Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра программной инженерии

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р физ.-мат. наук, профессор
_____ О. А. Змеев
«30» 05 2019 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

РЕФАКТОРИНГ ПРИЛОЖЕНИЯ PALTALK ДЛЯ ПЛАТФОРМЫ IOS

по основной образовательной программе подготовки магистров
«Управление проектами по разработке программного обеспечения»
направление подготовки 02.04.02 — Фундаментальная информатика и информационные технологии

Иванов Антон Евгеньевич

Научный руководитель ВКР
(магистерской диссертации)
доцент, канд. физ.-мат. наук
_____ В. А. Вавилов
«30» мая 2019 г.

Научный консультант:
_____ Л. С. Иванова
«30» мая 2019 г.

Автор работы
студент группы № 931709
_____ А.Е. Иванов

Томск-2019

Реферат

Выпускная квалификационная работа 47 с., 11 рисунков, 1 таблица, 20 источников.

IOS, IOS SDK, SWIFT, IPHONE, МОБИЛЬНОЕ ПРИЛОЖЕНИЕ, РЕФАКТОРИНГ, XCODE

Объект исследования: мобильное приложение «Paltalk» для платформы iOS.

Цель работы: провести рефакторинг приложения «Paltalk» для платформы iOS.

Результат работы: произведен рефакторинг мобильного приложения «Paltalk» для платформы iOS.

Содержание

Глоссарий

1. **Рефакторинг** — процесс изменения программы, не меняющий ее внешнее поведение с целью упрощения добавления изменений, улучшения читаемости кода проекта и его структуры[7].
2. **Код-ревью** — систематическая проверка исходного кода программы с целью обнаружения и исправления ошибок, которые остались незамеченными в начальной фазе разработки.
3. **User Interface** (по́льзовательский интерфейс) — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.
4. **Фреймворк** — заготовки, шаблоны для программной платформы, определяющие архитектуру программной системы; программное обеспечение, облегчающее разработку и объединение разных модулей программного проекта.
5. **Модуль предметной области (Domain)** — область программного кода, агрегирующая в себе классы и логику необходимую для реализации сценариев использования системы.
6. **Виджет** — элемент интерфейса, являющийся примитивом графического интерфейса пользователя, имеющий стандартный внешний вид и выполняющий стандартные действия.

Введение

В последнее время доля пользователей мобильных устройств на рынке стремительно растет. В силу своей компактности и портативности, смартфоны и умные устройства помогают своим владельцам всегда быть в курсе новостей, общаться со своими знакомыми, обмениваться информацией.

В связи с этим, доля мобильных приложений в разработке программного обеспечения неуклонно увеличивается. Популярность и востребованность того или иного приложения влечет за собой необходимость его доработок и развития. Мир постоянно изменяется, вносит свои корректировки и требования. Задача существующих программных продуктов — соответствовать новым изменениям и требования.

Неверные решения, принятые при проектировании, низкий уровень подготовки разработчиков, необходимость быстрого исправления ошибок и внесения изменений могут стать причиной резкого ухудшения качества кода. Это влечет за собой трудности в модификации и поддержке программного продукта, что, в свою очередь, негативно сказывается на стоимости разработки.

Рефакторинг выполняет задачу улучшения кода продукта, делая его код более читаемым, упрощает добавление нового функционала. Однако, следует учитывать, что внесение изменений в работающий код может привести к появлению ошибок в программе. Для решения данной проблемы необходимо использование модульных тестов.

В данной работе рассматривается приложение «Paltalk» для платформы iOS, которое предоставляет пользователям возможность

общаться в режиме реального времени, обмениваться подарками, совершать звонки, а также осуществлять совместные видео конференции.

Платформа iOS занимает весомую долю рынка мобильных устройств. Пользователей привлекает удобство платформы, ее надежность и качество выпускаемых для нее приложений. Поэтому современные разработчики не могут игнорировать необходимость выпуска своих программных продуктов для платформы iOS. Приходится учитывать особенности платформы, грамотно подходить к проектированию приложений.

Разработка приложения «Paltalk»[15] ведется в течение нескольких лет различными разработчиками на двух языках программирования: Objective-C и Swift. Следствием этого, а также специфики бизнес-процессов в компании является наличие некачественного кода в проекте. Так как процесс развития приложения продолжается, было принято решение провести рефакторинг его исходного кода.

Исходя из вышеизложенного была сформулирована цель данной работы: провести рефакторинг приложения «Paltalk» для платформы iOS.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Выявить проблемные места программного продукта, требующие изменений и улучшений.
2. Определить измеримые показатели эффективности проведения рефакторинга.
3. Подобрать решения для обнаруженных проблем.
4. Зафиксировать поведение текущего функционала в виде тестов.
5. Произвести улучшения кода.

6. Убедиться в работоспособности существующего функционала.
7. Оценить эффективность рефакторинга по установленным ранее измеримым показателям.

1 Рефакторинг исходного кода ПО

1.1 Определение рефакторинга

Помимо определения понятия “рефакторинг”, указанного в глоссарии, следует упомянуть ряд его важных характеристик.

Рефакторинг:

1. Не предназначен для исправления ошибок и добавления новой функциональности.
2. Не меняет видимого поведения программного обеспечения, которое продолжает выполнять прежние функции. Никто — ни конечный пользователь, ни программист — не сможет сказать по внешнему поведению, что что-то изменилось.
3. Не исправляет некорректное поведение программы.
4. Не добавляет нового функционала.
5. Не изменяет поведение программного продукта.
6. Выполняется для улучшения понятности кода или изменения его структуры, для удаления неиспользуемого кода.
7. Обычно производится тогда, когда когда программный код «мутный», непонятный.
8. Обычно выполняется частями.
9. Должны быть определены пределы, границы улучшений.
10. После изменения части программы ее интерфейс не изменился, поменялась только внутренняя часть.

Также стоит отметить, что рефакторинг вносит в архитектуру изменения, облегчающие поддержку и изменение существующего кода.

1.2 Определение необходимости рефакторинга

С течением времени, количество проблем в проекте часто возрастает, и как показывает практика, сами собой они не решатся. Но в большинстве случаев нет возможности сразу начать проводить рефакторинг, ведь могут быть сроки выпуска нового функционала продукта или другие причины, по которым нельзя проводить рефакторинг прямо сейчас.

Признаки, необходимости рефакторинга[9]:

1. Внесение новой функциональности требует неоправданно много времени.
2. Поведение кода является неожиданным.
3. Оценка реализации часто оказывается неверной.
4. Возникает необходимость многочисленных однотипных изменений в разных местах программы.
5. Стоит проводить рефакторинг в следующих ситуациях:
6. При третьем использовании проблемного кода.
7. При разработке нового функционала:

   Рефакторинг помогает быстрее разобраться как функционирует код. Если работа кода, в который нужно добавить новый функционал, недостаточно ясна, рефакторинг позволяет разбить на части и изучить каждую из них, увеличивая общее понимание проекта.

   Рефакторинг облегчает написание нового кода. После рефакторинга добавление нового функционала происходит с меньшими затратами.
8. При исправлении существующих ошибок в поведение программы:

   Из практики следует, что часто сложно понять место и причину неправильной работы программы. Проведение рефакторинга обнаруживает скрытые ошибки в поведении программы.

9. Во время код-ревью:

   Просмотр нового функционала, обычно, является последним этапом перед доставкой продукта к пользователям. Лучше всего проводить ревью вместе с автором кода. В этом случае, можно совместно принять решение, насколько сложно произвести то или иное изменение. При этом, небольшие изменения можно будет осуществлять очень быстро.

Если код сильно запутанный, при этом тратится много времени на его понимание и изменение, то проще и быстрее переписать все с самого начала, нежели подвергать данный код рефакторингу.

Также, следует воздерживаться от рефакторинга перед датой завершения проекта.

1.3 Измеримые показатели рефакторинга

По мнению автора данной работы, прежде чем начать воплощать любой процесс в жизнь, необходимо определить, какой результат работы можно считать успешным. В первую очередь, это необходимо, чтобы не распыляться на второстепенные задачи, на различные детали, а держать фокус на действительно важных вещах для достижения максимального результата. И процесс рефакторинга не исключение.

Прежде всего, необходимо помнить, что цель рефакторинга — это сделать код программы более легким для понимания, без этого рефакторинг нельзя считать успешным.

Рассмотрим следующие измеримые показатели кода, влияющие на уровень понимания его программистом:

1. Количество строк кода в одном классе.
2. Количество строк кода в одном методе.
3. Количество временных переменных.

4. Количество внешних зависимостей у одного класса.
5. Количество обязанностей в одном классе.
6. Количество классов, покрытых модульными тестами.

