COMPARISON OF ARCHITECTURAL PATTERNS WITHIN iOS APPLICATIONS

ПОРІВНЯННЯ АРХІТЕКТУРНИХ ПАТТЕРНІВ У ЗАСТОСУНКАХ iOS

by Mykyta Skrypchenko

Presented in Partial Fulfillment of the Requirements for the Degree

Master of Software Engineering

American University Kyiv

2024

APPROVED BY:

Tytenko Sergiy, Ph.D., Associate Professor

Dedication

To my mom and dad for unlimited love and support.

ABSTRACT

This work delves into the strategic selection of software architecture for iOS applications, underscoring the alignment of architectural decisions with specific project constraints and goals. Initial discussions centered around the challenges in using software metrics to compare various iOS architectures, leading to the proposal of a simplified framework aimed at aligning architectural choices with defined business objectives. The paper details the process of evaluating different architectural patterns — MVC, MVVM, VIPER, and TCA — considering these constraints and goals.

This work contributes to the field by providing a practical example of how architectural decisions can be tailored to specific project constraints and goals, offering insights that can be valuable for software architects and developers working on similar Swift application projects.

CONTENTS

1. INTRODUCTION

Mobile applications have witnessed an unprecedented surge in popularity and ubiquity over the past decade. This phenomenon can be attributed to several factors, including increasing smartphone adoption, changing consumer behavior, and the normalization of mobile-first approaches in businesses. As pointed out in [1], the penetration rate of smartphones reached 78.05 percent globally in 2020, and this figure is projected to rise to almost 87 percent in the United States by 2025. With this surge, mobile applications have become the primary gateway to the online world for a significant portion of the global population.

In 2010, only 27 percent of mobile users in the United States owned smartphones, a stark contrast to the anticipated 87 percent in 2025 [1]. This transformative shift has allowed not only enterprises but also individual developers and teams to create and distribute their applications to a worldwide audience. However, this diverse landscape comes with various challenges, ranging from differences in team sizes, budget constraints, and time-to-market pressures to distinct marketing strategies and business models.

Not only enterprises but individual developers and teams are allowed to create their applications and distribute them to a worldwide audience. The organizations vary in team sizes, budget constraints, time-to-market and marketing strategies, and business models.

Mobile applications have become integral to our daily lives, facilitating a wide range of activities, from online shopping to managing finances and accessing government services. Consequently, mobile application developers bear a considerable responsibility in ensuring that their products meet high standards of usability, availability, and reliability. This responsibility extends to both individual developers and large enterprises.

Amidst this dynamic landscape, the software architecture of a mobile application emerges as a pivotal factor. Software architecture encompasses the fundamental design decisions that dictate how an application will function and evolve. In the context of mobile applications, where agility, scalability, and user experience are paramount, the choice of software architecture becomes even more crucial.

High competition levels and therefore high levels of expectations from users make mobile application vendors careful in technical decisions and practices. The first thing every vendor should be aware of is the software architecture of its product.) — In the realm of mobile applications, a prescient perspective has been

validated, primarily attributed to the rapid pace of evolution within this domain. Each vendor must possess a profound understanding of the software architecture underpinning their product as a foundational consideration.

2. BACKGROUND OVERVIEW

2.1. Software Architecture and Quality Attributes for Mobile Applications Development.

How can you ensure that the software architecture you've selected is the most suitable for your needs? It is not an easy question, and there is no one right answer or a silver bullet. As stated in the work [2], software architecture is like a bet, a wager, wouldn't it be nice to know the outcome in advance? To know it, we need to step out and as we already did for the piece of code and then for software — we need to establish the rules, and measurements for a clear understanding of which architecture could fit better in certain conditions. However, the topic has already been well researched, we found plenty of space to fill towards the architecture of mobile applications.

Project Managers seek to achieve business requirements, Software Architects seek to manage the architecture according to these requirements. It can’t be managed what is not measured. But what can we measure in software architecture? While it was identified [3] the main quality attributes of the software overall, we would like to stop on those that are crucial specifically for mobile application development.

Performance. Performance pertains to the system's reactivity, which encompasses the time needed to react to stimuli (events) or the number of events processed during a specific period [3]. Performance holds significant importance in mobile software architecture due to the perpetual resource constraints of mobile devices, including limitations in terms of microprocessor power, storage capacity, and battery life.

Reliability. Reliability denotes the system's capacity to sustain continuous operation throughout its operational lifespan, often assessed through the metric known as mean time to failure [3]. For mobile systems we should consider that the network could be unstable, and some operations could fail due to the aforementioned constraints.

Availability. Availability refers to the percentage of time the system remains operational [3]. Availability is a critical consideration in mobile application architecture. Ensuring the availability of a mobile application means that it should be accessible and operational for users whenever they need it. Mobile applications can face challenges related to network connectivity, server

availability, and other factors that can affect their accessibility. To provide a positive user experience, mobile app architects need to design for availability, which may include strategies like offline capabilities, redundant servers, and efficient error handling to ensure the app remains usable even under less-than ideal network conditions.

Security. Security involves assessing the system's capacity to withstand unauthorized access attempts and service disruptions, all while maintaining its functionality for authorized users [3]. Security of Mobile apps involves implementing robust measures to protect user data and the application itself, ensuring that it remains resilient in the face of potential security risks.

Modifiability. Modifiability refers to the system's capacity for swift and cost-effective modifications, which is one of the key attributes for a large team, which is working on a mobile application.

Portability. Portability refers to a system's adaptability to function across diverse computing environments, which can encompass hardware, software, or a blend of both [3]. For a mobile architecture, it could address new versions of OSs or the device's screen size.

As stated in the work [2], quality attributes form the basis for architectural evaluation, but simply naming the attributes by themselves is not a sufficient basis on which to judge an architecture for suitability. In a perfect world, the quality requirements for a system would be completely and unambiguously specified in a requirements document.

In the [4] it was described the variability inherent in the design of the software architecture and how that variability argues for the use of measures that are tailored to the context of the specific organization. For instance, different roles seek to have different points of view and therefore different metrics, specifically, there was elaborated on Software Architect and Project Manager roles.

That's also true for different domains – there is no architecture of size-fits-all, no silver bullet, we should carefully integrate our requirements into Quality Attributes and therefore pick the right architecture for the solution.

A more in-depth examination underscores the significance of a comprehensive assessment of mobile software architectures with a focus on their thorough implementation to ensure the robustness of these attributes and more. Such examination should first of all rely on practical comparison of the different

architecture codebases. Furthermore, we have posed an essential research question: Is there a suitable method for selecting the right architecture based on the specific requirements of a given project? As the field of Mobile Application Development continues to expand, there is a growing need for a more in-depth exploration of this topic, particularly concerning the criteria used to evaluate the quality attributes of Mobile Software Architectures.

2.2. Metrics in Software Engineering

In the domain of software engineering, metrics are systematically categorized into two principal types: 'product metrics' and 'process metrics.' Product metrics encompass quantitative measures of the software artifact at various stages of its lifecycle, ranging from the requirement specification phase to the deployment of the fully functional system. These metrics may include, but are not limited to, the intricacies of software design architecture, the volumetric analysis of the program (measured in either source or object code), and the extent of documentation generated.

Contrastingly, process metrics are oriented towards the evaluation of the software development methodology. They encompass a range of measurements such as the cumulative duration of the development cycle, the specific methodologies employed in the process, and the average experience level of the development personnel.

Beyond this bifurcation, software metrics are further distinguished into 'objective' and 'subjective' categories. Objective metrics are characterized by their capacity to yield consistent values irrespective of the observer, assuming the observer is adequately qualified. A quintessential example of an objective product metric is the count of lines of code (LOC), which, given a uniform definition, should yield a consistent measurement across different evaluators. In contrast, subjective metrics are influenced by individual judgment and interpretation, leading to potential variability in measurements among equally qualified observers. An illustrative instance of a subjective product metric is found in the COCOMO cost estimation model, wherein software is classified as 'organic,' 'semi-detached,' or 'embedded.' [14] While classification is straightforward for most software products, those situated at the interstice of these categories may be subject to divergent classifications by different experts.

2.3. Goal-driven Measurement

Formulating architectural metrics for software systems presents a complex challenge, as the architecture significantly influences subsequent development

and project management strategies. These strategies include the allocation of coding tasks and defining developmental increments. Most prevalent metrics for functioning software systems tend to aggregate the outcomes of architectural, development, and management decisions, rather than directly evaluating the efficacy of the software architecture itself.

In the context of goal-oriented software development, organizations aim to align software creation with specific business and market objectives, such as enhancing market share, reducing production costs, or tailoring products to specific customer requirements. This necessitates a consideration of various factors, including the requisite skill sets for developers (e.g., programming languages, software tools), the partitioning of coding labor (e.g., number of required programmers), and the structuring of development phases (e.g., sequential completion stages of the software product).

In this framework, the implementation of goal-driven software measurement is imperative. This approach ensures that any metric, whether derived directly or adapted from existing literature, is both relevant and beneficial within the specific context of the organization. This goal-driven approach to measurement is integral to ensuring that the software development process is aligned with and effectively contributes to the achievement of the organization's overarching objectives [15].

In the context of contemporary mobile application development, which frequently employs scaled teams and methodologies, a heightened emphasis on specific product and process metrics is advocated. This is particularly pertinent given the singular binary nature of mobile applications, which stands in contrast to the decomposable structure of cloud-based services. Such metrics are not only applicable to mobile software but also extend to hardware domains like the Internet of Things (IoT), where resource constraints are a significant consideration. This expanded metric framework is vital for a comprehensive assessment and optimization of both the software artifact and the development process in these technology spheres.

2.4. Modern Mobile Application Metrics Schema

As was discussed earlier, there several dimensions of Metrics. There are also different viewpoints like project metrics and product metrics. Different quality attributes could be applied to the mobile application as well. As a key driver for the application there are Business Goals. All of these could be wrapped in Architectural schema.

Architectural schema (Figure 1) is proposed vision for aggregating all metrics to see the overall picture of defining and attributing the key metrics for the app.

