International Journal of Innovative Research in Engineering
Volume 4, Issue 2 (March-April 2023), PP: 173-183.
www.theijire.com
ISSN No: 2582-8746

Click To Pick - Shopping Application Using Swift

Sudhahar S[1], Hareesh T[2]
[1,2]Electronics And Instrumentation Engineering, Bannari Amman Institute of Technology, TN,India.

How to cite this paper:
Sudhahar S[1], Hareesh T[2], "Click To Pick - Shopping Application Using Swift", IJIRE-V4I02-173-183.

Abstract: *The idea was to reduce the time spent standing in the queue to purchase the products. By using this application we are able to place the order on the respective shop and go directly and pickup. Until now there are two possible ways to purchase the product, one is to go directly and purchase the product in person and another option is to purchase the product online. When we purchase the product in person, There is a possibility of standing in the queue to purchase the product. Because, nowadays the rush in the shop is becoming more and more. When we purchase through online, It takes some time to reach us. Minimum of two days is needed for the product to take a delivery. For an emergency purpose, Person wants to reach the shop for purchase. In this situation that person will face the above mentioned problems. With this application, all of us will be able to order our product through the application and visit the shop and pick up a product directly. So at this time it will get reduced.*

Key Word: *Product, Customer, Seller, Authentication*

I.INTRODUCTION

Nowadays online shopping has become more popular and most of the people choose online shopping for purchasing the products. In this there are lots who prefer the website for purchasing more products.

There are lots of advantages for the native application compared to the website like local DB storage, easy opening, low internet requirement, etc. So, I have planned to develop a native shopping application for the MacOS.

On through, There are many ways like type conversion to MacOS. I have planned to design an application using the native programming language that is swift. When we use the native technology to develop the application then, we are able to utilize many features in that domain. By developing through the native technology we are able to utilize all the threads available in that device. This will increase the efficiency of the application. So that user will work without any delay. This will enhance the user experience compared to other option to develop the application.

As of now I have developed an application with the option that the user that is the buyer can place the order in the application with the credentials. Seller is another type of user, Who is able to see the orders received. Add their product in the application.

1.1 Problem Statement

It is noticed that, When anybody places the order online it takes a minimum of two days to reach the product to home. For an emergency situation, there is no possibility to buy through online. If the person prefers offline, Person needs to visit the shop and purchase. Nowadays, Rush in the shop is becoming more and more. This will cause more delays.

With this application, the person will be able to order our product through the application and visit the shop and pick up a product directly. So at this time time will get reduced. There is no chance of standing in the queue.

1.2 Project Summary

Aim of the project is to make an application to make an order through the internet to reduce the time wastage in standing in the queue. Developing an application in the native environment to utilize the features of that Environment. Application is designed in the MacOS Environment using Swift programming language.

1.3 Purpose

We observed the few limitations in Existing system, such as

- In online order takes time to delivery
- Queue for shopping in person is very high
- There is no shopping in MacOS Domain

Purpose of my application is to overcome all the limitations including following features

- Direct purchase
- Saving of Time
- Increase the efficiency of application in native environment

II.LITERATURE REVIEW

Swift shopping in smart cities - S. Gowtham Raj, J. Rajeevrathan, S. Senthil Kumar, V. Rajaramanan. If a person enters a mall without a smart phone, he would be provided with one. The device is inbuilt with a QR code scanner application.

The person collects the item and reads the QR code using the device. After collecting all the items the bill will be generated and it will be transferred to the cashier through the Internet which is provided by the shopping mall. The customer only pays the cash on the counter. It is the IOS application developed only for billing work. Based on this backEnd work, I will use it to generate a bill for a purchased product.

Design And Development of a Multi Featured iOS Mobile Application using Swift 3 - Pankaj Kumar Sharma, Ravi Shankar Sharma. With the help of Swift 3, How to develop a effective IOS application. How to make and utilize the tools in swift. Based on the this knowledge, How to create an effective application and How to implement all the features are learned.

Augmented Reality Shopping Framework - S. Gowtham Raj, J. Rajeevrathan, S. Senthil Kumar, V. Rajaramanan. The application is developed to get the feedback of the shopping experience. Based on this paper, It planned to create the ratting of the product to help the other user to get the details about th products.

Online Shopping - An Overview - Edwin Gnanadhas. This article shows, How to develop an online shopping application. Based on the review of this papers, Got some ideas to develop the application. This is the common article to show how to develop the shopping application effectively.

III.TECHNOLOGY USED

3.1 Swift

Swift is an extremely powerful and user-friendly programming language for iOS, iPadOS, macOS, tvOS, and watch OS. Swift code is interactive and enjoyable to write, the syntax is compact yet expressive, and Swift offers current features that developers adore. Swift code is designed to be secure and provides software that runs at breakneck speed.

The Swift project's purpose is to provide the best available language for a wide range of applications, including system programming, mobile and desktop apps, and cloud services. Most significantly, Swift is intended to make it easier for developers to write and maintain proper applications. To accomplish this, we believe that the most apparent approach to create Swift code should also be:

Safe: The most obvious technique of writing code should also be safe. Undefined behavior is the enemy of safety, and developer errors should be identified before software is released to the public. Swift may appear severe at times when choosing safety, but we think that clarity saves time in the long run.

Fast: Swift is expected as a substitution for C-based dialects (C, C++, and Objective-C). Thusly, Swift should be practically identical to those dialects in execution for most assignments. Execution should likewise be unsurprising and reliable, not simply Swift in short blasts that require tidy up later. There are loads of dialects with novel elements — it is uncommon to be Swift.

Expressive: swift advantages from many years of headway in software engineering to offer grammar that is a delight to use, with current highlights designers anticipate. In any case, swift is rarely finished. We will screen language headways and embrace what works, constantly advancing to improve Swift even.

