УДК 004.6

В.В. СПИРИНЦЕВ, Н.А. ШИТИК
Днепропетровский национальный университет им. Олеся Гончара

АНАЛИЗ СОВРЕМЕННЫХ АРХИТЕКТУРНЫХ ШАБЛОНОВ, ИСПОЛЬЗУЕМЫХ ПРИ ПРОЕКТИРОВАНИИ ПРИЛОЖЕНИЙ В СРЕДЕ IOS

Представлены результаты исследования основных архитектурных шаблонов (*Model – View – Controller* (*MVC*) *и View – Interactor – Presenter – Entity – Router* (*VIPER*)), *используемых при проектировании приложений для мобильной платформы iOS. Выявлены их функциональные особенности, преимущества и недостатки. Освещены сферы их применения для выбора оптимальной стратегии при разработке корпоративных приложений.*

Ключевые слова: паттерны, MVC, VIPER, диаграмма классов.

В.В. СПІРІНЦЕВ, М.А. ШИТІК
Дніпропетровський національний університет ім. Олеся Гончара

АНАЛІЗ СУЧАСНИХ АРХІТЕКТУРНИХ ШАБЛОНІВ, ЩО ВИКОРИСТОВУЮТЬСЯ ПРИ ПРОЕКТУВАННІ ДОДАТКІВ У СЕРЕДОВИЩІ IOS

Представлені результати дослідження основних архітектурних шаблонів (*Model – View – Controller* (*MVC*) *і View – Interactor – Presenter – Entity – Router* (*VIPER*)), *що використовуються при проектуванні додатків для мобільної платформи iOS. Виявлено їхні функціональні особливості, переваги та недоліки. Висвітлено сфери їхнього застосування для вибору оптимальної стратегії при розробці корпоративних додатків.*

Ключові слова: паттерни, MVC, VIPER, діаграма класів.

V.V. SPIRINTSEV, M.A. SHYTIK
Dnepropetrovsk National University named after Oles Honchar

THE ANALYSIS OF MODERN ARCHITECTURAL PATTERNS OF THE APPLICATIONS USED FOR PLANNING IS IN ENVIRONMENT OF IOS

The results of research of the basic architectural templates (*Model – View – Controller* (*MVC*) *and View – Interactor – Presenter – Entity – Router* (*VIPER*)) *used for planning of appendixes for the mobile platform of iOS are presented. Their functional features, advantages and defects, are educed. The spheres of their application are lighted up for the choice of optimal strategy at development of corporate applications.*

Keywords: patterns, MVC, VIPER, diagram of classes.

Постановка проблемы

Корпоративные приложения чаще всего имеют дело с разнородными данными неограниченного объема и бизнес-правилами, логика которых усложнена большим количеством частных случаев, в условиях различия бизнес-процессов и представления данных. Кроме того, бизнес-требования могут изменяться во времени, и система должна быть адаптируемой к внесению изменений. В большинстве случаев, решаемые в процессе разработки таких приложений задачи являются однотипными. В связи с этим актуальными являются исследования в направлении проектирования и реализации шаблонного решения, которое одновременно должно быть достаточно гибким для реализации уникального для каждого приложения функционала. Шаблон (или паттерн) проектирования в разработке программного обеспечения – повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста [1]. Суть проектирования архитектуры приложения – использовать современные инструменты, подходы и технологии для получения максимальной выгоды, с учетом требований и ограничений, вытекающих из поставленных задач. При этом необходимо максимально учитывать современные тенденции и тенденции ближайшего будущего для обеспечения максимальной прибыли через масштабируемость, гибкость и обслуживание. В качестве признаков эффективной архитектуры выделяют: сбалансированное распределение обязанностей между сущностями с жесткими ролями; тестируемость, простота использования и низкая стоимость обслуживания.

