналу до того, як він потрапив в основну гілку репозиторію.

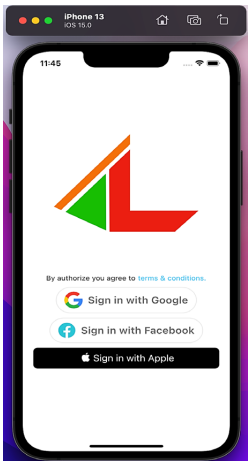


Рис. 12. Використання емулятора iOS для тестування

Після того, як деякий функціонал зливається в основну гілку, має місце закрите тестування. Apple дає можливість дистрибуції мобільного додатка в режимі закритого тестування у разі належного конфігурування акаунтів тестувальників. Для цього вони повинні встановити застосунок Test Flight та бути включеним до списку членів закритого тестування.

Після виконання зазначених вимог всередині інтерфейсу Test Flight тестувальник має змогу встановити певну версію додатка, до якої було надано доступ.

Розглянемо структуру користувацького інтерфейсу перепроєктованого мобільного додатка перегляду розкладу Житомирської політехніки. По-перше, було реалізовано можливість авторизації за допомогою таких сервісів: Google; Facebook; Apple (лише для IOS платформ). Такий функціонал було розроблено з метою синхронізації користувацьких налаштувань та властивостей між декількома пристроями. Так, зберігши належність до певної групи, дані будуть відновлені у разі авторизації цього акаунту на іншому пристрої. Після успішної авторизації користувач потрапляє на домашню сторінку. На поточний момент такою є список факультетів (рис. 13а).

Порівняно з попередньою версією інтерфейс майже не зазнав змін. Єдиним важливим зауваженням буде те, що додаток було налаштовано для виконання лише в портретній орієнтації, що дозволило спростити деякі моменти розробки, пов'язані з ускладненими вимогами до адаптивності.

Палітра кольорів також залишилася незмінною, а дизайн продовжує використовувати фреймворк Material.

Material design – адаптивна система рекомендацій, компонентів та інструментарію, яка слідує найкращим практикам побудови користувацьких інтерфейсів [19].

Дані про список факультетів, який необхідно відобразити на домашньому екрані, отримуються внаслідок використання пакета “cloud_firestore” для виконання запитів до однойменного сховища даних.

Вибір факультету є першим кроком для перегляду розкладу. У лівому верхньому кутку знаходиться елемент для відкриття бокового меню (рис. 13б). Елементи навігації було дещо змінено порівняно з попередньою версією. Було прибрано нереалізовані елементи, а також змінено їх позиціонування і сегментування. Також внаслідок появи функціоналу авторизації внизу меню було додано відповідний елемент керування для виходу з поточного облікового запису. Після вибору факультету користувач потрапляє на сторінку вибору групи та підгрупи (рис. 13в).

Дизайн другого кроку підготовки до перегляду розкладу також було перероблено. Зокрема, змінам піддалися заголовок сторінки, випадаючі списки і кнопка «Продовжити». Остання змінила свою позицію з центру до низу сторінки, а також стала займати всю ширину. Було додано можливість вибрати підгрупу для деяких груп за її наявності. Також тепер підгрупа зберігається в налаштуваннях акаунту за відповідно обраного чекбоксу «Це моя група».

Зробивши вибір, користувач потрапляє на фінальну сторінку – перегляд розкладу за вибраними параметрами.

На рис. 14 зображено оновлений екран розкладу занять. Було дещо змінено зовнішній вигляд елементів списку занять, а також було додано випадаючий знизу фрагмент для фільтрації за тижнем та підгрупою. Заголовок також відображає поточний параметр відображеного розкладу – групу. В майбутньому він може бути замінений на викладача або аудиторію.

У разі натискання на елемент розкладу з’являється спливаюче вікно з детальною інформацією про заняття. Нині наявна така інформація: викладач; групи, що присутні на занятті; аудиторія. Планується доробка функціоналу для надання посилання на віртуальну кімнату, якщо заняття проводяться дистанційно.