Devices are a basic piece of the Swift biological system. We endeavor to coordinate well inside a designer's toolset, to construct rapidly, to introduce incredible diagnostics, and to empower intuitive improvement encounters. Devices can make programming a lot more impressive, similar to Swift based jungle gyms do in Xcode, or an electronic REPL can while working with Linux server-side code.

Swift incorporates highlights that make code more straightforward to peruse and compose, while giving the designer the control required in a genuine frameworks programming language. Swift backings surmised types to make code cleaner and less inclined to errors, and modules take out headers and give namespaces. Memory is overseen naturally, and you don't have to type semi-colons. Swift likewise gets from different dialects, for example named boundaries presented from Objective-C are communicated in a spotless punctuation that makes APIs in Swift simple to peruse and keep up with.

The elements of Swift are intended to cooperate to make a language that is strong, yet enjoyable to utilize. A few extra highlights of Swift include:

- Closures brought together with capability pointers
- Tuples used for multiple return values
- Generics
- Swift and brief cycle over a range or collection
- Methods, extensions, and protocols are available in structs
- Strong implicit error handling available
- High level control stream with do, guard, defer, and repeat keyword

Security: Swift was planned from the start to be more secure than C-based dialects, and kills whole classes of hazardous code. Factors are constantly introduced before use, exhibits and numbers are checked for flood, and memory is overseen naturally. Language structure is tuned to make it simple to characterize your purpose — for instance, straightforward three-character catchphrases characterize a variable (var) or constant (let).

Another wellbeing highlight is that of course swift items can never be nothing, and attempting to make or utilize a nothing object results in a gather time mistake. This makes composing code a lot of cleaner and more secure, and forestalls a typical reason for runtime crashes. Nonetheless, there are situations where nothing is proper, and for these circumstances Swift has an inventive element known as optionals. A discretionary may contain nothing, however Swift linguistic structure compels you to securely manage it utilizing ? to show to the compiler you comprehend the way of behaving and will deal with it securely.

One of the most thrilling parts of creating Swift in the open is realizing that it is presently allowed to be ported across a large number of stages, gadgets, and use cases.

We want to give source similarity to Swift across all stages, despite the fact that the genuine execution systems might vary starting with one stage then onto the next. The essential model is that the Apple stages incorporate the Objective-C runtime, which is expected to get to Apple stage systems like UIKit and AppKit. On different stages, for example, Linux, no Objective-C runtime is available, on the grounds that it isn't required.

The Swift center libraries project expects to broaden the cross-stage abilities of Swift by giving versatile executions of key Apple systems (like Establishment) without conditions on the Objective-C runtime. Albeit the center libraries are in a beginning phase of advancement, they will ultimately give further developed source similarity to Swift code across all stages.

3.1.1: Apple Platforms

Open-source Swift can be utilized on the Macintosh to focus on all of the Apple platforms: iOS, macOS, watchOS, and tvOS. Besides, twofold forms of open-source Swift coordinate with the Xcode designer apparatuses, including total help for the Xcode construct framework, code fulfillment in the manager, and incorporated troubleshooting, permitting anybody to explore different avenues regarding the most recent Swift improvements in a natural Cocoa and Cocoa Contact advancement climate.

3.1.2: Linux

pen-source Quick can be utilized on Linux to construct Quick libraries and applications. The open-source twofold forms give the Quick compiler and standard library, Quick REPL and debugger (LLDB), and the center libraries, so one can bounce right in to Quick turn of events.

3.1.3: Windows

Open source Quick can be utilized on Windows to assemble Quick libraries and applications. The open source double forms give C/C++/Quick tool chains, the standard library, and debugger (LLDB), as well as the center libraries, so one can hop right in to Quick turn of events. Source Kit-LSP is packaged into the deliveries to empower engineers to be rapidly useful with their preferred IDE.

3.1.4: New Platforms

We can hardly stand by to see the new spots we can unite Quick —. We genuinely accept that this language that we love can make programming more secure, quicker, and simpler to keep up with. We'd adore your assistance to carry Quick to considerably additional figuring stages.

Fig 3.1: Swift Programming language

3.2 SQ Lite

SQLite is an in-process library that executes an independent, serverless, zero-setup, conditional SQL data set motor. The code for SQLite is in the public space and is subsequently free for use for any reason, business or private. SQLite is the most broadly sent data set on the planet with additional applications than we can count, including a few high-profile projects.

SQLite is an installed SQL information base motor. Dissimilar to most other SQL data sets, SQLite doesn't have a different server process. SQLite peruses and composes straightforwardly to normal circle records. A total SQL data set with numerous tables, records, triggers, and perspectives, is contained in a solitary circle document. The data set document design is cross-stage - you can uninhibitedly duplicate an information base between 32-bit and 64-digit frameworks or between enormous endian and little-endian structures. These elements go with SQLite a well known decision as an Application Record Organization. SQLite information base documents are a suggested stockpiling design by the US Library of Congress. Consider SQLite not as a substitution for Prophet but rather as a trade for f open ()

SQLite is a conservative library. With all highlights empowered, the library size can be under 750KB, contingent upon the objective stage and compiler improvement settings. (64-cycle code is bigger. Furthermore, some compiler enhancements, for example, forceful capability inlining and circle unrolling can cause the item code to be a lot bigger.) There is a tradeoff between memory use and speed. SQLite by and large runs quicker the more memory you give it. By and by, execution is generally very great even in low-memory conditions. Contingent upon the way things are utilized, SQLite can be quicker than direct file system I/O.