Количество строк кода в одном классе

Количество строк кода в одном классе является постоянной и наиболее болезненной проблемой каждого разработчика. Когда программист начинает исследовать класс, в котором более тысячи строк, а порой и более трех тысяч строк, то мотивация продолжать что-либо делать моментально пропадает. И это понятно почему:

- Разобраться с обязанностями класса в несколько тысяч строк практически невозможно, либо на это необходимо очень много времени.
- Что-то найти, исправить в большом классе очень сложно.
- Чем больше класс, тем больше изменений в нем происходит, а значит больше разработчиков участвуют в его модификации, что неизбежно ведет к рассогласованию, дублирующей логике и т.д.

Количество строк кода в одном методе

Функции с большим количеством кода создают большие проблемы на пути понимания разработчиком исследуемой области. Очень непросто вносить правки и добавлять новое поведение в такие методы.

Если в теле функции много кода, это уменьшает шансы на то, что она выполняет одну задачу, а значит название функции будет либо очень длинным, либо неполным, вводя в заблуждение.

Очень часто, для выполнения больших функций необходимо, в лучшем случае, большой набор входных параметров, а в худшем большое количество обращений к скрытым зависимостям, например, используя паттерн Одиночка (Singleton)[18].

Таким образом, чем меньше количество строк кода в одной функции, тем понятнее код в целом для программиста.

Количество временных переменных

Много проблем связано с временными переменными, потому что из-за них приходится пересылать массу параметров там, где можно без этого обойтись. Можно легко забыть, для чего эти переменные введены. Эта проблема особенно проявляется при использовании их в длинных методах.

Таким образом, автор приходит к выводу, что чем меньше временных переменных, тем понимание кода выше.

Количество внешних зависимостей у одного класса

Наличие большого количества зависимостей у класса нарушает общепризнанный принцип программирования — Слабая связность (Low Coupling)[16], согласно которому все объекты в системе должны знать друг о друге как можно меньше, т.е. имели как можно меньше внешних зависимостей.

Это дает следующие преимущества:

- Малое число зависимостей между классами.
- Слабая зависимость одного класса от изменений в другом классе.
- Высокая степень повторного использования подсистем.

Количество обязанностей в одном классе

Наличие большого количества обязанностей в одном классе нарушает общепризнанный принцип программирования — Принцип единственной ответственности (Single Responsibility)[6], согласно которому каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в классе. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

Всегда легче анализировать код, догадываться что делает та или иная его функция, когда класс выполняет строго одну задачу.

Таким образом, чем меньше обязанностей у класса, тем выше степень его понимания.

Количество классов, покрытых модульными тестами

Тестирование программного обеспечения – это один из самых важных этапов в процессе создания продукта. Не уделив ему должного внимания, нельзя добиться высокого качества продукта.

Большую часть времени разработчик тратит не на уяснение задачи, ее проектирование, и даже не на ее реализацию, а на отладку написанного кода после создания нового функционала или исправления ошибки, и это время важно всегда учитывать.

Таким образом, чем больше классов покрыто модульными тестами, тем с большей уверенностью мы можем считать, что новое изменение не навредило существующему проекту, а значит, не добавило в него новых ошибок.

2 Проблемы текущей реализации

Для рефакторинга была выбрана часть программного продукта, реализующая функционал чата между пользователями, где они совершают звонки между собой, а также обмениваются сообщениями, фотографиями, стикерами, виртуальными подарками, встроенными в приложение.

При анализе кода рассматриваемого модуля были выявлены следующие проблемы:

2.1 Дублирование кода

Дублирование кода представляет собой один из главных недостатков систем. Программы, в логике которых есть дублирование, трудно модифицировать. Если одно и то же делается в нескольких местах, то когда придется сделать что-то другое, надо будет редактировать больше мест, чем следует.

Причинами возникновения дублирования в коде могут служить как отсутствие коммуникации между членами команды разработки, так и низкая компетентность конкретного разработчика.

Дублирование поведения необходимо исключать. Хотя два дублирующихся метода прекрасно работают в существующем виде, они представляют собой питательную среду для возникновения ошибок в будущем. При наличии дублирования всегда есть риск, что при модификации одного метода второй будет пропущен. Обычно, находить дубликаты трудно.

Увидев одинаковые кодовые структуры в нескольких местах, можно быть уверенным, что если удастся их объединить, программа от этого только выиграет.

Относительно рассматриваемого приложения, проблема дублирования кода проявлялась в двух классах, *ChatViewController_iPad* и

ChatViewController_iPhone. Задача этих классов отображать экран чата для пользователей. Приложение поддерживает как iPhone так и iPad устройства, для этого было создано два этих класса.

В результате:

- 70% кода были идентичны в обоих классах.
- 10% кода выполняли одну и ту же задачу, но были реализованы по-разному.
- 20% кода реализовывали различия в поведении для iPhone и iPad устройств.

Таким образом, в 80% случаев, приходится менять код в обоих классах, что ведет ко всем вышеперечисленным проблемам.

2.2 Громоздкие методы

Программы, использующие короткие методы проще поддерживать. Программистам, не имеющим опыта работы с объектами, часто кажется, что никаких вычислений не происходит, а программы состоят из нескончаемой цепочки делегирования действий. Однако, тесно общаясь с такой программой на протяжении нескольких лет, вы начинаете понимать, какую ценность представляют собой все эти маленькие методы. Все выгоды, которые дает косвенность — понятность, совместное использование и выбор, — поддерживаются маленькими методами.

Практика показывает, что чем длиннее процедура, тем труднее понять, как она работает. Если раньше большая длина метода была оправдана из-за небольших ресурсных мощностей, то сейчас современные устройства имеют достаточно большой запас мощностей, чтобы больше не вынуждать программистов писать громоздкие методы.

Чем метод меньше, тем проще дать ему корректное и лаконичное название, описывающие задачу, которую он выполняет, что безусловно

увеличивает понимание проекта целиком разработчиком, так как правильное подобранное имя метода может избавить от необходимости изучать его реализацию.

Часто, методы с длинной реализацией сопровождаются набором комментариев для внесения ясности. Первым шагом по улучшению такого метода является вынесение кода, требующего комментария, в отдельную функцию. Название метода должно отражать назначение кода, а не то, каким образом он решает свою задачу. Таким подходом можно пользоваться, замещая любое количество строк, даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода.

При проведении рефакторинга исследуемого приложения, автором был замечен громоздкий метод в классе *ChatViewController* длиною в 521 строку кода. Заголовок метода представлен в листинге №1.

Листинг №1:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
     ...
}
```

Данный метод является необходимым для переопределения при создании таблиц с динамическим содержимым (компонент пользовательского интерфейса UITableView[19]). В теле такого метода необходимо создать и настроить ячейки таблицы (UITableViewCell[20]). Большое количество строк в данном методе является естественным

следствием растущей сложности и разнообразия отображаемых в таблице данных.

Объем данного метода является слишком большим для понимания и изменения метода. Пришлось потратить большое количество времени, чтобы разобраться как работает код и какую задачу он выполняет, т.к. название метода не может адекватно говорить, что именно делает метод, потому что неизбежно будет содержать в себе много логики.

Таким образом, следует активнее применять декомпозицию методов.

2.3 Божественный объект

Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него полей и методов. А если класс имеет слишком много полей, то это в скором времени приведет к дублированию кода.

Также, чем больше класс, тем больше задач он выполняет и тем больше разработчиков будут вносить в него свои изменения, что неизбежно приведет к росту ошибок, несогласованности класса.

Как и класс с чрезмерным количеством полей, класс, содержащий слишком много кода, создает питательную среду для повторяющегося кода, и будущих проблем.

В исследуемом проекте, основная часть работы по проведению рефакторинга была сосредоточена в классе *ChatViewController*, так как объем его кода составлял 3241 строчку кода, в котором встретились абсолютно все вышеизложенные проблемы. Данный класс является примером антипаттерна Божественный объект (God Object)[2] - это антипаттерн объектно-ориентированного программирования, описывающий объект, в котором реализуется основная часть функциональности программы. Такого вида объект, обычно, хранит большое количество данных и имеет много

методов, отчего присутствует большое количество использований этого класса по всему коду программы.

2.4 Сложная условная логика

Чаще всего при создания нового приложения, условная логика его методов проста в реализации и в понимании, потому что она состоит не более чем из нескольких строк кода. Но, к сожалению, с увеличением количества функционала в приложении увеличивается и сложность условной логики.

Методы с большим количеством проверок логических переменных — сложная задача для понимания разработчика. Часто эта проблема сопровождается плохим именованием булевских переменных, что еще сильнее затрудняет понимание.

В исследуемом коде была обнаружена функция со сложной условной логикой, в которой очень тяжело было разобраться. Такая функция представлена в листинге №2.

Листинг №2:

```
  @objc func checkPrivacy() -> Bool {
        let appDelegate = AppDelegate.sharedInstance()

        let status = UserStatus(csUserStatus:
appDelegate.myProfileHandler.status)

        guard status.privacyEnabled else {
            return true
        }

        // ugly anonymous check
        guard imSession!.opponent.userId.nick != "" else {
            return true
        }
```

```
        guard
appDelegate.contactListService.contactListLoaded else {
            return true
        }

        guard
!appDelegate.contactListService.containsUser(imSession!.opp
onent.nick, in: .LIST_CONTACT_LIST) else {
            return true
        }

        let noWarning =
appDelegate.userSettings.bool(forKey:
CFCI_SETTINGS_NOCONFIRMATION_PRIVACYTOSTRANGER)
        guard !noWarning else { return false }

        actionSheetType = .privacyStranger
        let actionSheet = UIActionSheet(title:
Localized.systemUserCantReplyWhileYouInPrivate,
                delegate: self,
                cancelButtonTitle: Localized.no,
                destructiveButtonTitle:
Localized.dontAskMeAgain,
                otherButtonTitles: Localized.yes)

        delegate?.chatLogControllerShow(actionSheet);

        return false
    }
```

В приведенном выше листинге по очереди проверяется пять логических переменных. При изменении такого метода, действительно, было

просто ошибиться в одной или нескольких проверках, например, забыв поставить отрицание перед какой-либо переменной.

В первую очередь было сложно понять, что именно проверяет метод, так как имена переменных не говорили сами за себя. Приходилось обращаться к другим разработчикам данного проекта, чтобы выяснить, что каждая из них означает.

В итоге было потрачено относительно много времени, чтобы стало понятно, при каком состоянии приложения, какое действие вызывается, что, естественно, негативно повлияло на скорость рефакторинга.

2.5 «Лодочный якорь»

Антипаттерн «Лодочный якорь»[8] описывает проблему в коде проекта, при которой существует неиспользуемые части системы, которые остались после оптимизации или рефакторинга.

Часто после рефакторинга кода, некоторые части кода остаются в системе, хотя они уже больше не используются. Также такие части кода могут быть оставлены на будущее, на случай, если придётся ещё их использовать. Такой код только усложняет систему, не неся абсолютно никакой практической ценности. Эффективным методом борьбы с лодочными якорями является рефакторинг кода для их устранения, а также процесс планирования разработки, с целью предотвращения возникновения якорей.

Часто неиспользуемый код, появляется когда требования к программному продукту изменились, либо были внесены какие-то корректировки, но удаление старого кода так и не была проведена.

Неиспользуемый код можно обнаружить и в сложном условном коде, где одна из ветвлений условий никогда не может быть исполнена, в виду ошибки или другого стечения обстоятельств.

В рассматриваемом приложении была и такая проблема. При появления нового сообщения, в верхней части экрана отображалось окно, в котором показывалось количество непрочитанных сообщений. За данный функционал отвечал класс *ChatNotificationView*.

Ранее, в системе класс *ChatNotificationView* был заменен системной нотификацией, обработкой которой занимается операционная система iOS.

Таким образом, вся логика для создания, отображения, наполнения, и обработки *ChatNotificationView* больше не требовалась, но существовала и затрудняла поддержку части приложения, где она использовалась.

2.6 Ленивый класс

Чтобы сопровождать каждый создаваемый класс и разобраться в нем, требуются определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданно в свое время, но уменьшившийся в результате рефакторинга, либо, добавленный для планировавшейся модификации, которая не была осуществлена. Такой класс является примером антипаттерна Ленивый класс [4].

Так в приложении был замечен класс, который был создан только для выполнения одного метода, состоящий из двух строк. Этот класс представлен в листинге №3.

Листинг №3:

```
class ContactIconPresenter: NSObject {
    @objc
    func reuseView(_ view: ContactIconView, item:
ContactListItem) {
        view.userStatus = UserStatus(csUserStatus:
item.status)
        view.viewModel = ContactIconViewModel(contactItem:
```

