ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
FACULTY OF INFORMATICS

THESIS

OE-NIK Student's name: **Shugaa Addin Raidan**
2025 Student's registration number: **T008883/FI12904/N**

Óbuda University
John von Neumann Faculty of Informatics
Institute for Cyber-physical Systems

THESIS

Name of the student: **Raidan Shugaa Addin**
Registration number: T008883/EI12904/N
Neptun's code: [redacted]

Title of the thesis:

Enhancing State Management in SwiftUI using TCA (The Composable Architecture)
Hatékony SwiftUI állapotkezelés TCA (The Composable Architecture) segítségével

Internal supervisor: Márk Benjámin Emődi
External supervisor: Attila Csaba Marosi, Ph.D. (HUN-REN SZTAKI)

Deadline: 15 December 2024

Subjects of the final exam: Computer Architectures
Cloud service technologies and IT security specialization

The task:

Software development begins with an initial project idea, accompanied by a set of expectations in the form of specifications, functionalities, and behaviors, and ends with a full implementation. While the final product may appear straightforward, there are many ways to achieve it through different design patterns. Choosing the right design pattern can be challenging and subjective, as it significantly impacts code maintainability, efficiency, scalability, and clarity. Furthermore, selecting the most appropriate pattern can reduce the time needed to fix bugs when they arise. Many developers overlook the importance of design patterns at the start of a project. However, if they proceed without a well-defined project structure, they would risk creating an app that works but has code that is difficult to maintain over time. On June 3, 2019, Apple introduced SwiftUI, a framework that revolutionized iOS app development with its declarative syntax and intuitive approach to building user interfaces. Although various design patterns exist, most do not fully align with SwiftUI's declarative nature.
The student is required to evaluate state-of-the-art methodologies in modern software architectures, with a focus on iOS application development. This evaluation will include reviewing architectural patterns such as VIPER and The Composable Architecture (TCA), with particular attention to TCA's state management capabilities in SwiftUI applications. The student will develop an iOS app using TCA and compare its scalability, maintainability, and overall development efficiency against other architectural patterns, while also demonstrating the challenges posed by declarative programming and how TCA helps address these issues.
The thesis must contain:

- provide an overview of the currently used methodology,
- evaluate the potential methods that can be utilized to maintainable software,
- propose a system design,
- create the implementation of the selected method,
- validate, test and compare the system's capabilities against the others,
- provide future work for the solution,
- conclude the achieved results.

Place of stamp

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

Fleiner R.

Dr. habil. Rita Fleiner
head of institute

This specification is valid until: **15 December 2026**
ÓE HKR Section 54, paragraph (10)

I consider the thesis suitable for submission:

external supervisor

internal supervisor

STUDENT'S DECLARATION

I the undersigned student hereby declare that this thesis is the result of my own work; I have disclosed the references and tools used in an identifiable manner. The results indicated in my thesis completed may be used by the university and the institution announcing the task for their own purposes free of charge.

Dated: Budapest, 10 December 2024..

.....
student's signature

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name: Shugaa Addin Raidan

Neptun code: [redacted]

Specialty: Computer Science Engineering

Phone number: [redacted]

Mailing address (e.g. domicile): [redacted]

Thesis / thesis work title in Hungarian:

Valósághű adatok előállítása és elemzése GAN felhasználásával

Thesis / thesis work title in English:

Generating and analyzing realistic data based on GANs

Internal supervisor: Márk Benjámin Emődi

External supervisor: Attila Csaba Marosi, Ph.D. (HUN-REN SZTAKI)

Please write the data in printed capital letters!

Occ.	Date	Content	Signature
1.	23/02	Discussing the roadmap	Emődi Márk
2.	29/02	Giving updates about my initial research findings	Emődi Márk
3.	07/03	Refining the thesis scope, outcome of the thesis	Emődi Márk
4.	15/05	Reviewing the thesis, revising	Emődi Márk

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Thesis I / Thesis II (BSc) or Thesis work 1 / Thesis work II / Thesis work III / Thesis work IV[3], (MSc) and is allowed to proceed for reporting / defense[4].

Dated: Budapest, May 16, 2024

Emődi Márk

internal supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name: Neptun code: Specialty:

..Shugaa Addin, Raidan..... [redacted] ...Computer Science Engineering.....

Phone number: Mailing address (e.g. domicile):

[redacted]

Thesis / thesis work[1] title in Hungarian:

.Hatékonyabb SwiftUI állapotkezelés TCA (The Composable Architecture) segítségével...

Thesis / thesis work[2] title in English:

..Enhancing State Management in SwiftUI using The Composable Architecture (TCA)....

Internal supervisor: External supervisor:

..Márk Benjámin Emődi..... ..Attila Csaba Marosi, Ph.D. (HUN-REN SZTAKI)..

Please write the data in printed capital letters!

Occ.	Date	Content	Signature
1.	23/08	Discussing the roadmap	Emődi Márk
2.	02/09	Giving some updates	Emődi Márk
3.	08/11	Consultation	Emődi Márk
4.	29/11	Asking some questions about thesis submission	Emődi Márk

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Thesis I / Thesis II (BSc) or Thesis work 1 / Thesis work II / Thesis work III / Thesis work IV[3], (MSc) and is allowed to proceed for reporting / defense[4].

Dated: Budapest, ...10th.of.December... 2024..

Emődi Márk
.....
internal supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

ABSTRACT

SwiftUI's declarative programming model has introduced a new paradigm for building user interfaces on Apple platforms, prompting the exploration of architectures that align more naturally with its unidirectional data flow. While traditional UIKit-based patterns like Model-View-ViewModel (MVVM) remain popular, they often rely on bidirectional data flows, making them less suitable for the declarative style of SwiftUI. In contrast, The Composable Architecture (TCA) promises a more structured approach tailored to SwiftUI's requirements.

This thesis evaluates the practical trade-offs between TCA and MVVM by implementing a podcast application with both architectures. The application includes real-world features such as audio playback, external library integration, and network communication. Using Xcode Instruments, the two implementations are compared in terms of memory usage, CPU consumption, and runtime performance. In addition, qualitative factors such as maintainability, complexity, and development experience are assessed.

The findings indicate that TCA offers clearer separation of concerns and more predictable navigation patterns, resulting in code that is easier to test and maintain at scale. Although TCA introduces a steeper initial learning curve and may involve additional overhead, it provides a more robust framework for managing complexity in modern SwiftUI applications. In contrast, MVVM's reliance on bidirectional flows and blending of responsibilities within view models can hinder long-term scalability and testability. Thus, for complex SwiftUI projects, TCA emerges as the more sustainable choice.

ABSZTRAKT

A SwiftUI deklaratív programozási modellje új paradigmát vezetett be a felhasználói felületek készítésében az Apple platformokon, ami arra ösztönöz, hogy olyan architektúrákat keressünk, amelyek jobban illeszkednek az egyirányú adatfolyamhoz. Míg a hagyományos UIKit-alapú minták, mint például a Model-View-ViewModel (MVVM), továbbra is népszerűek, gyakran kétirányú adatfolyamokra támaszkodnak, ami kevésbé alkalmassá teszi őket a SwiftUI deklaratív stílusához. Ezzel szemben a The Composable Architecture (TCA) egy strukturáltabb megközelítést ígér, mely a SwiftUI követelményeire illeszkedik.

Ez a szakdolgozat a TCA és az MVVM közötti gyakorlati különbségeket vizsgálja egy podcast alkalmazás mindkét architektúrával történő megvalósításával. Az alkalmazás valós funkciókat tartalmaz, mint például az audio lejátszás, a külső könyvtárak integrálása és a hálózati kommunikáció. Az Xcode Instruments segítségével a két implementáció összehasonlításra kerül a memória-használat, a CPU-fogyasztás és a futásidejű teljesítmény tekintetében. Ezenkívül olyan kvalitatív tényezőket is értékelünk, mint a karbantarthatóság, a komplexitás és a fejlesztési élmény.

Az eredmények fényében elmondható, hogy a TCA világosabb szétválasztást biztosít az egyes felelősségi körök között, és kiszámíthatóbb navigációs mintákat eredményez, ami könnyebben tesztelhető és karbantartható kódot eredményez. Bár a TCA nehezebben sajátítható el, robusztusabb keretet biztosít a komplexitás kezeléséhez a modern SwiftUI alkalmazásokban. Ezzel szemben az MVVM kétirányú folyamatokra való támaszkodása és a felelősségek keveredése a nézetmodelleken belül akadályozhatja a hosszú távú skálázhatóságot és tesztelhetőséget. Így a komplex SwiftUI projektek esetében a TCA a fenntarthatóbb választásnak bizonyul.

CONTENTS

1 INTRODUCTION

UIKit, since its introduction in 2008[1], has been the foundational framework for building user interfaces on Apple devices such as iPhones, iPads, and iPods. As an imperative framework[2], UIKit requires developers to define the state and behavior of user interfaces explicitly. For example, to create a screen with a label, we must manually create a UILabel, configure its properties (e.g., text, font, and color), add it to the view hierarchy using addSubview, and define its layout explicitly using Auto Layout or frame-based positioning. This approach often results in intricate codebases, especially for large-scale projects, and makes room for bugs as the more code we have, the higher the probability of bugs' occurrence. Additionally, as Apple's ecosystem expanded to include devices like Macs, Apple Watches, and Vision Pro, creating cross-platform applications with UIKit grew increasingly complex[3]. Developers often face challenges managing view hierarchies, Auto Layout constraints, and delegate patterns to each single platform, which can increase development time[3].

To address these limitations, Apple introduced SwiftUI in 2019—a modern declarative framework designed to simplify UI development[4]. By reducing boilerplate code and providing native support for Apple's growing range of devices, SwiftUI promotes efficiency and cross-platform compatibility[4][3], removing the need to maintain separate codebases for single platforms. Unlike UIKit, SwiftUI adopts a declarative programming paradigm, allowing developers to focus on describing the desired UI state rather than the steps to achieve it, which helps resolve many of the pain points associated with UIKit. Moreover, the ease of entry with learning SwiftUI has attracted many developers to code for Apple platforms, as its learning curve is not as steep compared to UIKit.

However, just like any framework, SwiftUI has its challenges, especially when the app grows. One of the challenges is managing App State. State is the data that drives the app's behavior and determines what is displayed on the screen at a given time. As the app grows in complexity, state management can become increasingly difficult, particularly when it is scattered across multiple views or components. This can lead to inconsistencies, where different parts of the app have conflicting versions of the same data or bugs that are hard to trace and debug.

Another significant challenge is side-effect management. SwiftUI does not provide a built-in, standardized way to handle asynchronous operations like API calls or file I/O, which can lead to hard-to-test logic if handled directly in the views. Furthermore, modularity and scalability can become problematic in SwiftUI. While SwiftUI encourages composing small views, it doesn't inherently enforce a separation of concerns between UI, state, and business logic. As features are added, this can result in tightly coupled components and duplicated logic, making the code harder to maintain and reuse. These issues become problematic as they can lead to buggy, unmaintainable code that is difficult to debug and test, especially in larger applications. The Composable Architecture (TCA) addresses these gaps by providing a structured approach to state management, modularity, and side-effect handling, ensuring clarity and predictability in SwiftUI development.

This thesis investigates the strengths and limitations of MVVM and The Composable Architecture (TCA). Through detailed analysis and practical implementation, this research demonstrates how TCA enhances state management in SwiftUI applications. To illustrate these benefits, an iOS application will be developed using TCA, showcasing its ability to improve scalability, maintainability, and overall code structure in modern app development. Throughout this thesis documentation, I will be referring to The Composable Architecture as TCA.

2 PROBLEM DESCRIPTION

The introduction of SwiftUI has significantly simplified the creation of user interfaces, making it more intuitive compared to UIKit [5]. SwiftUI is built on a declarative programming paradigm, enabling developers to describe what the UI should look like and how it should behave in various states. This is a departure from UIKit's imperative paradigm, which requires detailed, step-by-step instructions to manage and update UI elements [6]. Moreover, SwiftUI incorporates a unidirectional data flow, making state management more straightforward and reducing the potential for bugs caused by manual updates. These features make SwiftUI a powerful framework for modern app development.

However, SwiftUI presents challenges for developers accustomed to traditional architectures like MVVM and VIPER, which are commonly used in UIKit [7]. These architectures rely on explicit triggers and bidirectional data flow to manage UI changes, often leading to unpredictable behavior in complex applications. On the other hand, SwiftUI's declarative model ensures a consistent and predictable flow of data by automatically synchronizing the UI with state changes. This difference creates a gap between traditional architectures and SwiftUI's behavior, particularly in state management.

State management is the process of handling the data that drives the UI and ensuring synchronization between the UI and its underlying state[8]. In traditional architectures, the state is often managed through isolated components and manual updates, which can lead to inconsistencies. For example, in a shopping application, if a product list displays five items in stock and a user adds one item to their cart in one screen, the list should automatically update to show four items remaining in any other screen. Without a centralized state management system, other parts of the app, such as the cart view, might still display outdated information, resulting in a poor user experience.

These challenges have led to the development of architectures designed to better handle state management in SwiftUI. One such architecture is TCA, which draws inspiration from Redux [9]. TCA provides a structured and centralized approach to state management, which ensures consistent data handling across the application making it a good choice for managing complex applications in SwiftUI. Moreover, TCA integrates seamlessly with SwiftUI's declarative design, offering developers tools to build scalable, testable, and maintainable applications[10].

TCA's structured approach to state management and its integration with SwiftUI's declarative paradigm enhance modularity and testability in complex applications. However, concerns about increased boilerplate, abstraction layers, and the learning curve for developers remain. Additionally, existing comparisons tend to focus on theoretical differences rather than practical evaluations of metrics such as performance, scalability, or memory usage. Due to the limited research and case studies directly comparing it with traditional architectures like MVVM, this study seeks to fill this gap by providing a detailed, data-driven comparison of MVVM and TCA within the context of SwiftUI applications. In this thesis work, I aim to make a comparative analysis of MVVM and TCA in SwiftUI-based applications. Specifically, it will assess their relative strengths and weaknesses across key metrics such as performance, state management efficiency, scalability, and memory usage.

3 LITERATURE REVIEW

This section reviews relevant literature on architecture patterns, specifically MVVM, VIPER, and TCA within the scope of iOS development. The emphasis is on improving SwiftUI state management to enhance app maintainability, and testability, and ensure a clean, organized codebase. SwiftUI, introduced in 2019[4], is a relatively new framework, and as a result, there is limited research specifically on its use with established design patterns. Much of the existing literature focuses on theoretical aspects rather than practical applications or comparative analyses with traditional architectures.