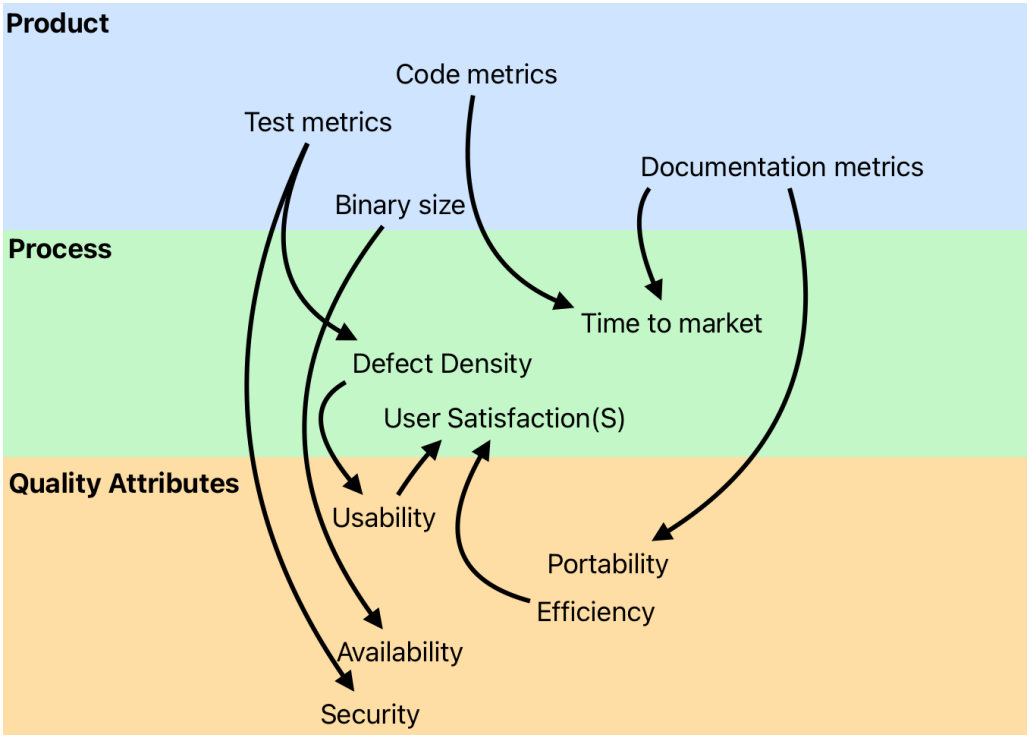

Figure 1. Architectural Metrics Schema

Figure 1 illustrates an architectural schema designed to aggregate various metrics, offering a comprehensive overview of the key performance indicators for a mobile application. The schema distinguishes between product and process metrics, emphasizing their interrelation and influence on quality attributes.

Product metrics include code metrics, which measure the characteristics of the codebase; test metrics, which assess the effectiveness and thoroughness of testing procedures; binary size, which indicates the compiled application size; and documentation metrics, reflecting the quality and comprehensiveness of documentation.

Process metrics are represented by defect density, providing insight into the number of defects relative to the size of the software; time to market, measuring

the period from ideation to launch; and user satisfaction, a subjective metric that gauges the end-user's contentment with the application.

Quality attributes form the foundation of the schema and are directly influenced by both product and process metrics. Usability denotes the ease with which users can interact with the application. Availability measures the app’s operational performance and readiness for use. Security represents the application's ability to protect against unauthorized access and vulnerabilities. Additionally, portability assesses the ease of transferring the app across different environments or platforms, and efficiency evaluates the application's performance in terms of resource utilization.

At the core of the architectural schema are business goals, which drive the entire development process. These goals ensure that the metrics align with strategic objectives, ultimately shaping the application’s development and refinement. This schema encapsulates the multifaceted nature of application development metrics, providing a strategic framework for continuous improvement and alignment with business objectives.

3. MOBILE APPLICATION ARCHITECTURES IN ACTION

3.1. Mobile Architectures

Over the years of mobile app development, the projects have rapidly become more and more complex and harder to maintain. Companies behind major platforms — Google and Apple have sought to develop guidelines for native app architectures. At the same time, developers have adapted existing architectural patterns for mobile platforms and continued to evolve them to adjust to growing complexity [5]. In this section we will briefly review the most popular patterns: MVC, MVVM, VIPER, and TCA architecture patterns.

Moreover, the continuous evolution of these architectural patterns is indicative of a broader trend in software engineering – the move towards more modular, scalable, and maintainable systems. As these patterns evolve, they often become more aligned with principles of clean code and sustainable software practices. This alignment is critical in an era where software systems are increasingly complex and integral to both business operations and daily life.

3.2. The business case

The Opora application, selected as a focal case study in the realm of mobile technology and community engagement, exemplifies a digital solution addressing the complexities of non-monetary aid collection in Ukraine. This mobile application serves as a vital platform for volunteers, streamlining the coordination and distribution of resources within a networked community.

At the heart of Opora's functionality lies a user-friendly interface facilitating essential operations. These include a secure login/logout mechanism, ensuring user authenticity and data protection. A pivotal feature of the app is the capability to create new aid collections or modify existing ones, which empowers volunteers to respond dynamically to evolving needs within the community.

The application boasts a comprehensive system for cataloging and accessing information about all collections and sponsors integrated within the network. This feature not only enhances transparency but also promotes efficiency in matching resources with requirements. The combination of these functionalities positions Opora as a potent tool in mobilizing and organizing non-monetary assistance, tailored to the specific context of Ukraine's volunteer community and context.

In the context of the Opora application, the research necessitated the definition of specific quality attributes, constraints, and business goals. These parameters were

established in a free form, tailored to the unique needs and circumstances of the project. The following outlines the key characteristics and objectives set forth for the Opora application:

Team Size:

The development team consists of up to three individuals. This small team size implies a need for an architecture that supports efficient collaboration, clear communication, and the ability to handle multiple aspects of the project with limited human resources.

Time to Market:

A critical emphasis is placed on the rapid deployment of the application, making time to market a crucial factor. This necessitates an architectural approach that facilitates quick development cycles, minimal complexity, and ease of deployment.

System Safety and Reliability:

The application is required to be safe and error-prone, indicating a high priority on robustness and fault tolerance. It should be designed to handle errors gracefully and maintain consistent performance under various conditions.

Maximum Availability:

Ensuring the application is highly available is a key goal. This involves strategies for redundancy, reliable hosting, and efficient handling of user load to ensure the application remains accessible at all times.

Geographical Focus:

The primary distribution of the app is intended for Ukraine, with global support not being a priority at this stage. This localization aspect influences the design decisions, particularly in terms of language support, cultural considerations, and regional compliance requirements.

Development Methodology:

Agile development practices are to be applied. This approach calls for flexibility in design, the ability to adapt to changing requirements, and a focus on iterative development with regular feedback loops.

These defined parameters served as guiding principles for the development of the Opora application, influencing both the selection of the software architecture and the overall development approach. They reflect a tailored strategy to meet the specific goals and constraints of the project, ensuring that the application is not only functional and reliable but also aligned with the project's targeted outcomes and operational context.

3.3. Micro-frontends as a modern pattern for large scale apps

Microfrontends in mobile development represent a paradigm shift, akin to the microservices approach in backend systems, but focused on the frontend development of mobile applications. This architectural style addresses the complexities inherent in modern mobile app development, particularly when teams are large or distributed, and the application itself is feature-rich or frequently updated.

The Microfrontend strategy can provide maximum reusability, reduce costs, improve efficiency, avoid resource waste, improve business operation efficiency, quickly and stably support company expansion or even multinational business while providing consistent user experience [13].

At its core, microfrontends involve decomposing a monolithic mobile application into smaller, more manageable pieces. Each 'microfrontend' is a semi-independent module responsible for a distinct feature or domain within the app. This modular structure allows for greater agility and scalability in development and maintenance. Teams can work on different features in parallel, reduce dependencies, and deploy updates for individual components without affecting the entire application. This approach is particularly beneficial in a multi-team environment, where coordination and integration of code can be challenging.

Furthermore, microfrontends in mobile development facilitate the use of a diverse set of technologies and frameworks, as each module can be developed with the most suitable technology stack without impacting others. This flexibility is vital in the rapidly evolving landscape of mobile development, where new tools and best practices emerge continuously.

In this research, microfrontends architecture would be used for the sake of reuse code that is not related to the architectures itself. It would simplify the development while would not affect the result. The Dependency Injection pattern would be used as a glue between different applications.

The final architecture would be as shown on Figure 2.

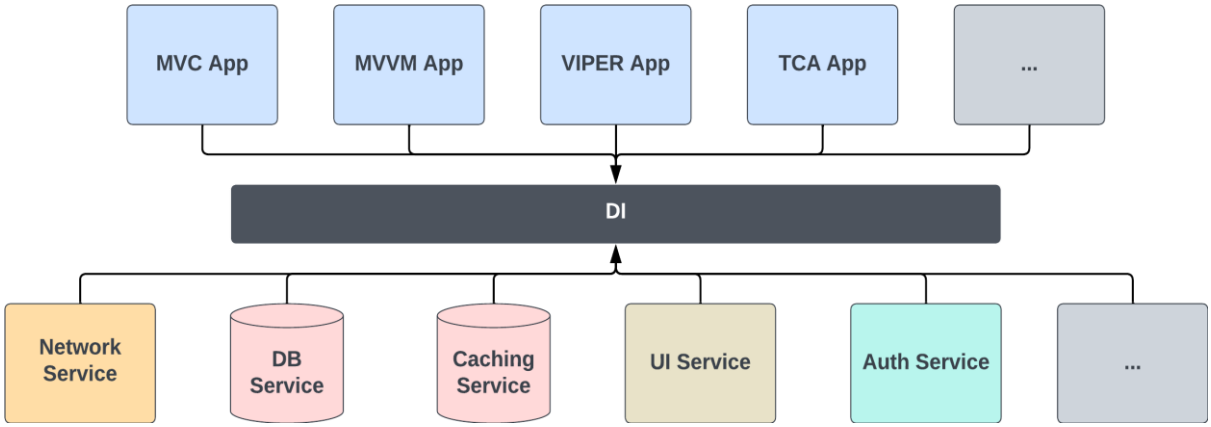

Figure 2. *Overall Architecture*

For the implementation of microfrontend architecture Tuist tool was picked.

