iOS Application Architecture

Srdan Stanic
srdan@hey.com
@srstanic

About this whitepaper

If you are interested in how to architect long-lasting iOS applications, you are in the right place. This whitepaper goes beyond the narrow architectural patterns like MVC, MVVM, and VIPER. It provides an in-depth analysis of what kinds of responsibilities are present in an iOS application. It categorizes these responsibilities and provides a mental model for structuring them into building blocks of a well-architected iOS application. By reading this whitepaper, you will develop a thinking framework required to architect applications that will support many years of development and maintenance.

About the author

Hey, my name is Srdan. I'm an iOS/Swift engineer with diverse software engineering and team leadership experience. Since I've professionally started programming in 2006, I've done projects for government, enterprise, and startups. The kinds of software I've built include backend systems, single-page front-end apps, hybrid, and native mobile apps. I have led teams as a technical team lead, a project manager, a product manager, and a CTO. I spend most of my time promoting clean code design, agile practices, and improving leadership skills in engineering teams.

Table of contents

Introduction

The goal of this whitepaper is to help iOS developers design readable and flexible code. These two characteristics of code design are crucial because almost every software is destined to be changed over time, whether it's because new features are introduced, existing ones are updated, or the environment the software is running in is changed. Whatever the reason for a change is, our job as software engineers is to spend a minimal amount of time implementing a change, while keeping the software working as expected, and designing these changes so that future changes can be implemented in an equally efficient manner.

Although the target audience for this whitepaper is iOS developers, the first chapter outlines the thinking framework for building software in general, while the later chapter goes into details and specifics of the iOS applications.

One reason for this approach is that the best software design practices apply to the iOS platform as much as any other platform.

The other reason is that the iOS application is often just one component within a larger application spanning across different platforms. It might be an application running in the Apple ecosystem consisting of a Mac app, an iOS app, and an iWatch app, all sharing the data through Apple's back-end services. Or it might be an application spanning across different ecosystems and consisting of a custom back-end API, a web app, an iOS app, and an Android app. Whatever the case, as iOS developers, we often need to integrate with other system components on different platforms. Being aware of which parts of the logic should be specific to the iOS application and which should be delegated to the back end or shared across platforms is an important aspect of code design. Note that in the context of this whitepaper, sharing logic across platforms doesn't necessarily mean the same code being executed on different platforms. It can be the case, but sharing logic in this context also includes having two or more separate implementations of the same logic in different programming languages.

This whitepaper is primarily written with Swift programming language in mind, but the main principles apply to Objective-C as well.

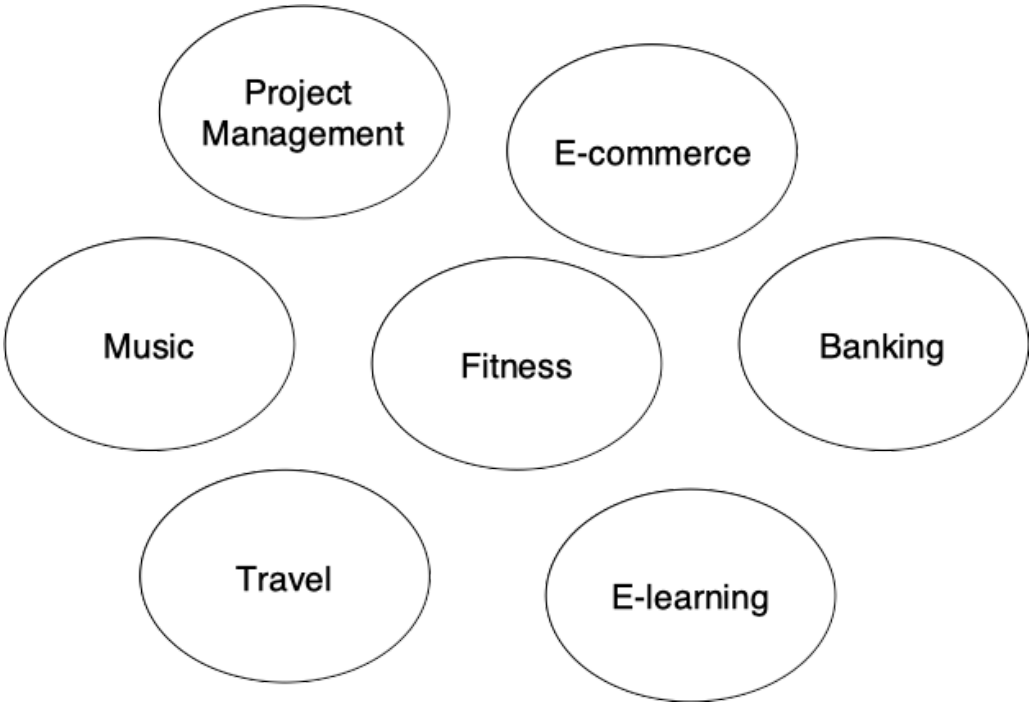
What are the main building blocks of application software?

An application software generally consists of three types of logic:

- domain logic
- application logic
- and infrastructure logic.

Let's take a closer look at each of them.

Domain logic

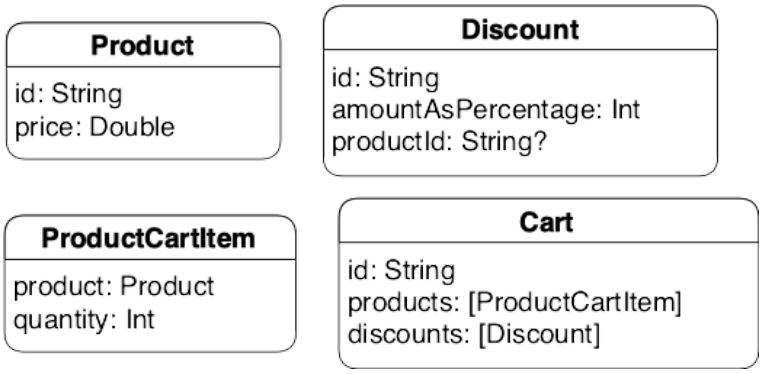


A domain is the subject area to which the software is applied. The domain can be very broad or very specific, depending on the software's complexity and how much of the domain the software covers. Some examples of domains are project management, e-commerce, music, fitness, banking, travel, and e-learning.