```
item)
    }
}
```

Таким образом, класс больше вводил в заблуждение, нежели помогал, что затрудняет как чтение кода, так и его поддержку.

2.7 Реализация большого количества ролей

Реализация классом большого количества задач является общепризнанной плохой практикой, выраженных в антипаттернах «Божественный объект» и «Швейцарский нож», которые описывают ситуацию, когда основная часть функциональности программы кодируется в одном объекте. Так как этот объект хранит большое количество данных и имеет много методов, его роль в программе становится сильно большой.

Вместо того, чтобы общаться друг с другом непосредственно, другие объекты полагаются на такие классы. Так как на классы с большим количеством выполняемых задач ссылается много кода, то его обслуживание, внесение изменений становится сложным, а также велик риск сломать существующую функциональность.

В данной работе существует пример такого класса — класс *ChatViewController*. Данный класс реализовывал задачу получения и отображения данных из сервера или хранилища данных, обработки этих данных, выполнял бизнес логику, реагировал на пользовательские действий, — отчего код становится настолько сложным, что изменить его функциональность становится большой проблемой.

Такой объект является большим и неповоротливым для поддержания своего существования, хранит большое количество данных и имеет много методов, является монолитом, претерпевает постоянные изменения со

стороны разработчиков. В нем сложно разбираться, и осуществлять поиск нужной части, отвечающий за какой либо функционал.

2.8 Большое количество внешних зависимостей

Когда у объекта есть зависимости, то это означает, что он не может функционировать без них. То есть, чем больше зависимостей, тем сложнее классу существовать.

Из практики следует, что чем больше у класса зависимостей, тем больше задач он выполняет, что ведет к ухудшению пониманию и поддержанию кода, то есть ведет к проблеме, описанной в главе «Реализация большого количества ролей».

Чем больше у класса зависимостей, тем сложнее покрыть этот класс модульными тестами, так как придется для тестирования реализовывать все эти зависимости, что может быть неоправданной тратой времени.

Стоит отметить, что внешние зависимости могут быть явными и неявными. Явные зависимости задаются через конструктор класса, либо объявлены в интерфейсе класса. Неявные зависимости сложно или даже невозможно обнаружить, анализируя интерфейс класса, так как, как правило, они используются напрямую в реализации класса, например, через паттерн Одиночка (Singleton).

Неявные зависимости несут множество проблем и неудобств. Нельзя быть уверенным, что изменение класса, реализующий паттерн Одиночка, не затронет класс с неявными зависимостями, ведь тогда придется изменять и такой класс тоже, что может оказаться незапланированным изменением, и привести к нарушениям сроков реализации.

В данной работе класс ChatViewController обладает семнадцатью внешними зависимостями, где 7 из них являются неявными. При просмотре такого класса, нельзя сразу сказать, что ему необходимо для его

функционирования, что может существенно усложнить процесс его переноса или переиспользования в дальнейшем. Класс ChatViewController с его зависимостями, можно увидеть на диаграмме, представленной на рис №2.

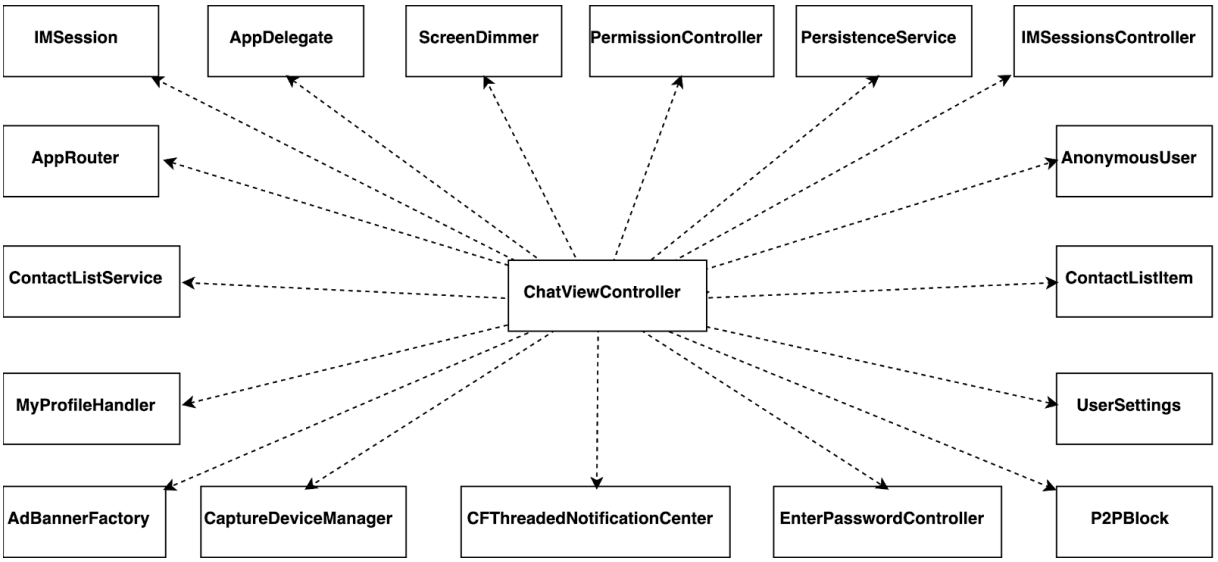


Рис №2: Диаграмма зависимостей класса ChatViewController.

2.9 Зависимость от деталей реализаций, а не от абстракций

Когда программа зависит от конкретного экземпляра объекта, создается зависимость от кода, что ограничивает возможность повторного использования этого объекта.

Зависимость объекта от конкретного класса всегда будет добавлять ограничения для этого объекта, т.к. такую зависимость нельзя заменить при необходимости другой реализацией. Однако, в случае, если объект зависит от интерфейса, то он сможет взаимодействовать с любым классом, который реализует необходимый интерфейс, тем самым не накладывая на себя жесткие ограничения.

Таким образом, практика показывает, что лучше выстраивать зависимость от интерфейса, где это возможно и не заставлять код программы зависеть от конкретных элементов. Ведь, чем меньше зависимостей, тем больше возможностей повторного использования кода, в то время как больше

число зависимостей, означает и больше число требований, что уменьшает возможность повторно использовать код.

В рассматриваемом приложении, класс *ChatViewController* зависел от конкретного фреймворка, реализующий работу с конкретным хранилищем данных, что является огромной проблемой, в случае необходимости перейти на другой способ хранения данных.

2.10 Нет возможности протестировать код модульными тестами

Модульное тестирование — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы[3].

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже проверенных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Автоматические тесты дают уверенность, что программа работает как задумано. Такие тесты можно запускать многократно. Успешное выполнение тестов покажет разработчику, что его изменения не нарушили существующее поведение программы.

Тест, не прошедший проверку, позволит обнаружить, что в коде сделаны изменения, которые меняют или нарушают поведение программы. Исследование ошибки, которую выдает тест, не прошедший проверку, и сравнение ожидаемого результата с полученным, даст возможность понять, где возникла ошибка, будь она в коде или в требованиях.

Модульные тесты могут служить в качестве документации к коду. Грамотный набор тестов, который покрывает возможные способы