Анализ последних исследований

Для снижения сложности системы путём абстракции и разграничения полномочий классов используют принципы SOLID (Single Responsibility Principle, Open-Closed Principle, Liscov Substitution Principle, Interface Segregation Principle и Dependency Inversion Principle). SOLID – это пять основных принципов объектно-ориентированного программирования и проектирования, предназначенных для повышения вероятности создания системы, которую будет легко поддерживать и расширять в течение длительного времени [2]. Для удовлетворения проектируемой системы различным атрибутам качества

применяются архитектурные шаблоны. Наиболее популярным шаблоном построения системы для платформы iOS является шаблон Model – View – Controller (MVC), адаптированный под особенности реализации стандартных фреймовков UIKit и Foundation [3] (такой подход рекомендует использовать Apple в своих гайдлайнах по разработке). Однако при разработке крупных корпоративных приложений при помощи шаблона MVC можно столкнуться с такими проблемами, как сосредоточение кода в контроллерах, усложнение тестирования, недостаточная модульность, высокая связанность классов и т.д. В качестве альтернативы MVC (особенно при использовании Swift в качестве языка разработки) при проектировании архитектуры все чаще используют шаблон View – Interactor – Presenter – Entity – Router (VIPER), который позволяет избежать перечисленных недостатков архитектуры, однако при этом гораздо увеличивается количество классов, интерфейсов, и как следствие – кода в целом [1, 4].

Формулирование цели исследования

Цель работы заключается в осуществлении анализа современных архитектурных паттернов, изучении сфер их применения для выбора оптимальной стратегии при разработке корпоративных приложений для мобильной платформы iOS.

Основная часть

При разработке корпоративных приложений для мобильных платформ одним из самых популярных является шаблон MVC. При использовании MVC каждый класс в приложении выполняет одну из трех ролей: модель, контроллер или представление, при этом шаблон определяет не только роли, но и то, как объекты взаимодействуют друг с другом. Модель инкапсулирует данные, логику и правила обработки данных. Так как объекты моделей представляют собой знания в конкретной предметной области, они могут быть переиспользованы для решения подобных задач. Объекты моделей не должны обладать знанием о своем представлении и связи с пользовательским интерфейсом. Взаимодействие пользователей со слоем представления, следствием которого является создание или обновление данных, обрабатывается контроллером, который в свою очередь обновляет слой данных [4]. После того, как модель получает обновления (к примеру, данные, полученные из сети; уведомления от гироскопа и т.д.), она оповещает контроллер, который обновляет соответствующие представления. Модели делятся на активные и пассивные: активные модели хранят логику и алгоритмы, пассивные – данные. При разработке приложений с помощью фреймворка Cocoa Touch, в роли пассивных моделей обычно выступают наследники базового класса NSObject. В случае, если модели необходимо хранить в базе данных, это могут быть наследники NSManagedObject (предоставленные стандартной оболочкой для работы с базами данных в iOS – CoreData) [5], либо наследники объектов из сторонних решений – RLMObject из библиотеки Realm, MTLModel из Mantle и т.д. Ответственностью объектов представления является отображение данных для пользователей, реакция на пользовательские действия и передача уведомлений о них контроллеру. Фреймворк UIKit предоставляет большой набор стандартных компонентов слоя представления, а непосредственно разработку пользовательского интерфейса можно производить как из кода, так и с помощью инструмента xCode Interface Builder. Контроллер выступает посредником между моделью и представлением, а также может координировать задачи приложения и управлять жизненным циклом других объектов. MVC стал стандартом в современной разработке программного обеспечения для мобильных платформ благодаря разделению кода на бизнес-логику, управление процессом выполнения задач и отображение, что способствует масштабируемости и поддерживаемости [6].

