**«Санкт-Петербургский государственный электротехнический университет**

**«ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ «ЛЭТИ»)**

|  |  |  |
| --- | --- | --- |
| **Направление** | 09.03.02 – Информационные системы и технологии | |
| **Профиль** | Название программы (профиля, специализации) | |
| **Факультет** | КТИ | |
| **Кафедра** | ИС | |
| *К защите допустить* |  | |
| Зав. кафедрой |  | Цехановский В.В. |

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

БАКАЛАВРА

Тема: Проектирование приложения под мобильную платформу Android с использованием паттернов mvp и Clean architecture

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент(ка) |  |  |  | Быстров Т.Д. |
|  |  | *подпись* |  |  |
| Руководитель |  |  |  | Синев В. Е. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
| Консультанты |  |  |  | Иванов И.И. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |

Санкт-Петербург

2020**ЗАДАНИЕ**

**на выпускную квалификационную работу**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой ИС |
|  | \_\_\_\_\_\_\_\_\_\_\_\_ Цехановский В.В. |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент(ка) | Быстров Т.Д. | | | |  | Группа | 6373 | |
| Тема работы: Проектирование приложения под мобильную платформу Android с использованием паттернов MVP и Clean Architecture | | | | | | | | |
| Место выполнения ВКР: СПбГЭТУ “ЛЭТИ” | | | | | | | | |
| Исходные данные (технические требования):  кратко указываются основные требования к ВКР | | | | | | | | |
| Содержание ВКР:  Кратко перечисляются основные разделы ВКР | | | | | | | | |
| Перечень отчетных материалов: пояснительная записка, иллюстративный материал, иные отчетные материалы | | | | | | | | |
| Дополнительные разделы: указывается наименование дополнительного раздела | | | | | | | | |
|  | | | | | | | | |
| Дата выдачи задания | | | Дата представления ВКР к защите | | | | | |
| «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | | | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | | | | | |
|  | | |  | | | | | |
| Студент | |  | | Быстров Т.Д. | | | |
| Руководитель | |  | | Синев В.Е. | | | |
| *(Уч. степень, уч. звание)* | |  | |  | | | |
| Консультант | |  | | Иванов И.И. | | | |
| *(Уч. степень, уч. звание)* | |  | |  | | | |

**календарный план выполнения**

**выпускной квалификационной работы**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой ИС |
|  | \_\_\_\_\_\_\_\_\_\_\_\_ Цехановский В.В |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент(ка) | Быстров Т.Д. |  | Группа | 6373 |
| Тема работы: Проектирование приложения под мобильную платформу Android с использованием паттернов MVP и Clean Architecture | | | | |

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование работ | Срок выполнения |
| 1 | Обзор литературы по теме работы | 00.00 – 00.00 |
| 2 | Разработка концепции приложения | 00.00 – 00.00 |
| 3 | Создание архитектуры приложения | 00.00 – 00.00 |
| 4 | Написание кода | 00.00 – 00.00 |
| 5 | Оформление пояснительной записки | 00.00 – 00.00 |
| 6 | Оформление иллюстративного материала | 00.00 – 00.00 |

|  |  |  |
| --- | --- | --- |
| Студент |  | Быстров Т.Д. |
| Руководитель |  | Синев В.Е |
| *(Уч. степень, уч. звание)* |  |  |
| Консультант |  | Иванов И.И. |
| *(Уч. степень, уч. звание)* |  |  |

**Реферат**

Пояснительная записка 00 стр., 00 рис., 00 табл., 00 ист., 00 прил.

РАЗРАБОТКА, МОБИЛЬНОЕ ПРИЛОЖЕНИЕ, ANDROID, АрХИТЕКТУРА, ПРИЛОЖЕНИЕ, ПОЛЬЗОВАТЕЛЬ, КОД

Объектом исследования являются архитектуры мобильных приложений

Цель работы – Познакомиться и реализовать с наиболее популярной архитектурной мобильных приложений.

Работа представляет собой построение мобильного приложения с использованием наиболее популярного презентационного шаблона Model View Presenter (далее MVP) и принципа Сlean Architecture. Паттерн MVP используется для работы с презентационной логикой приложения, а именно отделяет View (визуальное представление контента) от логики взаимодействия пользователя с контентом (Presenter), который в свою очередь является мостом между View и Model (бизнес – логикой приложения). Model представляет собой механизм, который удовлетворяет принципам Clean Architecture. В основе Clean Architecture лежит правило, что каждый компонент системы может быть независимо отвязан и переиспользован в другой системе, другими словами, мы можем спокойно вырвать компонент из иерархии компонентов или переиспользовать его в другом месте независимо от платформы разработки. С использованием этих подходов было написано приложения для чтения сохраненных файлов пользователя (в формате PDF). Приложение использует технологию SPRITZ, которая позволяет пользователю читать слова не перемещая глаз, тем самым увеличивая скорость восприятия контента. Для реализации была выбрана платформа Android, изучен механизм работы паттерна Dependency Injection посредством библиотеки Dagger2, а также использовался язык программирования Kotlin, рекомендованный разработчиками платформы Android. Также был изучен механизм жизненного цикла приложений в системе Android и способы написания собственных виджетов. Во время разработки, часть кода была покрыта тестами, тем самым, доказывая, что подход Clean Architecture позволяет очень просто менять компоненты местами или подкладывать вместо одного компонента другой без всякого рефакторинга системы.