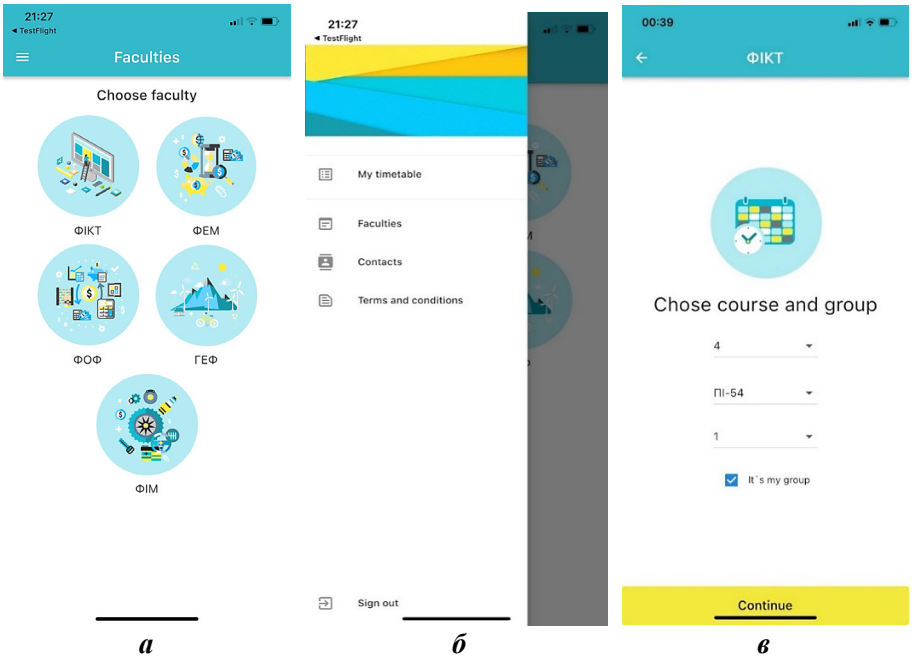
Крім спливаючого вікна, на рис. 15 також зображено новий функціонал інформування про зміни в розкладі.

ВИСНОВКИ

У цьому дослідженні було реалізовано архітектурне вдосконалення наявного програмного комплексу управління розкладом Житомирської політехніки для мінімізації впливу зовнішніх чинників на її архітектуру.

У результаті було розроблено мобільний додаток для перегляду розкладу, який є реімплементацією попередньої версії. Внаслідок цього було змінено архітектуру, покращено інтерфейс, а також додано новий функціонал. Реалізований програмний продукт спрямований на використання студентами та персоналом Житомирської політехніки.

Мінімізувавши кошти на розгортання та підтримку мобільного застосунку, можна перейти до планування щодо його публікації у відкритий доступ після належно проведеного закритого та відкритого тестувань.