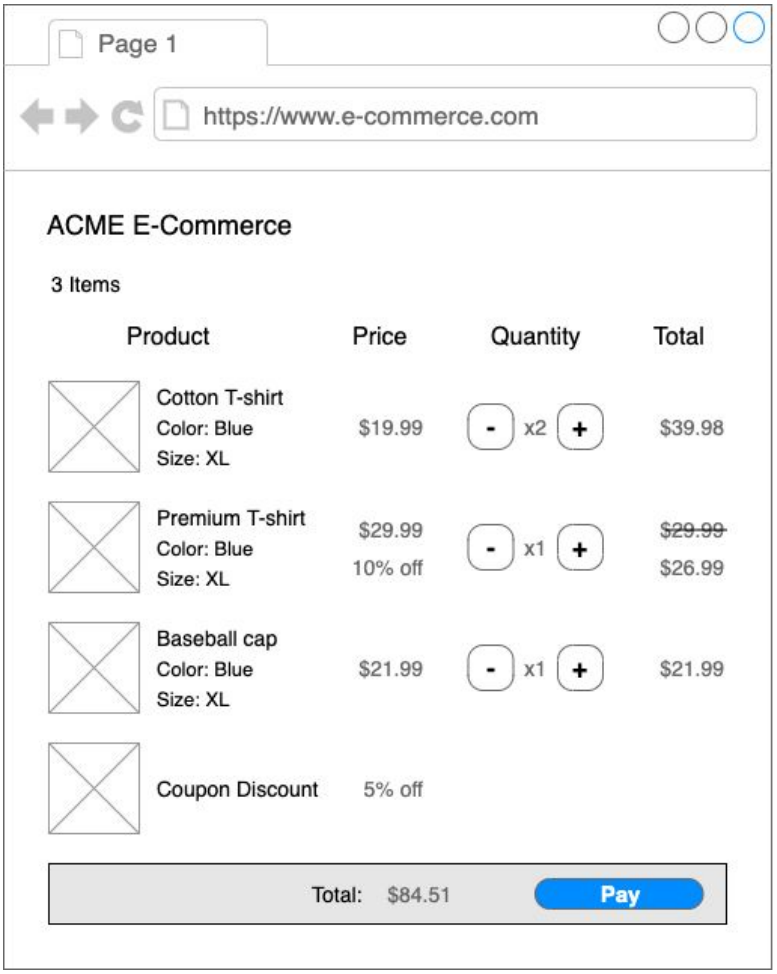
Domain logic includes domain-specific data structures with their relationships and domain-specific rules describing how these data structures interact with each other. Domain logic is a virtual representation of the domain's inner workings in the real world.

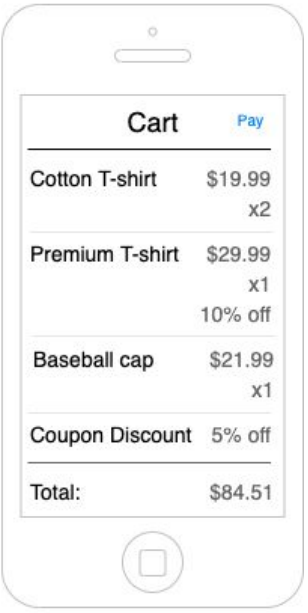
Domain logic is platform-agnostic and application-agnostic. It must be consistent, regardless of whether it is running on the server, web frontend, mobile, watch, console, or desktop.

As an example, let's take an e-commerce solution with a desktop web app and an iOS app. The core of the solution consists of products, discounts, and a cart. These types are domain-specific data structures.



Let's also take a use case where there are several products in the cart, one of them has a product-specific discount applied, and there is an additional discount on the whole cart. Before paying, a user needs to see the total price amount for the cart.





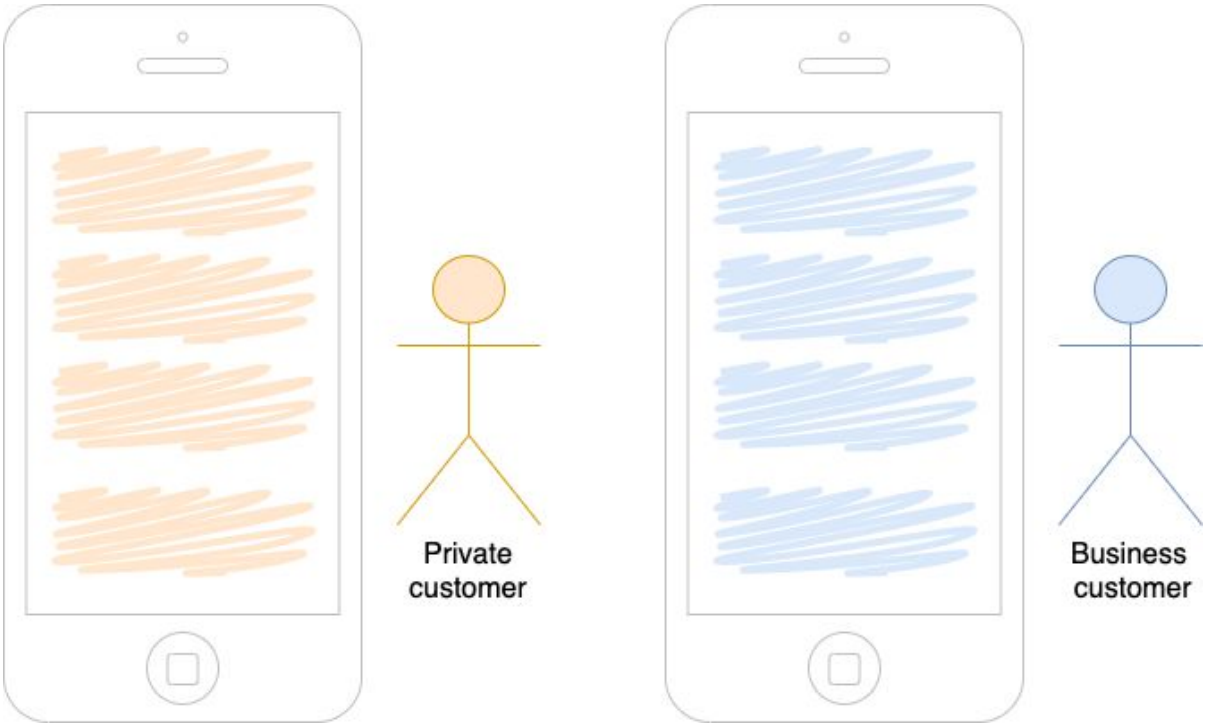
The logic for adding products and discounts to the cart and calculating price amounts are domain-specific rules. These rules are the same across all supported platforms.

CartRules

```
add(product: Product, to cart: Cart) -> Cart
add(discount: Discount, to cart: Cart) -> Cart
isDiscountApplied(to productId: String, in cart: Cart) -> Bool
calculateRegularPriceAmount(for productId: String, in cart: Cart) -> Double
calculateDiscountedPriceAmount(for productId: String, in cart: Cart) -> Double
calculateTotalPriceAmount(for cart: Cart) -> Double
```

A desktop web app and an iOS app will differ in many details, but the calculated total price amount for the cart needs to be the same on both platforms.

Furthermore, two separate iOS applications are serving two different groups of customers making purchases on this e-commerce platform. One group is private customers and the other group is business customers.