использования, ограничения и потенциальные ошибки является хорошей опорой для дальнейшего исследования кода программы.

Кроме того, модульное тестирование является необходимым этапом при проведении рефакторинга исходного кода, поскольку позволяет удостовериться, что внесенные изменения в процессе рефакторинга не нарушили работу приложения.

В программе для класса ChatViewContrller-а отсутствует возможность использования модульных тестов, так как он использует неявные зависимости, а значит, их нельзя задать извне при тестировании.

3 Используемые методы рефакторинга

3.1 Рефакторинг дублирования кода

Для решения проблемы дублирования кода, которая проявлялась в двух классах, ChatViewController_iPad и ChatViewController_iPhone, задача которых отображать экран чата для пользователей, была применена следующая последовательность действий:

1. Объединение классов ChatViewController_iPad и ChatViewController_iPhone в один — ChatViewController.
2. Выявление методов с одинаковой логикой с последующим удалением дублирующей реализации.
3. Добавление проверок для выполнения различающегося функционала для случаев, когда необходимо выполнить определенный код для устройства типа iPhone или iPad.
4. Удаление неиспользуемого кода, который остался в ходе слияния двух классов ChatViewController_iPad и ChatViewController_iPhone.

3.2 Рефакторинг антипаттерна «Лодочный якорь»

Для решении проблемы с отсутствием использования класса ChatNotificationView, который был заменен системной нотификацией, обработкой которой занимается операционная система iOS, был проделан следующий перечень действий:

1. Поиск и удаление переменных, используемых для работы с ChatNotificationView. Для этого необходимо проверить исследуемые методы на переопределение их в суперклассе или подклассе. Если таковы имеются, нужно проанализировать, используется ли там параметр. Если в какой-то из реализаций

параметр используется, необходимо воздержаться от удаления переменной.

2. Поиск и удаление методов, тем или иным образом, обращающиеся к ChatNotificationView, но которые теперь больше не используются.
3. Поиск и удаление объектов, которые используются только в классе ChatNotificationView.
4. После удаления всех внутренних зависимостей в пункте 3, необходимо произвести удаление класса ChatNotificationView.

3.3 Улучшение архитектуры

При дальнейшем исследовании работы, стало понятно, что приведенные ниже проблемы были сосредоточены в одном классе, ChatViewController-е:

1. Громоздкие методы.
2. Большие классы.
3. Сложная условная логика.
4. Реализация большого количества ролей.
5. Большое количество внешних зависимостей.
6. Зависимость от деталий реализация, а не от абстракций.
7. Нет возможности протестировать код модульными тестами.

Причина возникновения многих из них следует из большого количества задач, которые выполняет данный класс. ChatViewController реализует задачу получения и отображения данных из сервера или хранилища данных, занимается обработкой этих данных, выполняет правила бизнес логики, а также реагирует на пользовательские действия.

Таким образом, прежде чем предпринимать действия по устранению каждой проблемы в отдельности, было принято решение произвести

изменения в архитектуре проекта. Решением вышеперечисленных проблем может стать создание отдельных сущностей для выполнения относительно независимых задач, используя принципы чистой архитектуры, которые описаны в книге Роберта Мартина под названием «Чистая архитектура. Искусство разработки программного обеспечения». Автор данной книги утверждает, что архитектура должна[11]:

- Быть тестируемой:

  Функционал должен быть покрыт автоматизированными тестами, а не проверяться вручную каждый раз при добавлении новых изменений.

- Не зависеть от UI:

  Изменения касаемые графического представления для пользователя не должны затрагивать все приложения, и минимально влияют на программу в целом и не требовать больших временных затрат для изменения дизайна.

- Не зависеть от типа хранилища данных, внешних фреймворков и библиотек:

  Объекты должны зависеть от интерфейсов, которые исключают прямой зависимости от конкретных реализаций, например при использовании конкретного типа хранилища данных. Это правильно поможет безболезненно и быстро перейти с одного вида хранения данных к другому или какой-нибудь другой framework на другой, при необходимости.

- Быть разделена на слои:

  Выделяются слои в проекте с четко обозначенными обязанностями. Каждый класс созданный в проекте должен относиться к какому-либо слою. Такое разделение поможет повысить зацепление

объектов и предотвратить волнообразное изменение по всему проекту.

- Следовать правилу инверсии зависимостей:

  Правило инверсии зависимостей позволяет самым высокоуровневым слоям быть независимыми от низкоуровневых слоев. При необходимости использовать код в высокоуровневом слое из низкоуровневого, необходимо определить интерфейс в высокоуровневом слое, который уже будет реализовывать объект из низкоуровневого слоя. Таким образом, логика которая лежит в низкоуровневом слое будет выполняться в высокоуровневом слое, при сохранении независимости высокоуровневого низкоуровневого слоя.

3.3.1 Применение паттерна «Внедрение зависимостей»

Реализация экрана чата была разделена на следующие модули, как показано на рисунке №3.

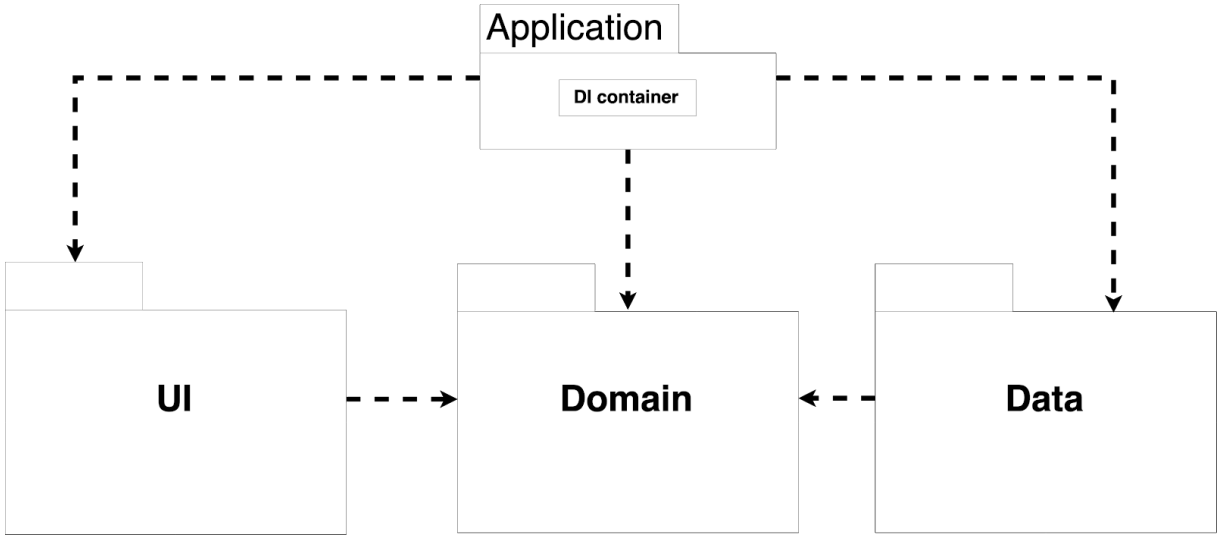


Рис №3: Разделение на модули экрана чата.

UI модуль — это фрагмент программы, реализующий пользовательский интерфейс, с отображением данных. Содержит в себе

логику обработки изменений данных приложения, а также логику обработки действий пользователя.