3.1 MVVM

3.1.1 Overview

The MVVM architecture is one of the most widely adopted design patterns. MVVM was created by Microsoft in 2005 as a new approach to building applications using the Windows Presentation Foundation (WPF) framework[11]. Since its introduction, its use has expanded beyond WPF.

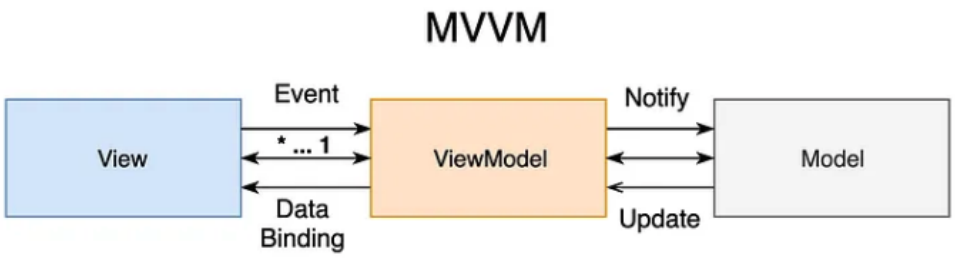


Figure 3.1: Model–View–ViewModel schema[12]

MVVM, as shown in Figure 3.1, has three components, Model, View, and ViewModel. The model handles business logic, data management, and external communication. It is responsible for tasks such as data persistence, managing external communications, and converting external data into model objects. The Model only interacts with the ViewModel, remaining unaware of the View's existence. This ensures that the Model is decoupled from the user interface, which improves separation of concerns and makes the application more modular and maintainable[13].

The View in MVVM consists of the SwiftUI views. Its primary role is to display updated information from the ViewModel, without containing any business logic of its own. The View can reference the ViewModel multiple times to retrieve data, but it remains a passive component that reacts to changes. Coordination tasks, such as navigation between screens, can be handled by a Controller, which may use the Delegation pattern. However, in more advanced implementations, this responsibility can be passed to a Coordinator, which gives rise to the MVVM-C architecture[14].

On the other hand, the ViewModel sits between the View and the Model. It manages the View's state and handles input and output operations that control the View's behavior. It acts as a mediator, translating the business logic and data from the Model into a form that the View can display. Importantly, the ViewModel is "owned" by the View, while the Model remains within the ViewModel's domain[15].

In MVVM, Data Binding automatically updates the View whenever the ViewModel's data changes. Data binding can be implemented in several methods, depending on the environment you are building the app for. One of these methods is by using Combine. Combine is a powerful framework developed by Apple that allows changes in objects' properties to be communicated efficiently across different parts of the application. It significantly reduces the amount of manual updating required between the View and the ViewModel, making the architecture more responsive and reducing boilerplate code[15].

3.1.2 Related Work

A study titled "SwiftUI: A Case Study of Code Quality Evaluation for Declarative Programming" investigates the use of static analysis tools to evaluate the code quality of SwiftUI projects[16].

Static analysis is a method used to review source code for potential issues without executing it. This study uses Codacy to analyze open-source SwiftUI projects found on GitHub. The researchers selected projects based on different architectures, such as Model View (MV), Model View ViewModel (MVVM), and Redux, and varying project sizes. By running these projects through Codacy, the study aimed to gather metrics on key aspects of code quality, including code complexity, issues related to performance and security, code duplication, and adherence to coding standards[16].

The findings revealed that Static analysis was effective in identifying coding standard violations, particularly in the area of code style[16]. Over 99% of the issues detected were related to style, such as naming conventions or whitespace errors, with very few significant security or performance concerns. Most of the analyzed projects received a B grade in terms of overall quality, reflecting minor issues rather than major flaws. However, although the research found that static analysis tools can provide valuable insights into SwiftUI projects implemented in different design architectures, the research does not touch on the benefits different design patterns would bring into a project, it was only focused on style and code standard violations, which leaves us the room for coming up with an in-depth look into design patterns and their impact on declarative programming, and in our case, SwiftUI.

Another research paper titled "ANALYSIS OF THE IMPLEMENTATION OF MVVM ARCHITECTURE PATTERN ON PERFORMANCE OF IOS MOBILEBASED APPLICATIONS"[17]. The paper discusses that the MVVM pattern can be implemented in both UIKit and SwiftUI, but the data binding mechanism differs depending on the framework. For applications built with SwiftUI, the Combine framework is recommended for handling data binding. The paper, however, implements RxSwift for UIKit-based applications, arguing that RxSwift provides relevant measurement data for the specific case study (AR Ruler). But if you're working with SwiftUI, Combine would be the natural choice.

while this paper focuses on UIKit and RxSwift, the SwiftUI implementation of MVVM would use Combine instead, ensuring a reactive data flow from model to view via the ViewModel. SwiftUI's declarative nature fits well with MVVM, allowing seamless updates between the ViewModel and the View. The paper primarily focuses on implementing the MVVM (Model View ViewModel) architecture pattern in iOS applications using UIKit and RxSwift. However, SwiftUI is briefly mentioned in the Implementation Results section.

3.2 VIPER

3.2.1 Overview

VIPER was introduced in 2014 by Jeff Gilbert and Conrad Stoll [18]. It stands for View, Interactor, Presenter, Entity, and Router, with each representing a separate layer in a module, Figure 3.2.

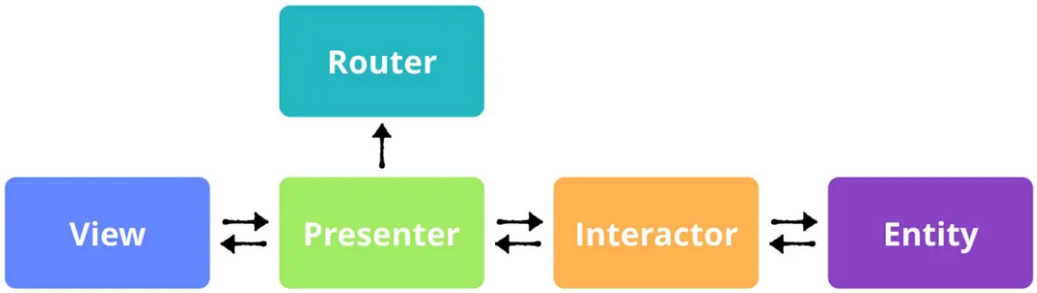


Figure 3.2: VIPER [19]

This structured approach allows for a clear separation of concerns, promoting the Single Responsibility Principle (SRP) and the Interface Segregation Principle (ISP) through the use of protocols for communication between the layers[20]. VIPER was designed to apply Clean Architecture principles by breaking down functionality into separate layers, each with a specific responsibility[20]. For example, the View is responsible for displaying information and handling user interface elements like buttons, labels, and views. It sends user interactions to the Presenter, which controls the logic but does not request information directly. Instead, the Presenter sends updates when needed. This keeps the View passive and focused only on presenting data to the user[20].

The Interactor handles the application's business logic. It is responsible for retrieving and processing data from databases or APIs. The Interactor is independent of the View and communicates with the Presenter via a delegate pattern. Importantly, it never passes entire Entities to the Presenter but instead sends simple data structures[20]. On the other hand, the Presenter acts as the central component in the VIPER architecture, mediating between the View, Interactor, and Router. It receives user inputs from the View, processes them, requests data from the Interactor, and then prepares the data for display by the View. Additionally, the Presenter manages navigation between different screens using the Router[20]. The Entities in VIPER represent simple data models, often implemented as structs. Only the Interactor interacts directly with these Entities, as they are not sent to the Presenter or View[20]. Finally, the Router is responsible for handling navigation and screen

creation. The Router functions in a similar way to the Coordinator pattern found in other architectures, such as MVVM-C[20].

3.2.2 Related Work

Before the release of SwiftUI, UIKit was the primary framework for creating iOS apps. However, after the introduction of SwiftUI, developers gradually started to transition to adopting SwiftUI, and in some cases full migration to SwiftUI. A research paper[21], by Babayev, explored the difficulties of this adoption, particularly the challenge of using traditional architectures in SwiftUI's declarative programming model.

One such architecture, VIPER, emphasizes a strict separation of concerns by organizing components into distinct layers: View, Interactor, and Presenter. SwiftUI's state-driven model and unidirectional data flow, however, differ fundamentally from VIPER's reactive logic and bidirectional communication patterns, creating unique obstacles for integration. The paper claims the need for intermediary components, such as ViewModels[21].

These ViewModels serve as a bridge, converting VIPER's callback-based logic into the state-driven mechanisms that SwiftUI requires. However, this adaptation introduces added complexity to application design, requiring careful layering to preserve VIPER's modularity and testability. While the paper provides valuable insights into these integration challenges, its proposed use of intermediary components like ViewModels, which rely on referenced types, is discouraged in SwiftUI development due to potential performance and architectural issues[21]. Furthermore, the research lacks a comprehensive exploration of alternative approaches to address these challenges effectively.

Another research paper[22], by Andika, investigates the performance implications of modularity within the VIPER architecture, particularly in the context of traditional UIKit-based development. the research examines the trade-offs involved in modularizing application components, which can improve maintainability and scalability but often introduce performance overhead due to increased inter-module communication. This analysis claims that while modular architectures like VIPER offer clear benefits in collaborative development and code reusability, they may suffer from reduced efficiency in terms of launch time, memory consumption, and CPU usage compared to monolithic designs[22]. Although this study provides useful insights into the balance between modularity and performance, it remains centered on VIPER within the UIKit paradigm. It does not address the integration of VIPER with declarative frameworks like SwiftUI, nor does it compare VIPER's performance and modularity with modern, SwiftUI-compatible architectures.

3.3 TCA

3.3.1 Overview

Since the primary focus of this thesis is an in-depth study of TCA, a dedicated section for its review is provided in Chapter 5.

3.3.2 Related Work

A comprehensive review of available literature and resources revealed no prior work about TCA. This lack of related work demonstrates the originality of this study and its potential contribution to the field. However, Rod Schmidt, who has used TCA for three years, shares a detailed and reflective account of his experience using TCA in a large-scale iOS application[23]. Schmidt's experience highlights TCA's strengths, particularly its robust testing capabilities and unidirectional data flow, which align well with modern development practices. However, he also provides a critical examination of the challenges developers face when adopting TCA, especially in complex, multi-team environments[23].

One of Schmidt's primary concerns is the steep learning curve associated with TCA. Developers must grasp advanced concepts like reducers, scoped states, and macros (@ObservableState, @Dependency), which diverge significantly from SwiftUI's simpler, object-oriented paradigm. This complexity, compounded by frequent framework updates, forces teams to continually adapt their codebases to leverage new features, leading to productivity challenges. Schmidt also critiques TCA's lack of encapsulation, noting that its flat reducer hierarchy can result in cross-team dependencies and sprawling application state management. Performance issues, such as stack overflow errors in large applications and slow action processing, are highlighted as additional drawbacks, exacerbated by TCA's functional nature and deviation from the platform's object-oriented optimization[23].

Schmidt further explores the organizational challenges of adopting TCA in a corporate environment. He discusses the difficulty of coordinating upgrades across eight independent teams, each relying on shared code, and the risks of adopting a third-party framework developed by just two individuals. While he acknowledges the maturity of TCA in its current state, he questions the suitability of such an architecture for large companies, where simpler alternatives like MVVM or Clean Architecture might offer equivalent benefits with fewer complexities[23].

4 SWIFTUI

4.1 Overview

Before SwiftUI was released, developers relied heavily on UIKit to build user interfaces. UIKit is Apple's long-standing framework for building graphical interfaces in iOS applications. Introduced in 2008[1], it uses an imperative programming model, where developers must define how the interface should behave, laying out views and specifying interactions step-by-step. Although UIKit provides more flexibility, it often requires more code and manual effort. However, with the release of SwiftUI, Apple has introduced a declarative paradigm into building User interfaces.

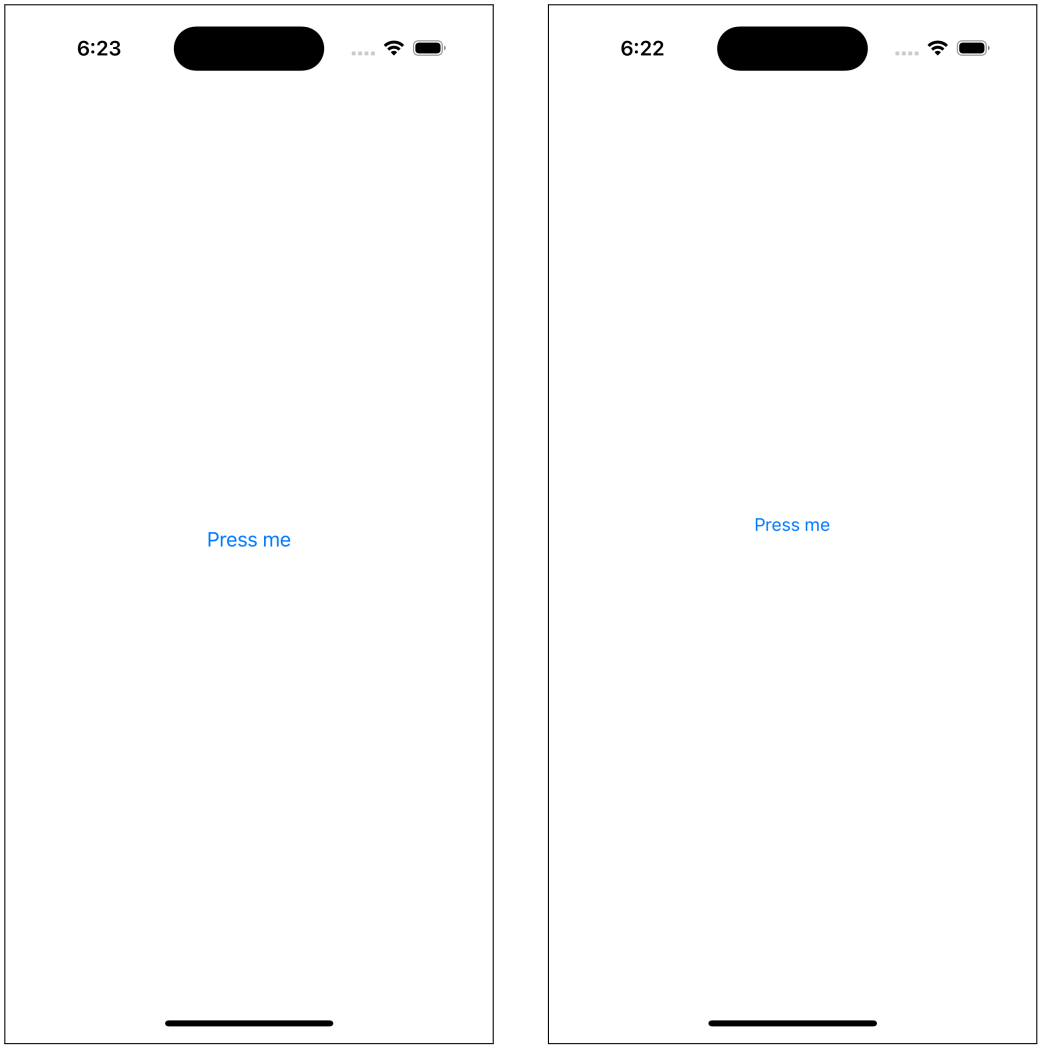
SwiftUI is a powerful UI framework that was developed by Apple and released in 2019 [4]. It provides developers tools to build user interfaces across all Apple platforms using just one set of tools and APIs[24]. Thanks to its declarative syntax, writing UI code becomes simpler and more intuitive. Compared to traditional UI frameworks like UIKit, you can achieve the same UI with significantly less code[25][24]. Here's an example of how SwiftUI and UIKit would implement a simple button:

UIKit:

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let button = UIButton(type: .system)
        button.setTitle("Press me", for: .normal)
        button.addTarget(self, action: #selector(buttonPressed),
        for: .touchUpInside)

        button.translatesAutoresizingMaskIntoConstraints = false

        view.addSubview(button)

        NSLayoutConstraint.activate([
            button.centerXAnchor.constraint(equalTo: view.centerXAnchor),
            button.centerYAnchor.constraint(equalTo: view.centerYAnchor)
        ])
    }

    @objc func buttonPressed() {
        print("Button pressed")
    }
}
```

SwiftUI:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Button("Press me") {
            print("Button pressed")
        }
    }
}
```



(a) SwiftUI Implementation *(b) UIKit Implementation*

Figure 4.1: SwiftUI vs. UIKit

The first thing that comes to mind when looking at those two code blocks is that although they both do the same thing, as shown in Figure 4.1, the UIKit code has almost twice the number of lines as the SwiftUI one.