SQLite is painstakingly tried preceding each delivery and has gained notoriety for being truly dependable. The greater part of the SQ Lite source code is given absolutely to testing and confirmation. A mechanized test suite runs a great many experiments including a huge number of individual SQL proclamations and accomplishes 100 percent branch test inclusion. SQLite answers smoothly to memory distribution disappointments and circle I/O blunders. Exchanges are Corrosive regardless of whether hindered by framework crashes or power disappointments. This is all checked by the mechanized tests utilizing unique test saddles which reenact framework disappointments. Obviously, even with this testing, there are still bugs. Yet, in contrast to a few comparative ventures (particularly business contenders) SQLite is transparent pretty much all bugs and gives bugs records and moment by-minute orders of code changes.

The SQLite code base is upheld by a worldwide group of engineers who work on SQLite full-time. The engineers keep on growing the abilities of SQLite and upgrade its unwavering quality and execution while keeping up with in reverse similarity with the distributed connection point spec, SQL sentence structure, and data set record design. The source code is totally free to anyone who needs it, yet proficient help is additionally accessible.

The SQLite project was begun on 2000-05-09. What's to come is in every case hard to foresee, yet the aim of the engineers is to help SQLite during that time 2050. Plan choices are made in light of that goal.

We the engineers trust that you find SQLite valuable and we implore you to utilize it well: to make great and lovely items that are quick, dependable, and easy to utilize. Look for grace for yourself as you excuse others. Furthermore, similarly as you have gotten SQLite for nothing, so likewise uninhibitedly give, paying the obligation forward.

Fig 3.2: SQLite

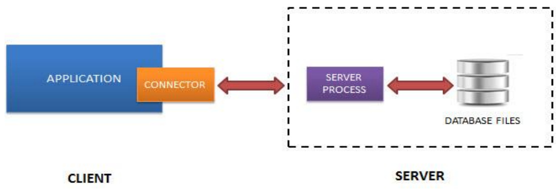


Fig 3.3: MySQLite SQL Request and Response

3.3 Clean Architecture

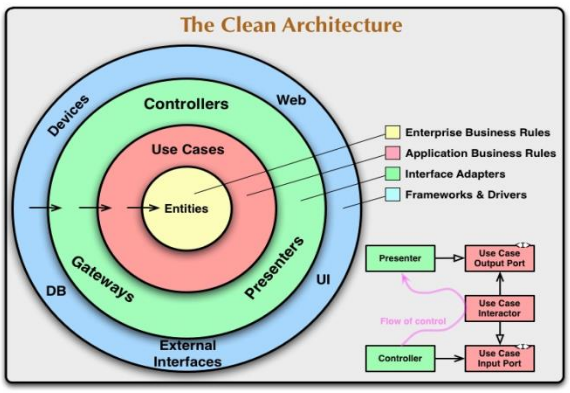


Fig 3.4: Clean Architecture

Advantages of clean Architecture

1. Independent of Structures. The design doesn't rely upon the presence of some library of element loaded programming. This permits you to involve such structures as instruments, instead of packing your framework into their restricted limitations.
2. Testable. The business rules can be tried without the UI, Data set, Web Server, or some other outside component.
3. Independent of UI. The UI can change effectively, without changing the remainder of the framework. An Internet UI could be supplanted with a control center UI, for instance, without changing the business rules.
4. Independent of Database. You can trade out Prophet or SQL Waiter, for Mongo, BigTable, CouchDB, or something different. Your business rules are not bound to the data set.
5. Independent of any external agency. As a matter of fact your business essentially knows nothing by any means about the rest of the world.

The diagram 3.4 is an attempt at integrating all these architectures into a single actionable idea.

The Reliance Rule*:* The concentric circles address various areas of programming. By and large, the further in you go, the more elevated level the product becomes. The external circles are instruments. The inward circles are strategies.

The superseding decide that makes this engineering work is The Reliance Rule. This standard says that source code conditions can point inwards. Nothing in an inward circle can know anything by any stretch of the imagination about something in an external circle. Specifically, the name of something proclaimed in an external circle should not be referenced by the code in the an internal circle. That incorporates, capabilities, classes. factors, or some other named programming substance.

All the same, information designs utilized in an external circle ought not be utilized by an inward circle, particularly in the event that those configurations are produce by a structure in an external circle. We believe nothing in an external circle should affect the internal circles.

The Reliance Rule: The concentric circles address various areas of programming. As a rule, the further in you go, the more significant level the product becomes. The external circles are components. The internal circles are approaches.

The abrogating decide that makes this design work is The Reliance Rule. This standard says that source code conditions can point inwards. Nothing in an internal circle can know anything by any stretch of the imagination about something in an external circle. Specifically, the name of something pronounced in an external circle should not be referenced by the code in the an internal circle. That incorporates, capabilities, classes. factors, or some other named programming element.

All the while, information designs utilized in an external circle ought not be utilized by an internal circle, particularly in the event that those organizations are produce by a system in an external circle. We believe nothing in an external circle should affect the internal circles.

Use Cases: The product in this layer contains application explicit business rules. It exemplifies and carries out all of the utilization instances of the framework. These utilization cases arrange the progression of information to and from the elements, and direct those substances to utilize their endeavor complete business rules to accomplish the objectives of the utilization case.

We don't anticipate that adjustments of this layer should influence the substances. We likewise don't anticipate that this layer should be impacted by changes to externalities like the information base, the UI, or any of the normal systems. This layer is disconnected from such worries.

We do, notwithstanding, anticipate that that changes should the activity of the application will influence the utilization cases and thusly the product in this layer. In the event that the subtleties of a utilization case change, some code in this layer will surely be impacted.