Domain модуль — это фрагмент программы, содержащий всю логику, ценную для бизнеса. Реализует бизнес-логику программы, инкапсулирует доменные объекты и всю необходимую для программы информацию об объекте предметной области.

Data модуль — это фрагмент программы, отвечающий за хранение данных в хранилище данных, их кэширование, а также получение данных из удаленных источников информации, таких как веб-сервис или сервер.

Для связывания классов между собой был использован паттерн Внедрение зависимостей[13], агрегирующий в себе всю логику по настройке объектов и установлению в них необходимых зависимостей. Он находится в модуле Application, чтобы знать обо всех модулях и собирать их воедино.

Ранее, UI модуль был представлен в виде одного класса — ChatViewController, в котором был реализован весь необходимый функционал для этого модуля. Теперь задачи выполняющиеся в UI слое выполняются не одним классом, а совокупностью классов, как это представлено на рисунке №4.

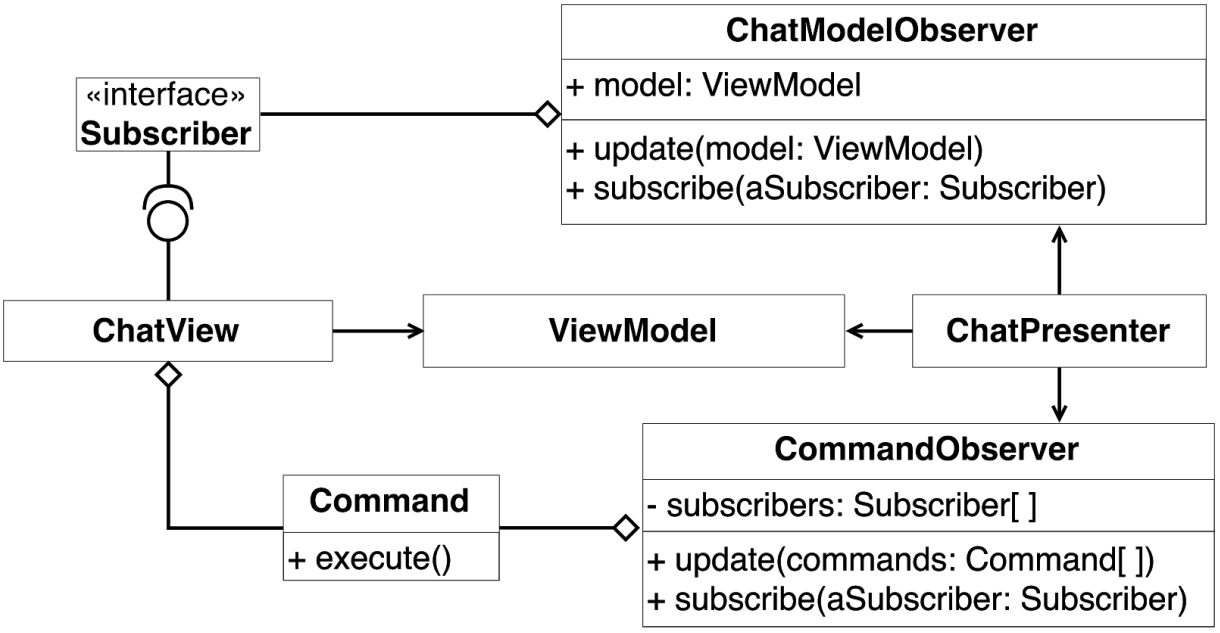


Рис №4. Пакет классов UI слоя.

3.3.2 Применение паттерна «MVP»

Для выделения обязанностей и объектов, был использован паттерн Model-View-Presenter (MVP), разделяющий модуль на *ChatView*, *ViewModel*, *ChatPresenter*[14].

Для наглядного разделения, используемые классы выделены на рисунке №5.

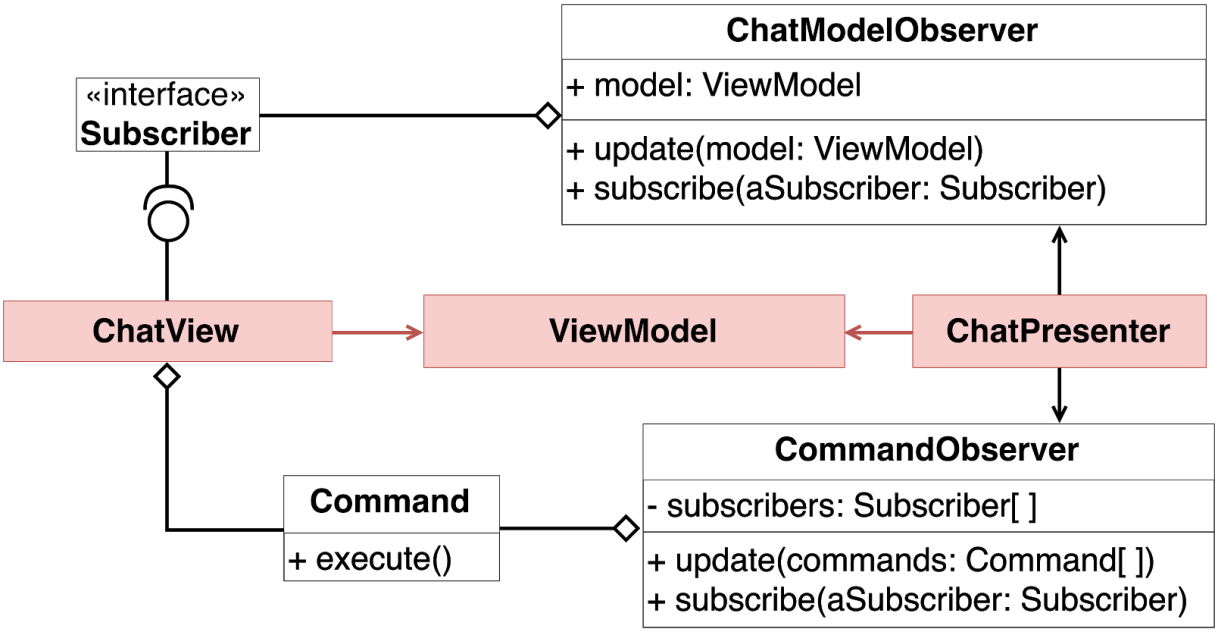


Рис №5: Применение паттерна Model-View-Presenter (MVP).

ViewModel — программный класс, агрегирующий в себе данные приложения и логику их получения и сохранения. Зачастую она основана на использовании хранилища данных или на данных полученных от веб-сервисов. В некоторых случаях потребуется ее адаптировать, изменить или расширить.

ChatView — представляет собой форму с виджетами. Пользователь может взаимодействовать с ее элементами, но когда какое-нибудь событие виджета будет затрагивать логику интерфейса, ChatView будет направлять его презентеру.

ChatPresenter — презентер содержит всю логику пользовательского интерфейса и отвечает за синхронизацию модели и представления. Когда представление уведомляет презентер, что пользователь что-то сделал, например, нажал кнопку, презентер принимает решение об обновлении модели и синхронизирует все изменения между моделью и представлением.

3.3.3 Применение паттерна «Наблюдатель»

Чтобы обеспечить низкую связность между ChatPresenter-ом и ChatView, но при этом уведомлять ChatView об изменениях модели, применен паттерн «Наблюдатель», который хранит в себе данные графического представления — ViewModel. При необходимости изменить данные на графическом представлении, ChatPresenter обновляет данные в объекте наблюдателе — ChatModelObserver, который в свою очередь уведомляет ChatView, так как он реализует интерфейс Subscriber, о необходимости обновить данные, передавая новые данные для отображения[10].

Наглядная диаграмма применения данного паттерна представлена на рисунке №6.

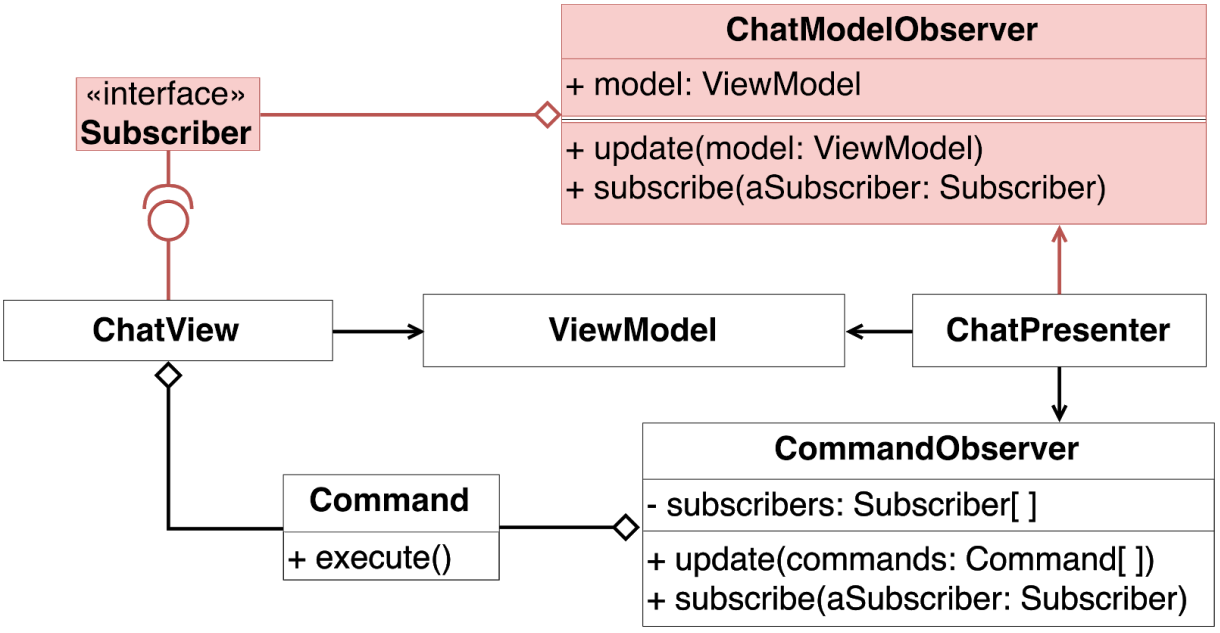


Рис №6: Применение паттерна «Наблюдатель».

3.3.4 Применение паттерна «Команда»

Чтобы избавиться от прямой зависимости между *ChatView* и *ChatPresenter*-ом, например, при нажатии пользователем на кнопку, применен паттерн «Команда»[1]. *ChatPresenter* реализует перечень команд, которые могут понадобиться представлению для реакции на пользовательские действия, а представление при помощи наблюдателя — *CommandObserver* их получает, по описанному выше принципу сохраняет себе и вызывает при необходимости тем самым совершая какое-либо действие, реализация которого находится в *ChatPresenter*.

Диаграмма применения паттерна «Команда» изображена на рисунке №7.

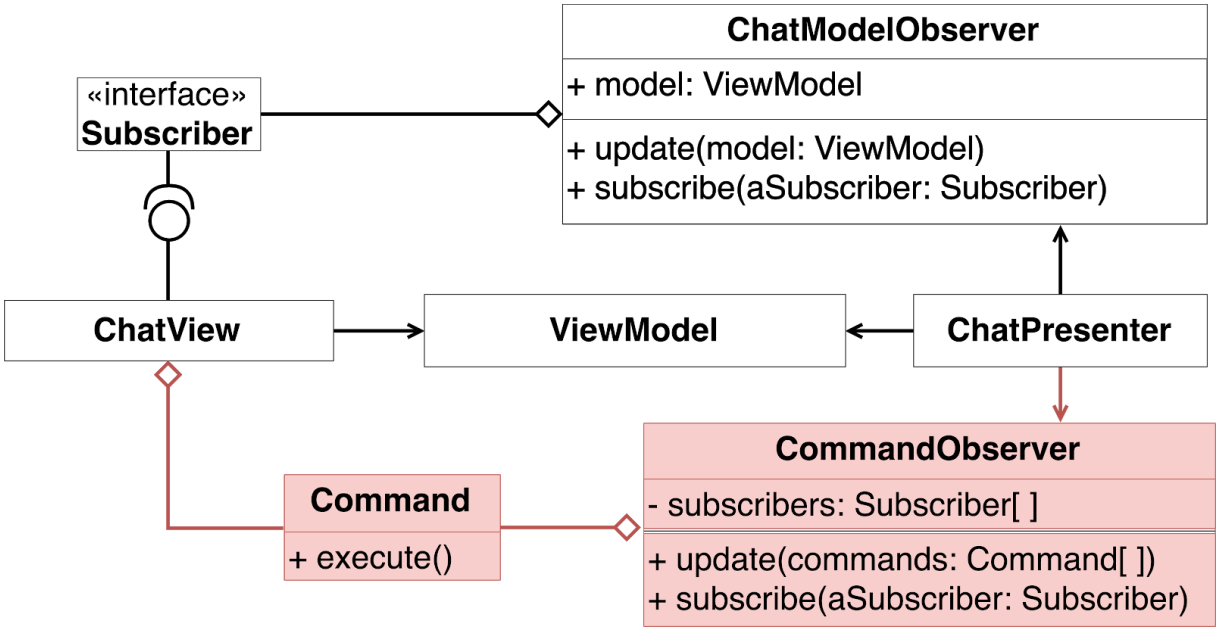


Рис №7. Применения паттерна «Команда».

3.3.5 Применение паттерна «Посредник»

ChatPresenter-у необходимо получать данные о сообщениях пользователей, статусе участника беседы, возможности позвонить пользователю или отправить изображение. Все эти данные находятся в различных сервисах. Чтобы *ChatPresenter* зависел от абстракций, а не от конкретных реализаций, и избавиться от необходимости явно ссылаться презентеру на сервисы, был применен паттерн «Посредник», который позволяет уменьшить связанность множества сервисов и презентера, благодаря перемещению этих связей в один класс — ChatServiceMediator, который, в свою очередь, передаст запрос, направленный из презентера конкретному сервису, который отвечает за эту задачу[5].

Для наглядности, схема применения паттерна изображена на рис №8.

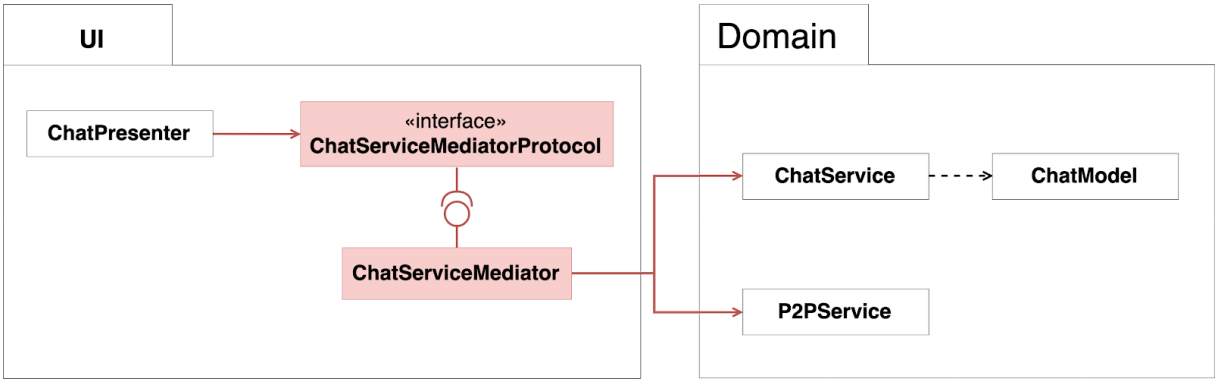


Рис №8: Применение паттерна «Посредник».

3.3.6 Применение паттерна «Репозиторий».

Так как модуль предметной области (Domain) реализует бизнес правила, то в нем нет логики получения данных. Поэтому получает он их из Data модуля. Для изоляции логики хранения данных, сокрытия деталей реализации и предоставления простого API, используется паттерн «Репозиторий» в том виде, в котором описывает его Мартин Фаулер в своих работах[17].

Диаграмма применение данного изображена на рисунке №9.

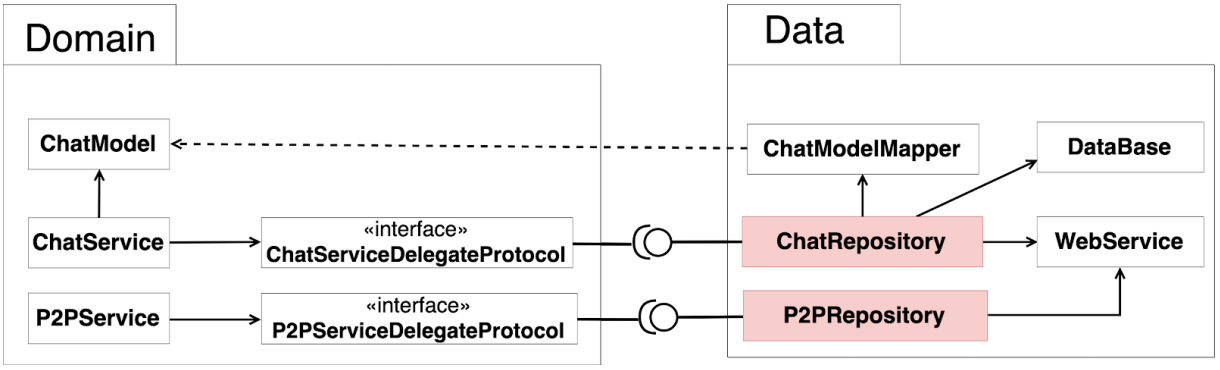


Рис №9: Применение паттерна «Репозиторий».

ChatRepository — это программный класс, инкапсулирующий в себе всё, что относится к способу хранения сообщений пользователей. Хорошей практикой считается создание отдельных репозиториев для каждого

бизнес-объекта или контекста, поэтому данные связанные с P2P звонками вынесены в отдельный P2PRepository.

Таким образом, приложение может внутри репозитория получать данные с сервера, или же получать информацию из хранилища данных, а также может поменять вид хранения данных без ущерба для всего приложения, при необходимости.

3.3.7 Применение паттерна «Объект передачи данных»