Another thing I would like to highlight is that the way we create user interfaces using SwiftUI is opinionated, meaning we must adhere to specific rules to display something on the screen. This is represented by the computed property body and the struct conforming to View. There isn't much flexibility in this regard.

4.2 Declarative vs Imperative Programming

Declarative syntax is a programming approach where you state what you want to accomplish without focusing on the specific steps required to achieve it. It's about stating the desired outcome rather than outlining the process. For example, in everyday speech, declarative syntax could sound like:

"I'd like a sandwich, please."

This is a simple request that focuses on the end result—getting a sandwich—without worrying about the individual steps involved in making it[26].

Imperative syntax, on the other hand, is a more detailed, step-by-step style of programming. Here, the programmer explicitly instructs how to achieve the desired result. A similar instruction in everyday speech would be: "I need two slices of bread, some butter, cheese, and lettuce. Spread the butter on the bread, layer the cheese and lettuce, and then put the slices together." This approach gives a specific sequence of actions to follow, focusing on the how rather than just the end goal. You could even go deeper by specifying "cut the cheese into thin slices" or "use whole wheat bread."[27]. Using the declarative coding style makes the code more readable and easier to comprehend. More significantly, the SwiftUI framework allows you to create user interfaces with substantially less code[24][25]. Additionally, as SwiftUI is a state-driven framework[10], the delegate patterns are no longer necessary, as declarative syntax now relies on States. By applying @State to a property, SwiftUI tracks that property, and whenever it is modified, it will automatically invalidate the current layout and trigger a reload.

For example, imagine a list displaying a collection of data. When the data updates, the list is automatically refreshed—there's no need to manually call for a refresh (like reloadData() in CollectionViews or TableViews)[28], therefore, making dynamic behaviors both predictable and easy to implement[10].

SwiftUI's declarative design fosters a consistent framework, requiring views to conform to the View protocol and ensuring updates are entirely state-driven[10]. This simplifies collaboration and facilitates onboarding for development teams. By unifying the development process, SwiftUI moves away from UIKit's fragmented structure, providing a more intuitive and modern approach to UI development[10].

4.3 State Management

In any application, regardless of the programming language, we can abstractly break it down into three fundamental components: the User Interface (UI), Logic, and Data, as can be seen in Figure 4.2.

The user interface is the visible layer with which end-users interact, while the logic handles validating user input and enduring data consistency and integrity; however, data are the single source of truth that other

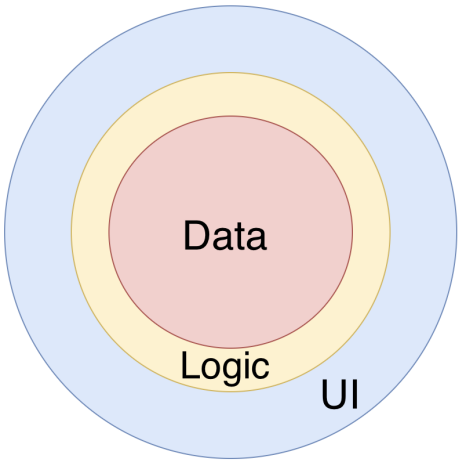


Figure 4.2: App Components

components rely on.

When the user interacts with the user interface the input data is validated by the logic layer, and eventually, the change is committed to the Data. If there is a change in data, the change has to be reflected instantly and displayed on the user interface. To achieve such a behavior, we need a way to inform the user interface to reload and update. For instance, if a string variable is displayed in the UI and its value changes, the UI will not automatically reflect this change unless it is notified. As a result, the UI must be refreshed or re-rendered to display the updated value.

This brings us to the concept of state management, which is responsible for handling how data changes are tracked and reflected throughout an application. State management refers to controlling and maintaining the state or condition of a system at any given time. In software development, "state" refers to the present values of variables, data, and settings that define an application's behavior at a given moment[29].

SwiftUI introduces a robust yet straightforward approach to state management, making it easier to maintain and update application data[10]. If we are handling some local state to individual views, the @State property wrapper provides a bindable value that automatically triggers re-renders whenever the state changes[10]. This ensures that the UI remains synchronized with the underlying data without requiring manual updates[10]. For more complex scenarios where state needs to be shared across multiple views, SwiftUI offers @ObservableObject and the @Published property wrapper[10]. These tools enable us to observe and react to shared state changes with minimal boilerplate. The transition from @ObjectBinding to @ObservableObject in later versions of SwiftUI further refined this approach, automating state observation and reducing repetitive code[10]. Despite these advancements, challenges remain when scaling state management for larger applications, as SwiftUI does not yet offer a comprehensive solution for organizing and decomposing global state into modular components.

5 THE COMPOSABLE ARCHITECTURE

5.1 Overview

Introduced in 2019, the Composable Architecture, also known as TCA, was originally designed to address the complexities of state management and side-effect handling, two common challenges in application development, especially after the introduction of SwiftUI. Since its creation, TCA has evolved into a mature tool widely used by individual developers and teams across organizations of all sizes. It provides a unified framework for building SwiftUI applications by integrating state, actions, side effects, and dependencies into a predictable and cohesive system. The primary goal of TCA is to enhance the maintainability, scalability, and testability of applications, enabling developers to create code that is both robust and easy to understand, and maintainable and scalable in the long run[30].

To understand TCA's approach, let me give you an insight into SwiftUI's architecture. Although SwiftUI is closed-source and not publicly documented, static analysis reveals that it operates as a single-entry system. This means that regardless of how many structs conform to 'View', everything eventually traces back to a single root: the parent view[31].

In SwiftUI, every view has one computed property called body, which describes its visual representation. A parent view is composed of child views, forming a hierarchy. Moreover, SwiftUI enforces that a view cannot have more than one body property. This strict design ensures a clear, predictable data flow within the application[31].

Moreover, SwiftUI's views are inherently composable, as demonstrated in the code block below. A single view can be divided into multiple child views, each handling a specific part of the UI, showcasing how views can be composed of smaller, reusable components[31]. Refer to Figure 5.1.

Listing 1: Composable SwiftUI View

```
import SwiftUI

struct ParentView: View {
    var body: some View {
        Text("Hello from ParentView!")
        Child1()
        Child2()
    }
}

struct Child1: View {
    var body: some View {
        Text("Hello from Child1!")
    }
}

struct Child2: View {
```

```
    var body: some View {
        Text("Hello from Child2!")
    }
}
```

Figure 5.1: Illustrating Composability

This contrast—strict enforcement in the UI layer served as the inspiration for TCA. The question arose: Could we achieve the same level of structure in the data layer as SwiftUI enforces in the UI layer?

TCA answers this question by filling that gap, providing a framework that introduces similar enforcement and structure to the data layer, creating a more unified and predictable development experience.

5.2 Core Components of TCA

TCA, like a SwiftUI view, enforces certain rules that act as the building blocks for any application built using it. These rules are organized into key concepts and tools: State, Action, Reducer, Store, and TestStore[30].

5.2.1 State

The State is a struct that encapsulates all the data, represented by its properties, required to execute the logic and render the user interface of a feature. It is designed to be immutable, meaning it can only be modified within a reducer in response to an action (Actions and Reducers will be discussed in detail below)[30].

As a value type, State offers several advantages over reference types, including easier debugging and tracing of changes. This immutability and value-type design ensure predictable state management, making it simpler to understand, maintain, and debug the application's flow[32].

Another advantage of using value types is their straightforward conformance to Equatable. This makes it easy to compare two or more states and identify differences. In TCA, this capability is especially useful for testing and debugging, as it allows developers to verify how state transitions occur in response to actions. This design simplifies the process of tracking and understanding state changes[32].

Listing 2: State Struct [30]

```
@ObservableState <---
struct State: Equatable {
    var count = 0
    var numberFact: String?
}
```

As can be seen above that, State was annotated with @ObservableState. Observable state is a state management pattern based on the observer pattern, where the state is monitored and notifies listeners (subscribers) of changes. Subscribers register to the observable state and receive updates when the state changes [33].

5.2.2 Action

Action is a type that encapsulates all possible events within a feature, including user interactions, notifications, and more[30].

Listing 3: Action enum [30]

```
@ObservableState
struct State: Equatable { /* ... */ }
enum Action {
  case decrementButtonTapped
  case incrementButtonTapped
  case numberFactButtonTapped
  case numberFactResponse(String)
}
```

Actions are modeled using enums which offer advantages in terms of modularity, testability, and debugging. By separating feature logic from its presentation, enums make it easier to add new behaviors without altering existing functionality. This approach supports advanced tools such as debugging utilities to log state changes (_printChanges()) or performance analyzers like signpost, which track and diagnose action bottlenecks. Enums also facilitate comprehensive testing, as frameworks like TestStore enable developers to simulate user interactions, verify state transitions, and validate effects in a step-by-step manner. Additionally, their structured and reusable nature simplifies extending features or implementing middleware to adjust reducers. While maintaining enums may initially appear to add complexity, the resulting codebase is more robust, scalable, and easier to manage, offering clarity and adaptability that justify the effort[32].

5.2.3 Reducer

Reducer serves as the central mechanism for managing a feature's logic, including state transitions and side effects. A reducer is a function that takes the current state and an action as inputs, producing an updated state along with any side effects[34].

Listing 4: Reducer Computed Property [30]

```
@ObservableState
struct State: Equatable { /* ... */ }
enum Action { /* ... */ }

var body: some Reducer<State, Action> {
  Reduce { state, action in
    switch action {
    case .decrementButtonTapped:
      state.count -= 1
      return .none

    case .incrementButtonTapped:
      state.count += 1
      return .none

    case .numberFactButtonTapped:
      return .run { [count = state.count] send in
        let (data, _) = try await URLSession.shared.data(
          from: URL(string: "http://numbersapi.com/\(count)/trivia")!
        )
        await send(
          .numberFactResponse(String(decoding: data, as: UTF8.self))
        )
      }

    case let .numberFactResponse(fact):
      state.numberFact = fact
      return .none
    }
  }
}
```

By centralizing the logic, this approach ensures deterministic and predictable application behavior, even in complex systems. Reducers are inherently composable, allowing developers to divide large applications into smaller, self-contained features that can seamlessly integrate into a cohesive system. This principle of composability extends beyond reducers to state and actions, making TCA an effective framework for constructing applications with nested or hierarchical features[35] [30].

5.2.4 Feature

The Feature is a standalone, self-contained module within an application that includes specific functionality. Feature consists of all the above components, State, Action, and reducer. [36]

Listing 5: Feature [30]

```
@Reducer
struct Feature {
  @ObservableState
  struct State: Equatable { /* ... */ }
  enum Action { /* ... */ }

  var body: some Reducer<State, Action> {
    Reduce { state, action in
      switch action {
      case .decrementButtonTapped:
        state.count -= 1
        return .none

      case .incrementButtonTapped:
        state.count += 1
        return .none

      case .numberFactButtonTapped:
        return .run { [count = state.count] send in
          let (data, _) = try await URLSession.shared.data(
            from: URL(string: "http://numbersapi.com/\(count)/trivia")!
          )
          await send(
            .numberFactResponse(String(decoding: data, as: UTF8.self))
          )
        }

      case let .numberFactResponse(fact):
        state.numberFact = fact
        return .none
      }
    }
  }
}
```

5.2.5 Store

The Store acts as the runtime system that links state management with the user interface. It observes changes to the state and serves as a centralized hub for state updates. When the state is modified, SwiftUI views subscribed to the Store automatically refresh, maintaining synchronization between the UI and the application's data. [30]

5.2.6 Side Effect

The Side Effect encapsulate asynchronous operations and define how their outcomes are processed within the system. This approach makes side effects declarative, composable, and easier to manage and test, while also providing tools to cancel unnecessary effects to prevent disruptions in application behavior. [37]

5.2.7 TestStore

The Test Store is a tool that is used to test how actions affect state and trigger effects. It is initialized with the feature's initial state and reducer, which defines the logic for processing actions. Developers send actions into the Test Store using its send method, and the resulting state changes are asserted within a closure to ensure accuracy. For actions that trigger asynchronous effects, the Test Store allows developers to use methods like receive to verify the effects' results and their impact on the state. [38]

5.3 Application Management in TCA

5.3.1 Dependency Management

TCA, as a library, offers a dependency management solution that makes an easy approach to handling external resources like clocks, API clients, and analytics services, which often complicate development and testing. TCA simplifies this process through its integration with a dependency management library, enabling developers to explicitly declare and override dependencies as needed. Dependencies are injected using the @Dependency property wrapper, making them easy to adapt for various scenarios, including production, previews, and testing. For instance, a feature requiring time-based effects can declare a dependency on the Clock protocol. During tests, a TestClock can simulate time progression, allowing tests to run instantly without waiting for real time to elapse. This approach enhances test reliability and reduces development complexity by providing a consistent and flexible framework for managing external systems[39].