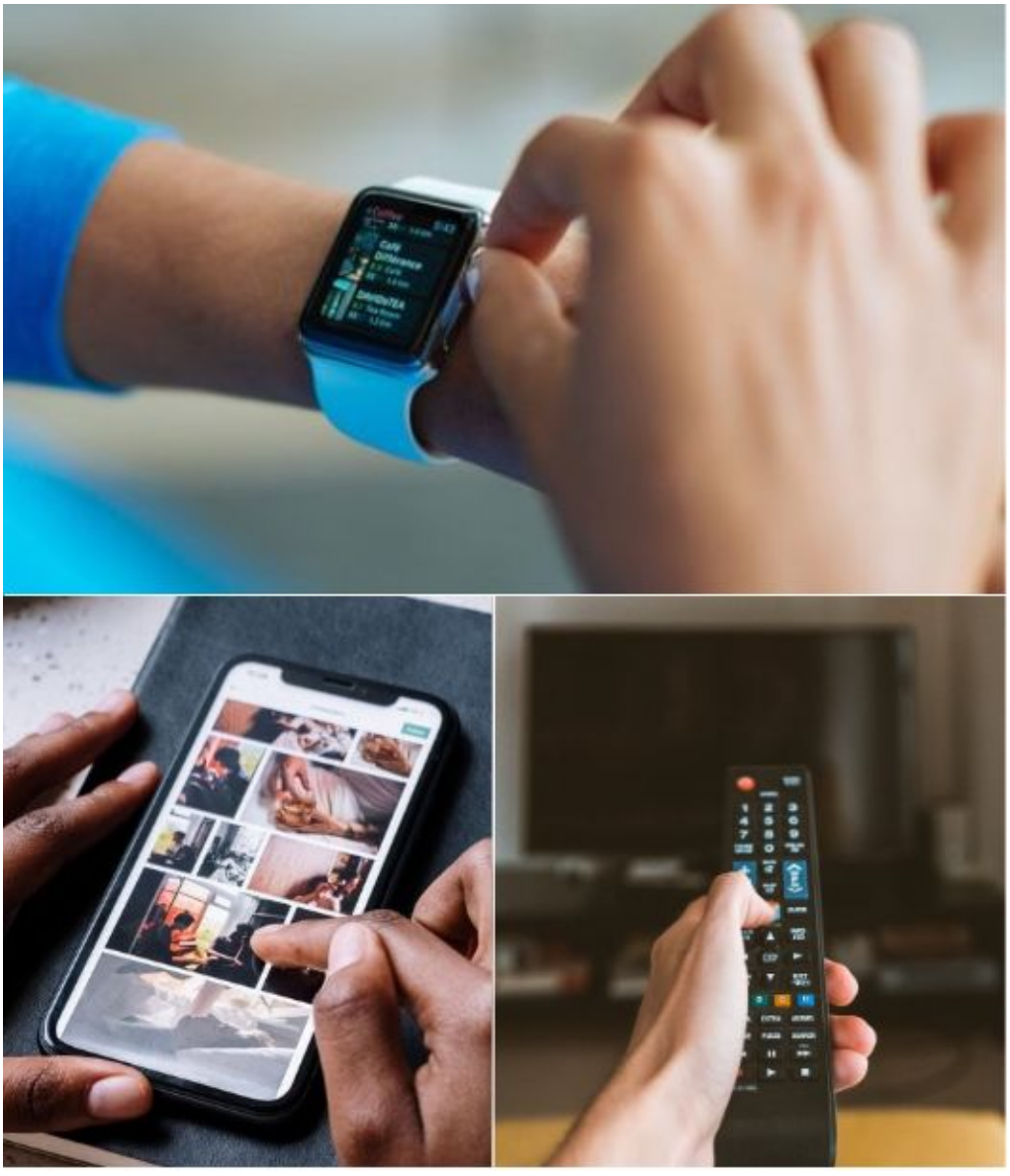
These two groups of customers have different needs. Thus applications serving them have different features, but the logic of adding products and discounts to the cart and calculating price amounts is the same across the applications.

Application logic

Application logic selectively exposes domain logic to the user in a manner appropriate for the environment the domain logic is running in. Additionally, it contains the logic required for the software solution to be developed and maintained.

Application logic is generally platform-specific and obviously application-specific.

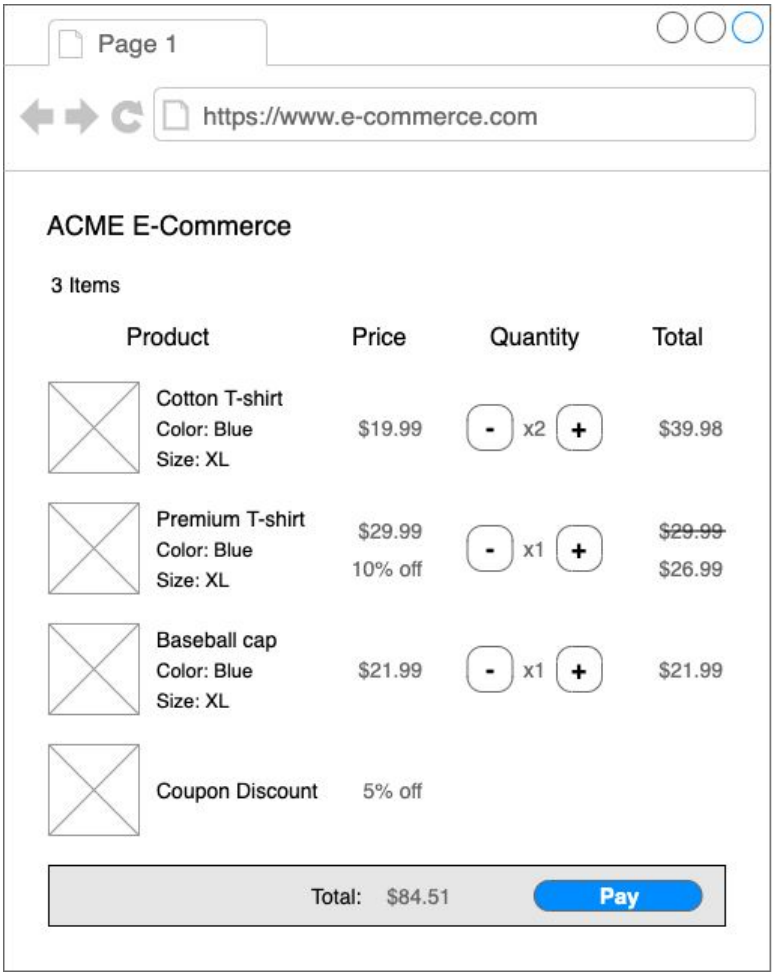
You might have the same application logic shared between different native mobile platforms, and even mobile web, because they are similar, but that same logic probably won't apply to the watch or the TV. Different platforms have different capabilities, different display sizes, and they allow for different types of user interaction, thus application logic generally differs across platforms.

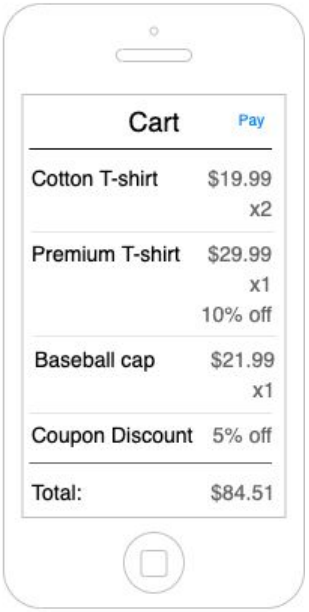


The application logic consists of high-level application logic and low-level application logic. High-level logic describes what work needs to be done and collaborates with the domain logic to do some of the work. The low-level logic uses the infrastructure components to do the remaining work.

Going back to the earlier e-commerce solution example, when a user wants to review the contents of her cart and initiate the purchase, there are several things she needs to be able to see:

- all the products in the cart with some basic information
- how many items for each product there are in the cart
- and the total price amount that will be charged.