Interface Adapters: The product in this layer is a bunch of connectors that convert information from the configuration generally helpful for the utilization cases and substances, to the organization generally helpful for some outside organization like the Data set or the Internet. It is this layer, for instance, that will completely contain the MVC design of a GUI. The Moderators, Perspectives, and Regulators all have a place in here. The models are logical just information structures that are passed from the regulators to the utilization cases, and afterward back from the utilization cases to the moderators and perspectives.

Likewise, information is changed over, in this layer, from the structure generally helpful for substances and use cases, into the structure generally advantageous for anything that diligence system is being utilized. for example The Information base. No code internal of this circle ought to know anything by any stretch of the imagination about the data set. In the event that the data set is a SQL data set, all the SQL ought to be limited to this layer, and specifically to the pieces of this layer that have to do with the data set.

Additionally in this layer is some other connector important to change over information from some outer structure, like an outside assistance, to the inward structure utilized by the utilization cases and elements.

Frameworks and Drivers: The furthest layer is for the most part made out of structures and apparatuses like the Information base, the Internet System, and so on. For the most part you don't compose a lot of code in this layer other than stick code that imparts to the following circle inwards.

This layer is where every one of the subtleties go. The Internet is a detail. The information base is a detail. We keep these things outwardly where they can cause little damage.

Just Four Circles? No, the circles are schematic. You might find that you really want something beyond these four. There's no standard that says you should constantly have quite recently these four. Notwithstanding, The Reliance Rule generally applies. Source code conditions generally point inwards. As you move inwards the degree of reflection increments. The peripheral circle is low level substantial detail. As you move inwards the product develops more unique, and typifies more significant level strategies. The internal most circle is the most broad.

Crossing boundaries: At the lower right of the graph is an illustration of how we cross the circle limits. It shows the Regulators and Moderators speaking with the Utilization Cases in the following layer. Note the progression of control. It starts in the regulator, travels through the utilization case, and afterward ends up executing in the moderator. Note likewise the source code conditions. Every last one of them focuses inwards towards the utilization cases.

We as a rule settle this obvious inconsistency by utilizing the Reliance Reversal Standard. In a language like Java, for instance, we would orchestrate connection points and legacy connections with the end goal that the source code conditions go against the progression of control at the perfect focuses across the limit.

For instance, consider that the utilization case necessities to call the moderator. In any case, this call should not be immediate on the grounds that that would abuse The Reliance Rule: No name in an external circle can be referenced by an inward circle. So we have the utilization case call a connection point (Displayed here as Use Case Result Port) in the inward circle, and have the moderator in the external circle carry out it.

A similar strategy is utilized to cross every one of the limits in the models. We exploit dynamic polymorphism to make source code conditions that go against the progression of control so we can adjust to The Reliance Rule regardless of what heading the progression of control is heading down.

What crosses crosses the boundaries: Ordinarily the information that crosses the limits is basic information structures. You can utilize fundamental structs or basic Information Move objects on the off chance that you like. Or on the other hand the information can basically be contentions in capability calls. Or on the other hand you can pack it into a has hmap, or develop it into an article. Significantly, disconnected, straightforward, information structures are passed across the limits. We would rather not cheat and pass Substances or Information base columns. We don't need the information designs to have any sort of reliance that abuses The Reliance Rule.

For instance, numerous data set systems return a helpful information design in light of a question. We could call this a Row Structure. We would rather not pass that line structure inwards across a limit. That would disregard The Reliance Rule since it would compel an inward circle to know something about an external circle.
So when we pass information across a limit, consistently in the structure is generally helpful for the internal circle.

Summary: Adjusting to these straightforward standards isn't hard, and will save you a ton of cerebral pains proceeding. By isolating the product into layers, and adjusting to The Reliance Rule, you will make a framework that is inherently testable, with every one of the advantages that suggests. At the point when any of the outer pieces of the framework become old, similar to the information base, or the web structure, you can supplant those out of date components with at least fight.

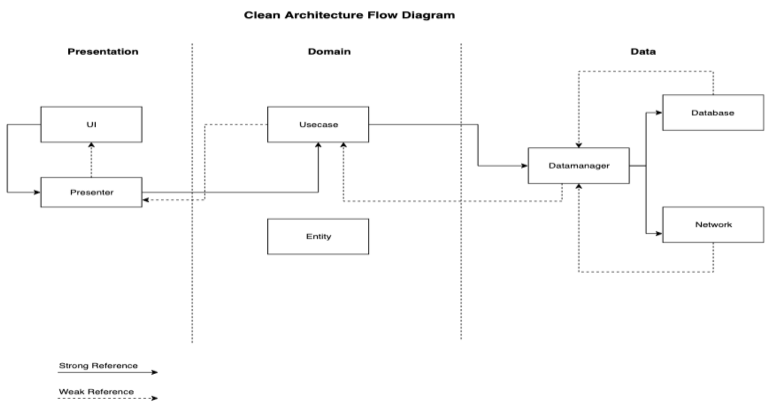


Fig 3.5: Three-Layers

3.4 Swift Ui

SwiftUI is a UI toolbox that allows us to plan applications in a definitive manner. That is an extravagant approach to saying that we let SwiftUI know how we believe our UI should look and work, and it sorts out some way to get that going as the client interfaces with it.

Definitive UI is best perceived in contrast with basic UI, which is the thing iOS designers were doing before iOS 13. In a basic UI we could cause a capability to be called when a button was clicked, and inside the capability we'd peruse a worth and show a mark - we consistently change the manner in which the UI looks and works in light of what's going on.

Basic UI leads to a wide range of issues, the majority of which spin around state, which is another extravagant term signifying "values we store in our code." We really want to follow what express our code is in, and ensure our UI accurately mirrors that state.