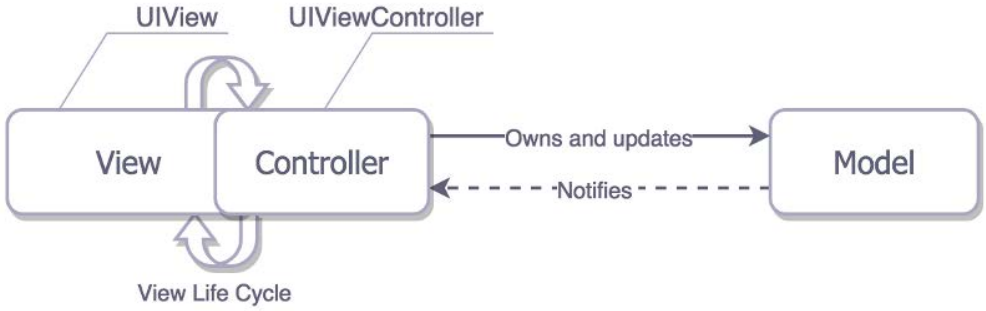

Рис. 1. Схема взаимодействия компонентов архитектуры MVC

Однако при значительном увеличении функционала использование MVC порождает некоторые проблемы. MVC не предусматривает разбиения функционала приложения на модули (как пример, частой ситуацией в разработке корпоративных приложений является существование синглтона NetworkClient, который выполняет запросы по авторизации, созданию пользовательского контента, поиску пользователей и т.д.). При существенном разрастании функционала количество таких универсальных классов увеличивается (сервисы, хелперы, модели), что значительно увеличивает связанность кода. Для наглядности рассмотрим схематическую диаграмму классов реального корпоративного приложения, построенную с помощью утилиты dependency-visualizer (каждый круг – это класс, стрелка между классами означает, что один из классов в своей реализации обращается к другому). Приложение построено на базе архитектуры MVC, но без разделения на модули.

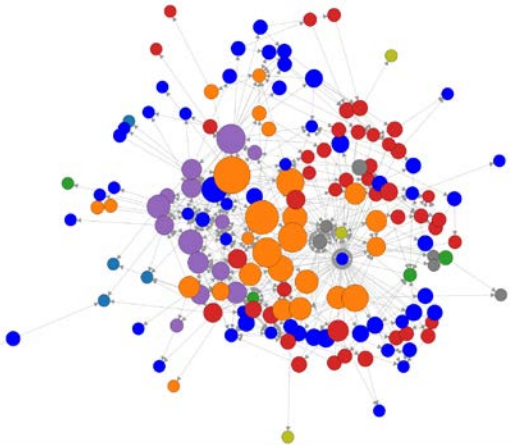

Рис. 2. Диаграмма классов приложения, разработанного с помощью MVC

Наличие большого количества связей между классами говорит о том, что необходимость изменения интерфейса одного из них повлечет за собой изменение реализации всех классов, которые от него зависят.

Одним из самых главных классов в Cocoa Touch является UIViewController, который управляет набором представлений, взаимодействием между ними и передачей обновлений от соответствующих моделей. Так как объекты-наследники UIViewController отвечают за весь отображаемый на экране интерфейс в момент времени, в них сосредотачивается весь код, который отвечает за получение данных из разнообразных источников (ответы на сетевые запросы; координаты от сервиса геолокации; уведомления от акселерометра и гироскопа и т.д.), обработку полученной информации, подготовку к отображению и непосредственно отображение. Такой подход нарушает принцип единой ответственности, усложняет тестирования, создает дополнительный порог сложности для понимания новыми членами команды.

Еще одним недостатком MVC при разработке для iOS является отсутствие явного выделения ответственности для навигации. В мобильных приложениях часто возникает необходимость осуществлять сложные переходы между пользовательскими сценариями, и, при проектировании с помощью шаблона MVC, код, который отвечает за навигацию (выбор того, на какой экран произойдет переход, наполнение соответствующего наследника UIViewController, осуществление анимированного перехода) остается в ViewController и добавляет еще одну ответственность к вышеперечисленным.

Для решения указанных проблем была предложена архитектура VIPER, которая является расширением MVC. Она более детально разделяет ответственности между слоями и вносит понятие модульности. Слой View совпадает в обоих архитектурах, Interactor является активной моделью и перенимает обязанности Model и ViewController, Presenter – слой, который отвечает за подготовку данных к отображению и передачу пользовательских событий к уровню Interactor (в MVC этими задачами занимался ViewController), Entity – пассивная модель, Router (часто также называемый Wireframe) – слой, отвечающий за навигацию.