The high-level application code describes what needs to be done:

1. pull the cart and products data from the available data storages
2. ask the domain logic to provide the needed price amounts
3. localize and transform all the data to string format for presentation
4. present the data

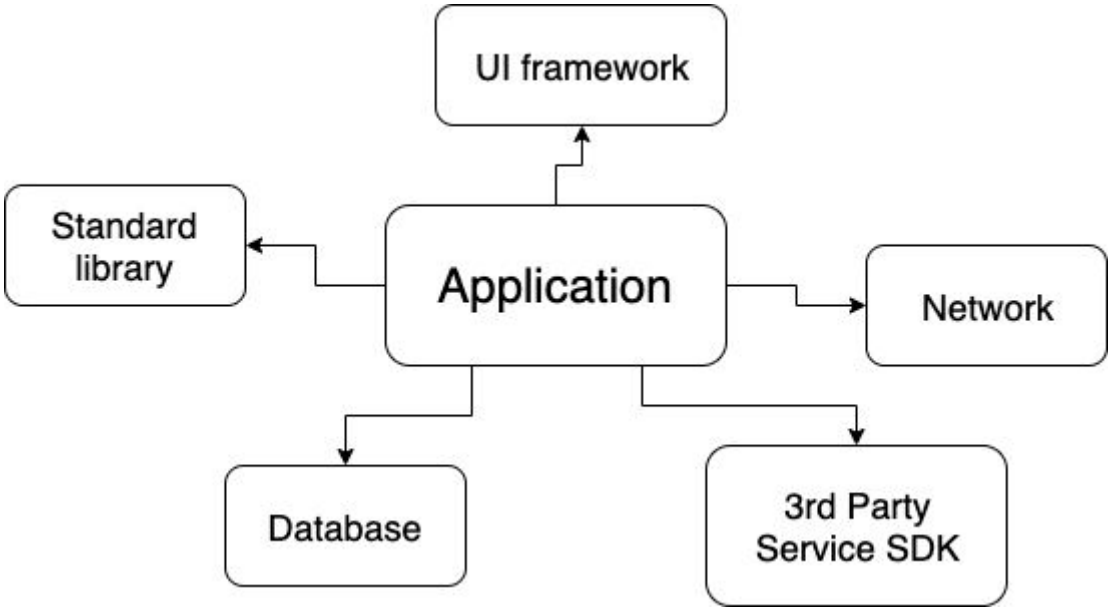
The low-level application code interfaces with the infrastructure to do the “dirty” low-level work:

1. Retrieves data from a local database or the remote server
2. Retrieves localization settings and localizes data
3. Lays out the view and displays the data

The application logic varies across the desktop web app and the iOS mobile app if you take a closer look. The desktop web app exposes more of the domain logic and presents more information to the user than the iOS mobile app. For example, the desktop web app displays more detailed product information and the regular price of a discounted product together with the discounted price. It also allows modifying the cart item quantity directly in the cart, unlike the mobile app.

Infrastructure logic

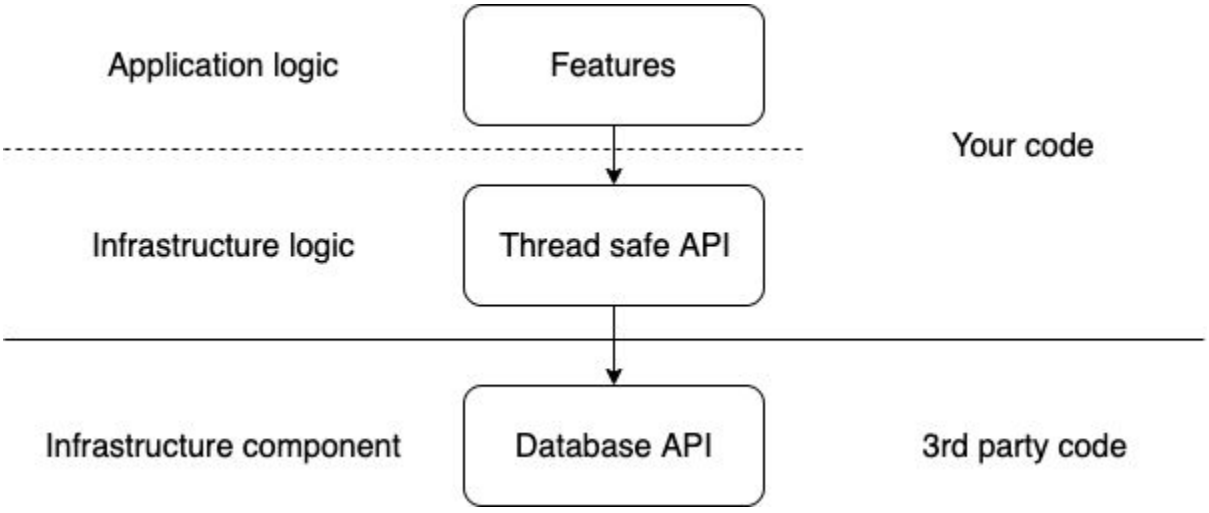
Infrastructure represents the environment the application is running in and all the tools the application uses to fulfill its purpose.



When programming an application, a developer uses the standard library of the programming language, platform-specific SDKs, third-party tools, and frameworks. All of them provide a set of functions and APIs that a developer can use to implement the application. Infrastructure logic is a layer of code that customizes these functions and APIs for the needs of the application code.

Infrastructure logic is platform-specific and application-agnostic. This logic can be shared between completely unrelated applications running on the same infrastructure.

To make it more specific, let's take an example from the e-commerce solution we are building. In a mobile app, this solution needs to persist the cart state on a mobile device so that the user can review it even when offline. The chosen persistent storage doesn't provide thread-safe APIs, so we need to add some wrapper code around the storage APIs to ensure thread safety. This logic is neither domain nor application-specific, so it can be reused across different apps using the same persistent storage.



Let's take another example. A user can refresh a product list by pulling to refresh. There are two different APIs to add a pull-to-refresh UI control to a list view. One works on the older versions of the OS, and the other, a bit simpler one, works on the newer versions of the OS. Both require separate steps to

1. instantiate a UI control
2. add it to the list view
3. and set the logic to be executed when the pull-to-refresh is triggered.

So we create a function that determines the OS version and uses OS-version-specific APIs to do all three steps. Now we can use this function every time we need to add a pull-to-refresh control to a list view. This logic is again neither domain-specific nor application-specific, and it can be reused across apps.

Precisely because the infrastructure logic is application-agnostic, if there is enough code around a certain area of the infrastructure, it can end up as a separate framework reused across different applications.

Why do I need to care about differentiating between domain, application, and infrastructure logic?

It's important to recognize and categorize code into these three layers because it makes a critical difference in the readability and flexibility of the codebase. It is generally a good practice to separate concerns in the code. And this is not the only separation you will make, but it is probably the most abstract one.

If there is a clear separation between these layers, the domain logic can be reused across platforms and applications, and the infrastructure logic can be reused across different applications on the same platform. For that to be the case, domain logic must not reference any application code or infrastructure code. And infrastructure code must not reference any domain code or application code.

There will be different reasons for change for each of the three layers, and the change will happen at different times. Whenever a change in any logic is introduced, our goal is to make sure that the change doesn't cause unintended new behavior or bugs.

Domain logic needs to be changed when the domain's inner workings have changed. It is a relatively rare occurrence, or at least less frequent than changes in application requirements or infrastructure components. But when it does happen, you want the logic to change across all applications and platforms at the same time.

Application logic needs to be changed when the requirements have changed for how the domain logic is presented or interacted with within a single application. In this case, we don't want the domain logic to change, and we don't want the change to affect the behavior of other applications using the same domain logic.

Infrastructure logic needs to change when the infrastructure APIs have changed or when we want to swap an infrastructure component. Think of OS updates and 3rd party service provider changes. Whatever the case is, we want the domain logic and application logic to stay intact while making this change, assuming the infrastructure change didn't also result in an application requirements change.