Assuming we have one screen with one Boolean property that influences the UI, we have two expresses: the Boolean may be on or off. On the off chance that we have two Booleans, An and B, we presently have four states:

An is off and B is off
An is on and B is off
An is off and B is on
An is on and B is on

Furthermore, in the event that we have three Booleans? Or on the other hand five? Or on the other hand numbers, strings, dates, and that's just the beginning? Indeed, then, at that point, we have parts greater intricacy.

Assuming you've at any point utilized an application that says you have 1 uninitiated message regardless of how frequently you attempt to let know if you've perused the darn thing, that is a state issue - that is a basic UI issue.

Interestingly, definitive UI allows us to educate iOS concerning all potential conditions of our application on the double. We could say in the event that we're signed in show a welcome message yet in the event that we're logged out show a login button. We don't have to compose code to move between those two states manually - that is the monstrous, basic approach to working!

All things being equal, we let Swift UI move between UI designs for us when the state changes. We previously told it what to show in light of whether the client was signed in or out, so when we change the confirmation state Swift UI can refresh the UI for our benefit.

That is the thing it implies by explanatory: we aren't making Swift UI parts show and stow away the hard way, we're simply telling it every one of the guidelines we believe it should adhere to and passing on Swift UI to ensure those guidelines are upheld.

Yet, Swift UI doesn't stop there - it likewise goes about as a cross-stage UI layer that works across iOS, macOS, tvOS, and even watch OS. This implies you can now learn one language and one design structure, then, at that point, convey your code anyplace.

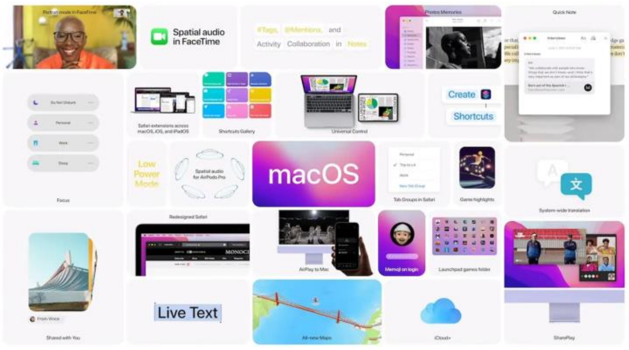


Fig 3.6: Swift UI

IV.METHODOLOGY PROPOSED AND WORKING

4.1 Project Planning and Scheduling

Our project is created utilizing explicit programming advancement lifecycle. Programming improvement approach is the most appropriate for the task relies upon the prerequisite and different elements. An interaction model is an improvement technique that is utilized to accomplish an objective that fulfills the necessities keeping the constraints.

4.1.1Iterative Water Flow Model

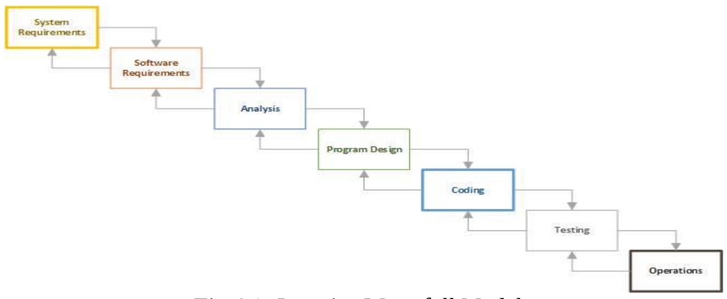


Fig 4.1: Iterative Waterfall Model

4.2 Backend Design

To develop the application, I have planned to use swift Programming language. To use that I have used the Xcode which is native for the MacOS and IOS. Planned to implement all Learned concepts such as Object Oriented Programming (OOPs), Inheritance, Multithreading, protocols, etc.

4.2.1Class Diagram Design

I have roughly developed the class diagram for the application. By using this class diagram only we are able to develop the whole backend of the application using the swift programming language. Class Diagram consists of Class names which I am going to use in the application, properties and methods in that class are also mentioned in the class diagram.

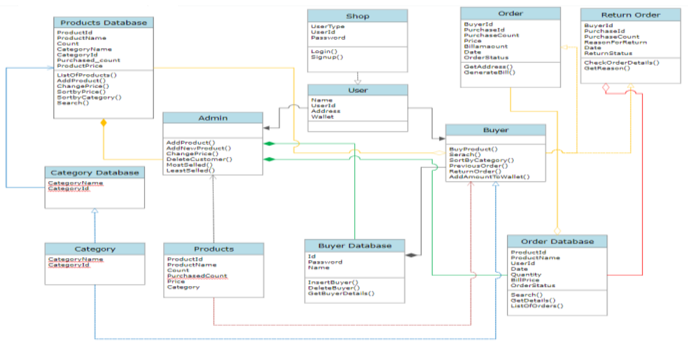


Fig 4.2: Class Diagram

4.2.2 Database Diagram Design

To use a database, We first need to know the column name which we are going to use in the database. And we also need to know the type of data stored in the column of the database. By using the SQL query in SQLite we are able to create the table in the SQLite database. In that table only I am going to save the details of the customer for login and other details such as address, mobile number etc. Details of the user both seller and buyer, are saved in the same table in the name of SELLER and BUYER.

All the orders received are available in the ORDER TABLE. PRODUCT table contain all the details of the product.

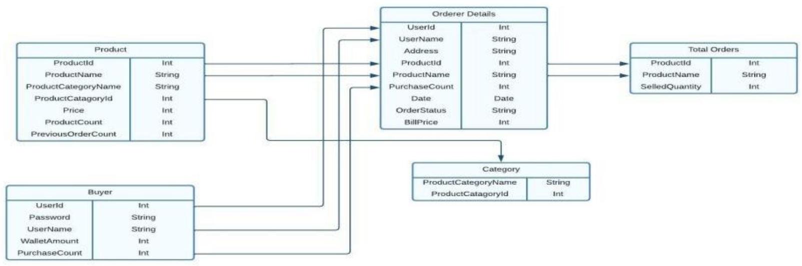


Fig 4.3: Database Diagram

4.2.3 E-R Diagram

An element relationship diagram (ERD) is a graphical portrayal of a data framework th11at shows the connection between individuals, objects, spots, ideas or occasions inside that framework. An ERD is an information demonstrating strategy that can assist with characterizing business processes and can be utilized as the establishment for a connection data set.