Tuist is a command-line tool that supports the microfrontend architecture in iOS and macOS projects. It leverages a declarative approach (Figure 3) for writing project configurations, allowing developers to define their project structures and dependencies in a clear, concise manner. By catering to microfrontend architecture, Tuist helps in managing multiple, smaller front-end modules within a large application, facilitating scalability and modular development. This declarative configuration approach not only minimizes errors but also streamlines the project setup and maintenance processes. Tuist automates tasks like project generation and dependency management, enhancing productivity and ensuring consistency in large, complex codebases worked on by development teams.

```
let project = Project(
    name: "Opora",
    organizationName: "OporaOrg",
    targets: [
        Target(
            name: "MVCApp",
            platform: .iOS,
            product: .app,
            bundleId: "com.oporaapp.MVCApp",
            infoPlist: .default,
            sources: ["Sources/MVCApp/**"],
            dependencies: [
                .target(name: "SharedServices")
            ]
        ),
        Target(
            name: "TCAApp",
            platform: .iOS,
            product: .app,
            bundleId: "com.oporaapp.TCAApp",
            infoPlist: .default,
            sources: ["Sources/TCAApp/**"],
            dependencies: [
                .target(name: "SharedServices"),
                .external(name: "ComposableArchitecture")
            ]
        ),
        Target(
            name: "ViperApp",
            platform: .iOS,
            product: .app,
            bundleId: "com.oporaapp.ViperApp",
            infoPlist: .default,
            sources: ["Sources/ViperApp/**", "Sources/SharedServices/**"],
            dependencies: [
                .target(name: "SharedServices")
            ]
        ),
        Target(
            name: "MVVMApp",
            platform: .iOS,
            product: .app,
            bundleId: "com.oporaapp.MVVMApp",
            infoPlist: .default,
            sources: ["Sources/MVVMApp/**"],
            dependencies: [
                .target(name: "SharedServices")
            ]
        ),
        Target(
            name: "SharedServices",
            platform: .iOS,
            product: .framework,
            bundleId: "com.oporaapp.SharedServices",
            infoPlist: .default,
            sources: ["Sources/SharedServices/**"],
            dependencies: [
                .external(name: "Swinject")
            ]
        )
    ]
```

Figure 3. Tuist configuration.

3.4. MVC Pattern

Model-View-Controller (MVC) architectural pattern was first introduced by Trygve Reenskaug while working on Smalltalk based system [6]. MVC architecture operates with three entities: model, view and controller. A model object encapsulates the data and implements the logic for manipulating the data. It can be as simple as storing a string, or more complex, like storing an object with multiple value fields or retrieving data remotely. A view object is responsible for the user interface. View object receives updates of model's state and updates the user interface accordingly. Controller object receives user input events from the view and changes the state of the model [7].

Over the years, the MVC architecture gained a lot of popularity among developers thanks to its clear separation of concerns [8]. The MVC pattern is shown in Figure 4. The Class Diagram is depicted in Figure 5.

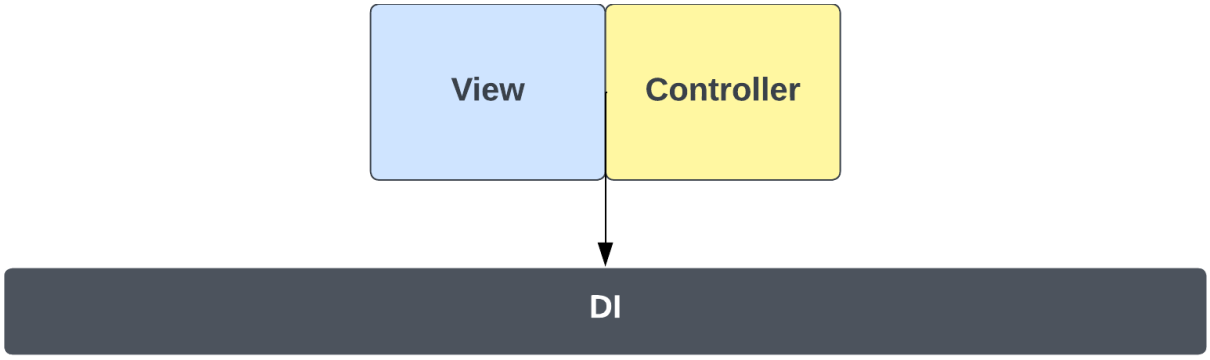

Figure 4. MVC Architecture

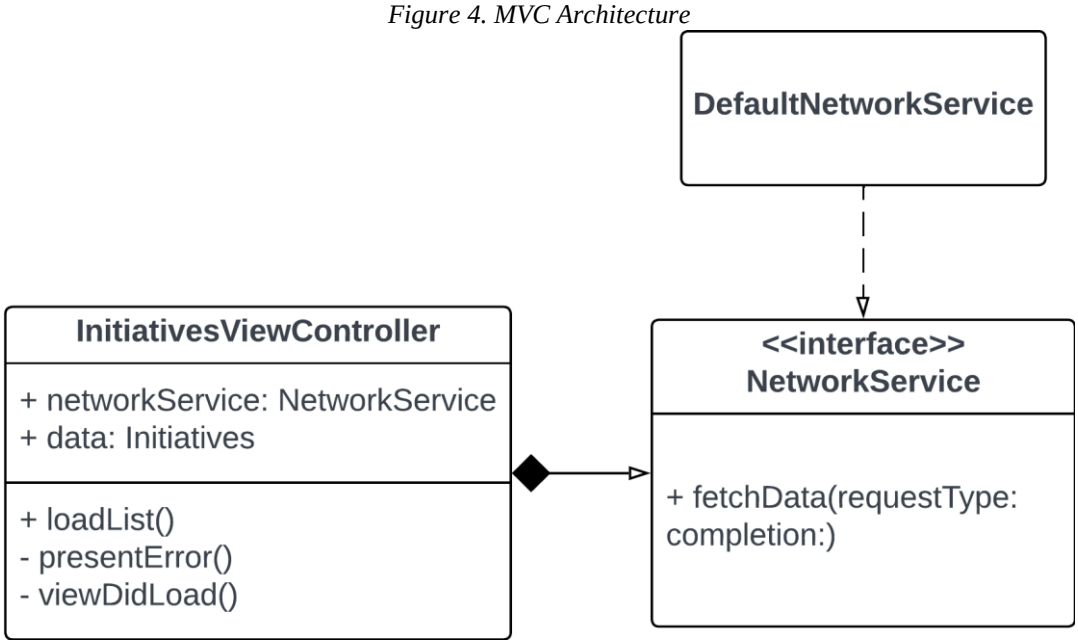

Figure 5. MVC Class Diagram

The folder structure (Figure 6) for one Feature in Opora app is quite simple and straightforward. Assuming, our model is provided by the DI and common for all architectures inside out Microfronted solution.

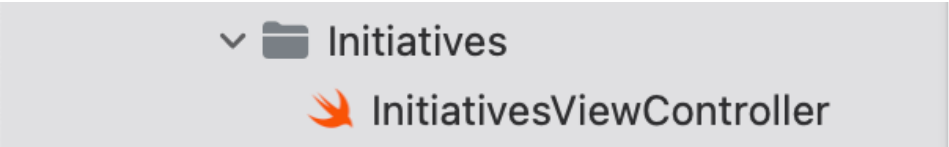

Figure 6. MVC Code Structure

```
15  class InitiativesViewController: UIViewController {
16      private let uiService: UIService
17      private let resolver: Resolver
18      private var tableView: UITableView = UITableView(frame: .zero, style: .insetGrouped)
19      private var data: Initiatives = []
20      private let networkService: NetworkService
21
22      init(resolver: Resolver = DIContainer.container,
23           uiService: UIService = UIServiceImpl(),
24           networkService: NetworkService = DefaultNetworkService()
25      ) {•••}
32
33      required init?(coder: NSCoder) {•••}
36
37      override func viewDidLoad() {•••}
55
56      private func loadList() {•••}
82
83      private func presentError() {•••}
86  }
87
88  extension InitiativesViewController: UITableViewDataSource {•••}
103
104 extension InitiativesViewController: UITableViewDelegate {•••}
110
```

Figure 7. Initiatives ViewController in MVC

For illustrative example, simple network call in the Opora application was picked to compare the architectures flows between each other. In MVC architecture all the logic is handled within the ViewController file as can be examined from Table 1.

Lifecycle action	**Description**
View Activation	The application's first screen is displayed by the OS, activating the View, which handles the user interface. Within the View we also trigger the network call to the Network Service, awaiting for the result and updating the view corresponding to the returned result from the Network Service.

Table 1. MVC Lifecycle.

The MVC architectural pattern offers several benefits and limitations when applied in software development. A primary advantage of MVC is its inherent simplicity, which facilitates ease of understanding and implementation. This

characteristic renders it particularly suitable for smaller-scale applications or for implementing discrete features within larger systems.

However, the MVC pattern also exhibits certain drawbacks. A notable limitation is the tight coupling observed between the View (user interface) and the Controller (business logic handler). It is clearly visible from structure of the code (Figure 3) as well as from the implementation in Swift (Figure 7). This interdependence often necessitates concurrent modifications in both the View and the Controller in response to changes in either component, leading to potential development inefficiencies.

Another concern associated with MVC pertains to scalability. In the context of large-scale applications, the Controller component may become increasingly complex and challenging to manage effectively. This complexity can hinder scalability and maintainability of the application.

Furthermore, the MVC pattern may not be ideally suited for applications with complex user interfaces (UIs). Managing intricate UI interactions within the MVC framework can become cumbersome and less efficient, potentially impacting the overall user experience and the ease of future modifications.