5.3.2 Navigation

Navigation between different screens in an app is considered one of the most challenging aspects of application development. Traditionally, when modeling navigation, developers usually rely on @State or @StateObject for local state. However, using TCA, the navigation state is explicitly modeled as part of the application's domain. This centralization of navigation state management serves as a single source of truth, enhancing clarity, predictability, and testability[40].

TCA comes with two navigation approaches:

- **Tree-Based Navigation**
- **Stack-Based Navigation**

Tree-based navigation models navigation state as optional properties within the parent domain. This approach is ideal for scenarios like presenting modals or sheets. By using TCA's '@PresentationState' property wrapper, advanced capabilities are unlocked, such as automatically managing to dismiss actions when a user swipes down on a sheet[40].

For example:

```
struct ParentState {
    @PresentationState
    var child: ChildState?
}
```

When the child state becomes non-nil, the sheet is presented. Once dismissed, the child state is cleared, maintaining consistency[40]. However, while tree-based navigation works well for hierarchically related features, stack-based navigation makes a good choice in drill-down scenarios, such as navigating through a list to detailed views. In this approach, the navigation stack is represented as an array, with each element corresponding to a feature in the navigation hierarchy[40].

This method aligns with SwiftUI's NavigationStack behavior and allows for straightforward management of multiple navigation levels programmatically. For instance, removing the last feature from the stack is as simple as popping the last element:

```
state.stack.removeLast()
```

All in all, making a decision on what approach for a project depends on its specific requirements, and context.

5.3.3 Domain Modeling

Domain modeling is the process of structuring and representing the concepts, behaviors, and rules of a specific problem or business area within a software application[41].

Domain modeling in TCA focuses on creating clear, consistent, and maintainable representations of an application's state, actions, and logic[42]. By using enums to model mutually exclusive states, TCA ensures that invalid or contradictory configurations are impossible, simplifying the logic required to manage application behavior[42]. For instance, navigation states can be represented as a single enum rather than multiple optional variables, reducing redundancy and guaranteeing that only one navigation destination is active at a time. Similarly, consolidating optional modal or sheet states into a single @PresentationState property backed by an enum further enhances clarity and simplifies state management[42].

TCA's structured approach to state and action design ensures that features remain modular and focused, with reducers scoping their logic to handle only the relevant parts of the application domain. Parent-child communication is elegantly handled through delegate actions, which allow child features to notify their parents about critical events, such as deleting an item or updating shared data, while maintaining encapsulation. This approach aligns state transitions with business logic, making applications easier to reason about and extend[42].

5.3.4 Persistence

Persistence in TCA is about saving and loading app data while ensuring the system remains flexible, testable, and predictable. This involves creating mechanisms to write data to the device's storage, like saving a list of meetings or updates to a feature, and then reloading that data when the app starts[43]. However, deciding when and how often to save can be tricky. Saving after every action can overwhelm the system while saving too infrequently risks losing updates[43].

One of the useful tools that came with TCA is a "debounced" saving strategy. This means saving happens only after a short delay following user actions, preventing frequent file operations while ensuring data remains current. The ‘Reduce‘ method in TCA is used to handle this, ensuring that saving occurs only after actions are processed. Additionally, a timer function ensures saves are not too frequent by "debouncing" the save operations[43].

6 PLANNING

I will explore the development and implementation of an iOS application using two distinct architectural patterns: The Composable Architecture (TCA) and Model-View-ViewModel (MVVM). The application, built with SwiftUI, enables users to search for podcasts, discover episodes, and stream playback through an intuitive and user-friendly interface. Initially, the app will be developed using TCA and then refactored into MVVM. By implementing the same application in MVVM and TCA, this project seeks to evaluate their performance, memory usage, storage efficiency, and scalability. The tool used for evaluating performance is Xcode Instruments.

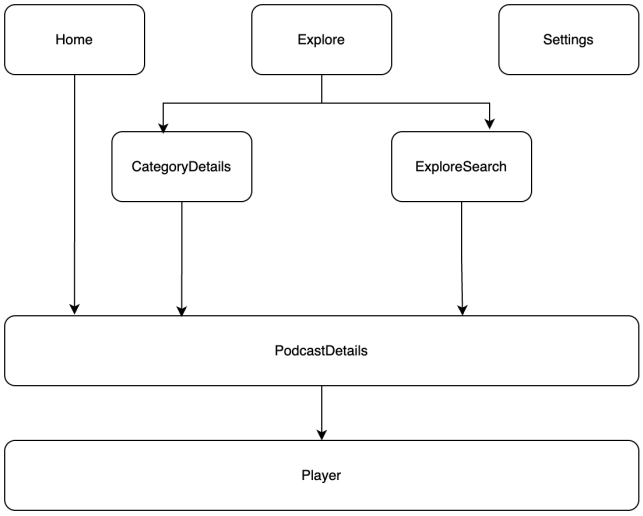


Figure 6.1: App Architecture

From broad discovery to focused playback, the App Architecture, as shown in Figure 6.1, illustrates the navigation flow and relationships between the screens in the podcast app, organized around three main tabs: Home, Explore, and Settings. The Home screen serves as the primary entry point, displaying trending podcasts and directly navigating to the Podcast Details screen when a user selects a podcast. From the PodcastDetails screen, users can view the selected podcast episode list and proceed to the Player screen to play specific episodes. The Explore tab provides two subviews for discovering podcasts: CategoryDetails, which organizes podcasts by genre and Explore Search, which allows users to search for specific podcasts or episodes. Both subviews navigate to the PodcastDetails screen upon podcast selection, maintaining a consistent flow toward the Player screen for playback. The Settings tab is focused on app management tasks, such as clearing cache, and does not directly connect to other screens. At the center of the app's navigation is the Podcast Details screen, which serves as the hub for podcast episode exploration and acts as a bridge to the Player screen.

7 IMPLEMENTATION

7.1 High-level Overview

7.1.1 Home

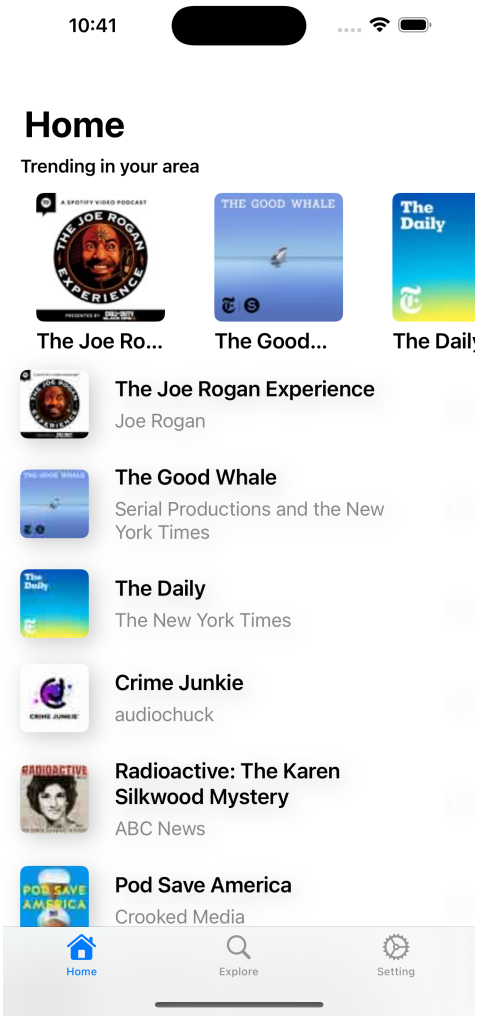


Figure 7.1: Home

The Home Screen, shown in Figure 7.1, provides a user-friendly experience. The home screen has a navigation bar that displays the title "Home," establishing the page's focus. Users can browse horizontal view cells of trending podcasts, and a vertical one just below it, with thumbnails and titles. The tapping of any podcast transitions smoothly to a detailed view for further exploration. During content loading, a progress indicator ensures users remain informed without interrupting the experience.

7.1.2 Explore

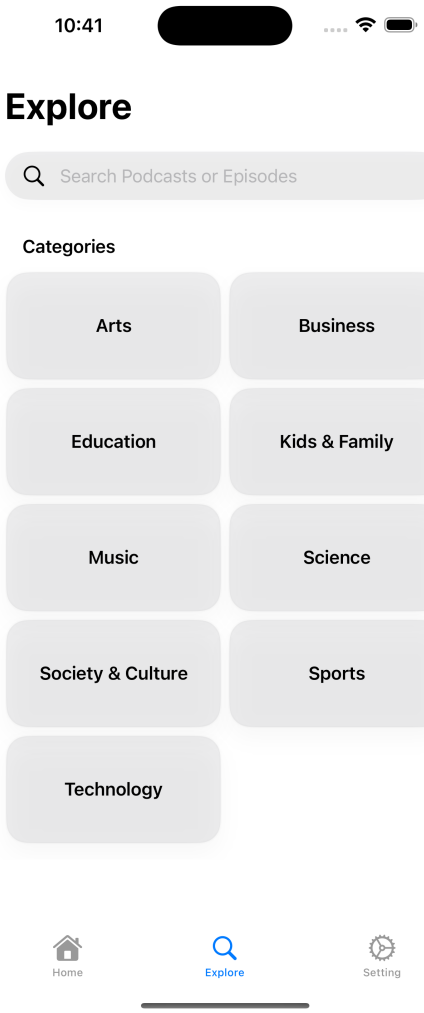


Figure 7.2: Explore

The Explore screen, shown in Figure 7.5, offers users a way to navigate various podcast categories and explore curated content. The explore screen has an expandable navigation bar that adjusts dynamically as users scroll. Users can search for podcasts or episodes using a search bar. Categories are presented in a grid format. A horizontal tab allows users to filter between all content, podcasts, and episodes seamlessly. During loading, a progress indicator keeps users informed. The view is designed to provide a smooth browsing experience.

7.1.3 Settings

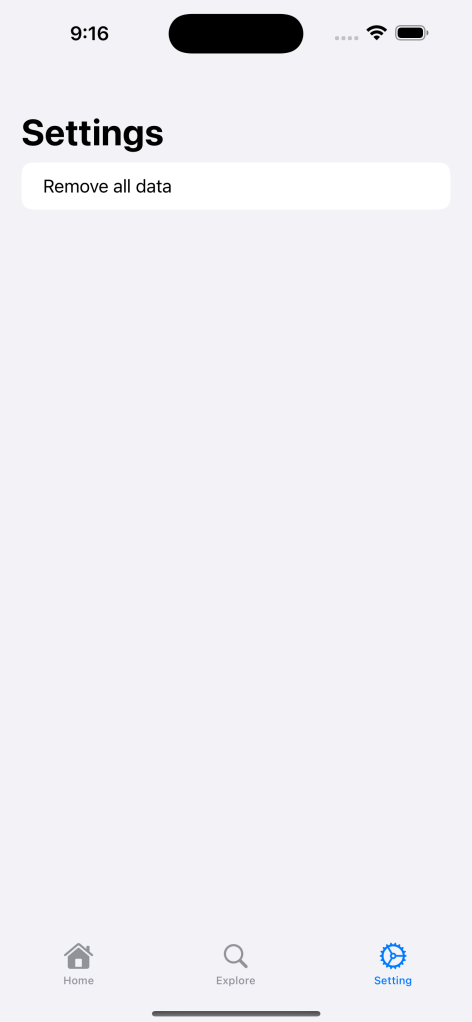


Figure 7.3: Settings

The Settings screen, as shown in Figure 7.3, provides the ability to clear app caches and temporary files by tapping a clearly labeled option. This triggers a confirmation dialog. If confirmed, a loading indicator appears while the cache is cleared, followed by an alert notifying users of successful completion. this functionality is particularly important when comparing the MVVM and TCA versions of the app, as clearing caches ensures a fresh start for accurate testing and evaluation.

7.1.4 Podcast Details

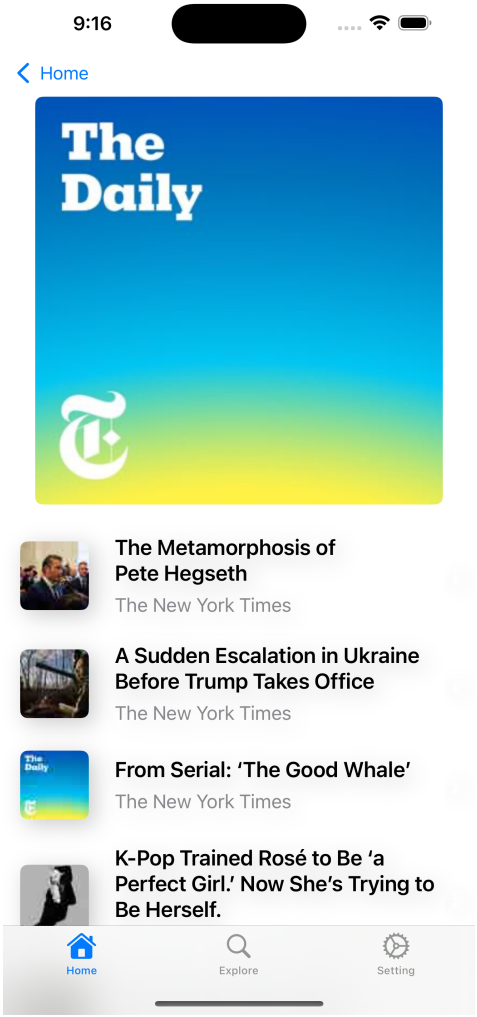


Figure 7.4: PodcastDetails

The Podcast Details screen, as shown in Figure 7.4, provides users with the ability to explore podcast episodes. At the top, a hero section displays the podcast's cover image. Below, users can scroll through a list of episodes, each presented with details like the title, author, and thumbnail. Tapping on an episode allows users to transition to a player screen for playback. While the content is loading, a progress indicator appears to let users know.