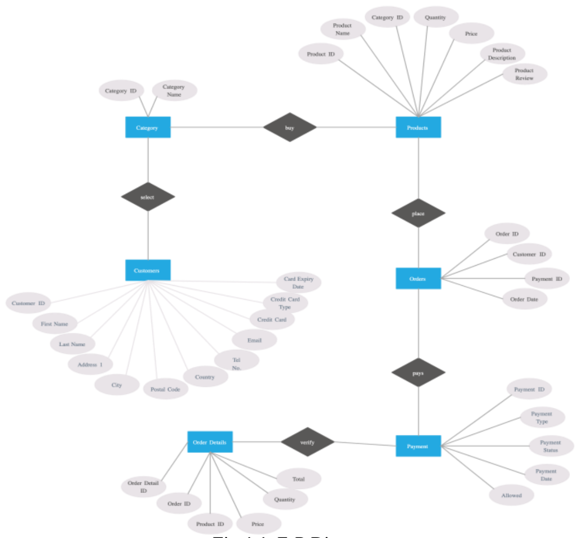
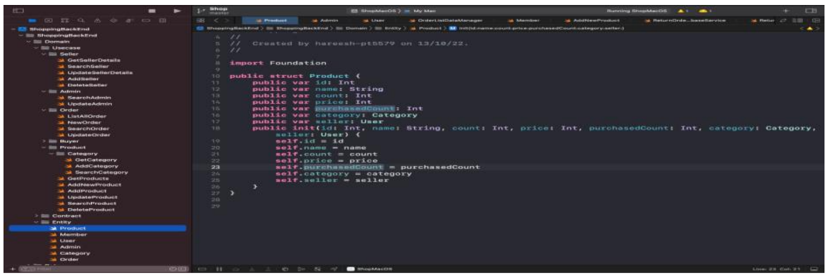


Fig 4.4: E-R Diagram

4.2.4 Code Completion

Based on the class diagram design, Started developing the backend work. Backend work is done using the swift code. All the backend code is written in the Xcode Software. Which is open source software for beginners in MacOS. For Every use case such as login, login check, etc separate class and separate file is created in the Xcode. application is divided into the three layers, such as UI, Domain, Database. All the usecase files are separated as a group. Because, It will increase the readability. Debugging also becomes easy when we do this. So that, swift code for backend is ready.

For the front end development, We are going to use the Swift UI. In this there is a special framework for the MacOS that is Appkit. By utilizing all features in the Appkit, We are going to develop the front End. For the database access, a Special file is written in the application and connection code also written in this file. Only one file will access the database. For every query, Separate common function is written. That function is called and query is gets executed.



```swift
//
//  Created by hareesh-pt5579 on 13/10/22.
//

import Foundation

public struct Product {
    public var id: Int
    public var name: String
    public var count: Int
    public var price: Int
    public var purchasedCount: Int
    public var category: Category
    public var seller: User
    public init(id: Int, name: String, count: Int, price: Int, purchasedCount: Int, category: Category,
        seller: User) {
        self.id = id
        self.name = name
        self.count = count
        self.price = price
        self.purchasedCount = purchasedCount
        self.category = category
        self.seller = seller
    }
}
```

Fig 4.5: Xcode Image

4.2.5 Database Table Creation

Based on the Database diagram, Started the creation of tables in SQLite. DB browser for SQLite is used as a workbench for working in the application. Created the table in the workbench by using the CREATE TABLE query. In the workbench we are able to see the created table visually. And also we are able to see the values of the table visually by opening it. So it is much easier to manage.

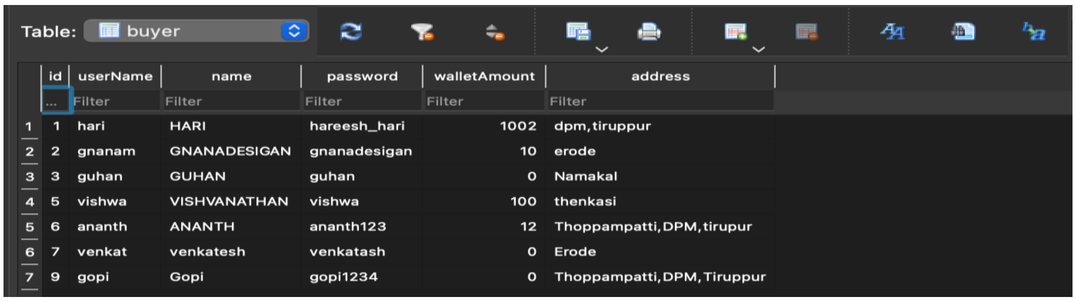


Table: buyer

	id	userName	name	password	walletAmount	address
1	1	hari	HARI	hareesh_hari	1002	dpm,tiruppur
2	2	gnanam	GNANADESIGAN	gnanadesigan	10	erode
3	3	guhan	GUHAN	guhan	0	Namakal
4	5	vishwa	VISHVANATHAN	vishwa	100	thenkasi
5	6	ananth	ANANTH	ananth123	12	Thoppampatti,DPM,tirupur
6	7	venkat	venkatesh	venkatash	0	Erode
7	9	gopi	Gopi	gopi1234	0	Thoppampatti,DPM,Tiruppur