3.5. MVVM Pattern

Model-View-ViewModel (MVVM) architecture was introduced by Microsoft in the early 2000s and was used in Silverlight and WPF [9]. It became prevalent in native app development for both Android and iOS.

MVVM operates with three objects: Model, View and ViewModel. Similarly to MVC, the Model is responsible for data access, data persistence, and communication. The View is dedicated to the user interface and should not contain any logic. The new object ViewModel is the core of the MVVM pattern, and its place is between Model and View. The view is bound to ViewModel via data binding. The data binding updates the View automatically when ViewModel's state changes. A View can have multiple references to ViewModel. However, the ViewModel should not have any dependencies on View (Figure 8). This is a big difference compared to MVC, where presenter or controller will set view in code [10]. This way MVVM offers even greater separation of concerns and greater testability of ViewModel [11].

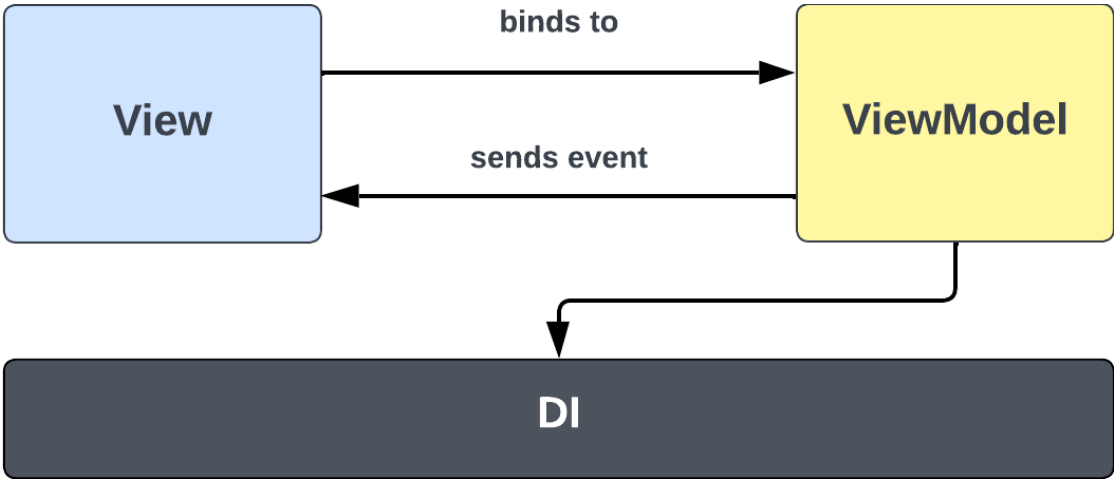

Figure 8. MVVM Architecture

That difference from the MVC pattern is clearly visible from the Code Structure (Figure 9) perspective as well.

A critical feature of MVVM is the use of data binding to connect the View with the ViewModel. This binding allows for automatic updates to the View in response to changes in the state of the ViewModel. It enables a single View to reference multiple ViewModel instances, enhancing flexibility and scalability.

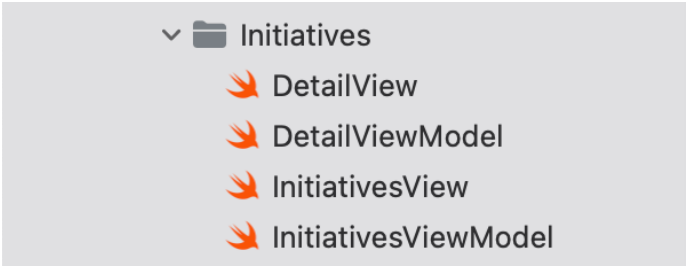

Figure 9. MVVM Code Structure

For Opora case, MVVM with data binding was implemented (Figure 10, Figure 11). It can be examined that it is a decoupled version of the same functionality that was implemented for MVC earlier. Now, our codebase has two separated concerns View (Figure 11) and ViewModel (Figure 10). The first one represents the view user can interact with. The view only responsible for drawing the resources, wait for user interaction consequently pass the results of interaction to the ViewModel. The latter is responsible for loading data and preparing it for presentation in the View.

```
6   class InitiativesViewModel: ObservableObject {
7       enum LoadingState {
8           case loading
9           case loaded(items: Initiatives)
10          case error(Error)
11      }
12
13      @Published var selectedItem: Item?
14      @Published var loadingState: LoadingState = .loading
15      private var cancellables: Set<AnyCancellable> = []
16
17      private let uiService: UIService
18      private let networkService: NetworkService
19
20      init( ••• ) { ••• }
27
28      func loadList() { ••• }
51
52      func itemTapped(item: Item) { ••• }
55  }
```

Figure 10. MVVM Initiatives ViewModel.

```
struct InitiativesView: View {
    @ObservedObject var viewModel: InitiativesViewModel
    @Environment(\.uiService) var uiService: UIService

    var body: some View {

        VStack {
            switch viewModel.loadingState {
            case .loading:
                ProgressView("Loading...")
            case let .loaded(data):
                List {
                    ForEach(data) { element in
                        ZStack {
                            uiService.createInitiativeCard(initiative: element)
                            NavigationLink (
                                destination: DetailView(viewModel: DetailViewModel(itemID: element.id))
                                    .navigationBarTitleDisplayMode(.inline)
                            ) { EmptyView() }
                                .opacity(0)
                        }
                        .listRowInsets(EdgeInsets(top: 0, leading: 0, bottom: 0, trailing: 0))
                    }
                }
                .navigationTitle("Ініціативи")
            case let .error(error):
                Text(error.localizedDescription)
            }
        }
        .onAppear {
            viewModel.loadList()
        }
    }
```

Figure 11. MVVM Initiatives View

MVVM implementation of the simple network call within Opora app would contain steps described in Table 2. The Class diagram is show on Figure 12.

Lifecycle action	**Description**
View Activation	The OS displays the first screen, activating the View, which is responsible for the user interface.
View - ViewModel Interaction	The View interacts with the ViewModel, typically through data binding or event handling, triggering a specific action.
Network Call	The ViewModel, handling the application's logic, initiates an asynchronous network call. It processes the action received from the View and deals with the networking logic.
State Update	Once the network response is received, the ViewModel updates its state based on the outcome of the network call.
View Update	The View, observing changes in the ViewModel through data binding, automatically updates the UI to reflect the new state.

Table 2. MVVM Lifecycle

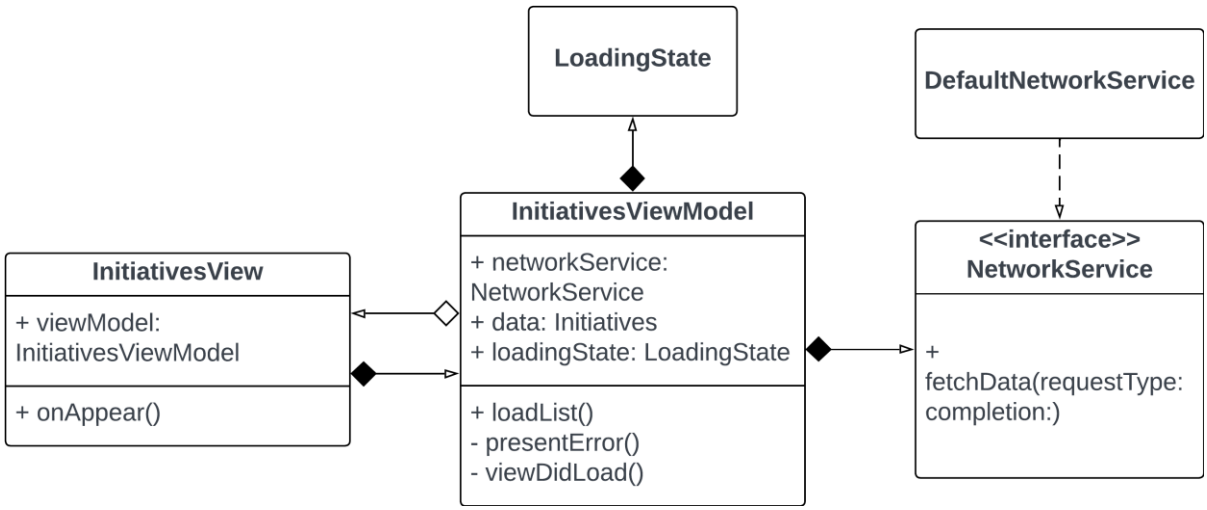

Figure 12. MVVM Class diagram

The MVVM architecture offers a cohesive blend of advantages and drawbacks. On the plus side, data binding significantly reduces boilerplate code by enabling automatic updates of the view in response to changes in the data model. This feature streamlines the development process and enhances the responsiveness of the application. In terms of testability, the ViewModel's design allows for straightforward unit testing, independent of the user interface, thus facilitating a more robust and reliable codebase.

A key benefit of MVVM is the clear separation of development roles it provides. This separation enables developers to concentrate on the business logic while allowing designers to focus exclusively on the user interface. Such specialization not only improves productivity but also enhances the quality of both the application's functionality and its user experience. Additionally, MVVM is particularly well-suited for applications with complex user interfaces and rich user interactions, offering improved scalability and flexibility in managing intricate UI components and workflows.

However, MVVM is not without its challenges. It presents a steeper learning curve compared to the MVC pattern, requiring developers to gain a deeper understanding of its underlying principles. This complexity can be daunting, especially for those new to the architecture. Moreover, MVVM can introduce additional complexity and overhead, which may be particularly noticeable in smaller applications where the benefits of the pattern might not fully justify the extra effort. Finally, limited platform support is another consideration; not all development frameworks fully support MVVM, which could restrict its use in certain environments or necessitate additional workarounds.

3.6. VIPER Pattern

It was introduced by Mutual Mobile as a way of applying the Clean Architecture to iOS applications. The name of the architecture is derived from the acronym View, Interactor, Presenter, Entity, and Routing [12].