**Abstract**

This work is a construction of a mobile application using the most popular presentation pattern Model View Presenter (hereinafter MVP) and the principle of Clean Architecture. The MVP pattern is used to work with the presentation logic of the application, namely, it separates the View (visual representation of content) from the logic of user interaction with the content (Presenter), which in turn is a bridge between the View and the Model (business logic of the application). Model is a mechanism that meets the principles of Clean Architecture. At the heart of Clean Architecture is the rule that each component of a system can be independently unlinked and reused in another system. In other words, we can safely pull a component out of the component hierarchy or re-use it elsewhere, regardless of the development Board. Using these approaches, applications were written to read the user's saved files (in PDF format). The app uses SPRITZ technology, which allows users to read words without moving their eyes, thereby increasing the speed of content acceptance. The Android platform was chosen for implementation, the mechanism of the Dependency Injection pattern was studied using the Dagger2 library, and the Kotlin programming language was used, recommended by the developers of the Android platform. We also studied the application lifecycle mechanism in the Android system and ways to write custom widgets. During development, part of the code was covered with tests, thus proving that the Clean Architecture approach makes it very easy to swap components or replace one component with another without any refactoring of the system.

**Содержание**

# Введение

1. Обзор предмета создания приложения
   1. RSVP
   2. ORP
2. Обзор приложения SpritzReader
   1. Экран имеющихся книг
   2. Экран текущей страницы книги
   3. Экран настроек скорости чтения
   4. Экран скоростного чтения
3. Зачем нужна архитектура мобильному приложению?
   1. Какую проблему решает архитектура?
      1. Зона ответственности архитектурных слоев
      2. Агрегация и композиция
      3. Инверсия зависимостей
      4. Единоотвественность
      5. Открытость/Закрытость
      6. Разделение протоколов
   2. Основные сведения об архитектуре SpritzReader
      1. Структура пакетов
      2. Data
      3. DI
      4. Domain
      5. Presentation
      6. Views
      7. Модель архитектуры SpritzReader
4. View и Presenter
   1. Проблемы Android разработки
   2. Контракты взаимодействия презентационной логики
   3. Пример взаимодействия View и Presenter
   4. Тестирование Presentation – логики
5. Navigation – View или Presentation?
6. Model как реализация подхода Clean Architecture
   1. Путь модели от запроса до экрана
   2. Работа с данными внутри приложения. Слои приложения
      1. Data-слой
      2. Domain-слой
      3. Presentation-слой
   3. Паттерн Repository и паттерн DataSource
   4. Interactor vs. UseCase или как работать с бизнес логикой
   5. Тестирование бизнес логики
7. Реализация MVP в Android
   1. Использование Andoird виджетов как View
   2. Асинхронная работа с данными
   3. Внедрение зависимостей с помощью Dagger 2
   4. Хорошие практики
   5. Плохие практики
8. **ВВЕДЕНИЕ**

Современные мобильные приложения – это сложные, комплексные системы, имеющие способность быстро изменяться во времени. Как и в любой области пользователе-ориентированных продуктов, за кажущейся простотой пользовательского интерфейса скрываются сложные механизмы передачи, трансформации, преобразования данных. Мобильные приложения это не только о том, как показать картинку, но и о том, как это переиспользовать в другом месте, расширить и быстро внедрить при условии наличия команды разработчиков, которые постоянно друг другу мешают.

История становления ПО подарило мобильной разработке много примеров, паттернов и принципов проектирования, однако в большинстве случаев мобильное приложение как живой организм, должно постоянно мутировать – пополняться новым функционалом, видоизменять и поддерживать старый, уметь переходить на новые технологии. Для этого на помощь мобильным разработчикам приходит проектирование архитектуры приложения. Архитектуры мобильного ПО обладают своей вариативностью, поскольку зависят от большого количества факторов:

* Идея продукта
* Срок запуска приложения для пользователя
* Количество человек в команде
* Система сборки
* Платформа, на которой разрабатывается приложение

В данной работе невозможно оценить то огромное количество принципов и подходов в проектировании мобильного ПО, поэтому мы рассмотрим частный пример, а именно проектирование архитектуры приложения на платформе Android.

1. **ОБЗОР ПРЕДМЕТА СОЗДАНИЯ ПРИЛОЖЕНИЯ**

В рамках изучения проектирования мобильного приложения на платформе Android за основу была взята идея скорочтения. Мне всегда хотелось научиться читать быстро, не теряя смысл прочитанного. Но обычно все скорочтение заканчивается на том, что вы, набрав определенный темп сбиваетесь с нужной вам строки, выпадая из контекста прочитанного. Данную проблему могут решить подходы, который называется RSVP и ORP.