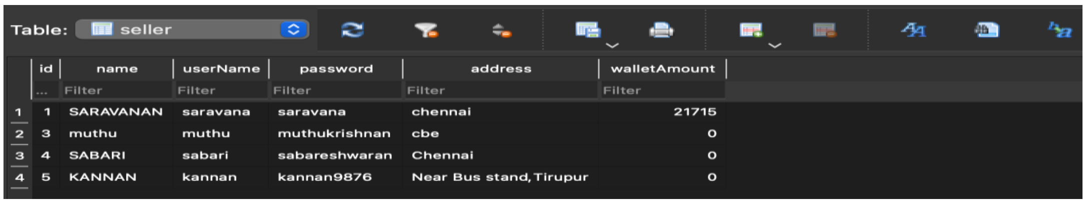


Table: seller

	id	name	userName	password	address	walletAmount
1	1	SARAVANAN	saravana	saravana	chennai	21715
2	3	muthu	muthu	muthukrishnan	cbe	0
3	4	SABARI	sabari	sabareshwaran	Chennai	0
4	5	KANNAN	kannan	kannan9876	Near Bus stand,Tirupur	0

Table 4.1: User table

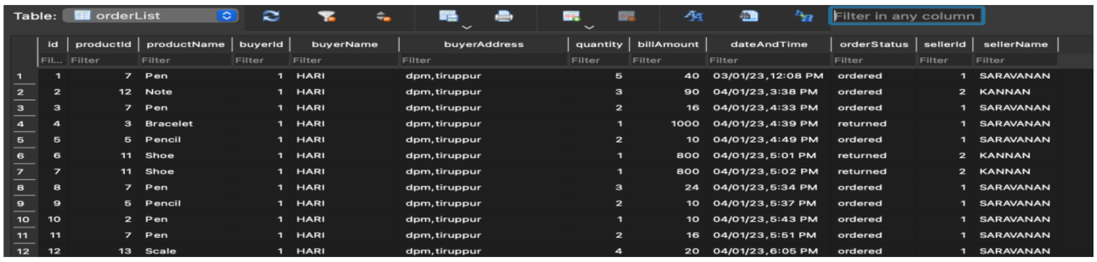


Table: orderList

	id	productId	productName	buyerId	buyerName	buyerAddress	quantity	billAmount	dateAndTime	orderStatus	sellerId	sellerName
1	1	7	Pen	1	HARI	dpm,tiruppur	5	40	03/01/23,12:08 PM	ordered	1	SARAVANAN
2	2	12	Note	1	HARI	dpm,tiruppur	3	90	04/01/23,3:38 PM	ordered	2	KANNAN
3	3	7	Pen	1	HARI	dpm,tiruppur	2	16	04/01/23,4:33 PM	ordered	1	SARAVANAN
4	4	3	Bracelet	1	HARI	dpm,tiruppur	1	1000	04/01/23,4:39 PM	returned	1	SARAVANAN
5	5	5	Pencil	1	HARI	dpm,tiruppur	2	10	04/01/23,4:49 PM	ordered	1	SARAVANAN
6	6	11	Shoe	1	HARI	dpm,tiruppur	1	800	04/01/23,5:01 PM	returned	2	KANNAN
7	7	11	Shoe	1	HARI	dpm,tiruppur	1	800	04/01/23,5:02 PM	returned	2	KANNAN
8	8	7	Pen	1	HARI	dpm,tiruppur	3	24	04/01/23,5:34 PM	ordered	1	SARAVANAN
9	9	5	Pencil	1	HARI	dpm,tiruppur	2	10	04/01/23,5:37 PM	ordered	1	SARAVANAN
10	10	2	Pen	1	HARI	dpm,tiruppur	1	10	04/01/23,5:43 PM	ordered	1	SARAVANAN
11	11	7	Pen	1	HARI	dpm,tiruppur	2	16	04/01/23,5:51 PM	ordered	1	SARAVANAN
12	12	13	Scale	1	HARI	dpm,tiruppur	4	20	04/01/23,6:05 PM	ordered	1	SARAVANAN

Table 4.2: Order table

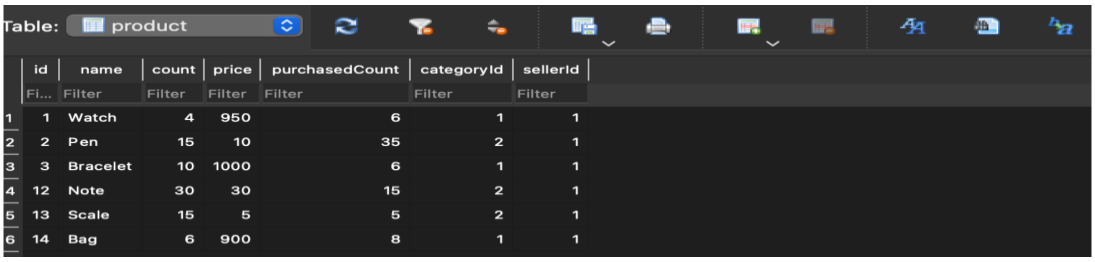


Table: product

	id	name	count	price	purchasedCount	categoryId	sellerId
1	1	Watch	4	950	6	1	1
2	2	Pen	15	10	35	2	1
3	3	Bracelet	10	1000	6	1	1
4	12	Note	30	30	15	2	1
5	13	Scale	15	5	5	2	1
6	14	Bag	6	900	8	1	1

Table 4.3: Product table

4.3 Front End Design

To Develop a Front end, Swift UI in the Appkit framework is used.

In the Starting page, we get the input such as user name and password for login. If we are a new user, We have an option Create new account. By using this users are able to create new account.