Even if you have a single platform application and no expectations on the horizon to share code across different applications and platforms, separating these three layers makes it easier to change any one of them without unintended changes in the other two.

What about business logic and presentation logic?

In architectural discussions, code is often separated into business logic and presentation logic. So the question arises, how does the business/presentation separation fit into the domain/application/infrastructure concept?

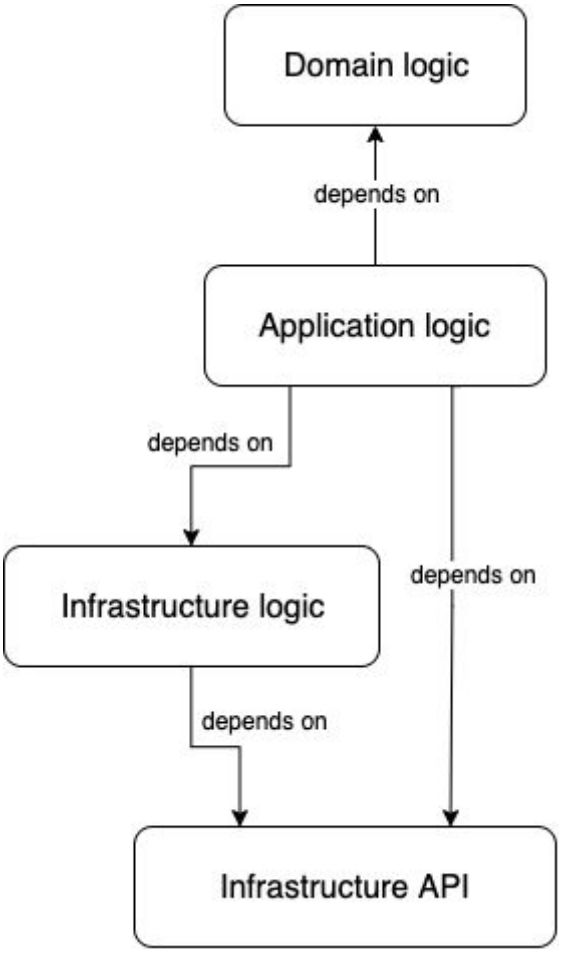
Business logic describes how the business works and how the business is applied to software - hence it is either domain logic or high-level application logic.

Presentation logic describes user interaction and information presentation on a high level in a specific application - hence it is high-level application logic.

How to architect an iOS application?

To quickly summarize the previous chapter

- most applications have a domain logic, application logic, and infrastructure logic
- domain logic has no references to application logic, infrastructure logic, or infrastructure APIs
- infrastructure logic has no references to application logic or domain logic
- application logic references domain logic, infrastructure logic, and infrastructure APIs to implement the features of the application



With that in mind, let's look at how to architect an iOS application.

Domain logic design

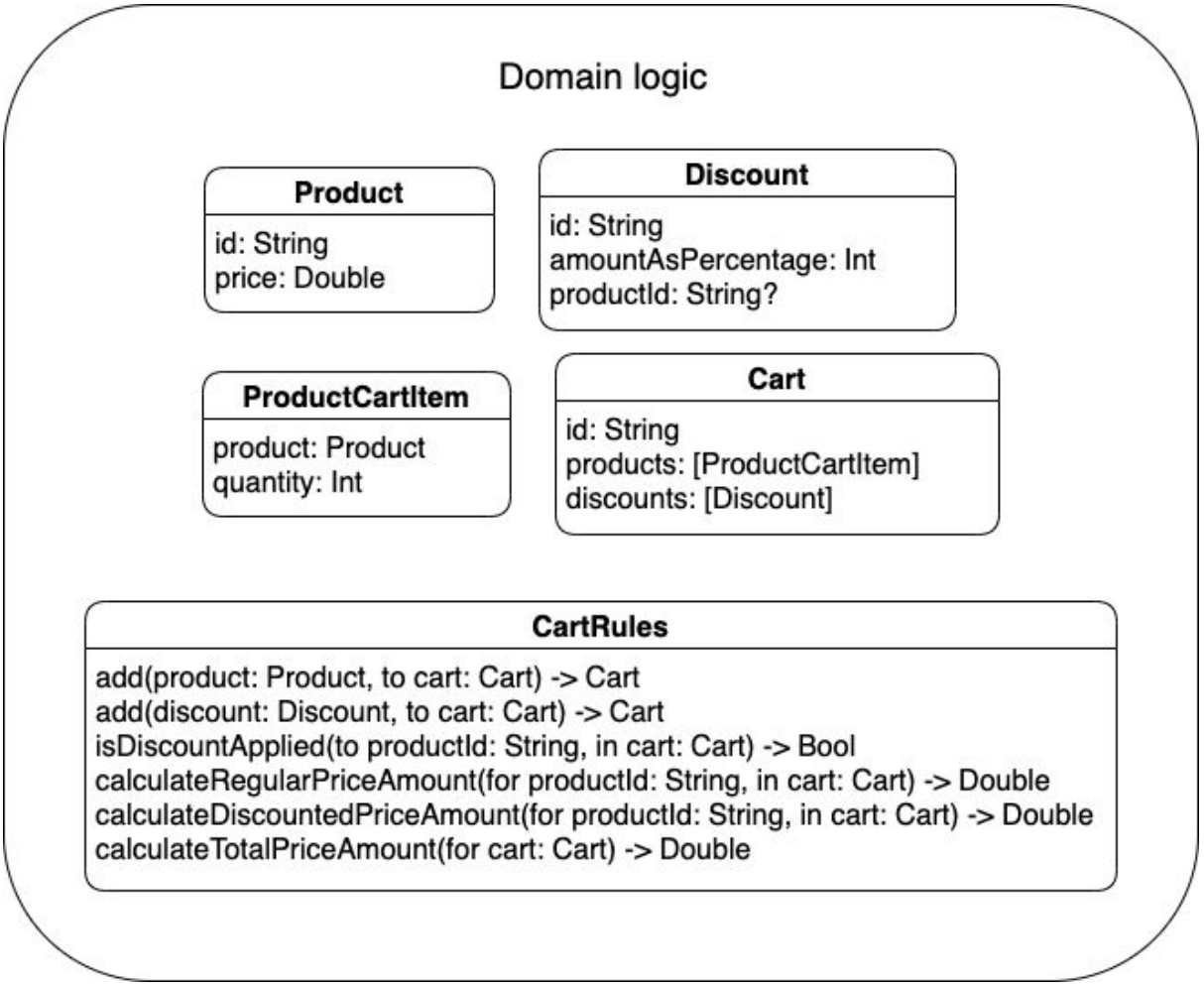
The amount of domain logic in an iOS app can vary a lot. Suppose the application is a thin client for a back-end service. It might not contain any domain logic because the only thing it does is send user requests to the server and present information received from the server. On the other hand, if the application doesn't use any back-end services, except maybe a

remote storage service such as iCloud for data backup, it will contain all of the domain logic needed to implement the requirements.

In case there is domain logic in the app, it should consist of structs and functions.

Domain structures and functions should have no references to any application logic, infrastructure logic, or infrastructure APIs.

Improper state management is one of the biggest causes of bugs. So the goal is to minimize state management throughout the app and localize it in specific parts of the app. Specifically, let the application logic deal with state management. Treat the domain logic as a function - pass the values into the domain logic, and values should come out of it.