Так как все модули зависят от модуля предметной области (Domain), то все сущности на уровне приложения расположены именно в нем. Для удобства обмена данными между слоями применен паттерн «Объект передачи данных», содержащий в себе все необходимые данные, поддающиеся сериализации и не содержащий в себе логику[12].

Диаграмма применения данного паттерна представлена на рисунке №10.

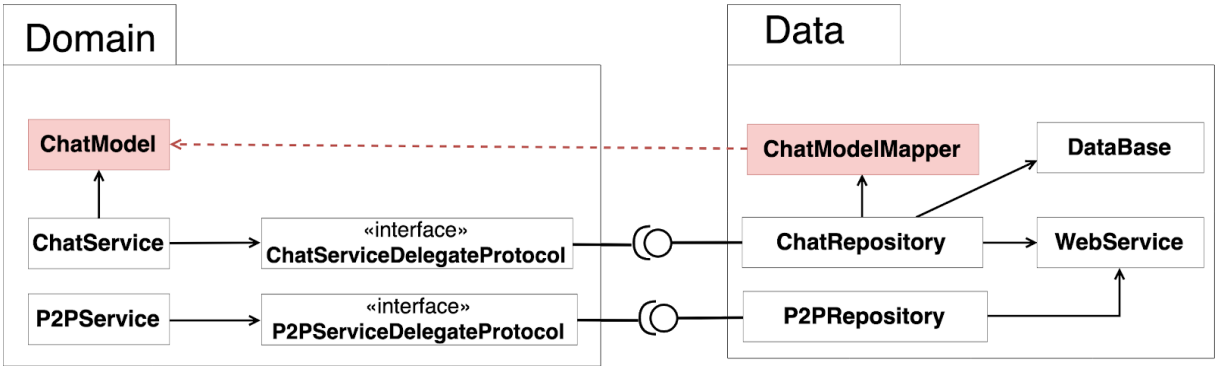


Рис №10: Применение паттерна «Объект передачи данных».

3.3.8 Применение правила инверсии зависимостей

Как можно было заметить на диаграмме разделения на модули на рисунке №3, модуль предметной области (Domain) использует Data модуль для получения данных, но не зависит от него напрямую, так как это самая высокоуровневая часть приложения, которая не должна ни от чего зависеть.

Чтобы добиться такой независимости и при этом осуществлять взаимодействия модуля предметной области с Data модулем, применено правило инверсии зависимости[13], благодаря которому создан интерфейс — ChatServiceDelegateProtocol, описывающий все необходимые методы для получения данных для модуля предметной области. Особенность заключается в том, что ChatServiceDelegateProtocol находится в модуле предметной области, но реализуется он в модуле Data, классом ChatRepository. Таким образом, модуль предметной области использует Data модуль, получая из него список сообщений с пользователями, но ничего про него не знает. Только при сборке приложения, модуль предметной области получит конкретную реализацию из Data благодаря описанному паттерну «Внедрение зависимостей».

Диаграмма применения данного паттерна представлена на рисунке №11.

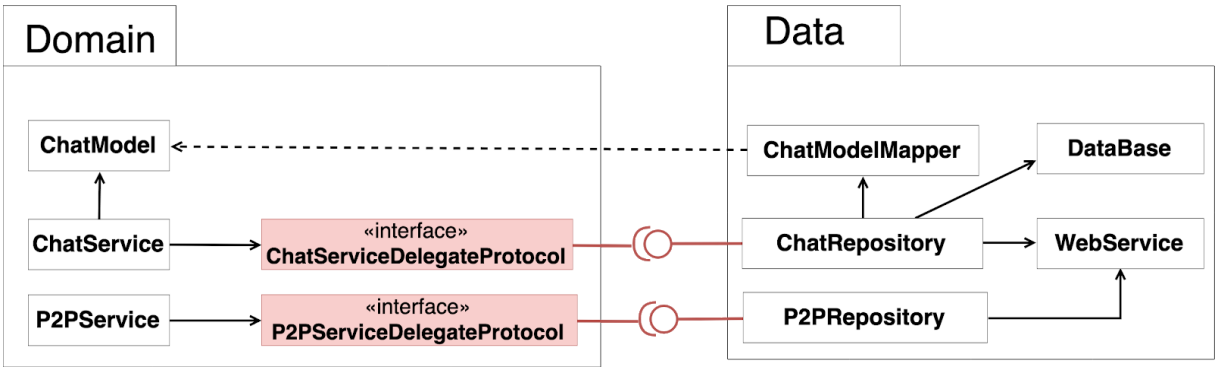


Рис №11: Применение правила инверсии зависимостей.

Применив все описанные выше улучшения к архитектуре в процессе рефакторинга, были решены оставшиеся проблемы.

Таким образом, все выявленные проблемы были решены.

4. Результаты рефакторинга

В главе «Измеримые показатели рефакторинга», автором были сформулированы показатели, которые помогут проанализировать насколько полезно и успешно был произведен рефакторинг.

1. Количество строк кода в одном классе

Максимальное количество строк кода:

- **Изначально**: 3241
- **Результат рефакторинга**: 656

Изначально существовал один большой класс под названием ChatViewController, состоящий из 3241 строк кода. После рефакторинга, часть кода из этого класса была переработана, улучшена и вынесена в другие классы. Название этих классов и количество строк кода, которые они занимают, представлены в таблице №1.

Таблица №1: Классы исследуемого модуля с количеством строк после рефакторинга.

Название класса	Количество строк кода
ChatViewController	656
ChatPresenter	548
ChatRouter	142
ChatServiceProvider	270
ChatService	360
ChatServiceDelegate	238
ChatServiceErrors	43
P2PService	126

Заключение: Рефакторинг выполнен эффективно по данному показателю.

2. Количество строк кода в одном методе

Максимальное количество строк кода:

- **Изначально**: 521
- **Результат рефакторинга**: 54

Заключение: Рефакторинг выполнен эффективно по данному показателю.

3. Количество временных переменных

- **Изначально**: 7
- **Результат рефакторинга**: 0

Заключение: Рефакторинг выполнен эффективно по данному показателю.

4. Количество внешних зависимостей у одного класса

- **Изначально**: 17
- **Результат рефакторинга**: 4

До рефакторинга класс ChatViewController имел 17 зависимостей, после рефакторинга удалось заменить 17 зависимостей одной — класс ChatPresenter.

Ниже представлен список сущностей, от которых ранее зависел класс ChatViewController:

1. AppDelegate
2. ScreenDimmer
3. IMSession
4. AppRouter
5. CaptureDeviceManager
6. MyProfileHandler
7. AdBannerFactory
8. AnonymousUser

9. EnterPasswordController
10. CFThreadedNotificationCenter
11. ContactListItem
12. UserSettings
13. P2PBlock
14. ContactListService
15. IMSessionsController
16. PersistenceService
17. PermissionController

Заключение: Рефакторинг выполнен эффективно по данному показателю.

5. Количество обязанностей в одном классе

Максимальное количество обязанностей в классе:

- **Изначально**: 7
- **Результат рефакторинга**: 2

До рефакторинга класс ChatViewController выполнял 7 обязанностей:

1. Управление интерфейсом и обработка действий пользователя.
2. Осуществление P2P звонков.
3. Получение данных из сети.
4. Взаимодействие напрямую с фреймворками, конкретными типами хранилищ данных, конкретными реализациями, а не абстракциями.
5. Реализация правил бизнес логики.
6. Трансформация данных модуля предметной области для данных UI модуля.
7. Навигация между экранами.

После рефакторинга класс ChatViewController стал выполнять лишь только одну обязанность: *Управление интерфейсом и обработка действий пользователя.*

Но стоит отметить, что в данной реализации выбранного архитектурного решения, класс ChatPresenter совмещает в себе две обязанности: *Реализация правил бизнес логики* и *Трансформация данных модуля предметной области для данных UI модуля.*. Поэтому, максимальное количество обязанностей в одном классе равно двум.

Заключение: Рефакторинг выполнен эффективно по данному показателю.

6. Количество классов, покрытых модульными тестами

Количество классов покрытых тестами в исследуемом модуле:

- **Изначально**: 0
- **Результат рефакторинга**: 5

После рефакторинга, были покрыты модульными тестами следующие классы:

1. ChatViewController
2. ChatPresenter
3. ChatService
4. P2PService
5. ChatLogDelegate

Заключение: Рефакторинг выполнен эффективно по данному показателю.

Заключение