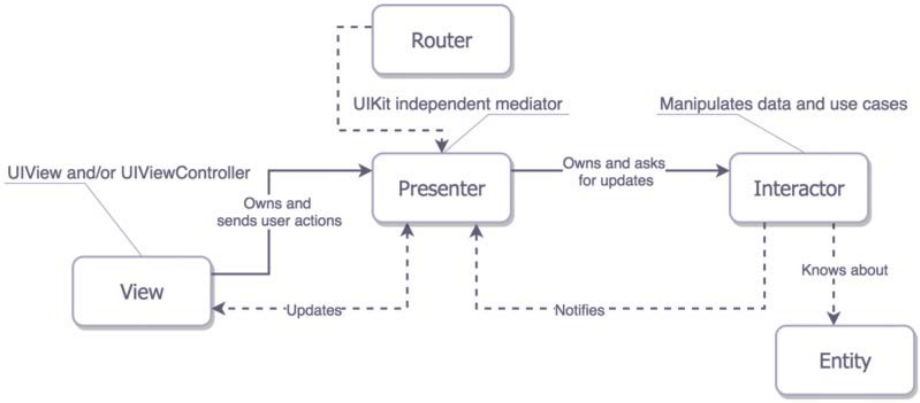

Рис. 3. Схема взаимодействия компонентов архитектуры VIPER

Взаимодействие между слоями происходит через интерфейсы (у класса, принадлежащего к любому из VIPER-слоев кроме Entity есть два протокола – ввода и вывода), поэтому один VIPER-модуль содержит как минимум пять классов и восемь интерфейсов. Большое количество шаблонного кода является главным недостатком VIPER, с которым частично можно бороться с помощью кодогенерации. Одной из широко используемых утилит для генерации Swift-кода является Generamba, разработанная iOS-командой компании Rambler, которая создает пустой VIPER-модуль со всеми классами и интерфейсами, добавляет его в файловую систему проекта и в файл проекта .pbxproj.

Пример модуля, который реализует отображения профиля текущего пользователя в корпоративном приложении.

```
// Слой Entity - пассивная модель
struct TextInfo {
   var collectionsDescriptionText = ""
   var descriptionText = ""
   var productsText = ""
}
// Слой Interactor - отвечает за получение и обработку данных из сети и оповещение слоя Presenter
об обновлениях с помощью протокола InteractorOutput
protocol InteractorInput: class {
   weak var output: HomeInteractorOutput! { get set }
   func fetchHomeInfo()
}
protocol InteractorOutput: class {
   func interactorDidFetchTextInfo(homeInfo: TextInfo)
   func interactorDidFetchImageInfo(homeInfo: ImageInfo)
   func interactorHomeInfoFetchDidFail(error: NSError)
   func interactorDidFetchFullInfo()
}
class Interactor: InteractorInput {
   weak var output: InteractorOutput!
   private let apiClient: APIClient
   private var homeTextInfo = TextInfo()
   private var homeImageInfo = ImageInfo()
   // MARK: - InteractorInput
   func fetchHomeInfo() {
       let executor = Executor.mainThreadExecutor()
       var tasks = [Task]()
       let textInfoQuery =  Content.homeQuery("key", containedIn: [TextInfoKey, ImageKey])
       let textInfoTask = textInfoQuery.findObjectsInBackground()  { [weak self] task in
               if let homeTextInfo = task.result as? [Content] {
                   self?.parseTextServerResponse(homeTextInfo)
               }
           }
       )
       tasks = [textInfoTask]
       let combineTask = Task(forCompletionOfAllTasks: tasks)
       combineTask.continueWith({ [weak self] task in self?.didFinishLoading(task.error)})
   }
   // MARK: Response handling
   private func parseTextServerResponse(textContent: [Content]) {
       if let index = textContent.indexOf( {$0.key == TextInfoKey }) {
           let collectionsTextInfo = textContent[index]
           homeTextInfo.collectionsDescriptionText = collectionsTextInfo.textValue!
       }
       output.interactorDidFetchTextInfo(homeTextInfo)
   }
}
// Слой Presenter - отвечает за пользовательский ввод, запрос информации из слоя Interactor и
обновление предсталения
protocol PresenterOutput: class {
   func homePresenterDidSelectMenu(presenter: Presenter)
   func homePresenterDidSelectCollections(presenter: Presenter)
}
class Presenter: ViewOutput, InteractorOutput {
   var interactor: InteractorInput!
   weak var output: PresenterOutput!
   weak var view: ViewInput
   private var homeInfoLoaded = false
   // MARK: ViewOutput
   func viewIsReady() {
       interactor.fetchHomeInfo()
   }
   func viewDidSelectCollections() {
       output.homePresenterDidSelectCollections(self)
   }
   // MARK: HomeInteractorOutput
   func interactorDidFetchTextInfo(homeInfo: HomeTextInfo) {
       view.applyTextInfo(homeInfo)
   }
   func interactorDidFetchImageInfo(homeInfo: HomeImageInfo) {
       view.applyImageInfo(homeInfo)
   }
}
// Создается модуль с помощью builder-метода. Важно заметить, что в архитектуре MVC класс
UIViewController относился к слою Controller, а в VIPER - к слою View.

   private func presentHome(animated: Bool) {
       let viewController = StoryboardScene.Profile.instanciateViewController()
       modulePresenter = Presenter()
       modulePresenter.output = self
```

```
        let interactor = Interactor(container: session.container)
        modulePresenter.interactor = interactor
        interactor.output = modulePresenter
        modulePresenter.view = viewController
        viewController.output = modulePresenter
        navigationController.setViewControllers([homeViewController], animated: animated)
    }
```