7.1.5 Explore Search

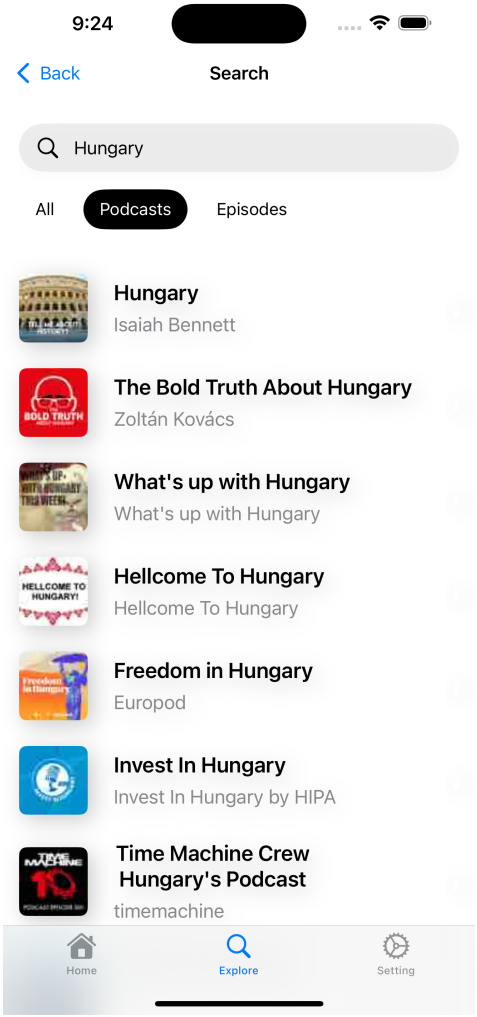


Figure 7.5: ExploreSearch Screen

Explore search, as shown in Figure 7.5, provides users with the ability to discover podcasts and episodes through a search interface. Users have the ability to input keywords and filter their results by selecting from three tabs, "All," "Podcasts," or "Episodes". As the results populate, users can browse a list with podcast and episode information, including titles, authors, and thumbnails. Each result is tappable, leading to further exploration or playback.

7.1.6 Category Details

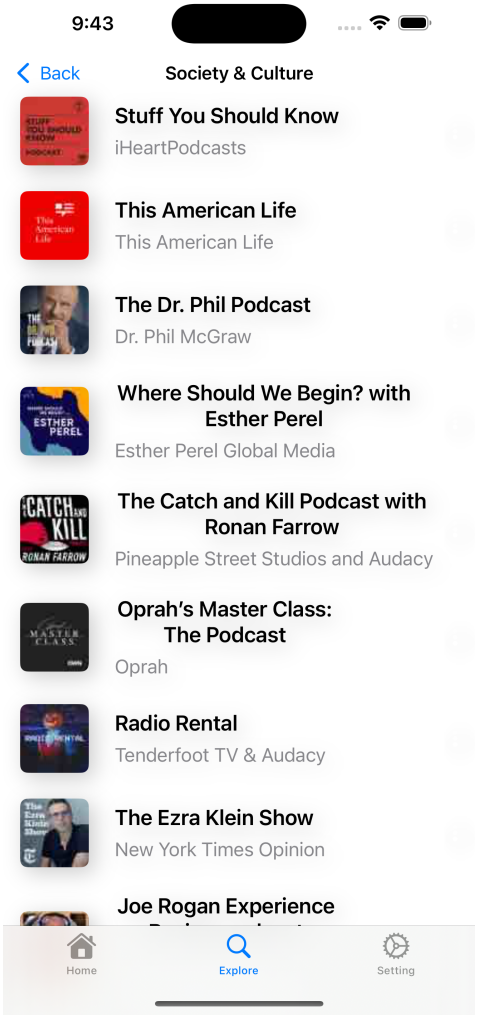


Figure 7.6: CategoryDetails Screen

The Category Details screen, as shown in Figure 7.6, provides users with a way to explore podcasts within a specific category. At the top, the category name is displayed as the navigation title. Users can scroll through a vertically arranged list of podcasts. Each podcast entry is tappable, enabling navigation to more in-depth details.

7.1.7 Player

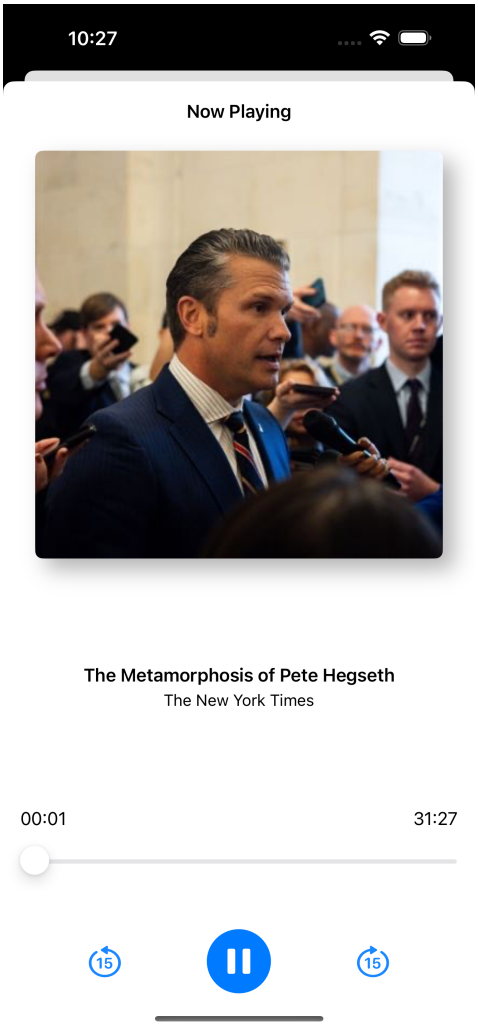


Figure 7.7: Player Screen

The Player screen, as shown in Figure 7.7, provides the ability to play audio, allowing users to listen to podcast episodes. Below, users can see the playback progress with elapsed and total time displayed alongside a draggable slider for precise navigation within the episode.

7.2 Technical Overview

When building an application, we need to focus on two primary components: the user interface (UI) and the domain or data layer. In TCA, the data layer is represented as a Feature. A Feature is a modular unit of functionality that encapsulates a specific aspect of an application's behavior. Three main components are required to define a feature: state, reducer, and action. These components are the foundational building blocks for the Feature to function within the composable architecture. Optionally, a Path or destination can also be defined to enable navigation functionality.

On the other hand, when implementing the same application in MVVM, we need to create a ViewModel in addition to the UI. The ViewModel is where the core logic resides. It acts as an intermediary between the View and the Model, managing the View's state and handling input and output operations that dictate the View's behavior. Moreover, the ViewModel translates business logic and data from the Model into a format that the View can easily present.

This thesis work brings a podcast application that was built in TCA, and then refactored into MVVM, and as previously mentioned, the app consists of seven screens. This makes a total of 21 components for each State, Reducer, and Action, within the context of TCA, and 7 viewModel classes within the context of MVVM. For this reason, including the complete code and detailed descriptions of all components in this thesis documentation is not practical. Instead, the source code has been uploaded to GitHub, and I will focus on explaining the key aspects of the code implementation.

7.3 App Screens

7.3.1 Home

The home screen is the first screen users see upon launching the application, allowing them to fetch and view trending podcasts. Tapping on a podcast cell navigates the user to the PodcastDetails screen, where episodes of the selected podcast are displayed. In TCA, defining the HomeFeature involves three components: State, Action, and Reducer. State is used to declare the properties of the feature, while Action defines all permissible actions within the HomeFeature, and Reducer is the only location for mutating state and triggering actions. This structure ensures that changes and calls are easy to track, preventing state modifications and enforcing a clear separation of concerns. In contrast, the MVVM architecture uses a ViewModel, which is a class containing functions for handling user interactions and properties for managing data.

Listing 6: TCA

```
import ComposableArchitecture

@Reducer
public struct HomeFeature: Sendable {
    @ObservableState
    public struct State: Equatable {
    }

    public enum Action {

    }

    public init() {}
    public var body: some ReducerOf<Self> {
        Reduce { state, action in
            switch action {

            }
        }
    }
}
```

Listing 7: MVVM

```
import Foundation

class HomeViewModel: ObservableObject {
    @Published @MainActor var podcasts: [Podcast]?

    init() {
    }
}
```

In TCA, structs are used as the primary data structure, whereas in MVVM, a class data structure is used. This is the first major difference between the two architectures. The choice of data structure impacts the nature of the architecture itself. TCA is a unidirectional architecture where the struct is a value type, while MVVM is bidirectional and relies on the class, which is a reference type.

Another distinction lies in how state updates are managed. In MVVM, we use @Published, which publishes changes to the UI. When the value it holds is updated, the UI reflects those changes, meaning that changes to the data drive the UI. In TCA, we achieve similar behavior using @ObservableState, which serves the same purpose by notifying the system of state changes.

Additionally, when defining functions or actions that can be performed within a feature or ViewModel,

TCA organizes them in an enum called Action, encapsulating all possible actions. In contrast, MVVM simply declares these as functions within the ViewModel. For example:

Listing 8: TCA

```
public enum Action {
    case fetchTrendingPodcasts
    case trendingPodcastResponse(PodcastResult)
}

public var body: some ReducerOf<Self> {
    Reduce { state, action in
        switch action {
        case .fetchTrendingPodcasts:
            state.isLoading = true
            return .run { send in
                try await send(
                    .trendingPodcastResponse(
                        podhubClient.getLocalTrendingPodcasts(50)
                    )
                )
            }
        case .trendingPodcastResponse(let result):
            state.podcasts = result.podcastList
            state.isLoading = false
            return .none
        }
    }
    .ifLet(\.$destination, action: \.destination)
    .forEach(\.path, action: \.path)
}
```

Listing 9: MVVM

```
func getTrendingPodcasts() {
    isLoading = true
    Task {
        defer {
            isLoading = false
        }
        podcasts = try await podHubManager.getLocalTrendingPodcasts()
        .podcasts
    }
}
```

In the Action enum in TCA, we define a blueprint of all the actions that can be performed within a feature. These blueprints are implemented in the Reducer, as it is the only place where states can be mutated. One of the key advantages of this approach is that, by using an enum for actions, the switch statement on the actions must be exhaustive. This ensures that no actions are left unhandled, reducing the risk of unused actions or boilerplate code as the project scales.

Another key difference between TCA and MVVM is how they handle network calls, reflecting their architectural purposes. In MVVM, a single function typically handles both fetching data and assigning the result. This approach, while straightforward, consolidates multiple responsibilities into one function. For example, the MVVM code provided shows how the getTrendingPodcasts function not only initiates the network call but also directly updates the UI state (isLoading) and assigns the fetched podcasts to the podcasts property. While this works well for simpler use cases, it can lead to tightly coupled code as the application grows, making it harder to test and scale individual components.

In contrast, TCA adopts a more modular approach by separating the network call into two distinct actions: one to initiate the fetch and another to handle the response. In the example provided, the Action enum defines fetchTrendingPodcasts to start the network operation and trendingPodcastResponse to handle the data once it is returned. This separation is implemented in the Reducer, where each action is processed in isolation. The fetchTrendingPodcasts action updates the state to reflect a loading state (state.isLoading = true) and triggers the network call, while the trendingPodcastResponse action processes the returned data, updates the state with the result, and resets the loading flag.

This modular approach in TCA provides several advantages. By decoupling the network call from the handling of its response, each step becomes easier to test and reason about independently. The reducer acts as a centralized place to manage state transitions, ensuring predictable and traceable changes. Additionally, this design is inherently composable, allowing different actions and reducers to be combined seamlessly within a feature or across features.

7.3.2 Explore

The Explore screen allows users to browse podcast categories, search for specific podcasts, and navigate to detailed views. Tapping on a category navigates the user to the CategoryDetails screen, where podcasts within the selected category are displayed. Similarly, users can search for podcasts directly on the Explore screen. Upon initiating a search, they are taken to the ExploreSearchResults screen, where the results are displayed, and further searches can be performed without returning to the Explore screen.

Listing 10: TCA Implementation

```
import ComposableArchitecture

@Reducer
public struct ExploreFeature: Sendable {
    @ObservableState
    public struct State: Equatable {
        var podcasts: [Podcast]?
        var isLoading: Bool = false
```

```
        var searchTerm: String = ""
    }

    public enum Action {
        case fetchPodcasts
        case searchForPodcast(String)
        case fetchPodcastsResponse(PodcastResult)
    }

    public var body: some ReducerOf<Self> {
        Reduce { state, action in
            switch action {
            case .fetchPodcasts:
                state.isLoading = true
                return .run { send in
                    try await send(
                        .fetchPodcastsResponse(
                            podHubClient.getTrendingPodcasts()
                        )
                    )
                }
            case .fetchPodcastsResponse(let result):
                state.isLoading = false
                state.podcasts = result.podcastList
                return .none
            case .searchForPodcast(let term):
                state.isLoading = true
                return .run { send in
                    try await send(
                        .fetchPodcastsResponse(
                            podHubClient.searchFor(term)
                        )
                    )
                }
            }
        }
    }
}
```

Listing 11: MVVM Implementation

```
import Foundation

class ExploreViewModel: ObservableObject {
    @Published var podcasts: [Podcast]?
```

```
    @Published var isLoading: Bool = false

    func fetchPodcasts() {
        isLoading = true
        Task {
            defer { isLoading = false }
            podcasts = try await podHubManager.getTrendingPodcasts()
            .podcastList
        }
    }

    func searchForPodcast(_ term: String) {
        isLoading = true
        Task {
            defer { isLoading = false }
            podcasts = try await podHubManager.searchFor(term).podcastList
        }
    }
}
```

In TCA, same to the home screen, the ExploreFeature uses a Reducer to define how actions modify the state, separating concerns into distinct actions such as fetchPodcasts and fetchPodcastsResponse. This modularity ensures that state transitions are predictable and explicitly manage side effects. On the other hand, the MVVM implementation centralizes logic within the ExploreViewModel, where asynchronous methods directly update properties like podcasts and isLoading. While this approach is simpler for smaller features, it tightly couples data handling and UI state updates.

7.3.3 Settings

The Settings screen allows users to manage app settings, mainly, clearing cached data, which is very important when comparing the MVVM and TCA versions of the app because clearing caches makes sure testing starts fresh and results are accurate. The screen includes UI elements like alerts and confirmation dialogs for user interactions. Clearing the cache involves presenting a confirmation dialog before executing the action and notifying the user with an alert upon completion.

In the TCA implementation, alerts and confirmation dialogs are represented as part of the state using @Presents, which ensures clear and predictable state transitions. Actions such as alertButtonTapped and confirmationDialogButtonTapped handle user interactions and define the behavior of these UI elements. The reducer processes these actions, updating the state and triggering side effects like clearing the cache.

In contrast, the MVVM implementation uses a SettingsViewModel to manage the state directly. The clearAllCaches method encapsulates the logic for performing the cache clearing operation, updating the isLoading and didClearCache properties to reflect the operation's progress and result. While this approach simplifies the implementation, it lacks the modularity and explicit state handling provided by TCA.

7.3.4 Podcast Details

The Podcast Details screen allows users to view the episodes of a selected podcast. When a user selects a podcast, the app retrieves the podcast's RSS feed URL, which contains metadata about its episodes. This URL is used to fetch and parse the RSS feed into episode objects. These episode objects are then displayed on the screen for the user to browse and interact with.