В результате выполненной работы был произведен рефакторинг приложения «Paltalk» для платформы iOS. Все поставленные задачи были выполнены. Достигнутые значения измеримых показателей эффективности проведения рефакторинга показали, что рефакторинг произведен успешно. Результат работы был зафиксирован актом о внедрении (приложение №1), а произведенные изменения были внесены в приложение и опубликованы в магазине.

Список источников

1. Фаулер М., Архитектура корпоративных программных приложений // М.Фаулер. — М.: Издательский дом "Вильяме", 2006. — 544 с.
2. Божественный объект [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://en.wikipedia.org/wiki/Божественный_объект (дата обращения 22.03.2019)
3. Модульное тестирование [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://en.wikipedia.org/wiki/Модульное_тестирование (дата обращения 30.04.2019)
4. Гамма Э., Приемы объектно—ориентированного проектирования // Э.Гамма. Р.Хелм, Р.Джонсон. — СпБ: Питер, 2001. — 368 с.
5. Ларман К., Применение UML и шаблонов проектирования. –2 изд. – М.: Издательский дом «Вильямс», 2004. – 624 с.
6. Принцип единственной ответственности [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://en.wikipedia.org/wiki/Принцип_единственной_ответственности (дата обращения 20.02.2019)
7. Рефакторинг. [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://ru.Wikimedia.org/wiki/Рефакторинг (дата обращения 23.03.2019)
8. Кериевский Д., Рефакторинг с использование шаблонов // Д.Кериевский. — М.: Издательский дом "Вильяме", 2006. — 400 с.
9. Фаулер М., Рефакторинг: улучшение существующего кода // М.Фаулер. — СПБ: Символ-Плюс, 2003. — 432 с.

10. Якобсон А., Буч Г., Рамбо Дж., Унифицированный процесс разработки программного обеспечения. – 3 изд. – СПб.: Питер-принт, 2002. – 492с.
11. Мартин Р., Чистая архитектура. Искусство разработки программного обеспечения // М.Фаулер. — СПб.: Питер, 2018. — 352 с.
12. Data Transfer Object. [Электронный ресурс] // Martinfowler. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html (дата обращения 25.04.2019)
13. Inversion of Control Containers and the Dependency Injection pattern. [Электронный ресурс] // Martinfowler URL: https://martinfowler.com/articles/injection.html (дата обращения 20.03.2019)
14. Model-View-Presenter [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://en.wikipedia.org/wiki/Model-View-Presenter (дата обращения 12.02.2019)
15. Paltalk - Group Video Chat App. [Электронный ресурс] // Apple, Inc. URL:https://itunes.apple.com/ru/app/paltalk-group-video-chat-app/id466970942?mt=8 (дата обращения 30.04.2019)
16. Reducing Coupling. [Электронный ресурс] // Martinfowler. URL: https://www.martinfowler.com/ieeeSoftware/coupling.html (дата обращения 25.04.2019)
17. Repository. [Электронный ресурс] // Martinfowler. URL: https://www.martinfowler.com/ieeeSoftware/repository.html (дата обращения 20.04.2019)

18. Singleton pattern. [Электронный ресурс] // Wikimedia Foundation, Inc. URL: https://en.wikipedia.org/wiki/Singleton_pattern (дата обращения 14.04.2019)
19. UITableView. [Электронный ресурс] // Apple, Inc. URL:https://developer.apple.com/documentation/uikit/uitableview (дата обращения 27.02.2019)
20. UITableViewCell. [Электронный ресурс] // Apple, Inc. URL:https://developer.apple.com/documentation/uikit/uitableviewcell (дата обращения 28.02.2019)

ПРИЛОЖЕНИЕ 1

СПРАВКА

о внедрении результатов выпускной квалификационной работы (магистерской диссертации)

на тему РЕФАКТОРИНГ ПРИЛОЖЕНИЯ PALTALK ДЛЯ ПЛАТФОРМЫ IOS

Выдана студенту 2 курса очной формы обучения магистратуры Института прикладной математики и компьютерных наук Томского Государственного Университета
наименование высшего учебного заведения (полностью)

Иванову Антону Евгеньевичу
Фамилия, имя, отчество

в том, что в практику работы ООО «ТОМСКСОФТ»
наименование организации

в 2019 году внедрены следующие результаты (выводы, рекомендации) выпускной квалификационной работы (магистерской диссертации): Изменение кодовой базы мобильного приложения "Paltalk" для платформы iOS, произведенные А.Е. Ивановым в ходе рафакторинга.

Руководитель организации ООО «ТОМСКСОФТ» И.В.Безходарнов
наименование организации *подпись, печать*

МП
ТомскСофт
ИНН 7017049173
ТОМСК
РОССИЯ
ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ

АНТИПЛАГИАТ
ТВОРИТЕ СОБСТВЕННЫМ УМОМ

Отчет о проверке на заимствования №1

Автор: iae95@yandex.ru / ID: 4368889
Проверяющий: (iae95@yandex.ru / ID: 4368889)

Отчет предоставлен сервисом «Антиплагиат»- http://users.antiplagiat.ru

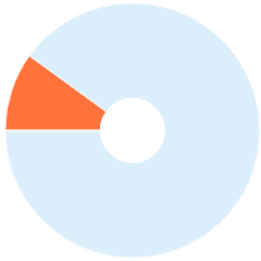
ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 30
Начало загрузки: 03.06.2019 21:32:14
Длительность загрузки: 00:00:09
Имя исходного файла: Диссертация - Рефакторинг
Размер текста: 1264 кБ
Символов в тексте: 50414
Слов в тексте: 6074
Число предложений: 433

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Последний готовый отчет (ред.)
Начало проверки: 03.06.2019 21:32:23
Длительность проверки: 00:00:04
Комментарии: не указано
Модули поиска: Модуль поиска Интернет

ЗАИМСТВОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
10,12%	0%	89,88%



Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Доля в тексте	Источник	Ссылка	Актуален на	Модуль поиска	Блоков в отчете	Блоков в тексте
[01]	2,35%	2,35%	фаулер м.рефакторинг.улуч...	http://inethub.olvi.net.ua	23 Апр 2014	Модуль поиска Интернет	11	11
[02]	0,4%	1,72%	Лабораторная работа №4 Ц...	http://rerefat.ru	23 Мар 2016	Модуль поиска Интернет	3	12
[03]	1,07%	1,07%	фаулер м.рефакторинг.улуч...	http://inethub.olvi.net.ua	23 Апр 2014	Модуль поиска Интернет	3	3

Еще источников: 13
Еще заимствований: 6,32%