**2.1 RSVP (Rapid Serial Visual Presentation)**

Большинство взрослых людей способны читать текст со скоростью от 200 до 250 слов в минуту. Обычно, когда человек читает, его глаза перескакивают со слова на слово с помощью очень быстрых передвижений глазом, названных соккадами, меняющих положение нашего глаза от одного слова к другому. Мы видим и фиксируем большинство слов. Однако, те слова, которые являются наиболее короткими или предсказуемыми, мы стараемся пропускать. И напротив, слова наибольшей длины либо менее предсказуемые мы фиксируем лучше всего. В результате количество времени, в течение которого наши глаза задерживаются на каждом слове, и расстояние, на которое они прыгают, меняется от слова к слову тем самым увеличивая сложность понимания прочитанного. Согласно Spritz, их революционная технология избавляет читателя от необходимости двигать глазами во время чтения. Но в чем же заключается идея технологии? Технология построена на понятии RSVP, т.е. на отображении слов с большой частотой на длительном промежутке времени в фиксированном месте. Этот метод родился достаточно давно и имеет большую историю в психологии, где для участников экспериментов показывали слово в течении 1/10 секунды в зафиксированном положении. Так как все слова были зафиксированы в своем положении, больше не было необходимости перемещать глазом, а значит трудности, которые возникали до этого, больше не мешали достигать высоких скоростей чтения и понимания. Со временем в Spritz придумали, как еще можно увеличить скорость чтения, назвав подход ORP

**2.2 ORP (Optimal recognition point)**

Идея данного подхода состоит в том, чтобы зафиксировать глаз не на целом слове а на конкретной точке, которая находится в центре либо чуть левее центра слова. Исследователи чтения признают эту идею как версию двух феноменов, наблюдаемых в научных исследованиях чтения:

* Эффект оптимального положения просмотра (распознавание слов происходит быстрее всего, когда слово центрировано вокруг точки фиксации)
* Эффект предпочтительного положения просмотра (при нормальном чтении точка фиксации имеет тенденцию быть немного левее оптимального положения просмотра)

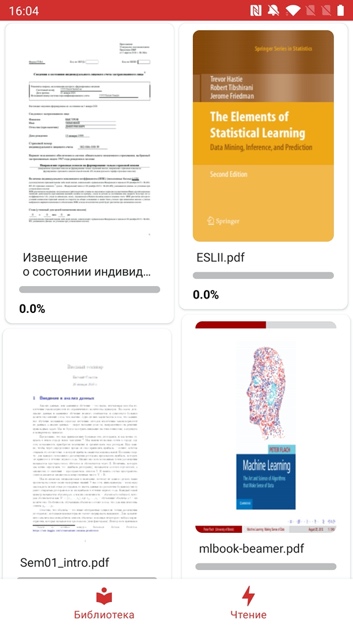
Данные подходы дают значительную прибавку в скорости чтения текста. Но возможно ли прочитать “Гарри Поттера” за полтора часа, полностью осознав то, что хотел передать нам автор? Думаю нет. Поэтому, необходимо добавить, что RSVP в связке с ORP не является серебряной пулей в скорочтении и скорее подходит для маленьких рассказов и небольших статей.

1. **ОБЗОР ПРИЛОЖЕНИЯ SpritzReader**

В функционал приложения входят 5 основных функции:

* Отображение имеющихся книг (pdf – файлов)
* Отображение прогресса чтения каждой из книг пользователя
* Отображение последней страницы книги, которую выберет пользователь
* Экран настроек скорости чтения, где пользователь выбирает количество слов в минуту
* Экран скорочтения, на котором пользователь видит слова согласно правилам RSVP и ORP

**3.1 Экран имеющихся книг**

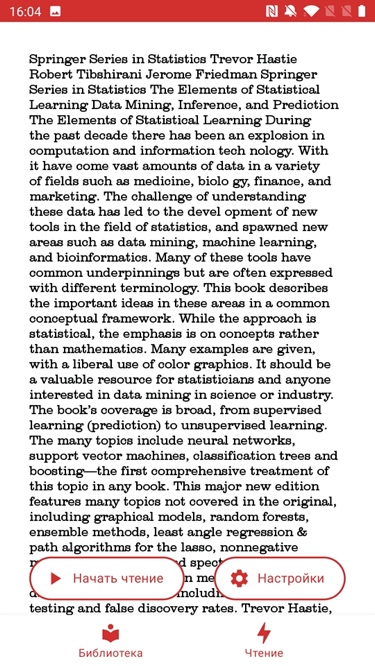


Приложение способно получать список книг из локального хранилища пользовательского устройства, путем обхода директорий Documents и Downloads. Также на этом экране отображаются обложки, названия и прогресс чтения книг и документов. Снизу находятся управляющие кнопки, отвечающие за навигацию между экранами. Прогресс отображается в процентом соотношении, и также отмечен полосой, где полностью серая полоса означает, что книга не прочитана, а полностью бордовая полоса означает полное прочтение данного документа.

Следовательно, на данном экране можно выделить три пользовательских сценария:

* Получение списка книг
* Отображение обложек книг
* Подсчет и отображение процентного соотношения прочитанного материала, к общему количеству страниц

**3.2 Экран текущей страницы книги**



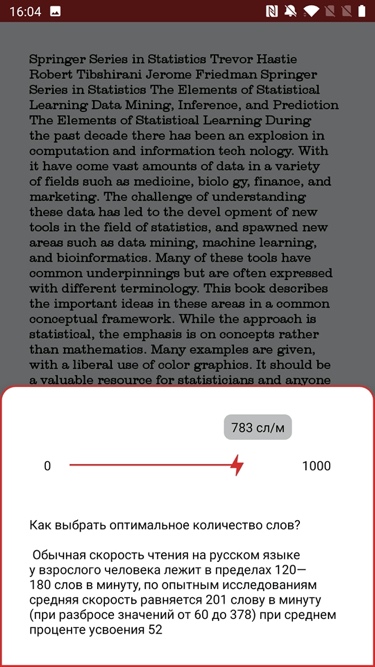
Приложение умеет отображать последнюю страницу, на которой остановился читатель.