In TCA, the state of the screen includes properties such as the selected podcast, an optional list of episodes, a loading indicator, and a player state for playing a selected episode. Actions are defined to handle fetching episodes (fetchEpisode), tapping on an episode cell (cellTapped), and managing playback (playEpisode). When the fetchEpisode action is triggered, the RSS feed URL from the selected podcast is parsed asynchronously into episodes. Once parsing is complete, the episodes are returned via the episodeResponse action, which updates the state to include the fetched episodes and stops the loading indicator. If a user taps on an episode, the cellTapped action updates the state to present the episode in the player.

In the MVVM implementation, the ViewModel initializes with the selected podcast and its RSS feed URL. The ViewModel uses @Published properties to manage the loading state and list of episodes, allowing the UI to reactively update as changes occur. The fetchEpisodes function fetches and parses the RSS feed asynchronously, assigning the resulting episodes to the episodes property and updating the loading state appropriately. Both implementations outsource RSS parsing to the FeedKit library.

Since we are comparing two architectures (TCA and MVVM), it would be ideal to outsource the RSS parsing logic to FeedKit. Using FeedKit as a dedicated library for parsing ensures consistency and reliability in handling RSS feeds across both implementations. This approach keeps the parsing logic separate from the architectural layers, allowing TCA and MVVM to focus on their core responsibilities without being burdened by feed processing.

7.3.5 Explore Search

The Explore Search screen allows users to search for podcasts or episodes by entering a search term. Users can specify whether they want to search across all content or within a specific tab, such as podcasts or episodes. Once a search is initiated, the application retrieves results asynchronously and displays them to the user.

In TCA implementation, the state includes properties for storing the search results (`searchResult`), a list of episodes (`episodes`), the current loading state (`isLoading`), the search term (`searchTerm`), the active search tab (`activeTab`), and the player state for playing an episode (playEpisode). The `Action` enum defines user interactions, such as entering a search term (`searchTermChanged`), initiating a search (`searchForPodcastTapped`), and viewing results (`showSearchResults`). When a user initiates a search, the `searchForPodcastTapped` action updates the loading state, clears previous results, and triggers an asynchronous search via the injected `podHubClient`. Once the results are fetched, the showSearchResults action updates the state with the retrieved data and resets the loading state. Users can also tap an episode to play it, handled by the `cellTapped` and `playEpisode` actions. The reducer ensures all actions are processed in a predictable, unidirectional flow.

In the MVVM implementation, the `ExploreSearchViewModel` manages the state and interactions. It

uses `@Published` properties to reactively update the UI with loading states and the list of podcasts. The `searchForPodcast` method handles the search logic, updating the `podcastList` with results fetched asynchronously using the injected `podHubManager`.

7.3.6 Category Details

The Category Details screen allows users to view podcasts within a selected category. When a user selects a category, the application fetches a list of podcasts associated with that category and displays them.

In the TCA implementation, the state includes properties such as the selected category (`category`), the loading status (`isLoading`), and a list of podcasts (`podcasts`). The `Action` enum defines actions for initiating a fetch (`fetchPodcastList`) and handling the response (`podcastResponse`). When the `fetchPodcastList` action is triggered, the podcasts are cleared, the loading state is set to `true`, and an asynchronous call is made to fetch podcasts using the injected `podHubClient`. Once the data is fetched, the `podcastResponse` action updates the state with the list of podcasts and resets the loading status. The reducer ensures a clear and predictable flow of actions and state transitions.

In the MVVM implementation, the `CategoryDetailsViewModel` manages the state with @Published properties, allowing the UI to update as data changes. The ViewModel uses the getPodcastList method to fetch the podcast list for the selected category asynchronously. This method updates the `isLoading` state while the data is being retrieved and assigns the fetched podcasts to the `podcasts` property once the call is complete.

7.3.7 Player

The Player Screen allows users to play, pause, and manage the playback of a podcast episode. It uses an underlying `AVPlayer` instance for audio playback and provides controls for handling play, pause, and resume actions.

In the TCA implementation, the state includes key properties such as the `AVPlayer` instance (`player`), the playback status (`isPlaying`), the episode's audio URL (`audioURL`), and a shared `RunningItem` instance that tracks the currently playing episode and its playback times. The `Action` enum defines user interactions, such as handling play actions (`handlePlayAction`), updating playback state (`updateIsPlaying`), and managing the episode's current and total times (`onCurrentTimeChange` and `onTotalTimeChange`).

The reducer processes these actions predictably. For instance, when `handlePlayAction` is triggered, the reducer checks the playback state and either plays, pauses, or resumes the episode using the AudioPlayerManager singleton. The `immediatelyPlay` action ensures seamless playback of a newly selected episode, resetting the current state if a different episode is being played.

In the MVVM implementation, the `PlayerViewModel` handles playback logic and state management. The `PlayerViewModel` initializes with the selected episode and uses `@Published` properties to track the playback state (`playerStatus`), total duration (`totalTime`), and elapsed time (`elapsedTime`). The `subscribe` method establishes Combine subscriptions to observe playback events from AudioPlayerManager. These updates ensure that changes to the playback state or time are reflected in the UI.

The `play` and `handlePlayButton` methods manage playback controls. Depending on the current state, these methods instruct `AudioPlayerManager` to play, pause, resume, or restart playback. This approach encapsulates playback logic within the ViewModel, allowing for direct UI binding via Swift's reactive properties. Overall, while both architectures achieve similar functionality, TCA offers a more modular approach suitable for complex state management, whereas MVVM simplifies the implementation for smaller-scale features.

7.4 Common App Managers and Frameworks

Some libraries have been used for tasks like logging, image caching, and RSS feed parsing to establish a neutral, optimized starting point for both the TCA and MVVM versions of the application. By outsourcing these details to external frameworks, any measured differences in performance, memory, or CPU usage can be traced back to each architecture's inherent characteristics rather than minor implementation discrepancies. This ensures that the comparison between TCA and MVVM is both fair and meaningful, accurately highlighting how each architectural pattern influences the app's overall efficiency and responsiveness.

7.4.1 Logger Manager

Logger manager is logging system for the application. It is active only in debug builds and provides several logging functions, PODLogError, PODLogWarn, PODLogInfo, PODLogDebug, and PODLogVerbos for emitting messages at various levels of severity. Within debug mode, controlled by the #if DEBUG condition, shouldLog is set to true, ensuring that log messages will be printed. Each public logging function uses a shared PODLog class to handle message formatting and routing. The PODLog class assigns log message types (e.g., .error, .warning) and constructs a descriptive output that includes the filename, function name, and line number where the log was called. The PODLogFormatter singleton is responsible for adding timestamps, tagging each message with an emoji and severity label, and then printing the final message. It also employs extensions to String to neatly extract file name information from the file parameter.

7.4.2 PodHub Manager

The PodHub Manager acts as a centralized client for asynchronously retrieving and managing podcast data from the iTunes API. It provides functions for searching podcasts or episodes based on specific criteria, fetching trending shows either from a locally inferred region or a chosen country, and retrieving podcasts categorized by genre.

7.4.3 Player Manager

The Player Manager is responsible for controlling and tracking the playback of audio content, such as podcasts. It uses Apple's AVPlayer to load and play audio streams, and integrates with system services like MPRemoteCommandCenter and MPNowPlayingInfoCenter to provide lock screen and control center interactions. Features include queue management, setting up time observers to continuously track playback progress, and handling remote commands like play, pause, skip forward, and skip backward. It also responds to audio interruptions, updates the user interface with details like the current track and artwork, and manages audio sessions

for seamless background playback.

7.4.4 ItunesPodcast Manager

ItunesPodcast Manager is a specialized component designed to help discover, search, and retrieve podcasts from two key data sources. First, it utilizes the official iTunes API to run customized searches based on various parameters like search terms, country, genre, or language, ensuring listeners find content that matches their preferences. Second, it reaches out to Apple's Marketing Tools API to identify and gather popular, trending podcasts for a given region.

7.4.5 FeedKit Framework

In the implemented podcast app, the FeedKit framework is utilized to simplify the process of converting raw RSS feeds into organized episode data. Instead of manually parsing the XML, FeedKit automatically translates the incoming RSS feed into a collection of episodes complete with titles, descriptions, authors, and release dates.

7.4.6 KingFisher Framework

The KingFisher framework enhances the visual experience by efficiently handling podcast artwork and episode thumbnails in the implemented podcast app. When images are needed, KingFisher retrieves them from the network and stores them in a local cache, greatly reducing loading times and bandwidth consumption. As a result, navigating through episodes and browsing various shows feels smooth and responsive, even under slower network conditions.

7.4.7 SwiftyJSON Framework

The SwiftyJson framework simplifies the management of data received from external services that deliver results in JSON format. In the implemented podcast app, complex JSON responses, such as search results or trending lists, are parsed with minimal overhead.

8 TEST AND MEASUREMENT

In this section, I will conduct a series of tests on the developed application using a physical device to monitor its activity and behavior. The tests are performed on an iPhone 15 Pro Max running iOS 18.1. The testing and monitoring are carried out using Xcode Instruments as the primary tool.

8.1 Memory Consumption

The testing involved four distinct scenarios to evaluate memory consumption for two app architectures, TCA and MVVM. Each scenario was executed three times to compute reliable averages. Below are the scenarios and the corresponding results:

1. App Launches with No Action Performed
This scenario measured memory usage when the app was launched but no further interaction occurred.

- TCA: Average memory consumption was 49.86 MB.
- MVVM: Average memory consumption was 56.76 MB.

2. App Launches and Scrolls to the Bottom of the Trending Podcast List
Here, the app was launched, and the user scrolled to the bottom of the trending podcast list to simulate basic interaction.

- TCA: Average memory consumption was 52.2 MB.
- MVVM: Average memory consumption was 66.5 MB.

3. App Launches and Plays a Specific Episode
In this scenario, the app was launched, and a specific podcast episode was played to assess memory usage during media playback.

- TCA: Average memory consumption was 65.5 MB.
- MVVM: Average memory consumption was 66.2 MB.

4. App Launches and Searches for a Specific Term
This test involved launching the app and performing a search for a specific term to evaluate memory consumption during a search operation.

- TCA: Average memory consumption was 137.6 MB.
- MVVM: Average memory consumption was 147.0 MB.

Table 8.1: Memory Consumption Measurements for TCA and MVVM

Scenario	**First Run (MB)**	**Second Run (MB)**	**Third Run (MB)**	**Average (MB)**
TCA				
No action performed	49.9	49.4	50.3	49.86
Scrolling to the bottom of the trending podcast list	52.8	52.2	51.6	52.2
Playing a specific episode	69.6	58.9	68.0	65.5
Searching for a specific term	136.0	137.0	140.0	137.6
MVVM				
No action performed	57.5	56.6	56.2	56.76
Scrolling to the bottom of the trending podcast list	68.0	66.4	65.1	66.5
Playing a specific episode	66.1	66.8	65.7	66.2
Searching for a specific term	145.0	147.0	149.0	147.0

8.1.1 Summary & Observations

TCA demonstrates lower memory consumption compared to MVVM across most scenarios tested. For instance, in the "No action performed" scenario, TCA averages 49.86 MB, significantly less than MVVM's 56.76 MB. Similarly, when scrolling through the podcast list, TCA's memory usage averages 52.2 MB, markedly lower than MVVM's 66.5 MB.

However, the differences come closer in scenarios involving more complex interactions. For example, when playing a specific episode, TCA averages 65.5 MB, slightly below MVVM's 66.2 MB. Searching for a term results in the highest memory usage for both architectures, with MVVM consuming 147.0 MB compared to TCA's 137.6 MB, reflecting a smaller but consistent advantage for TCA.

8.2 CPU Usage

8.2.1 CPU Usage in TCA

When launching the application, the first 30.857 seconds show normal behavior, except for the initial 2.5 seconds, which show a noticeable CPU spike, as shown in Figure 8.1. This spike occurs because, upon launch, the application initializes and immediately fires several network calls to fetch trending podcasts, causing a temporary increase in CPU usage.

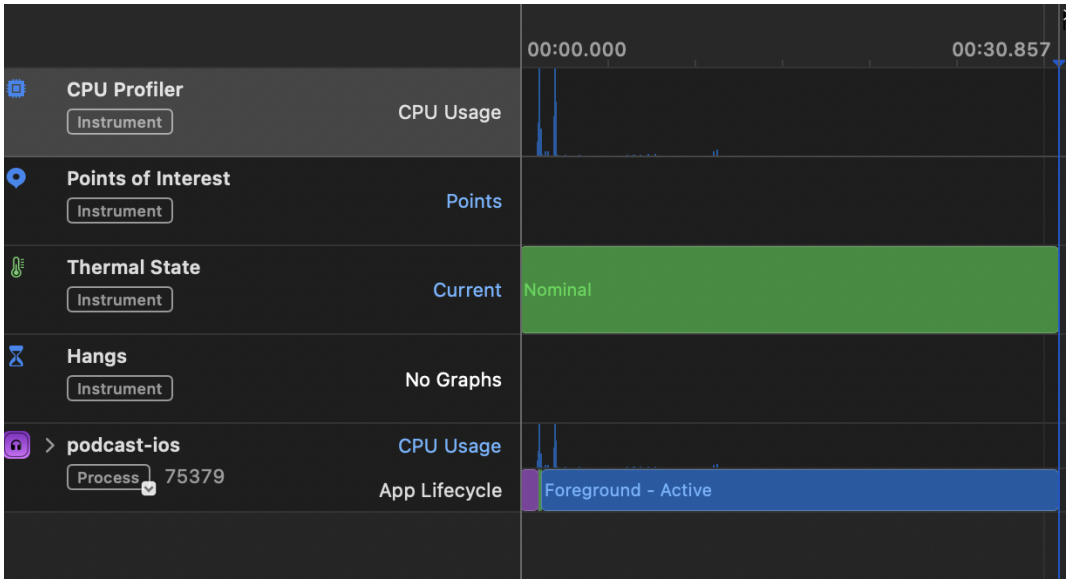


Figure 8.1: First 30-second Application Launch: TCA

When launching the application, navigating to a podcast cell, and selecting any episode, the app transitions to the player screen. Upon starting playback of an episode, the process consumes up to 4% of CPU resources, as illustrated in Figure 8.2.