При правильном разбиении функционала приложения на модули диаграмма классов может выглядеть следующим образом:

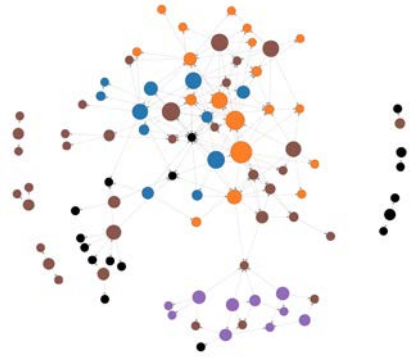

Рис. 4. Диаграмма классов приложения, разработанного с помощью VIPER

Как видно из рис. 4, VIPER-приложение содержит набор классов (которые являются ядром приложения) и набор модулей, каждый из которых является обособленной группой классов. Модуль соединяется с ядром с помощью классов – ModuleBuilder. Таким образом, логика приложения распределяется по модулям, следовательно, уменьшается связанность функционала, упрощается добавление или удаление модулей.

Выводы

В работе рассмотрены наиболее распространенные архитектурные шаблоны для мобильной платформы iOS: MVC и VIPER. Указанные паттерны реализуют разделение ответственностей между классами, что обеспечивает высокую переиспользуемость и гибкость кода. Однако, при масштабировании приложения абстракций MVC недостаточно для того, чтобы разделить функционал между классами и полностью придерживаться принципов SOLID. Использование шаблона VIPER позволяет решить данную задачу. С другой стороны, VIPER предполагает большое количество классов и шаблонного кода, поэтому использование его в небольших проектах, либо в проектах, где нет переиспользования кода, является избыточным.

Перспективы исследований

Большинство мобильных приложений представляют собой Create – Read – Update – Delete (CRUD) клиенты, разработанные для решения повседневных задач: заказ товаров и услуг, обмен графическими и текстовыми данными, новостная агрегация и т.д. В большинстве случаев, решаемые в процессе разработки таких приложений задачи являются однотипными. В связи с этим актуальными являются исследования в направлении проектирования и реализации шаблонного решения. Выбор правильной масштабируемой архитектуры обеспечивает качество программного продукта, а решение этой задачи позволяет значительно сократить затраты на разработку приложений с таким функционалом и увеличить их стабильность за счет тестирования шаблона во многих продуктах.

Список использованной литературы

1. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес: пер. с англ. – С.-Петербург: Питер, 2016. – 368 с.
2. Мартин Р. Быстрая разработка программ. Принципы, примеры, практика / Р. Мартин, Дж. Ньюкирк, Р. Косс: пер. с англ. – М.: Вильямс, 2004 – 752 с.
3. Neuburg M. iOS 9 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics / M. Neuburg- S.: O'Reilly Media, 2015. – 604 p.
4. Фримен Э. Паттерны проектирования / Э. Фримен, К. Сиерра, Б. Бейтс: пер. с англ. – С.-Петербург: Питер, 2016. – 656 с.
5. Zarra M. Core Data: Data Storage and Management for iOS, OS X, and iCloud [Text] / M. Zarra – N.-Y.: Pragmatic Bookshelf, 2013. – 43 с.
6. Бек К. Шаблоны реализации корпоративных приложений: пер. с англ. / К. Бек. – М.: Вильямс, 2000. – 224 с.