Infrastructure logic design

The infrastructure layer is most likely the lightest one in an iOS application. It usually consists of extensions to infrastructure classes from different frameworks like Foundation, UIKit, CoreData, and others. As an example of a repository of reusable Swift extensions, take a look at EZSwiftExtensions.

For applications that have very specific technical requirements, the amount of infrastructure logic can grow significantly. And if there is a need to publish this code separately or use it in a different application, it can be extracted into a separate framework.

An example of infrastructure logic being extracted into a framework is the Texture framework (previously known as AsyncDisplayKit). Texture is an iOS framework built on top of UIKit, which helps optimizing view rendering and keeps the most complex iOS user interfaces smooth and responsive. Another example is RxSwift/RxCocoa. RxSwift provides basic constructs for the reactive programming paradigm, and RxCocoa provides reactive UIKit extensions.

Infrastructure logic should have no references to any application logic or domain logic.

Application logic design

The application layer is most likely the heaviest one in an iOS application. There are lots of components and concerns in this layer, and numerous architectural patterns are describing how to organize this layer:

- Model-View-Controller
- Model-View-Presenter
- Model-View-View-Model + Coordinator
- View-Interactor-Presenter-Entity-Router (VIPER)
- View-Interactor-Presenter (CleanSwift)

Technically, these patterns cover both the application and the domain layer. Still, most of them don't make an explicit separation between the two, except maybe VIPER, which has the Entity component to stand for the domain logic. In others, the domain logic is a part of the Model component.

The biggest misconception about these patterns is that the application cannot consist of any other components but those named by the pattern. The most notorious example of that is the Massive-View-Controller - an alternative name for the MVC pattern. The ViewController component becomes massive because the Model and the View have a pretty clear and relatively small amount of responsibility, so everything else ends up in the ViewController.

Different patterns are a better fit for different apps, and even for different use cases within a single app. But more importantly, these patterns are high-level concepts that may vary a lot in implementation from team to team. This whitepaper will not go into details of any of these patterns. Instead, it will try to distill all the different responsibilities of components within an app, and then you can choose whatever pattern you feel works best for your use case.

Regardless of the architectural pattern used, the application logic will typically consist of two sub-layers - high-level application logic and low-level application logic.
High-level application logic consists of use cases and business services. A use case implements the high-level logic of a single application feature. When multiple use case components share a block of logic, this block can be extracted into a business service component and then reused within multiple use case components.

High-level application logic components can reference domain models and domain functions, but not vice versa.

High-level application logic components have dependencies to which they delegate implementation details of interaction with the infrastructure components. These dependencies represent low-level application logic. Typical low-level application logic components are the following:

- View - UI presentation and interaction component
- Coordinator - UI navigation component
- Tracker - analytics tracking component
- Data Store - persistent storage components
- Infrastructure Service - other infrastructure interfacing components

The main requirement for the high-level application logic is to keep it unaware of any specific implementation of its dependencies through the use of protocols. This approach is called a dependency inversion principle, and it allows high-level modules to be independent of the low-level module implementation details.

Any objects or structures referenced by the high-level application logic must also be high-level. This includes domain structures and high-level presentation structures passed to the view component for displaying in the UI.

References to framework-specific objects from CoreData, Realm, Firebase, UIKit, SwiftUI, and others are not allowed in the high-level application logic. Any mapping between framework-specific objects and domain structures or high-level presentation structures is done in the low-level application logic.

The following chapters will dive into more details about high-level and low-level application logic components.

Use case

A use case describes an event initiated by an actor and the effects of that event.

An actor can be either the user or one of the system components.

An event can be either a user interaction with the device, or some kind of a trigger, or an update from one of the system components.

The effects include changing the state of the application, notifying different system components of that change, and reflecting the change in the information presented in the user interface.

Here are a couple of examples of use cases:

- A user (actor) taps on the button (event) to complete a purchase - the button tap is tracked in the analytics service, the complete purchase request is sent to the server, the success or the error alert is displayed in the UI depending on the response, and if an error happens it is logged in the logging service (effects)
- A user (actor) taps on the list item (event) to transition from the items list view to the item details view (effects)
- As the device is moved, the OS (actor) sends location updates to the app (event), which then caches the latest position and updates the position of the user icon on the map in the UI (effects)
- As a new chat message is received on the server (actor), it sends a push notification to the client (event), which then persists the new message locally and displays it in the UI (effects)

A use case describes what needs to happen on a high level, but it doesn't define any specific details of how the event or the effects are materialized in the infrastructure.

Each use case can be implemented in a separate component, but it can be more convenient to have multiple closely related use cases implemented in a single component because such use cases usually have shared logic and the same dependencies.

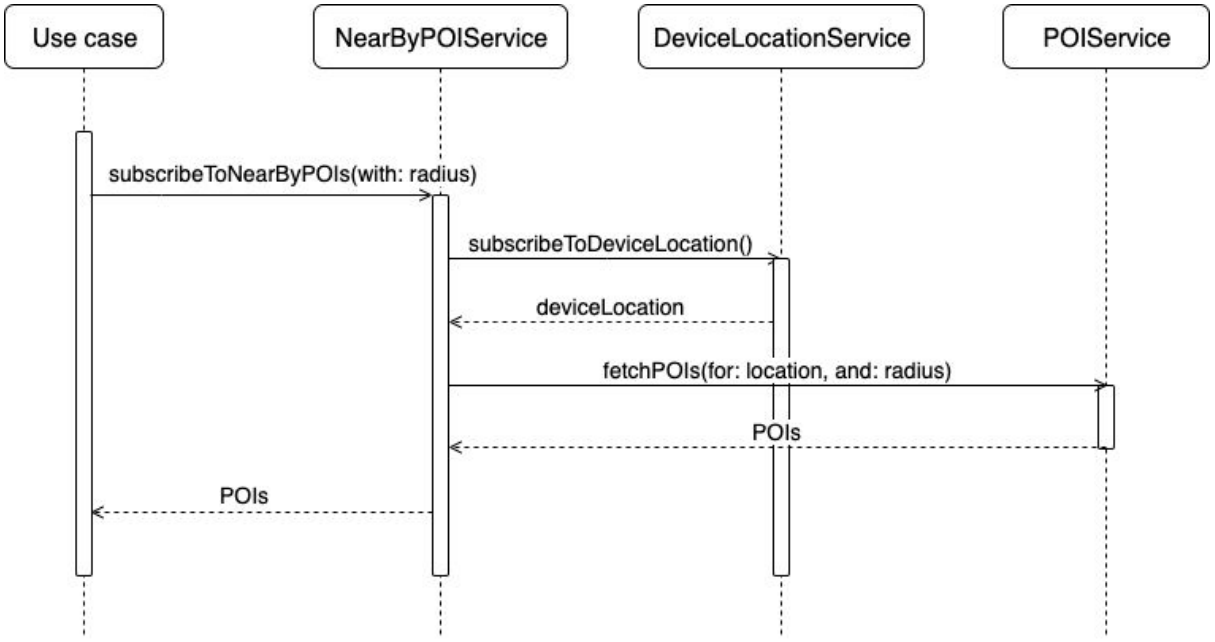
A use case is responsible for mapping information between domain structures and high-level presentation structures. A simple 1-on-1 mapping can be defined as a static function in the presentation structure. For more complex cases that combine data from multiple domain structures into one presentation structure, the mapping can be delegated to a separate presenter component.