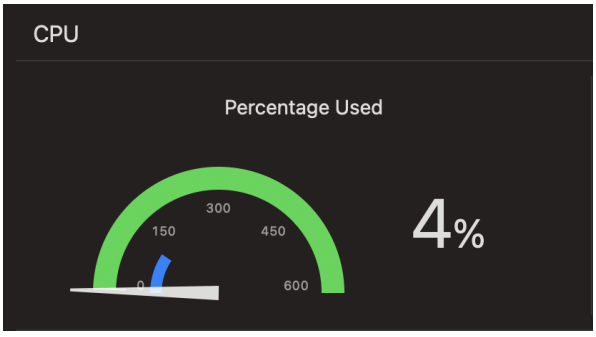


Figure 8.2: CPU Usage: TCA

8.2.2 CPU Usage in MVVM

When launching the application, the first 30.711 seconds show normal behavior, except for the initial 2.3 seconds, which show a noticeable CPU spike, as shown in Figure 8.3. Just like in TCA, this spike occurs because, upon launch, the application initializes and immediately fires several network calls to fetch trending podcasts, causing a temporary increase in CPU usage.

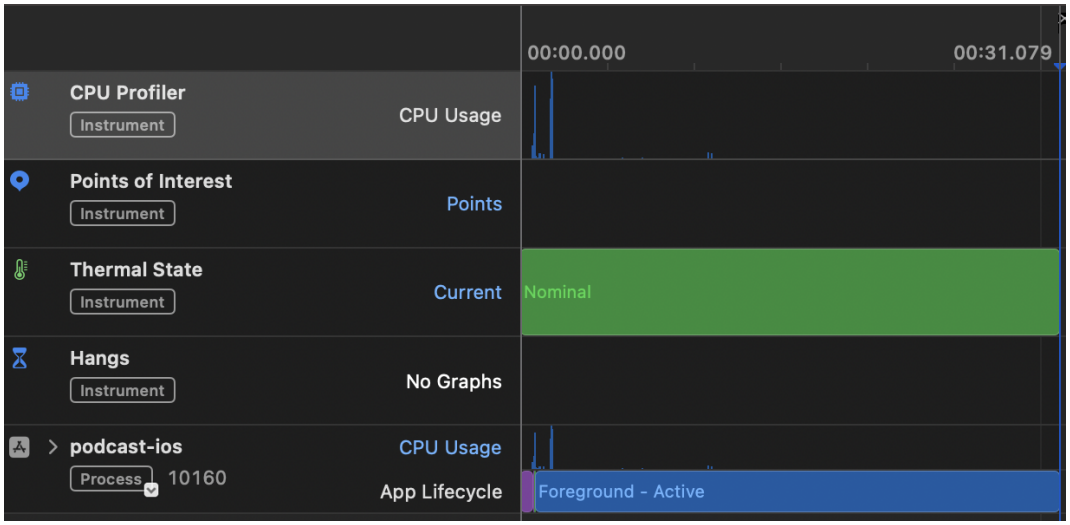


Figure 8.3: First 30-second Application Launch: MVVM

However, when launching the application, navigating to a podcast cell, and selecting any episode, the app transitions to the player screen. Upon starting playback of an episode, the process consumes up to 3% of CPU resources, as illustrated in Figure 8.4.

8.3 Unit Testing

All tests were conducted using Testing Library Version 94 on an arm64e architecture compatible with Apple devices running iOS 15.6. Both Unit Testing in TCA and MVVM achieved a perfect 100% success rate, with no failures or unexpected outcomes. The tests were executed efficiently across different feature suites.

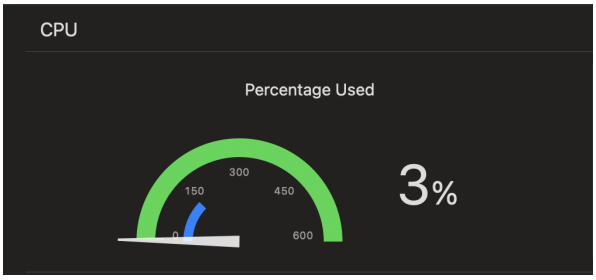


Figure 8.4: CPU Usage: MVVM

8.3.1 Unit Testing in TCA

The test run assessed eight tests across three feature suites-SettingsFeatureTests, ExploreFeatureTests, and HomeFeatureTests-completing in a total of 0.348 seconds.

- SettingsFeatureTests: This suite executed a single test, testInitialState, which validated the initial application state and passed in 0.003 seconds.
- ExploreFeatureTests: This suite ran four tests:
  - testInitialState: Verified proper state initialization (0.005 seconds).
  - testSearchTermTest: Validated search term functionality (0.342 seconds).
  - testCategoryUserFlow: Checked category navigation flows (0.346 seconds).
  - testFetchTrendingPodcastFirstItem: Ensured accurate fetching of the first trending podcast item (0.346 seconds).
- HomeFeatureTests: This suite executed three tests:
  - testInitialState: Verified proper state initialization (0.346 seconds).
  - testFetchTrendingPodcastFlow: Validated the retrieval of trending podcast data (0.346 seconds).
  - testFetchTrendingPodcastCount: Confirmed the correctness of the podcast count (0.346 seconds).

8.3.2 Unit Testing in MVVM

The test run assessed seven tests across three feature suites: SettingsFeatureTests, ExploreFeatureTests, and HomeFeatureTests-completing in a total of 0.210 seconds.

- HomeViewModelTests: This suite included three tests:
  - testInitialState: Validated the initial application state (0.001 seconds).
  - testFetchTrendingPodcastFlow: Ensured correct podcast data retrieval (0.209 seconds).
  - testFetchTrendingPodcastCount: Confirmed the accuracy of the podcast count (0.133 seconds).
- ExploreViewModelTests: This suite comprised four tests:
  - testInitialState: Verified proper state initialization (0.079 seconds).
  - testSearchTermTest: Validated search functionality (0.185 seconds).
  - testFetchTrendingPodcastFirstItem: Confirmed accurate retrieval of the first trending podcast item (0.162 seconds).
  - testCategoryUserFlow: Checked category navigation flows (0.125 seconds).

8.3.3 Case Study: CategoryUserFlow

Listing 12: Testing Category User Flow in MVVM

```
@Test
func test_categoryUserFlow() async throws {
    @Injected(\.podHubManager) var podHubManager: PodHubManagerProtocol
    podHubManager = MockPodHubManager()
    let sut = ExploreViewModel()
    let firstCategory = sut.catagoryList.first!
    #expect(firstCategory.id == .arts)
}
```

Listing 13: Testing Category User Flow in TCA

```
@Test
func test_categoryUserFlow() async throws {
    let store = TestStore(initialState: ExploreFeature.State()) {
        ExploreFeature()
    } withDependencies: {
        $0.podHubClient = .mock()
    }
    let firstCategory = store.state.catagoryList.first!
    #expect(firstCategory.id == .arts)
    #expect(store.state.path.isEmpty)
    await store.send(.catagoryTapped(firstCategory))
    #expect(!store.state.path.isEmpty)
}
```

Testing the Category User Flow in MVVM is significantly more challenging compared to TCA because of the tight coupling between the view and navigation logic, typically through constructs like NavigationLink. In MVVM, the navigation behavior is closely tied to the ViewModel's state, which is observed by the view to trigger navigation. This coupling makes it difficult to test the navigation flow in isolation, as the test must either simulate the view's behavior or rely on indirect assertions about the ViewModel. For example, determining if a navigation occurred often depends on implicitly verifying the state mutations within the ViewModel, which can lead to brittle and hard-to-maintain tests. In contrast, TCA decouples navigation logic from the view by encapsulating it within the feature's state and actions. Using await store.send(.catagoryTapped(firstCategory)), testers can explicitly simulate user interactions and assert on the resulting state transitions, such as verifying changes to store.state.path. This declarative and structured approach in TCA not only makes navigation flows testable without involving the view but also ensures clarity and precision in verifying intended behavior. Consequently, TCA's design simplifies flow testing, making it more predictable, maintainable, and robust compared to the implicit and tightly coupled nature of MVVM testing.

9 OVERVIEW

9.1 Seperation of Concern

When choosing an architecture for a project, separation of concerns is one of the most critical factors developers consider. This principle ensures that each function or component is responsible for a single, specific task and performs only the job it was designed for. It avoids overloading functions with multiple responsibilities, making the codebase easier to understand, maintain, and scale. Adhering to this principle helps promote modularity and clarity, as each part of the system has a well-defined role and does not exceed its intended scope.

TCA is inherently Composable, this means that large, complex problems can be broken down into smaller, manageable components, each with its own distinct responsibilities. The architecture not only allows features to be decomposed into smaller parts but also enables these parts to be recomposed back into the domain of the larger feature, maintaining a cohesive structure. This composability is evident in the podcast app when managing complex workflows such as fetching and parsing RSS feeds, handling search functionality, managing category-specific podcasts, and controlling playback. For example:

Listing 14: TCA

```
public enum Action {
    case fetchTrendingPodcasts
    case trendingPodcastResponse(PodcastResult)
}

public var body: some ReducerOf<Self> {
    Reduce { state, action in
        switch action {
        case .fetchTrendingPodcasts:
            state.isLoading = true
            return .run { send in
                try await send(
                    .trendingPodcastResponse(
                        podhubClient.getLocalTrendingPodcasts(50)
                    )
                )
            }
        case .trendingPodcastResponse(let result):
            state.podcasts = result.podcastList
            state.isLoading = false
            return .none
        }
    }
    .ifLet(\.$destination, action: \.destination)
    .forEach(\.path, action: \.path)
}
```

On the other hand, MVVM consolidates these responsibilities within a single function, such as getTrendingPodcasts, which both retrieves the data and directly assigns it to a property. While simpler, the MVVM approach can blur the boundaries of responsibility, making it harder to scale and test in complex scenarios. For example:

Listing 15: MVVM

```
func getTrendingPodcasts() {
    isLoading = true
    Task {
        defer {
            isLoading = false
        }
        podcasts = try await podHubManager.getLocalTrendingPodcasts()
        .podcasts
    }
}
```

9.2 Navigation

In the Composable Architecture (TCA), navigation is managed explicitly through the state, ensuring a modular and predictable approach. For example, the `HomeFeature` uses the `path` property in the `State` struct to manage navigation hierarchies:

- Explicit Navigation State: Navigation is defined using the `path` property, a `StackState` type that handles the navigation stack. For example:

  ```
  public var path = StackState<Path.State>()
  ```

  When the user taps on a podcast, the `podcastDetailsTapped` action appends a new state for the `PodcastDetails` feature to the navigation stack:

  ```
  case .podcastDetailsTapped(let podcast):
      state.path.append(
      .podcastDetails(
      PodcastDetailsFeature.State(
      podcast: podcast
      )))
      return .none
  ```

- Navigation Logic in Reducer: Navigation actions are processed in the reducer, ensuring that navigation is tightly integrated with the application's state and actions. This makes deeplinking straightforward because appending to the `path` directly reflects the desired navigation.

– Deeplinking: Deeplinking in TCA involves modifying the `path` state. For instance, if a deeplink points to a specific podcast:

```
state.path.append(
.podcastDetails(
PodcastDetailsFeature.State(
podcast: podcast
)))
```

This approach ensures that navigation state is always consistent and easily testable.

In the MVVM architecture, navigation is handled directly in the view layer, relying on SwiftUI's `NavigationLink` for transitioning between screens.

– UI-Driven Navigation: Navigation logic is embedded in the UI code rather than being centralized in the state. For example:

```
NavigationLink {
    PodcastDetailsView(
    viewModel: PodcastDetailsViewModel(
    podcast: podcast
    ))
} label: {
    ListViewHero(
    imageURL: podcast.image ?? URL(string: ""),
    title: podcast.title
    )
        .frame(width: 150, height: 150)
}
```

The `PodcastDetailsView` is created inline with the required `PodcastDetailsViewModel`.

– Deeplinking: Handling deeplinks in MVVM requires external coordination. A deeplink would need to trigger the creation of a PodcastDetailsViewModel and presentation of the PodcastDetailsView. Since navigation is tied to the view hierarchy, additional logic must be implemented outside the ViewModel to handle deeplinking consistently.

In TCA, navigation is tightly integrated with the application state and reducer, making it predictable, modular, and well-suited for handling complex workflows like deeplinking. In contrast, MVVM relies on UI-driven mechanisms like `NavigationLink`, simplifying implementation for basic flows but lacking the centralized control and scalability provided by TCA.

Feature	TCA Navigation	MVVM Navigation
Navigation Logic	Managed explicitly through `path` in the `State` struct and processed in the reducer for predictability.	Embedded in the UI using `NavigationLink`, tying navigation directly to the view layer.
Modularity	Navigation is modular and centralized, enabling better testability and reuse across the application.	Navigation is decentralized, requiring inline logic within SwiftUI views, reducing modularity.
Deeplinking	Easily supported by appending the desired state to `path`.	Requires external handling to construct the ViewModel and present the appropriate view.

Table 9.2: Comparison of Navigation in TCA and MVVM

9.3 Testing

TCA has a clear advantage over MVVM due to two key factors: the testing process itself and the testability of the components being tested. As TCA is a framework, the testing process is enhanced by tools like the TestStore, which enforces exhaustiveness by requiring all actions and state transitions to be explicitly tested, which can be disabled, too, if not needed. Moreover, TestStore also allows developers to override dependencies during tests, creating controlled and predictable scenarios that ensure comprehensive and focused test coverage. Additionally, TCA's decoupled architecture, with explicit separation of state, actions, and side effects, makes individual components highly testable in isolation. This means reducers, for instance, can be tested independently without relying on external systems. In contrast, MVVM often couples logic, state, and reactive bindings within the ViewModel, making it harder to isolate specific functionality. Tests in MVVM may depend on other parts of the system, such as the UI or external services, leading to less reliable tests and increased maintenance complexity. TCA's structured design, therefore, shines in both facilitating the testing process and improving the testability of individual components, making it a more test-friendly architecture compared to MVVM.

9.4 Maintainability and Scalability

The Composable Architecture (TCA) fosters maintainability and scalability by breaking application logic into well-defined, independent components. Its structured approach—dividing features into `State`, `Action`, and `Reducer`—ensures that each piece has a clear, focused responsibility. The `State` struct holds the feature's data, the `Action` enum defines all events (user interactions or external triggers), and the reducer handles how state changes in response to these actions. This explicit separation clarifies data flow, makes the codebase more readable, and allows each part to be developed, tested, and maintained in isolation. As a result, large and complex applications built with TCA remain more manageable over time.

Developers benefit from TCA's well-defined guidelines for organizing features. They know where to place logic (in the reducer), where data resides (`State`), and how user interactions are handled (`Action`). Moreover, since state properties are immutable and can only be updated through the reducer, all state changes are traceable and predictable. For instance, actions like `fetchTrendingPodcasts` or `trendingPodcastResponse` indicate precisely when and how the state is updated—such as setting `isLoading` or changing the `podcasts`

list. This makes it easier to locate the source of issues, even in a large codebase.

Additionally, TCA's `TestStore` simplifies testing by providing a controlled environment to simulate actions and verify state transitions. This leads to early detection of edge cases, more comprehensive test coverage, and reduced risk of regressions as the codebase grows.