Это необходимо, чтобы дать пользователю возможность восстановить контекст прочитанного ранее, обычно одной страницы достаточно, чтобы понять на чем остановился пользователь. Отображение происходит с помощью специального шрифта, который имитирует книжную печать, чтобы пользователю было удобно воспринимать текст. Также на экране присутствуют кнопки управления экраном, позволяющие вернуться на список с книгами. Из управляющих кнопок на самом экране есть кнопка “Начать чтение”, ответственная за переход на экран со скорочтением, а также кнопка “Настройки”, позволяющая пользователю перейти на экран настроек скорочтения

Следовательно, на данном экране можно выделить три пользовательских сценария:

* Начать чтение книги
* Настроить скорость чтения
* Вернуться на экран с книгами

**3.3 Экран настроек скорости чтения**

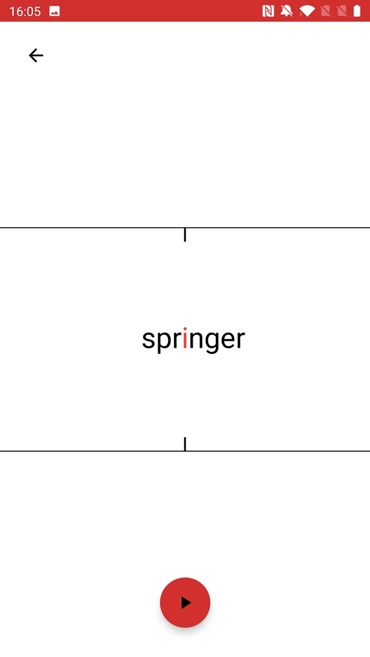


Приложение умеет настраивать скорость смены слов на экране скорочтения, а именно выставлять количество слов в минуту. В самом верху находится текст с текущим количеством слов в минуту, а под ним ползунок, меняющий данное значение. Стоит отметить, что текст меняет свое положение на экране в зависимости от положения ползунка, тем самым дает знать пользователю какое значение слов в минуту он выбрал на данный момент. Слева и справа от ползунка отображены минимальное и максимальное значение соответственно. Под ползунком находится подсказка для пользователя о средних значениях скоростей чтения взрослого человека для того, чтобы незнающий пользователь мог ориентироваться в значениях.

Следовательно, на экране можно выделить один пользовательский сценарий

* Регулирование скорости чтения

**3.4 Экран скоростного чтения**

 Приложение умеет запускать экран с чтением, где согласно принципам RSVP и ORP отображаются слова с переодичностью, которую пользователь указал на экране настроек. На экране можно приостановить чтение использовав круглую кнопку снизу, этой же кнопкой можно возобновить чтение. По середине экрана находится виджет отображения слов, каждое слово находится фиксировано в центре экрана, удовлетворяя требованию RSVP, а буква для фокуса подсвечена красным цветом, удовлетворяя ORP. Нажав на кнопку в верхнем левом углу, можно вернуться на предыдущий экран.

Следовательно, на экране можно выделить два пользовательских сценария

* Остановка/Запуск механизма быстрого чтения
* Возвращение на предыдущий экран

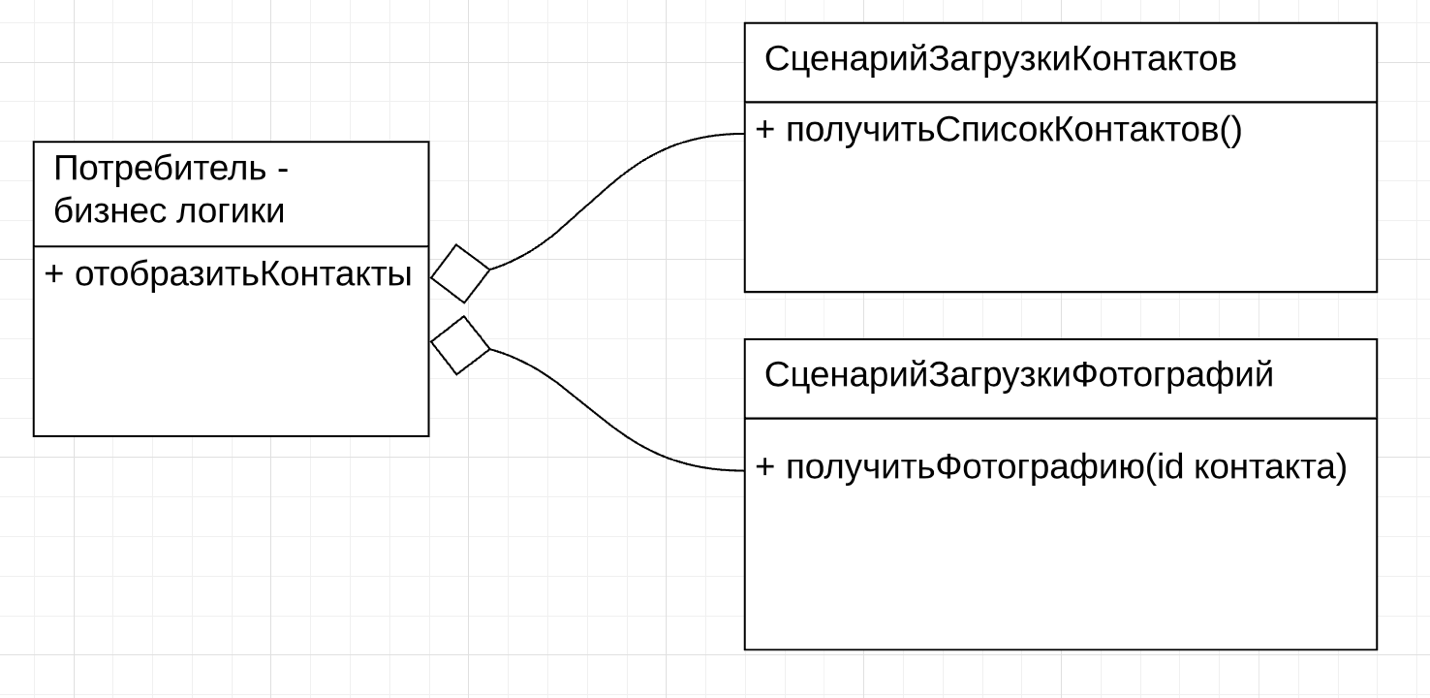
1. **Зачем нужна архитектура мобильному приложению?**