The View is responsible for the user interface, displaying information to the user and capturing user inputs. The Interactor handles business logic and prepares data for the Presenter. The Presenter, acting as a mediator, takes data from the Interactor, formats it for display, and sends it to the View. Entities are simple data models used by the Interactor. Lastly, the Router manages navigation and data flow between different parts of the application, ensuring a clear and decoupled approach to application architecture (Figure 13). This structured division not only

simplifies the development process but also makes unit testing and debugging more straightforward, leading to more robust and maintainable iOS applications.

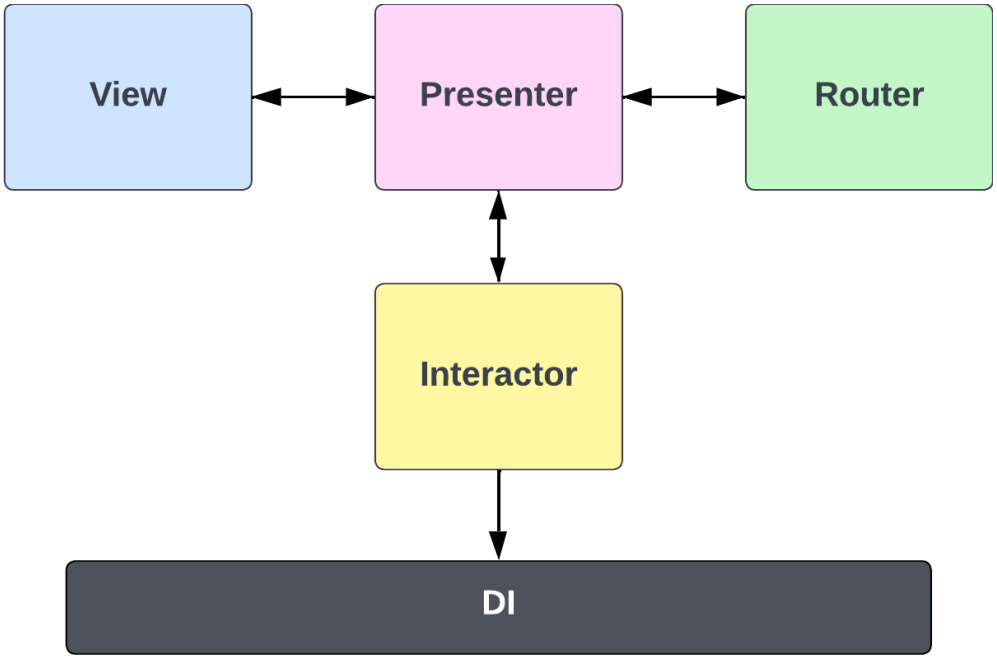

Figure 13. VIPER Architecture

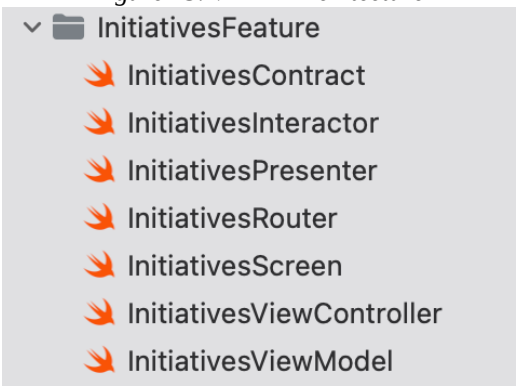

Figure 14. VIPER Code Structure

The network call flow for the Opora app would contain the following steps as it depicted on Table 3.

Lifecycle action	**Description**
View Layer Activation	OS displays the first screen of the application. In response, the View layer, responsible for UI presentation, becomes active.

Presenter Communication	Upon the activation of the View, it communicates with the Presenter. This communication is triggered by user interactions or lifecycle events. Presenter, serving as the intermediary between the View and the rest of the architecture, receives this input.
Interactor Involvement	Presenter, after processing the input from the View, forwards the relevant action to the Interactor. The Interactor handles the business logic of the application, which in this case involves making a network call. This step ensures that the business logic is kept separate from the UI logic.
Network Call	The Interactor performs the network call asynchronously. It manages the details of the network request, including waiting for the response. This encapsulation of network logic within the Interactor aligns with the single responsibility components.
State Management and Response Handling	Once the network call is completed and a response is received, the Interactor processes this response.
Presenter Updates	Interactor communicates the outcome back to the Presenter. Presenter then prepares the data in a format suitable for display.
View Update	Presenter sends the formatted data back to the View, which updates the UI. This involve displaying data received from the network or updating the screen based on the results of the network call.

Table 3. VIPER Lifecycle

```swift
protocol InitiativesView: ErrorViewPresentable {
    func setViewModel(_ viewModel: InitiativesViewModel)
    func setErrorView(_ viewModel: ErrorViewModel, visible: Bool)
}

protocol InitiativesEventHandler: AnyObject {
    func didLoadView()
    func didTapOnItem(with id: String)
}

protocol InitiativesInteractorInput: AnyObject {
    func retrieveList()
}

protocol InitiativesInteractorOutput: AnyObject {
    func didReceiveList(with initiatives: Initiatives)
    func didReceivedError()
}

protocol InitiativesWireframe: AnyObject {
    func openItem(id: String)
}
```

Figure 15. VIPER InitiativesContract.

VIPER architecture, adhering to a protocol-oriented or contract-oriented development approach (Figure 15), emphasizes the definition and adherence to specific protocols or contracts between its components. This method of development underpins the interaction model within the VIPER framework.

By following this protocol-oriented approach, VIPER ensures that each component remains independent and loosely coupled with others. This design not only facilitates easier testing and maintenance but also enhances the reusability of components. Developers can focus on specific aspects of the application in isolation, leading to cleaner, more manageable code. It reflects on the VIPER file structure (Figure 14).

The Class Diagram for VIPER architectural pattern can be observed on Figure 16.

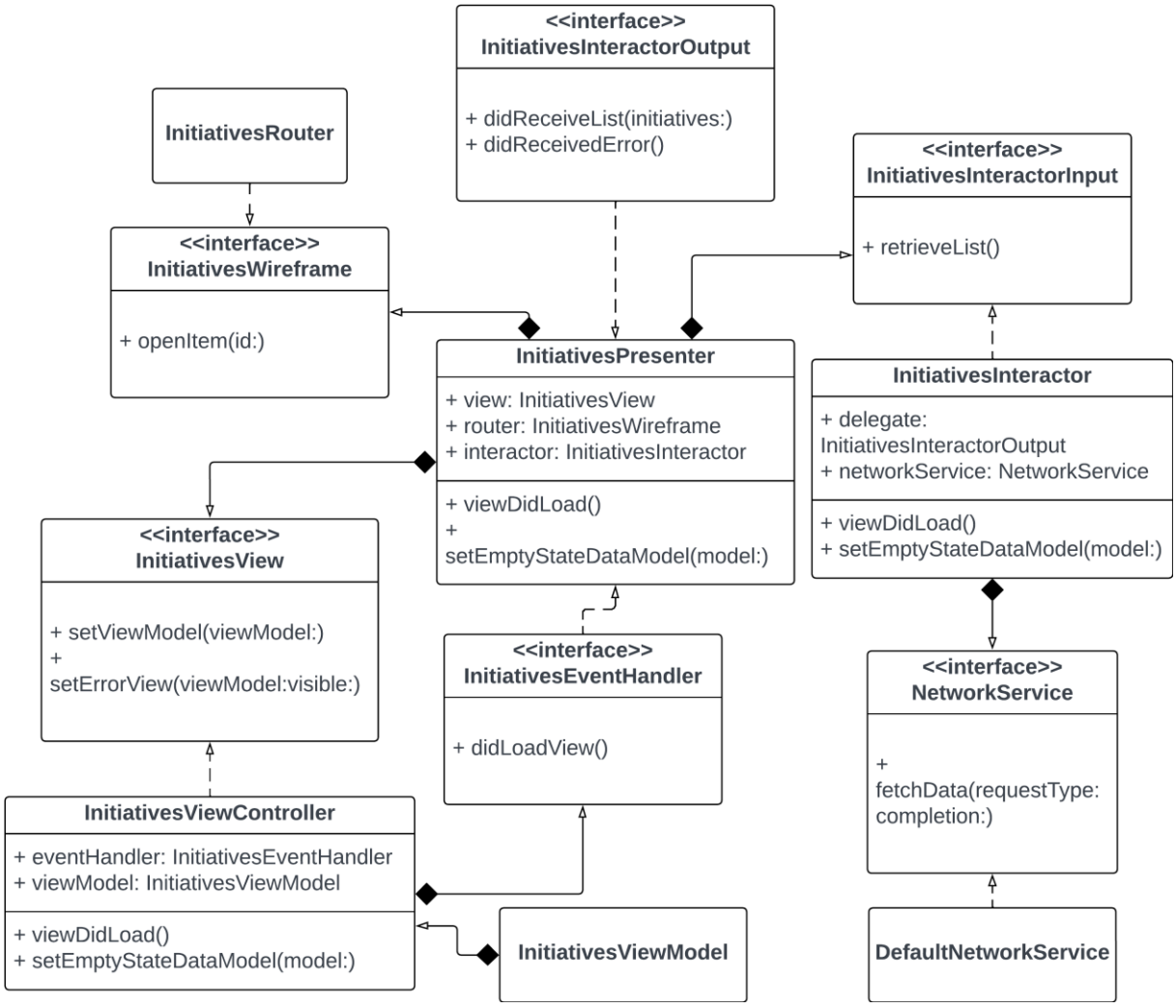

Figure 16. VIPER Class Diagram

Among its pros, VIPER is highly celebrated for its modular structure, promoting a clean separation of concerns with distinct roles allocated to each component. This modularization not only clarifies the structure of the application but also enhances its readability and maintainability. Another significant advantage is the ease of testing it offers; each component in the VIPER architecture can be independently tested, allowing for thorough and efficient unit testing. This aspect is particularly beneficial for ensuring the reliability and stability of the application. Additionally, VIPER is well-suited for large-scale applications. Its modular nature greatly aids in scalability and maintainability, making it an ideal choice for complex projects where these factors are critical.