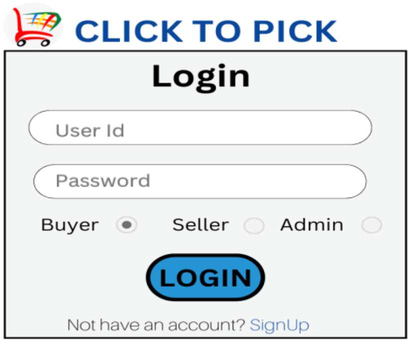


Fig 4.6: Front page

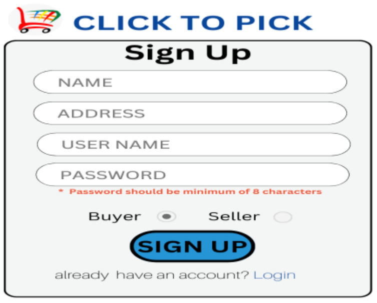
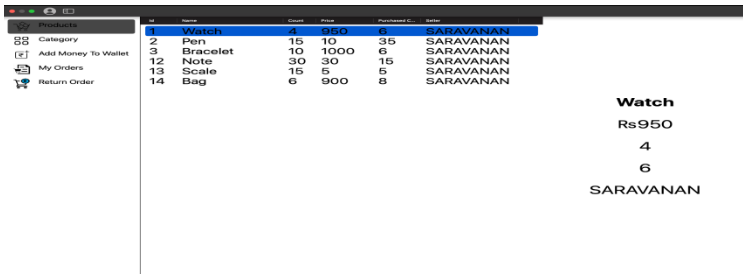

CLICK TO PICK
Sign Up
NAME
ADDRESS
USER NAME
PASSWORD
* Password should be minimum of 8 characters
Buyer
Seller
SIGN UP
already have an account? Login

Fig 4.7: Create new account page

The remaining Front end is the UI part of the application in the developing stage. That will be developed by a separate team. After that Full UI design, We will able to connect the front end with back to run as a full application.

Buyer page design sample is shown in the figure 4.8.



Id	Name	Count	Price	Purchased C...	Seller
1	Watch	4	950	6	SARAVANAN
2	Pen	15	10	35	SARAVANAN
3	Bracelet	10	1000	6	SARAVANAN
12	Note	30	30	15	SARAVANAN
13	Scale	15	5	5	SARAVANAN
14	Bag	6	900	8	SARAVANAN

Watch
Rs950
4
6
SARAVANAN

Fig 4.8: Buyer Page Design

4.4 System Requirement

4.3.1 Hardware Requirement

Laptop or PC

- MacOS with minimum of 6 core
- 100 GB ROM or higher

4.3.2 Software Requirement

Laptop or PC

- Xcode
- SQLite for database

V.RESULT AND DISCUSSION

By using my application we are able to order the products needed through the application. User can visit the shop and pick up the products we order. So this saves the time of visiting in person and waiting until packing and getting the products. In online shopping, When we place an order, It takes a minimum of two days to reach us. For emergency purposes, Online shopping is not feasible. So Direct buying from the shop is good. To avoid the crowd, buyer utilize my application.

There is no shopping application available in the MacOS App store. So, This application is developed to run on MacOS. We will use the same back end code to run in IOS platform.

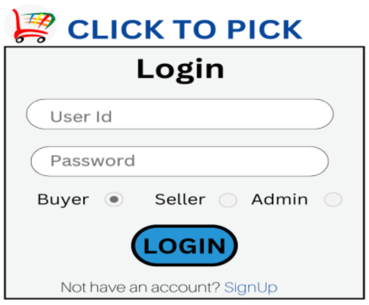


Fig 5.1: Sign in page

VI.CONCLUSION

The application created will run on the MacOS domain. So In this application, we can easily purchase the product, Which we need. Then, We will visit the shop and pick up the product which we ordered.

This application saves the time spent on searching and packing the products in the shop. And also saves the delivery time when we order through online like amazon, Flipkart, etc.

Buyer are able to see the nearest shop products. So we can easily pick up the order in which we are placed.

All details are stored in the SQLite Database, So it will be secure and Buyer is able to access any time in an easier manner.

References

1. *Marcel Rebouças, Gustavo Pinto, Felipe Ebert, Weslley Torres, Alexander Serebrenik, Fernando Castor (2016) "An Empirical Study on the Usage of the Swift Programming Language" , Software Analysis, Evolution, and Reengineering (SANER).*
2. *Barua A, Thomas S W and Hassan A E (2014) "What are developers talking about? an analysis of topics and trends in stack overflow", Empirical Softw. Engg., vol. 19, no. 3, pp. 619-654.*
3. *Chandra S S and Chandra K (2005) "A comparison of Java and C#", J. Comput. Sci. Coll., vol. 20, no. 3, pp. 238-254.*
4. *Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood (2005) "Building Customized Program Analysis Tools with Dynamic Instrumentation", Sigplan Not. 40, 190--200.*
5. *Hadjerrouit S (1998) "Java as first programming language: A critical evaluation", SIGCSE Bull, vol. 30, no. 2, pp. 43-47.*
6. *Malte Kraus and Vincent Haupert (2018) "The Swift Language from a Reverse Engineering Perspective", Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium.*
7. *Pinto G, Castor F and Liu Y D (2014) "Mining questions about software energy consumption", MSR, pp. 22-31*
8. *Pramila Kole, Nikita Jagtap, Kamlesh Kawade, Deveshree Wankhede (2022) "Smart Shopping System using RFID" .*
9. *Pratt T W and Zelkowitz M V (2000) "Programming Languages: Design and Implementation", Prentice Hall PTR*
10. *Porter M F (1997) "Readings in information retrieval. chapter An Algorithm for Suffix Stripping", Morgan Kaufmann, pp. 313-316.*