**4.1 Какую проблему решает архитектура?**

Архитектура – это основополагающий фундамент любого программного обеспечения. Архитектуру можно представить как большую стройку, в которой фундаментом выступает архитектурная база приложения (его слои данных), а кирпичиком выступает каждый отдельный модуль, внедренный в слой данных. Но для чего нужны все эти аналогии со строительством и какой практический смысл несет проектирование архитектуры?

* + 1. **Зона ответственности архитектурных слоев**

Архитектура позволяет разделить ваш продукт на зоны ответственности или логические куски бизнес – логики. Бизнес – логика представляет собой какой-нибудь обособленный пользовательский сценарий. В хорошо спроектированной архитектуре любую логику можно сконструировать из кусочков уже имеющихся пользовательских сценариев. Например, пользователю необходимо отобразить список контактов, но при этом к этому списку нужно добавить его фотографию, которая находится на определённом веб-сервере, получить которую можно по названию контакта, тогда зависимости могут выглядеть примерно так

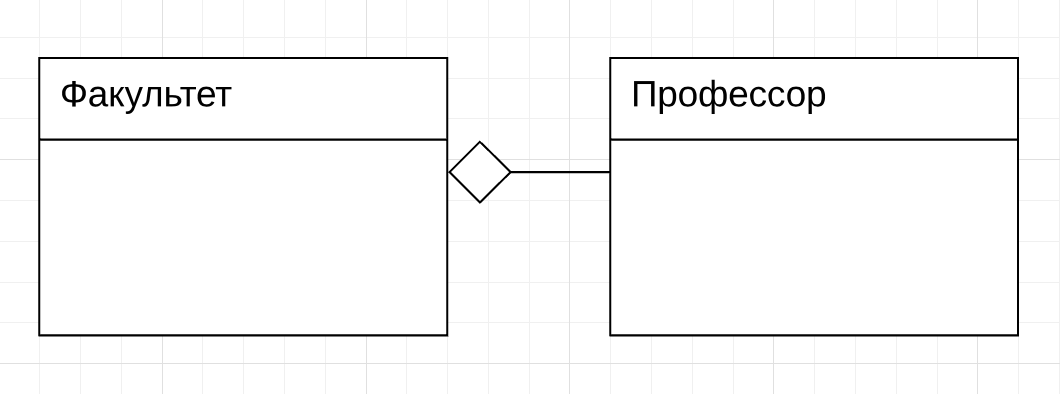


* + 1. **Агрегация и Композиция**

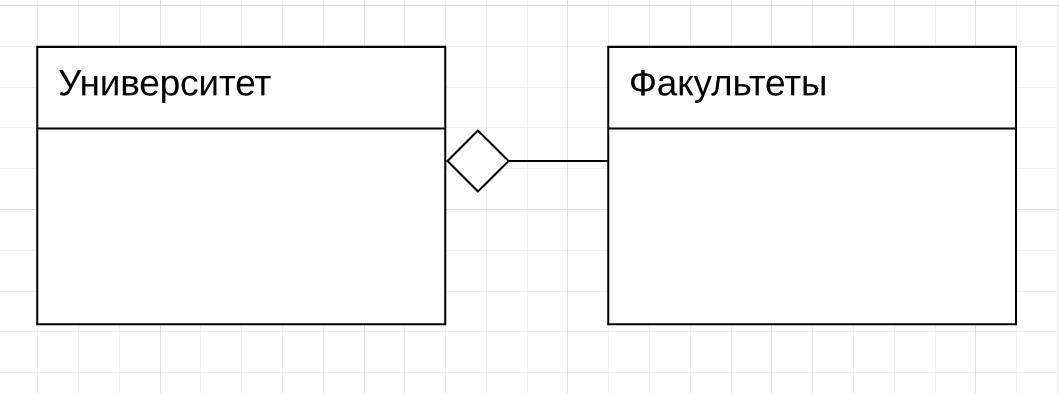
Следствием первого пункта является, является то, что разработчики сами выбирают способы инициализации и построения графа зависимостей компонентов в проекте, а именно решают проблему того, каким образом каждый потребитель бизнес-логики будет получать свои компоненты. Тут можно выделить несколько способов:

* Агрегация (делегирование)
* Композиция

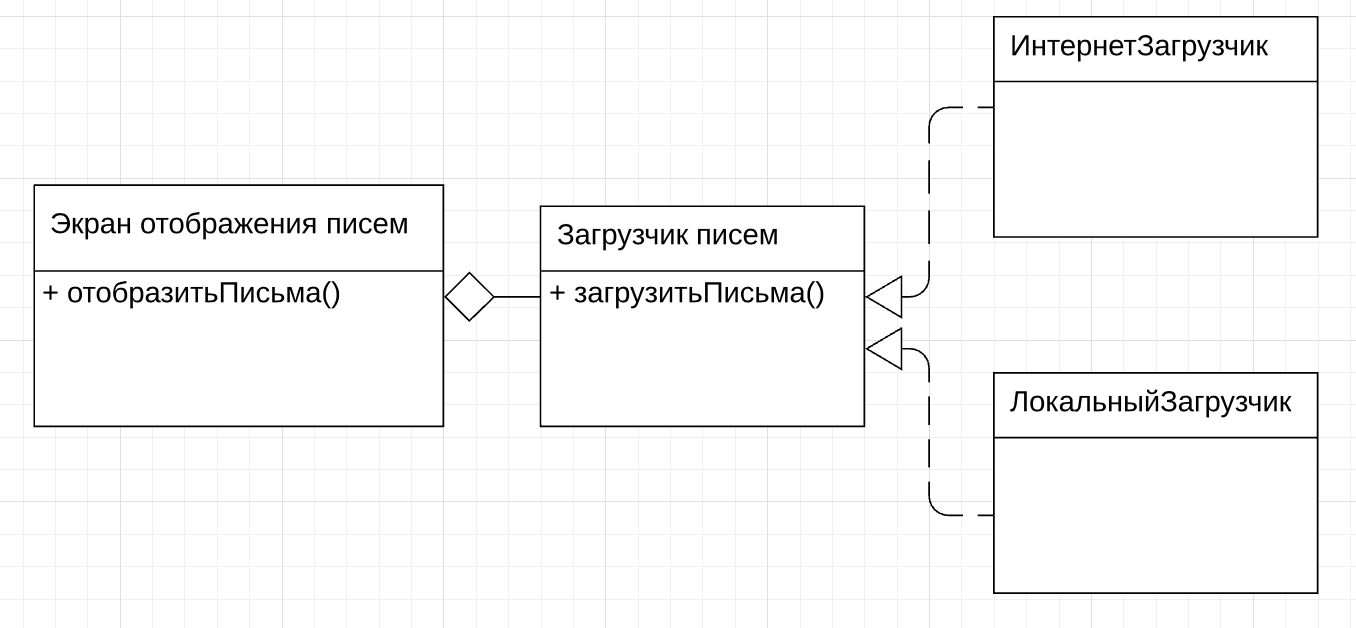
Агрегация – это отношение сущностей, когда одна сущность является контейнером для другой сущности, но при этом объект, который является зависимостью создается и настраивается вне своего контейнера и может существовать самостоятельно. Примером может послужить отношение “Факультет - Профессоры”



Композиция – это более строгий вид агрегации. В условиях композиции зависимости создает сам объект контейнер, поскольку ему важно, как именно создалась его зависимость, и сущности, которые находятся в композиции не несут какой-либо ценности без своего контейнера. Примером может послужить отношение “Университет - Факультеты”



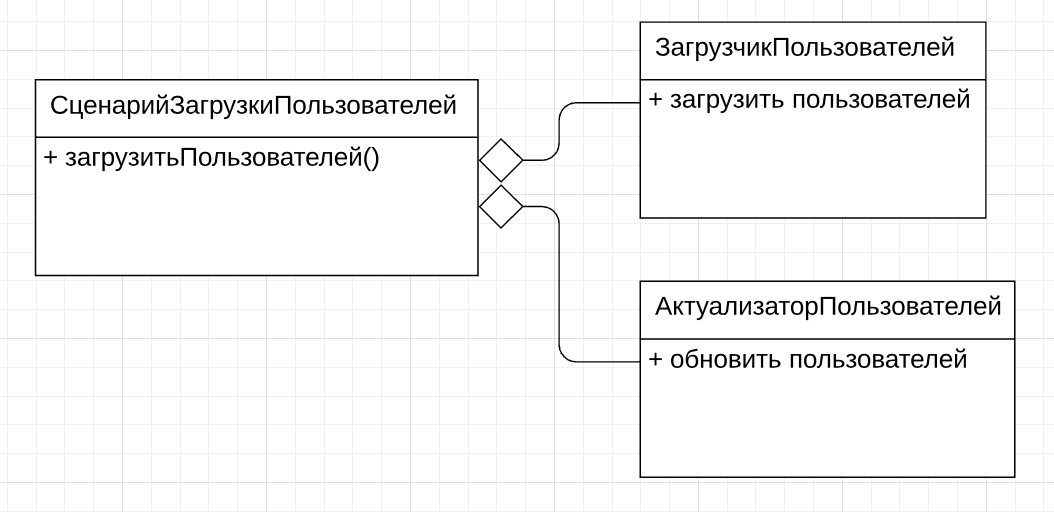
**4.1.3 Инверсия зависимостей**

Так как существуют механизмы настраивания зависимостей, то можно воспользоваться более абстрактными сущностями для того, чтобы использовать различные реализации одного и того же протокола. Это позволяет улучшить переиспользоваемость модулей внутри проекта, заставляет писать и поддерживать меньше кода, поскольку все изменения происходят лишь на определенном уровне абстракции, а потребитель пользуется публичным протоколом. Примером может стать загрузка писем в почтовом клиенте. В случае, когда нет интернета, клиент может загружать письма из локального хранилища, иначе отображаются письма из удаленного хранилища