On the flip side, the VIPER architecture is more complex, especially when compared to more straightforward frameworks like MVC or MVVM. This complexity is accompanied by a steeper learning curve, which can be a barrier for developers new to this architectural pattern. For small-scale projects, VIPER

might lead to overengineering, introducing unnecessary complexity that could slow down development and add to the cognitive load without tangible benefits.

Furthermore, the inherent complexity and the multitude of components involved in VIPER architecture can potentially increase development time. This increase is often due to the additional planning and coordination required to manage the multiple layers and interactions within the architecture.

3.7. TCA Pattern

The Composable Architecture (TCA) (Figure 17) in Swift is a modern framework designed to handle complex state management in a coherent and predictable manner. Rooted in the principles of functional programming, TCA promotes unidirectional data flow, making the state changes in an application easy to follow and reason about. Central to this architecture is the concept of a "store" that holds the application's state. This state is modified by sending "actions" to a "reducer", a pure function that takes the current state and an action and returns a new state.

TCA emphasizes composability, allowing developers to break down application logic into smaller, reusable components. Each component has its own store, reducer, and state, which can be composed into larger ones. This modular approach facilitates scalability and reusability, making it easier to manage large applications and share functionality across different parts of the app.

Another key feature of TCA is its integration with SwiftUI, Apple's declarative UI framework. TCA works seamlessly with SwiftUI's reactive paradigm, enabling developers to build responsive and state-driven user interfaces with less boilerplate code. Additionally, TCA supports powerful side-effect management and integrates well with Combine, Apple's framework for handling asynchronous events. This integration provides a robust environment for handling complex workflows, asynchronous tasks, and more, all while maintaining a clear and maintainable codebase.

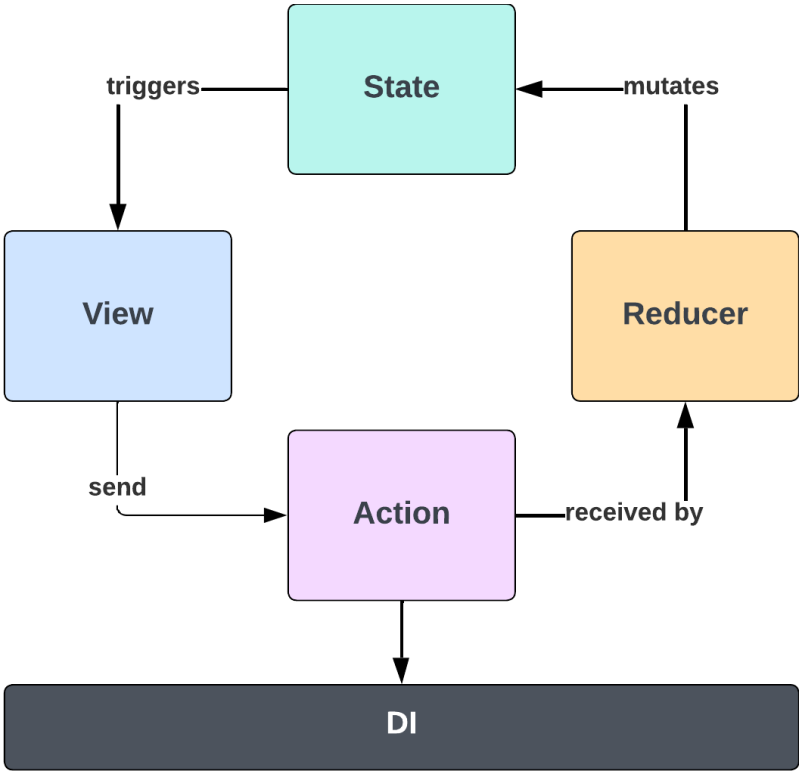

Figure 17. TCA Architecture

From the code structure perspective (Figure 18), for each feature in most cases there will be two files: View (Figure 20) and Feature, which contains State (Figure 19) — type that specifies the necessary data for a feature to execute its logic and display its user interface, Reducer (Figure 21) — a function that delineates the process of transitioning the current state of an application to its subsequent state in response to an action; additionally, it is tasked with identifying and returning any effects that need to be executed, like API calls, which can be accomplished through the return of an 'Effect' value, and Action (Figure 22) — type that represents all of the actions that can happen in your feature, such as user actions, notifications, event sources [16].

Figure 18. TCA Code Structure.

```
struct State: Equatable {
    var isLoading: Bool = false
    var data: Initiative?
    var id: String
}
```

Figure 19. TCA State structure.

```
struct InitiativesView: View {
    let store: StoreOf<InitiativeFeature>
    @Dependency(\.uiService) var uiService

    var body: some View {
        NavigationStackStore(self.store.scope(state: \.path, action: { .path($0) })) {
            WithViewStore(self.store, observe: { $0 }) { viewStore in
                VStack {
                    if viewStore.isLoading {
                        ProgressView("Loading...")
                    } else if let data = viewStore.data {
                        List {
                            ForEach(data.indices, id: \.self) { index in
                                ZStack {
                                    let element = data[index]
                                    uiService.createInitiativeCard(initiative: element)
                                    NavigationLink(state: DetailFeature.State(id: element.id)) {
                                        EmptyView()
                                    }
                                    .opacity(0)

                                }
                                .listRowInsets(EdgeInsets(top: 0, leading: 0, bottom: 0, trailing: 0))
                            }
                        }
                        .navigationTitle("TCA List")
                    } else if let error = viewStore.error {
                        Text("Error: \(error.localizedDescription)")
                    }
                }
                .onAppear {
                    viewStore.send(.loadData)
                }
            }
```

Figure 20. TCA Initiative View.

```
struct InitiativeFeature: Reducer {
    @Dependency(\.uiService) var uiService
    @Dependency(\.networkService) var networkService

    struct State: Equatable { ••• }

    enum Action { ••• }

    var body: some ReducerOf<Self> {
        Reduce { state, action in
            switch action {
            case .loadData:
                let request = APIRequest.getInitiatives
                state.isLoading = true
                return .run { send in
                    let data = try await networkService.fetchData(for: request)
                    do {
                        let initiatives = try JSONDecoder().decode(Initiatives.self, from: data)
                        await send(.dataLoaded(initiatives))
                    }
                    catch {
                        await send(.dataLoadingFailed)
                    }
                }
            case let .dataLoaded(data):
                state.isLoading = false
                state.data = data
                return .none
            case .dataLoadingFailed:
                state.isLoading = false
                return .none
            case .path:
                return .none
            }
        }
        .forEach(\.path, action: /Action.path) {
            DetailFeature()
        }
    }
}
```

Figure 21. TCA Reducer struct.

```
enum Action {
    case loadData
    case dataLoaded(Initiative)
    case dataLoadingFailed
}
```

Figure 22. TCA Action enumeration.

The networking flow for the Opora application would be as follows on Table 4.

Lifecycle action	**Description**
View Activation	The app waits until the OS shows the first screen and trigger the subscriber via the WithViewStore mechanism, passing the Action as a parameter.
Reducer processing	The received action will be processed by the Reducer.

Network Call	The Reducer will trigger the network call asynchronously and await the result.
View Update through binding	Reducer will change the state according to the network call result.

Table 4. TCA Lifecycle.

The Composable Architecture (TCA), known for its emphasis on composability and unified state management, presents a unique set of advantages and challenges within software development.

The Class diagram for TCA architecture is presented on Figure 23.

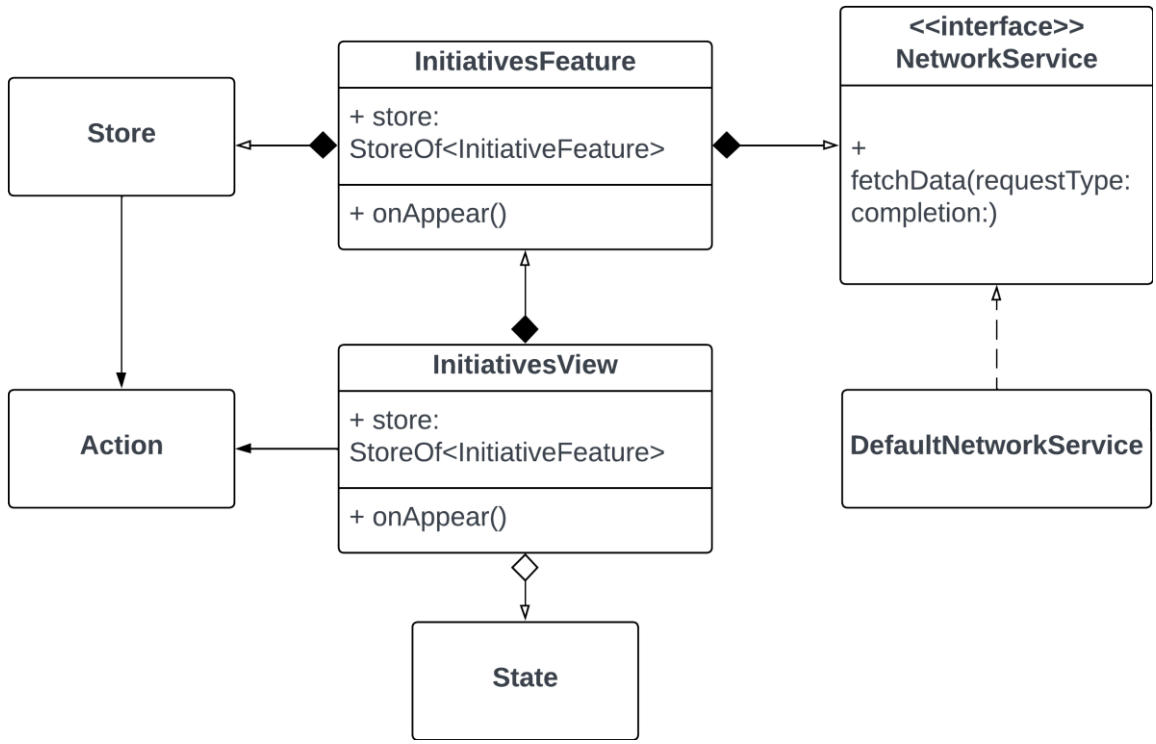

Figure 23. TCA Class Diagram

One of the primary benefits of TCA is its emphasis on composability. This aspect allows developers to build more complex functionalities by seamlessly combining simpler, more manageable elements. This modular approach facilitates the construction of intricate features while maintaining code clarity and reusability. Another significant advantage of TCA is its approach to unified state management. By providing a clear and structured framework for handling application state, TCA makes it easier to track, manage, and debug state-related issues. This feature is particularly beneficial in maintaining consistency and

predictability across the application. Additionally, TCA's architecture greatly enhances testability. Its predictable and centralized state management system allows for more straightforward and thorough testing, ensuring a more robust and reliable application.