A use case is generally going to have one view, one coordinator, and one tracker. The number of business services, data stores, and infrastructure services may vary between zero and several.

Depending on the architectural pattern chosen, the use case logic will likely be found in the Controller (MVC), Presenter (MVP), ViewModel (MVVM), or Presenter and Interactor (VIPER and CleanSwift). Some of the clean architecture patterns (VIPER and CleanSwift) make a special effort to create boundaries between the components that deal with high-level business structures and high-level presentation structures. This can be useful if the same business logic of the use case is reused with different presentations. But that's rarely the case, and the additional complexity and cost associated are most likely not worth it.

Business service

A business service represents a separate chunk of high-level business logic reused across different use cases. It often depends on one or more data stores and one or more infrastructure services. Usually, it is introduced when multiple use cases are depending on the same block of business logic.
Let's take an example of a treasure hunt type application with two use cases. One use case is the user seeing an augmented reality view with nearby points of interest appearing on the screen. The other use case is the user seeing a map with the user's location and points of interest present on the visible part of the map. Both of these use cases require a chunk of business logic, which uses the DeviceLocationService and the POIService backed by a remote API to determine a list of points of interest needed for the use case.



This shared chunk of high-level application logic can be extracted into a business service named NearByPOIService and then reused in the two use cases.

UI presentation and interaction

The user interface is represented by the view layer of the application. A view component defines how information and interaction controls are laid out and rendered on the screen. It translates between high-level presentation structures and implementation-specific data types of the UI framework. And it interprets the user's gestures.

To keep the code flexible, the view component needs to be easily interchangeable. Whether there is a need to switch from a UITableView implementation of a list view to a UICollectionView implementation, or there is a need to move from a UIKit based implementation of the view layer to a SwiftUI based implementation, it needs to be as easy as possible. This flexibility is achieved with a boundary between the use case component and the view component. This boundary is defined by two protocols, one for the presentation information and configuration passed from the use case to the view, and the other for the user interaction events and data passed from the view to the use case.

The boundary between the use case and the view doesn't only protect the use case from referencing a specific view implementation, but it also allows for the view component to be used by different use cases or driven by a temporary mock object. The latter is useful in several scenarios:

- prototyping the view layer before the business logic is yet implemented
- unit-testing the view layer rendering with various configuration and test data
- for larger teams with specialist roles where the high-level application logic and view components are implemented and maintained by different people

UI navigation

The user interface navigation component is responsible for transitioning UI elements from the screen and onto the screen. This logic lives in the same environment as the UI presentation and interaction logic - either UIKit and SwiftUI.

It is defined as a separate dependency, and not a part of the view interface, because of reusability, flexibility, and testability. Having a separate navigation component makes it easy to reuse the navigation behavior when a specific view needs to be accessed from multiple different views. It's also easier to change view flows in the app as the requirements change.

The navigation dependency is often split into two components, a builder component and a navigation component. The builder component creates the use case component and its dependencies, including the view component (UIViewController), and links them all together. The navigation component is responsible for the transition between the current view and the newly assembled view.

In simpler cases, the navigation dependency can be implemented by the UIViewController, which uses the builder component to assemble the new scene and then transitions to it.

As long as there is a clear boundary with the navigation protocol from the use case perspective, the implementation can easily be changed to be simple or more complicated, depending on the requirements.

Some architectural patterns explicitly prescribe defining the navigation component, like the wireframe in VIPER and Coordinator in MVVM+C.

Persistent storage

It is impossible to develop anything but a trivial app without persistent storage. There needs to be a place where the application can store application data and user data so that it persists even after the application is terminated and wiped out of memory.

The component providing CRUD operations for the application and user data is often called a repository or a data store. Most of the use cases will have one or more data store dependencies for one or more data types.

A data store for a specific data type can be implemented as a local storage or a remote storage depending on the requirements. In theory, the use case wouldn't even care about whether the store is local or remote, thus different kinds of data stores would all implement the same protocol. In practice, there will usually be a difference in user experience, depending on whether the storage is local or remote. Remote storage usually takes longer to complete operations, so additional content loading indication in the UI is needed. Remote storage can also fail to complete operations because of connectivity issues, resulting in specific user-facing error messages. There are also use cases where a combination of local and remote storage is used, with one being the source of truth. This use case again may require different user experience.

Local storage can be implemented with CoreData, SQLite, RealmDB, or even with files on a local disk. Remote storage is likely implemented with a networking layer that communicates with a web server. There are also hybrid solutions like Firebase Realtime Database and Realm Sync, which act as a local database, but also synchronize all the data with the back end, and notify the application when new data is coming from the backend.

Hiding the implementation details of a data store from the use case through a protocol helps with flexibility and testability. This includes the data store's responsibility to map from the framework-specific objects to the domain structures. It's much easier to swap between two data store implementations if the details are contained and hidden behind a protocol. It is very costly to replace the persistence solution once the framework-specific objects returned by it have been referenced all around the business logic and presentation logic.

Analytics tracking

Most of the iOS apps have either custom or off-the-shelf analytics system integration. The goal of the analytics system is to track important user behavior in the app, so that a product manager can better understand how the application is used. Understanding user behavior in the app drives decisions on how to develop the application further. Tracking user behavior is also important from the marketing perspective. Providing feedback on the marketing's performance helps grow the business and keep the application on the market.

It's impossible to know upfront which analytics services will be used throughout the lifetime of an application. There might be one at first, then two or three in combination, and then only one again sometime later. Depending on how the needs of product management and marketing teams change, the need for integrating different analytics solutions will change as well. If there are multiple analytics services integrated, not all events will be tracked on all analytics services. And different analytics services may require different data formats for various types of events and their parameters.

To keep the code flexible and make changes in the analytics tracking as easy as possible, it's advisable to specify a high-level analytics tracking protocol for every use case component. This protocol contains methods to be called on significant events within the use case with high-level models as arguments. Whether there are one or more analytics services called and how the high-level models are converted to analytics event parameters is an implementation detail, and not a concern of the use case component.

Infrastructure service

An infrastructure service provides access to infrastructure components such as platform frameworks, 3rd party frameworks, and external web services. It serves as a translator from the high-level language that use cases and business services speak to the low-level language that infrastructure components speak. It hides the complexity of the infrastructure APIs and exposes only the functionality needed by the high-level application code. As a result, whenever the infrastructure API changes, there is only one place in the application code that needs updating.

An example of an infrastructure service is a service that provides access to the photo library on the device. Assuming the use case requires adding photos from the Internet to the photo library on the device, this service would have one or more methods to ensure the user has provided required access to the photo library and to save a photo in the photo library.

Another example of an infrastructure service is a service that provides access to the Facebook SDK features. Assuming the use case requires displaying a list of user's friends on Facebook, this service would expose the ability to connect with Facebook and get a list of user's friends.

The amount of code in the infrastructure services can vary. If the infrastructure API is very complicated and doesn't suit your needs very well, there might be a lot of code to expose the needed functionality through a service protocol. If the infrastructure API is simple and suits your needs perfectly, you might simply create a protocol with the same method signatures used by the infrastructure class and extend the infrastructure class with this protocol.