**4.1.4 Единоответственность**

Важно, чтобы каждый элемент бизнес-логики отвечал за определенный пользовательский сценарий. Плюс такого подхода в том, что логика в данном случае является атомарной и работает только с одними данными, не создавая никаких side-эффектов. Sidе-эффект это явление, когда код, который по смыслу должен делать одно, но при этом выполняет еще некоторое стороннее действие явно не указанное в контракте сценария. Бывают нейтральные side-эффекты и негативные. К нейтральным относится логгирование каких-нибудь данных, к негативным относится, например, обновление базы данных в случае, когда клиент просто хочет получить список записей из неё. В таком случае необходимо создать еще один бизнес-сценарий, который обновляет данные в базе данных и тогда, каждый из сценариев будет удовлетворять принципу единой ответственности. Соблюдение этих правил напрямую влияет на расширяемость и компонентов архитектуры приложения.

Сценарий с side-эффектом



**4.1.4 Открытость/Закрытость**

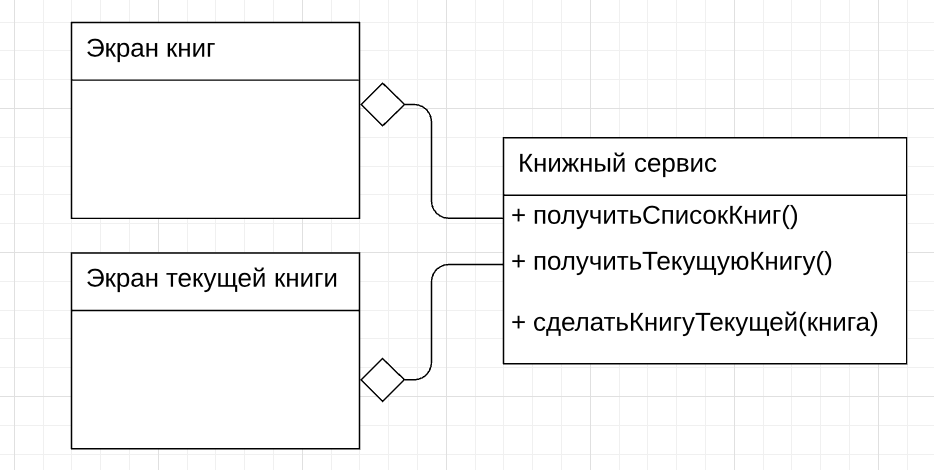
Правильно спроектированная архитектура позволяет с легкостью управлять данными различного происхождения, приводя их к определенному протоколу. С помощью такого приема разработчики могут с легкостью управлять моделями, создавая другие модели, расширяющие определенную базовую сущность. Это приводит к универсальности написанного кода, большей читаемости и переиспользованию, написанных модулей. Допустим нам необходимо уметь считать площадь определенных фигур в определенной графической системе, тогда мы хотим иметь протокол для подсчета площадей каждой фигуры. И в момент, когда добавиться новая фигура, она будет наследником данного протокола с помощью которого, мы определим, как считать ее площадь, тем самым мы можем работать не с определенным видом фигур, а с последовательностью абстракций. Этот подход улучшает гибкость проектирования архитектуры, потому что, закрывая данные обобщенным протоколом вы можете использовать большее количество данных самого различного происхождения несмотря на то, что они совершенно друг на друга не похожи. Этот принцип несет название “Открытости/Закрытости” и гласит, что любая модель в системе должна уметь расширяться, но не изменяться.



**4.1.5 Разделение протоколов**

Еще один важный момент в построении гибкой архитектуры — это предоставление атомарных, клиент ориентированных интерфейсов, вместо одного общего. Зачастую работа в приложениях с большой архитектурой ведется командами параллельно, и бывают моменты, когда командам необходимо разрабатывать один и тот же функционал, предварительно разделив его на соответствующие слои. В хорошо спроектированном программном обеспечении определенные слои логики общаются между собой с помощью протоколов, тем самым скрывая логику внутри реализации этого протокола. У этого подхода есть еще один плюс: команды могут заранее договориться о протоколе взаимодействия их слоев и приступить к параллельной разработке. Но есть небольшая проблема, а именно объем этих протоколов. Зачастую создается протокол взаимоотношений с через чур большим функционалом и это приводит код к состоянию “прибитой гвоздями” бизнес-логики, что является камнем преткновения в большой и гибкой системе. Такую логику сложно переиспользовать, а если и получается, то она тащит за собой кучу лишнего функционала, в котором соответственно нужды никакой нет. Из-за этого кодовая база растет, а зависимости становятся все жёстче. Например, пусть у нас есть система с книгами пользователя. Она умеет возвращать список всех книг пользователя, сохранять выбранную пользователем книгу в раздел текущих, и соответственно получать ее обратно. И пусть у нас есть два экрана: экран общего списка книг и экран первой страницы выбранной книги. Мы спроектировали нашу бизнес-логику так, что один сценарий умеет выполнять сразу все операции над книгами. Пользуясь таким сценарием мы заведомо допускаем ошибку, поскольку на экране списка книг на необходимо уметь получить все книги и сохранить выбранную пользователем книгу, а значит функционал получения сохранённой книги нам здесь не нужен и его можно вынести в отдельный сценарий, который будет использоваться только на экране текущей книги. Тем самым мы разделили один общий сценарий на два поменьше и увеличили гибкость, а также переиспользвание модуля.