However, TCA also has its challenges. One of the main drawbacks is the complexity involved in state management, especially in large-scale applications. As applications grow and become more complex, managing the state within the TCA framework can become increasingly challenging, potentially leading to complications in maintaining and scaling the application. Another hurdle is the learning curve associated with TCA. Understanding and effectively implementing TCA requires a good grasp of functional programming concepts, which may be challenging for developers who are not familiar with these principles. Lastly, TCA's effectiveness often depends on specific tools or libraries. This dependency can limit its applicability in certain projects where such tools may not be available or preferred.

3.8. Comparative analysis of architectures and measurement results

In the pursuit of establishing a methodology for selecting the most suitable architecture for a given project, this study employs several open-source tools: SwiftLint [17], SonarQube [18], and swift-code-metrics [19]. These tools were utilized to measure static objective metrics across various architectures. However, it's important to note that these tools alone do not comprehensively address all the testing requirements for the Opora application, as elaborated in the 'Conclusions and Future Work' section.

Notably, the Opora application's simplicity permits the evaluation of certain metrics such as LOC (Lines of Code) and NOC (Number of Classes) using a straightforward grep command [19]. This approach is indicative of the applicability of basic tools in assessing simpler software structures, although it may not suffice for more complex evaluation needs

For LOC the following command was used:

```
find . -name '*.swift' | xargs grep -h '' | wc -l
```

For counting number of classes (including structures and enumerations) was used the next grep command:

```
find . -name '*.swift' | xargs grep -E 'class |struct
|enum ' | wc –l
```

	MVC	MVVM	VIPER	TCA
LOC	348	346	929	403
NOC	4	13	19	15
Binary Size	5265172 bytes	5401150 bytes	5394256 bytes	8008348 bytes

Table 5. Objective metrics for given architectures

Table 5 provides a clear depiction of the lack of correlation between the number of classes or lines of code and the binary size. It's noteworthy that the TCA possesses an additional dependency, resulting in an increase of up to 40% in its binary size compared to other architectures. In contrast, the VIPER architecture exhibits a significantly higher count of classes— up to four times more than MVC and approximately a third more than both MVVM and TCA. Despite this disparity in class count, VIPER maintains a binary size relatively comparable to that of MVVM. This observation underscores the non-linear relationship between class count, lines of code, and the resultant binary size in different software architectures. The analysis reveals a series of notable correlations between the number of classes in a software system and various performance metrics, based on empirical data gathered from a diverse range of software development projects. The key findings are summarized as follows:

1. Impact on Market Speed for Different Team Sizes

    a. Small Teams: There is a significant inverse relationship between the number of classes and time to market. Large code base and hight cohesion leads small teams to a slowdown in market delivery, likely due to heightened complexity and coordination demands.

    b. Large Teams: Conversely, for larger teams, the data shows a positive correlation. The decomposition into more classes appears to enhance market speed, likely because large teams can leverage parallel development streams.

Architecture patterns like MVC have low cohesion, usually having all logic inside one big class. On the contrary, architectures like VIPER have loose coupling, therefore, it is more suitable for large teams.

2. Testability and Its Ripple Effects

    a. User Satisfaction: A direct positive correlation between the number of classes and testability was observed, which in turn correlates positively with user satisfaction. This suggests that more modular systems, allowing for detailed testing, result in a higher quality, more user-friendly product.

    b. Security: There is a positive relationship between testability and security. Systems with a higher number of classes, and therefore better testability, exhibited *Table 5. Objective metrics for given architectures*

    c. fewer security vulnerabilities, contributing to a more secure software product.

3. Binary Size and Availability

The analysis indicates a notable correlation between smaller binary sizes and increased availability. This finding is underpinned by the fact that smaller binaries are easier to distribute and require less storage, making the software more accessible to a wider user base. For developing countries, it is crucial to have smaller binary sizes due to limitation of internet speed and cost. In some cases it should be a major concern for worldwide business.

These results elucidate the multifaceted implications of structural design decisions in software development. They particularly highlight the need for careful consideration of team size and project objectives when deciding on the architecture and complexity of a software system.

3.9. Architecture for Opora case

In the context of the Opora application project, a detailed analysis was undertaken to select an appropriate software architecture that aligns with the predefined business requirements and quality attributes. These requirements were as follows:

- Team size of up to three people.
- Time to market is a critical factor.
- The system must prioritize safety and be error-prone.

- The application's primary market is Ukraine, with no immediate need for worldwide support.
- Agile development methodologies are to be implemented.

To determine the most suitable architecture for the Opora application, it was essential first to define the relative Quality Attributes, referencing the framework presented in Figure 1.

Given the paramount importance of the time-to-market metric, Code metrics such as Lines of Code (LOC) were considered vital for eliciting the architectural pattern. Additionally, the chosen architecture needed to be highly testable to ensure a low crash rate. Another critical factor was the small team size.

Upon review, the VIPER architecture was recognized as highly testable due to its loose coupling between classes. However, the adoption of VIPER could negatively impact the time-to-market metric, suggesting it might not be the ideal choice for this project. Similarly, the MVC architecture was deemed unsuitable for a team size of three, due to its high complexity and the scalability considerations for potential future worldwide distribution.

This analysis narrowed the choices down to two architectures: MVVM and TCA. Each presents its own advantages and disadvantages in the context of this project. The final decision was influenced by subjective metrics, such as the team's proficiency level and the project's size.

The TCA architecture, while offering numerous benefits, is complex due to the intricacies of functional programming. This complexity might pose a challenge for teams with less experience. Additionally, there was a concern regarding the application size when using TCA, due to the necessity of incorporating a third-party library, which could increase the app size by approximately 3-5MB.

For the Opora project, the team composition was assumed to comprise senior developers for the build phase and mid-level developers for maintenance. The application was classified as a small-to-medium project, without the requirement for extensive feature capabilities. Based on these considerations, the TCA (The Composable Architecture) was selected as the most suitable architecture.

The decision to choose TCA was influenced by several factors:

Team Expertise: The presence of senior developers in the initial development phase aligns well with the complexity of TCA, as they are more likely to possess the necessary skills to effectively implement functional programming paradigms.

Maintenance Considerations: Mid-level developers in the maintenance phase would benefit from TCA's modular and scalable structure, facilitating easier updates and bug fixes.

Application Size Concerns: Although TCA requires the integration of an additional library, the projected increase in application size (3-5MB) was deemed acceptable. This increase is relatively modest and unlikely to pose significant distribution challenges, especially considering the localized nature of the application’s deployment.

Alignment with Agile Practices: TCA's modular nature fits well with the agile development methodology adopted for the project. It allows for more flexible and iterative development, with the ability to easily adapt and modify components as the project evolves.

In conclusion, the selection of TCA for the Opora project was a strategic decision, balancing technical considerations with the project's specific requirements and team dynamics. This choice underlines the importance of a thorough architectural evaluation process, considering various factors such as team composition, project scope, and specific business objectives, to ensure the successful alignment of the software architecture with the overall goals of the project.

3.10. Other projects

In the realm of software development, the selection of an appropriate architectural pattern is critically influenced by the specific requirements and constraints unique to each project. This paper has demonstrated that these varying requirements directly inform the choice of architectural pattern. For instance, in the context of large-scale projects characterized by global reach and a team size exceeding five members, the VIPER architecture emerges as a suitable choice. VIPER's modular and segmented approach is particularly conducive to managing complex, expansive projects. Additionally, for such projects, it may be prudent to contemplate a modular approach wherein different components of the project utilize distinct architectural patterns. This strategy facilitates easier scaling and maintenance as the project grows.
Conversely, for smaller-scale projects or those managed by a single engineer, the MVC architecture warrants consideration. The simplicity and straightforward nature of MVC make it an attractive option for projects with limited scope and resources. However, it is imperative for teams opting for MVC to be cognizant of the potential drawbacks associated with this pattern. These include issues related to scalability and the risk of codebase becoming unmanageable as the

project evolves. Awareness of these challenges is essential for ensuring long-term sustainability and maintainability of the software.

In summary, the choice of software architecture is a decision that must be carefully calibrated to align with the project's scale, team size, and specific operational objectives. This tailored approach ensures that the chosen architecture not only facilitates the immediate development goals but also supports the project's evolution over time.

CONCLUSIONS

In this research, the use of software metrics for evaluating iOS app architectures was thoroughly explored, and a framework to guide the selection of mobile app architectures based on business goals and operational limits was developed. The study focused on creating Opora, an iOS app designed to help volunteers gather non-monetary aid in Ukraine, employing various architectures like MVC, MVVM, VIPER, and TCA, integrated using a microfrontend approach to handle the complexity of using multiple architectures simultaneously. The framework was successfully applied to choose the TCA architecture for Opora, aligning with specific requirements and constraints.

This work contains the following results and achievements.

1. Overview of measurement of the software architectures

A comprehensive overview was achieved in measuring software architectures, particularly emphasizing the necessity of establishing precise rules and measurements. Key attributes specific to mobile applications such as Performance, Reliability, Availability, Security, Modifiability, and Portability were meticulously examined. The establishment of a holistic evaluation methodology underscored the importance of a thorough assessment beyond merely listing quality attributes. The relevance of categorizing software metrics into 'product' and 'process', and further into 'objective' and 'subjective' categories was highlighted, ensuring the precision and contextual relevance of metrics. The implementation of a goal-driven measurement approach successfully aligned software development with business and market objectives, demonstrating that metrics are integral tools contributing towards the project's overarching goals.