On the other hand, MVVM is easier to get started with but presents challenges as projects scale. In MVVM, the `ViewModel` often contains both state properties and logic, frequently tied directly to the UI via mechanisms like `@Published`. While this can be convenient for small features, it risks creating tightly coupled components that are harder to test and maintain over time. For example, in the Explore Search feature, fetching data and updating the `podcastList` property occur directly within the `ViewModel`. Without a clear division between fetching actions and state updates, it can become difficult to isolate issues as the codebase grows.

Scalability also suffers in MVVM due to the lack of a standardized structure. Developers may organize code in different ways, resulting in inconsistencies across features. Navigation is often intertwined with view logic through elements like `NavigationLink`, complicating complex navigation flows and deep linking support. As the application expands, these ad hoc patterns can make it harder to maintain a consistent, predictable, and scalable codebase.

9.5 Comparative Summary

Aspect	**TCA**	**MVVM**
Separation of Concerns	Decomposes features into `State`, `Action`, and `Reducer`, each with a clear role, enhancing modularity and clarity.	Combines state, logic, and updates in the ViewModel, potentially blurring responsibilities and increasing coupling.
Navigation	Explicitly managed in state, handled by the reducer. This facilitates predictable deep linking and complex flows.	Managed through the view (e.g., `NavigationLink`), tying navigation directly to UI elements and complicating complex flows.
Testing	The `TestStore` ensures exhaustive testing of actions and state transitions. Decoupled components simplify isolation and coverage.	Coupling within the ViewModel makes isolation harder. Tests often rely on external elements, reducing reliability and test coverage.
Maintainability & Scalability	Clear structure, immutable state, and explicit actions enable easier debugging, scalability, and consistency in large codebases.	Lack of standardization across features can cause inconsistencies and make large-scale maintenance and debugging more challenging.

Table 9.3: Comparative Summary: TCA vs. MVVM

10 CONCLUSION

This thesis explored the comparative strengths and weaknesses of MVVM and The Composable Architecture (TCA) within the context of SwiftUI-based application development. Through a detailed analysis and practical implementation, the findings highlight the transformative potential of TCA for modern app development, particularly in large-scale and complex applications.

The results show TCA's advantages in modularity, scalability, and testability. By adhering to a strict separation of concerns and leveraging components such as State, Action, and Reducer, TCA facilitates a highly structured and predictable approach to state management. This structure not only enhances code maintainability but also simplifies the implementation of features like deeplinking and complex navigation workflows. Additionally, TCA's built-in tools for state transition and side-effect testing ensure a high degree of reliability and ease in verifying app behavior.

Memory usage analysis further supports TCA's scalability. In scenarios like idle states or standard interactions, TCA consistently consumes less memory than MVVM. However, the performance gap narrows, or even reverses, in more interactive scenarios like playing episodes or searching terms. This suggests that while TCA excels in handling static or moderately dynamic content, its performance in highly interactive use cases should be an area for future optimization.

By contrast, MVVM's simplicity and reduced initial development overhead make it well-suited for smaller applications. However, the architecture's tendency to consolidate responsibilities within ViewModels can lead to tightly coupled components, hindering modularity and long-term maintainability. The embedded navigation logic and less robust testing capabilities further limit MVVM's scalability and predictability, particularly in larger projects.

In conclusion, this research demonstrates that while MVVM remains a viable choice for small-scale applications, TCA's structured design, robust state management, and comprehensive testing tools position it as the superior architecture for teams prioritizing scalability, maintainability, and performance. These findings provide insights for developers and organizations aiming to optimize their SwiftUI applications and encourage further exploration of TCA to address its current limitations in high-interaction scenarios.

11 ÖSSZEFOGLALÓ

A szakdolgozat az MVVM-et és a The Composable Architecture-t (TCA) hasonlította össze erősségeit és gyengeségeit vizsgálva SwiftUI-alapú alkalmazásfejlesztés kontextusában. Részletes elemzés és gyakorlati megvalósítás során keletkezett megállapítások rávilágítanak a TCA transzformatív potenciáljára a modern alkalmazásfejlesztésben, különösen nagyméretű és összetett alkalmazások esetében.

Az eredmények a TCA előnyeit mutatják a modularitás, a skálázhatóság és a tesztelhetőség terén. Azáltal, hogy a TCA betartja a problémák szigorú szétválasztását, és olyan komponenseket használ, mint a State, Action és Reducer, a TCA lehetővé teszi az állapotkezelés rendkívül strukturált és kiszámítható megközelítését. Ez a struktúra nem csak a kód karbantarthatóságát javítja, hanem leegyszerűsíti az olyan funkciók megvalósítását is, mint a deeplinking és az összetett navigációs munkafolyamatok. Emellett a TCA beépített eszközei az állapotátmenetek és a mellékhatások teszteléséhez nagyfokú megbízhatóságot és egyszerűséget biztosítanak az alkalmazás viselkedésének ellenőrzésében.

A memóriahasználat elemzése alátámasztja a TCA skálázhatóságát. Az olyan forgatókönyvekben, mint az inaktív állapotok vagy a standard interakciók, a TCA következetesen kevesebb memóriát fogyaszt, mint az MVVM. A teljesítménybeli különbség azonban csökken, vagy akár megfordul az interaktívabb szenáriókban, például epizódok lejátszása vagy kifejezések keresése során. Ez arra utal, hogy míg a TCA kiválóan kezeli a statikus vagy kevés dinamikus tartalmat, a teljesítménye a masszívan interaktív használati esetekben egy további optimalizálási terület.

Ezzel szemben az MVVM egyszerűsége és a csökkentett kezdeti fejlesztési költségei alkalmassá teszi kisebb alkalmazásokhoz. Az architektúra azon tendenciája azonban, hogy a ViewModel-ekben konszolidálja a felelősségeket, szorosan összekapcsolt komponensekhez vezethet, ami akadályozza a modularitást és a hosszú távú karbantarthatóságot. A beágyazott navigációs logika és a kevésbé robusztus tesztelési képességek tovább korlátozzák az MVVM skálázhatóságát és kiszámíthatóságát, különösen nagyobb projektekben.

A konklúzió a megvalósult munka során, hogy míg az MVVM továbbra is életképes választás a kis méretű alkalmazásokhoz, a TCA strukturált kialakítása, robusztus állapotkezelése és átfogó tesztelési eszközei a kiválóbb architektúrává teszik a skálázhatóságot, a karbantarthatóságot és a teljesítményt előnyben részesítő csapatok számára. Ezek az eredmények betekintést nyújtanak a SwiftUI-alkalmazások optimalizálására törekvő fejlesztők és szervezetek számára, és ösztönzik a TCA további vizsgálatát, hogy kezelje annak jelenlegi korlátait a magas interakcióval rendelkező használati esetekben.

12 REFERENCES

[1] GetStream. "Uikit vs swiftui: A detailed comparison." Accessed: 2024-09-13. (2023), [Online]. Available: https://getstream.io/blog/uikit-vs-swiftui/.

[2] AppMaster.io. "Swiftui vs uikit: Ui frameworks for ios apps." Accessed: 2024-11-22. (2024), [Online]. Available: https://appmaster.io/blog/swiftui-vs-uikit-ui-framework-for-ios-apps.

[3] M. Technolabs. "Swiftui vs uikit: Which framework should you choose?" Accessed: 2024-11-22. (2024), [Online]. Available: https://www.moontechnolabs.com/blog/swiftui-vs-uikit/.

[4] A. Developer. "Introducing swiftui: Building apps faster and easier." Accessed: 2024-09-13. (Jun. 2019), [Online]. Available: https://developer.apple.com/news/?id=06032019b.

[5] AppMakers.dev. "Swiftui vs uikit: Declarative vs imperative programming." Accessed: 2024-11-22. (2024), [Online]. Available: https://appmakers.dev/swiftui-vs-uikit-declarative-vs-imperative-programming/.

[6] V. Shkodich. "Architectures comparing for swiftui." Accessed: 2024-11-22. (2024), [Online]. Available: https://medium.com/@vladislavshkodich/architectures-comparing-for-swiftui-6351f1fb3605.

[7] H. C. Studio. "Imperative vs declarative ui: A deep dive." Accessed: 2024-11-22. (2024), [Online]. Available: https://habitualcs.io/imperative-vs-declarative-ui-a-deep-dive/.

[8] J. Sundell. "A guide to swiftui state management." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.swiftbysundell.com/articles/swiftui-state-management-guide/.

[9] D. Lupich. "The composable architecture (swift): Guide to tca." Accessed: 2024-11-22. (2024), [Online]. Available: https://medium.com/@dmitrylupich/the-composable-architecture-swift-guide-to-tca-c3bf9b2e86ef.

[10] Point-Free. "Swiftui and state management: Part 3." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/swiftui-and-state-management/ep67-swiftui-and-state-management-part-3.

[11] J. Gossman. "Introduction to model/view/viewmodel pattern for building wpf apps." Accessed: 2024-09-13. (2005), [Online]. Available: https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps.

[12] O. Tech. "Clean architecture and mvvm on ios." Accessed: 2024-09-13. (2020), [Online]. Available: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3.

[13] Microsoft. "Mvvm pattern in .net maui." Accessed: 2024-11-22. (2024), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm.

[14] Q. Studios. "Coordinator pattern in swiftui." Accessed: 2024-11-22. (2024), [Online]. Available: https://quickbirdstudios.com/blog/coordinator-pattern-in-swiftui/.

[15] M. Freddo. "Mastering mvvm: A comprehensive guide to the model-view-viewmodel architecture." Accessed: 2024-11-22. (2024), [Online]. Available: https://dev.to/mochafreddo/mastering-mvvm-a-comprehensive-guide-to-the-model-view-viewmodel-architecture-221g.

[16] G. Imbugwa, E. Ewane, H. Saliu, and M. Mazzara, *Swiftui: A case study of code quality evaluation for declarative programming*, Dec. 2020. DOI: 10.13140/RG.2.2.30949.22241.

[17] D. Indrawan, D. Kusumo, and S. Puspitasari, "Analysis of the implementation of mvvm architecture pattern on performance of ios mobile-based applications," *JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika)*, vol. 8, pp. 59–65, Feb. 2023. DOI: 10.29100/jipi.v8i1.3293.

[18] C. Zullo, "Clean ios architecture pt.6: Viper – design pattern or architecture?," 2018, Accessed: 2024-09-17.

[19] R. F. García, *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift*. Apress, 2023.

[20] R. Ferrer García, *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift*. Apress, 2023. DOI: 10.1007/978-1-4842-9069-9.

[21] N. Babayev, *Exploring swiftui: A study on modern ui frameworks*, Accessed: 2024-11-22, 2024.

[22] M. R. Andika, N. Selviandro, and G. S. Wulandari, "Understanding the impact of modularity in ios app performance using viper architecture pattern," in *2023 3rd International Conference on Intelligent Cybernetics Technology Applications (ICICyTA)*, 2023, pp. 358–363. DOI: 10.1109/ICICyTA60173.2023.10428901.

[23] R. Schmidt. "Composable architecture: My experience." Accessed: 2024-11-22. (2024), [Online]. Available: https://rodschmidt.com/posts/composable-architecture-experience/.

[24] SwiftUI. "Mastering swiftui - sample pdf." Accessed: 2024-09-13. (2024), [Online]. Available: https://swiftui.s3-us-west-2.amazonaws.com/mastering-swiftui-sample.pdf.

[25] AppCoda, *Swiftui basics: Learn swiftui for ios development*, Accessed: 2024-12-07, 2024.

[26] J. Lloyd, "Practical advantages of declarative programming," English, in *Unknown*, Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming, 1994, pp. 3–17.

[27] IONOS. "What is imperative programming?" Accessed: 2024-09-13. (2024), [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/.

[28] P. Publishing. "Learn swiftui." Accessed: 2024-09-13. (2020), [Online]. Available: https://www.packtpub.com/free-ebook/learn-swiftui/9781839215421.

[29] GeeksforGeeks, *Handling state and state management system design*, Accessed: 2024-09-17, 2023.

[30] Point-Free. "Getting started with the composable architecture." Accessed: 2024-11-22. (2024), [Online]. Available: https://pointfreeco.github.io/swift-composable-architecture/main/documentation/composablearchitecture/gettingstarted.

[31] Point-Free. "Point-free live: Dependencies stacks." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/livestreams/livestreams/ep221-point-free-live-dependencies-stacks.

[32] Point-Free. "Composable architecture frequently asked questions." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/blog/posts/141-composable-architecture-frequently-asked-questions.

[33] B. Lemley. "Observable state." Accessed: 2024-11-22. (2024), [Online]. Available: https://medium.com/@bradfordlemley/observable-state-feac950850b.

[34] Point-Free. "Reducers and stores in the composable architecture." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/reducers-and-stores.

[35] Point-Free. "Tour of the composable architecture 1.0: The basics." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/composable-architecture-1-0/ep243-tour-of-the-composable-architecture-1-0-the-basics.

[36] Point-Free. "Swift composable architecture on github." Accessed: 2024-11-22. (2024), [Online]. Available: https://github.com/pointfreeco/swift-composable-architecture.

[37] Point-Free. "Adding side effects in the composable architecture." Accessed: 2024-11-22. (2024), [Online]. Available: https://pointfreeco.github.io/swift-composable-architecture/main/tutorials/composablearchitecture/01-02-addingsideeffects/.

[38] Point-Free. "Composable architecture: Testing." Accessed: 2024-11-22. (2024), [Online]. Available: https://pointfreeco.github.io/swift-composable-architecture/main/documentation/composablearchitecture/testing/.

[39] Point-Free. "Tour of the composable architecture 1.0: Dependencies." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/composable-architecture-1-0/ep248-tour-of-the-composable-architecture-1-0-dependencies.

[40] Point-Free. "Tour of the composable architecture 1.0: Navigation." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/composable-architecture-1-0/ep245-tour-of-the-composable-architecture-1-0-navigation.

[41] ScienceDirect. "Domain modeling." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.sciencedirect.com/topics/computer-science/domain-modeling#:~:text=Domain%20Modeling%20refers%20to%20the,world%20for%20a%20software%20application..

[42] Point-Free. "Tour of the composable architecture 1.0: Correctness." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/composable-architecture-1-0/ep247-tour-of-the-composable-architecture-1-0-correctness.

[43] Point-Free. "Tour of the composable architecture 1.0: Persistence." Accessed: 2024-11-22. (2024), [Online]. Available: https://www.pointfree.co/collections/composable-architecture/composable-architecture-1-0/ep249-tour-of-the-composable-architecture-1-0-persistence.

13 LIST OF FIGURES