Unit Testing

Unit tests are a vital prerequisite necessary to introduce change into existing software with minimal time. Having unit tests is especially helpful if a developer is modifying code written a while ago or code written by someone else entirely. These situations are risky, especially in more complex projects. There is usually not enough time to build a complete and accurate mental model of relationships between components involved and all the possible code paths. Heaving clean architectural boundaries and components with well-separated responsibilities, as described in earlier chapters, helps with limiting the size of the mental model a developer needs to build when introducing change. Unintended change can slip in anyway, and unit tests can help catch them.

There is no one definition of what a unit is in the context of unit testing. It can be a function/method, or a class, or a module. Consider a module in this context as a set of interconnected classes representing a well-rounded feature, and a feature as a use case or a group of closely related use cases.

Popular opinion is that a unit test should only ever test a single class and that all of its dependencies should be mocked. But this style of unit testing requires deep coupling between the implementation code and the test code. As a result, the software is less flexible because almost any change in the implementation will break the tests and require changing them.

To provide the desired flexibility, the unit tests should test feature behavior as a black box, thus allowing refactoring of the module structure without any changes in the test code. With this in mind, it is advisable to design unit tests to test behavior around major architectural boundaries and mock the dependencies on the other side of the boundary. Specifically, this means testing the following blocks of logic and treating them as a black box (unit):

- domain logic
- high-level application logic (use cases, business services) with all the dependencies mocked
- infrastructure logic

Domain logic and high-level application logic are the core value that the application provides to its users, and they need to have maximum unit test coverage.

Infrastructure logic can also be important to test if its components are used in many places throughout the app or if there is complex logic involved.

Testing low-level application logic (views, navigation, data stores, analytics tracking, infrastructure services) has a lower return on investment because it usually consists of simple components that map a high-level interface to a low-level interface. This logic is also usually more cumbersome to test because it ties directly to the infrastructure. Depending

on the maturity and the criticality of the application, it may be acceptable and even advisable to skip unit-testing low-level application logic or unit-test it sporadically.

To provide an example of how treating the high-level application logic as a test unit can be useful, let's imagine that the beforementioned treasure hunt app at first only had a use case with a map view. At that time, the logic of getting the device location and fetching the nearby POIs was a part of a use case. This use case depended on the POIService and the DeviceLocationService. And it has been tested as a unit. Then a feature request came in to introduce the augmented reality view, which also needs this logic of getting the device location and fetching the nearby POIs. It is now possible to refactor this logic and move it into a business service used in the existing use case and the new use case, without changing the existing tests. At the same time, these unmodified tests will verify that the refactoring did not cause any unintended behavior in the existing use case.

Summary

In summary, every iOS application consists of application dependencies and application code.

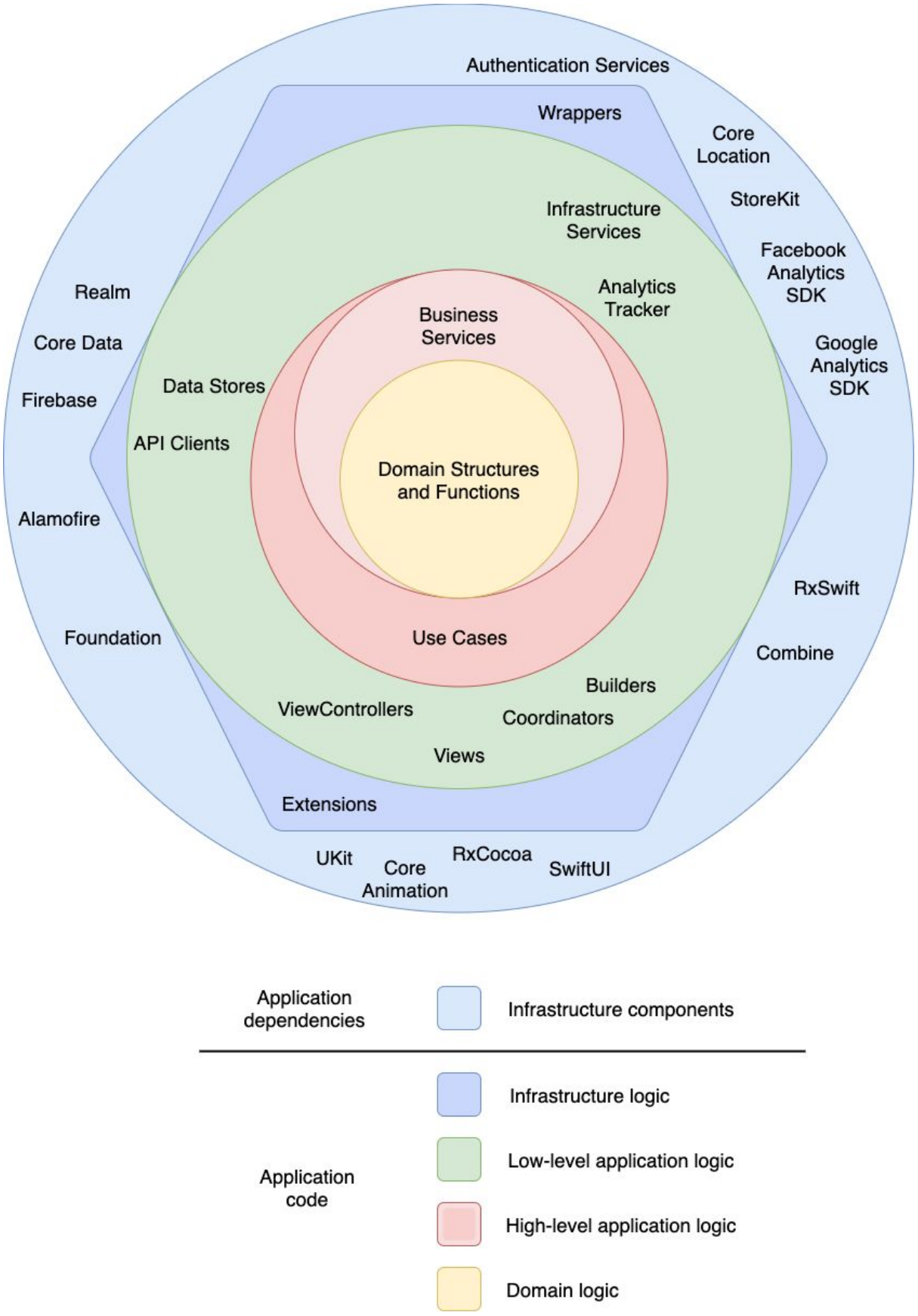
Application dependencies usually include:

- standard library
- platform-specific SDKs
- third-party tools
- and frameworks.

Application code consists of:

- infrastructure logic
- low-level application logic
- high-level application logic
- and domain logic.

Dependency arrows always must point from lower-level logic to higher-level logic. Looking at the diagram on the next page, no inner circle component can reference an outer circle component. When an inner circle component needs to communicate with an outer circle component, it uses the dependency inversion principle.


Authentication Services
Wrappers
Core Location
StoreKit
Infrastructure Services
Facebook Analytics SDK
Realm
Analytics Tracker
Business Services
Core Data
Google Analytics SDK
Firebase
Data Stores
API Clients
Domain Structures and Functions
Alamofire
RxSwift
Foundation
Use Cases
Combine
Builders
ViewControllers
Coordinators
Views
Extensions
UKit
Core Animation
RxCocoa
SwiftUI
Application dependencies
Infrastructure components
Application code
Infrastructure logic
Low-level application logic
High-level application logic
Domain logic