а *б* *в*

Рис. 13. Список факультетів

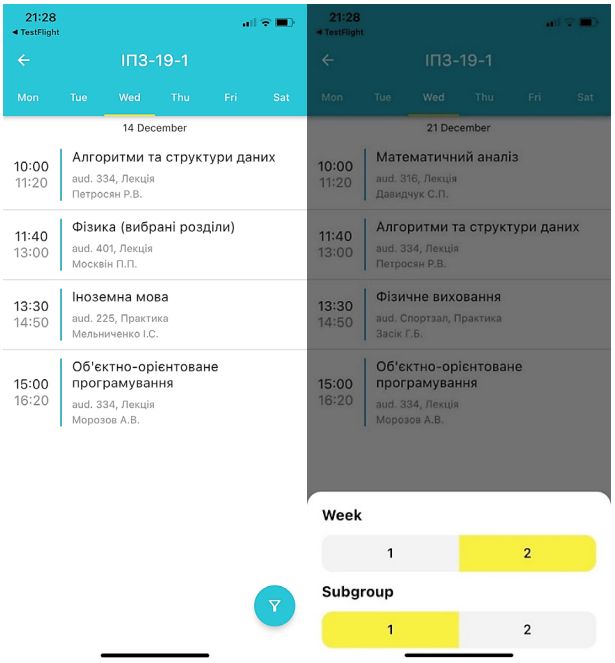


Рис. 14. Розклад занять

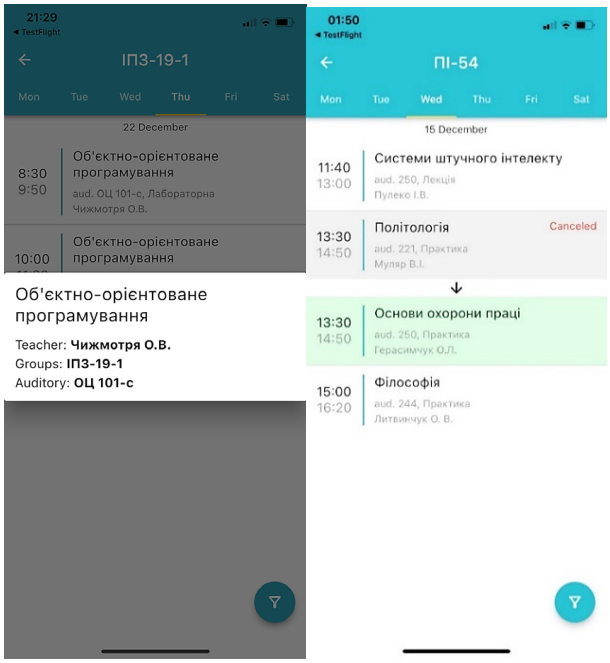


Рис. 15. Новий функціонал розкладу

На поточний момент програмна інженерія розвивається стрімкими темпами, з'являються нові дослідження і досвід проєктування системних і програмних архітектур. Засоби і підходи до побудови гнучких і довготривалих програмних архітектур, реалізовані у цій роботі, можуть стати прецедентом для подальшого їх використання у розробці програмних систем.

У ході роботи було виявлено деякі зовнішні чинники, які мають вплив на архітектури програмних продуктів, а також було досліджено масштаби зазначеного впливу. Вони призвели до нівелювання певних затрачених на розробку зусиль і повторної реалізації й перепроєктування деяких компонентів системи.

Отже, доцільно в майбутньому продовжити дослідження видів впливів на системні і програмні архітектури програмних комплексів, що дозволить зробити ще один крок до покращення процесів інженерії програмного забезпечення.

REFERENCES

[1] What is Software Architecture. Retrieved from: https://www.castsoftware.com/glossary/what-is-software-architecture-tools-design-definition-explanation-best.

[2] What is Layered Architecture – Medium. Retrieved from: https://syedhasan010.medium.com/layered-architecture-5d6e500da9ec.

[3] Domain driven design – MartinFlower. Retrieved from: https://martinfowler.com/tags/domain%20driven%20design.html.

[4] Inversion of Control – Tutorials Teacher. Retrieved from: https://www.tutorialsteacher.com/ioc/inversion-of-control.

[5] What is Singleton – Techopedia. Retrieved from: https://www.techopedia.com/definition/15830/singleton.

[6] Repository pattern – DevIQ. Retrieved from: https://deviq.com/design-patterns/repository-pattern.

[7] Flutter – Build apps for any screen. Retrieved from: https://flutter.dev.

[8] Dart programming language – Dart. Retrieved from: https://dart.dev.

[9] What is Framework – Tech Monitor. Retrieved from: https://techmonitor.ai/what-is/what-is-a-framework-4945801.

[10] What is code review – Smart Bear. Retrieved from: https://smartbear.com/learn/code-review/what-is-code-review.

[11] Redux Fundamentals. Redux overview. Retrieved from: https://redux.js.org/tutorials/fundamentals/part-1-overview.

[12] MVP architecture pattern in Android with example. Retrieved from: https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example.

[13] Top key differences between MVP and MVVM. Retrieved from: https://www.educba.com/mvp-vs-mvvm.

[14] What is Model-View-ViewModel. Retrieved from: https://whatis.techtarget.com/definition/Model-View-ViewModel.

[15] Which pattern to choose from MVVM and BLOC. Retrieved from: https://flutteragency.com/which-pattern-to-choose-from-mvvm-and-bloc.

[16] Inversion of Control. Retrieved from: https://www.tutorialsteacher.com/ioc/inversion-of-control.

[17] Firebase Crashlytics. Retrieved from: https://firebase.google.com/products/crashlytics.

[18] What is unit testing. Retrieved from: https://smartbear.com/learn/automated-testing/what-is-unit-testing.

[19] Develop Flutter Material Design. Retrieved from: https://material.io/develop/flutter.

[20] Nick Rozanski, Eóin Woods (2011) Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 704 p. Retrieved from: https://books.google.com/books?id=ka4QO9kXQFUC.

[21] Eoin Woods (2008) Top 10 Architecture Mistakes, 42 p. Retrieved from: http://jaoo.dk/dl/jaoo-aarhus-2008/slides/EoinWoodsTop10Mistakes.pdf.

[22] Byron J. Williams, Jeffrey C. Carver (2010) Characterizing software architecture changes: a systematic review. *Information and Software Technology*.Volume 52, Issue 1, Pp. 31–51. DOI: 10.1016/j.infsof.2009.07.002

[23] Jianjun Zhao, Hongji Yang, Liming Xiang, Baowen Xu. (2002) Change impact analysis to support architectural evolution. *Journal of Software Maintenance and Evolution Research and Practice,* Vol. 14(5), Pp. 317–333. DOI: 10.1002/smr.258.

Дата надходження статті до редакції: 12.04.2022
Дата затвердження статті до друку: 21.04.2022