2. Microfrontend architecture discussed

This research articulated the pivotal role of Microfrontend architecture in modern mobile app development, mirroring the backend's microservices approach to address complexities, particularly in large or distributed teams. Highlighting the architecture's capacity for enhancing reusability, efficiency, and cost-effectiveness, the study delved into the decomposition of monolithic apps into manageable, autonomous microfrontends, each responsible for specific app features. This modular approach not only fosters development agility and scalability but also allows the use of diverse technologies, enhancing flexibility in the dynamic mobile development landscape. The practical application of microfrontends, facilitated by the Tuist tool, was underscored, emphasizing its

utility in project configuration, task automation, and ensuring consistency across complex, large-scale codebases, marking a significant stride in modular, efficient mobile app development.

3. **Created the Opora application using 4 different architecture patterns: MVC, MVVM, VIPER, TCA**

During the work, the Opora application was developed using four different architecture patterns: MVC, MVVM, VIPER, and TCA, each addressing the complexities of non-monetary aid collection in Ukraine. A user-centric interface, incorporating secure authentication and dynamic aid management, was successfully integrated across all architectural models, streamlining resource coordination and distribution. The application also featured a comprehensive and transparent cataloging system for collections and sponsors, enhancing operational efficiency and transparency. Key project parameters, including a compact, agile development team, rapid deployment, robust system safety and reliability, and maximum application availability, were meticulously defined and adhered to. The project's focus on the specific geographical context of Ukraine influenced design decisions, ensuring local relevance and compliance. This multifaceted approach, guided by clearly defined goals and constraints, led to the successful development of the Opora application, demonstrating the application's functionality, reliability, and alignment with the operational needs and strategic goals of Ukraine's volunteer community across various architectural frameworks.

4. **Discussed architecture patterns in depth**

Explored various architectural patterns in software development, emphasizing their distinct characteristics, benefits, and limitations. The Model-View-Controller (MVC) pattern, known for its simplicity and clear separation of concerns, was discussed, highlighting its suitability for smaller applications but noting its limitations in scalability and the tight coupling between the View and Controller. The Model-View-ViewModel (MVVM) pattern, with its strong separation of concerns and enhanced testability, was examined, pointing out its steeper learning curve and potential complexity in smaller applications. The View-Interactor-Presenter-Entity-Routing (VIPER) pattern was explored for its modular structure and ease of testing, suitable for large-scale applications, but with noted complexity and potential for increased development time. Finally, The Composable Architecture (TCA) was presented for its emphasis on composability and unified state management, enhancing testability and maintainability of complex functionalities, while acknowledging the complexity involved in state management and the steep learning curve associated with functional

programming concepts. Each architectural pattern offers unique advantages and faces specific challenges, making the choice of architecture crucial and dependent on the specific requirements and scale of the project.

5. **Created a framework that helps to pick architecture pattern according to the inputs like constraints and requirements**

During this research, the complexities involved in employing software metrics for the assessment and comparative analysis of various iOS application architectures were extensively examined. Additionally, a streamlined framework was conceptualized and proposed. This framework is specifically designed to facilitate the decision-making process in selecting mobile app architectures, aligning with predefined business objectives and operational constraints.

A case study central to this research was the development of a project named Opora. Opora is an iOS application service crafted to assist volunteers in amassing non-monetary aid within Ukraine. The development process of Opora was characterized by the utilization of several widely recognized iOS application architectures, including MVC, MVVM, VIPER, and TCA. To effectively manage and mitigate the complexities associated with the integration of these diverse architectures, a microfrontend architectural approach was employed. This strategy was instrumental in addressing the multifaceted challenges presented by the simultaneous use of multiple architectural paradigms.

It was shown previously that the aim of the architecture is to achieve the business goals, through providing the software decisions. Business goals are set by product or board person, that means that to pick the right architecture there should be a framework or rules. Architectural schema (Figure 1) depicts the first step of this framework.

With the help of this Framework, and according to the requirements and constraints presented TCA architectural pattern for Opora application was picked as the best suited one.

Let’s discuss some limitations found during the work.

Limitations

In the realm of software development, particularly for Swift-based applications, certain limitations have been identified in the currently available tools. These limitations primarily pertain to the support for microfrontend architecture and the extent of metrics support for Swift applications. The detailed analysis of these limitations is presented below.

Low to No Support for Microfrontend Architecture

None of the existing code metrics tool for Swift, has witnessed sufficient tooling support for microfrontend architecture. Microfrontend architecture, a design approach that structures frontend applications as a composition of distinct features or components that can be developed and delivered independently, is gaining traction for its flexibility and scalability. However, Swift tools have not yet evolved to support this architectural paradigm effectively. This gap in tooling can lead to challenges in implementing microfrontend architecture in Swift applications, such as difficulties in component integration, increased complexity in managing dependencies, and challenges in ensuring consistent build and deployment processes.

Insufficient Metrics for Swift Applications

Another notable limitation is the lack of comprehensive metrics support in Swift development tools. Metrics, which are crucial for assessing code quality, performance, and maintainability, are inadequately catered for in the existing Swift tool ecosystem. Tools that do provide metrics support often have limitations in terms of the depth and breadth of the metrics offered. This deficiency hampers developers' ability to perform thorough code analysis, optimize performance, and ensure high-quality, maintainable codebases. The lack of robust metrics support can also impede the ability to track and improve software quality over time, particularly in larger or more complex Swift projects.

Limited reviewed metrics

Reviewed metrics are limited and insufficient for creating the full framework for eliciting the architectural pattern from the business goals and requirements. However, it is important to acknowledge the presence of an initial model that serves as a starting point for this endeavor. This model, albeit in its nascent stage, provides a baseline from which further development and enrichment are possible. Enriching this model would require an expansion beyond the conventional

metrics, incorporating a more holistic set of criteria that encapsulate both technical and business perspectives.

Further work

Advanced static analyser for Swift

Modern static analysers are lacking support of microfrontend architecture. These limitations underscore the need for the development of more advanced and specialized tools in the Swift ecosystem. Enhancing tool support for microfrontend architecture and expanding the range of available metrics would significantly benefit developers working with Swift, ultimately leading to more efficient development processes and higher-quality applications. The lack of support in static analyzers for these aspects results in a significant gap in the tooling ecosystem, particularly for developers and teams adopting microfrontend approaches in their Swift applications.

More metrics to review

The framework illustrated in Picture 7 should increase the coverage of common Quality Attributes to suit more fluently for a greater number of possible business goals.

The current framework seeks to expand on these attributes, ensuring that they are not only comprehensive but also sufficiently flexible to accommodate varying business needs. This expansion is particularly crucial in today's dynamic business environment, where the agility to adapt to changing market demands and user expectations is key.

Software tool

Considering the insights and findings discussed previously, there emerges a significant opportunity to develop a software tool. This tool would be designed to assist Software Architects and business stakeholders in potentially identifying the most suitable architecture, based on a variety of inputs. These inputs include specific requirements, the type of application, business goals, and constraints such as team size and budget. This tool would function as an advanced decision-support system, integrating various parameters and constraints to recommend architectural patterns that align optimally with the given inputs.

REFERENCES

1. Laricchia, F. (2023, September 28). Global smartphone penetration 2016-2022. Statista. https://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/
2. Clements, P., Kazman, R., & Mark. Klein. Evaluating Software Architectures: Methods and Case Studies, Addison Wesley., Dec 6, 2001.18
3. Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.
4. Chastek, Gary J.; Ferguson, Robert W. (2018). Toward Measures for Software Architectures. Carnegie Mellon University. Report. https://doi.org/10.1184/R1/6585371.v1
5. Bilohub D., Skrypchenko M., Tytenko S.,"QUALITY ATTRIBUTES AND ARCHITECTURAL PATTERNS OF MODERN MOBILE APPS" Modern engineering and innovative technologies 29-03 (2023): 33-38 https://doi.org/10.30890/2567-5273.2023-29-01-056
6. Trygve Reenskaug. The Model-View-Controller (MVC). Its Past and Present Java Zone, Oslo 18–19 September 2003
7. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R.,, Stafford, R. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.
8. Campos, E., Kulesza, U., Coelho, R., Bonifácio, R., & Mariano, L. (2015, April). Unveiling the architecture and design of android applications. In Proceedings of the 17th international conference on enterprise information systems (Vol. 2, pp. 201-211).
9. J. Grossman, Introduction to Model/View/ViewModel pattern for building WPF apps, Microsoft, 8 October 2005. [Online]. Available: https://blogs.msdn.microsoft.com/johngossman/2005/10
10. Lou, T. (2016). A comparison of Android Native App Architecture MVC, MVP and MVVM.
11. García, R. F. (2023). IOS architecture patterns MVC, MVP, MVVM, Viper, and VIP in swift. Apress. ISBN: 978-1-4842-9069-9
12. J. Gilbert, C. Stoll, Architecting iOS Apps with VIPER, 1 June 2014. [Online]. Available: https://www.objc.io/issues/13-architecture/viper/
13. N. Yang, Four principles of micro frontends for mobile, 17 March 2023. [Online]. Available:

https://www.thoughtworks.com/insights/blog/mobile/four-principles-mfes-mobile-pt1
14. Everald E. Mills , Software Metrics: Sei Curriculum Module Sei-Cm-12-1.1, Software Engineering Institute, 1988
15. Robert E. Park, Wolfhart B., G. William, A. Florac, Goal-Driven Software Measurement — A Guidebook August 1996
16. B. Williams, S Celis. [Online]. Available: https://github.com/pointfreeco/swift-composable-architecture
17. SwifLint official documenation. Available: https://github.com/realm/SwiftLint
18. SonarQube 10.3 official documentation. Available: https://docs.sonarsource.com/sonarqube/latest/
19. M. Campolese, Swift-code-metrics. Available: https://github.com/matsoftware/swift-code-metrics
20. Grep command line utility. Available: https://en.wikipedia.org/wiki/Grep#:~:text=grep%20is%20a%20command%2Dline,which%20has%20the%20same%20effect.