Пример плохого разделения протоколов



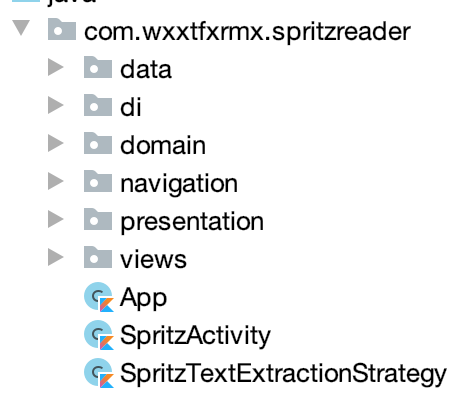
Пример, когда разделение интерфейсов, убрало лишние зависимости



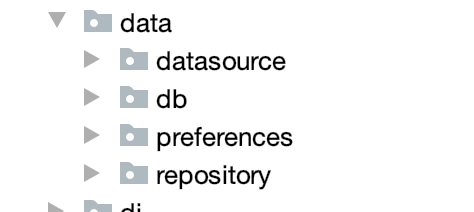
**4.2 Основные сведения об архитектуре приложения SpritzReader**

**4.2.1 Структура пакетов**

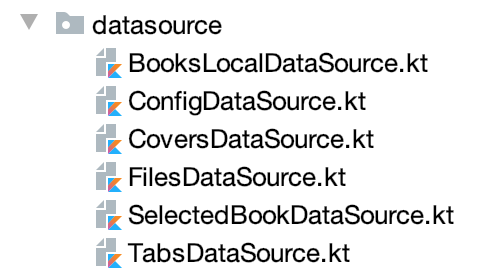
Проект разделен на слои, в составе которых, как и архитектурные модули, так и модули платформеннозависимые:

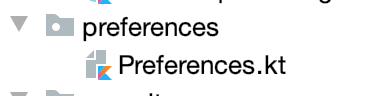


**4.2.1 Data**

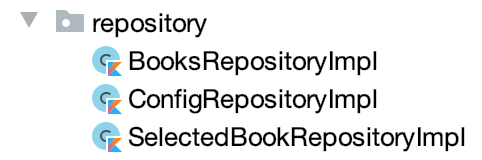
Пакет является архитектурным фундаментом, в котором лежит самая низкоуровневая реализация локальных хранилищ. 

В его состав входят:

* Работа с так называемыми “Источниками данных” (DataSource). 
* Обертка для работы с мобильной версией SQL (SQLite3).
* Работа с персистентным хранилищем самого приложения (SharedPreferences)



* И имплементации паттернов Repository, поскольку согласно принципам Clean Architecture, имплементации репозиториев являются частью data – слоя приложения.

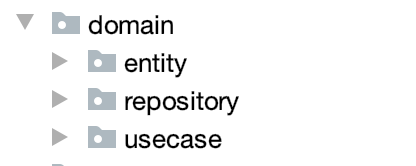


**4.2.2 DI (Dependency Injection)**

Пакет внедрения зависимостей. Данный модуль представляет собой реализацию системы модулей в проекте. Этот пакет классов зависит как от платформы, так и от инструментов, использованных при построении графа зависимостей. SpritzReader был написан под платформу Android, поэтому был использован самый популярный механизм для внедрения зависимостей – Dagger2. DI – пакет проекта принято не считать частью архитектуры, поскольку он очень вариативен и при смене инструмента часто необходимо заменить полностью весь механизм внедрения зависимостей, что противоречит правилам хорошей архитектуры.

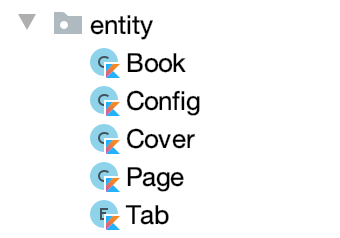
**4.2.3 Domain**

Пакет классов, который отвечает за все бизнес сценарии приложения. Именно в этом модуле находится вся логика пользовательских сценариев. Модуль несет самую большую бизнес ценность, поэтому необходимо чтобы он был максимально атомарным. Причина тому – часто меняющиеся бизнес сценарии. Сегодня заказчик требует показать картинку на одном экране, завтра уже на другом, а в рамках хорошо спроектированного domain-слоя перенос логики с экрана на экран не составит труда

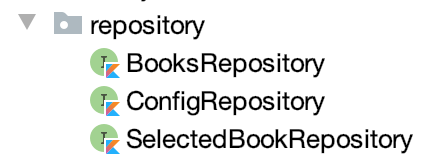


В состав domain – слоя входят:

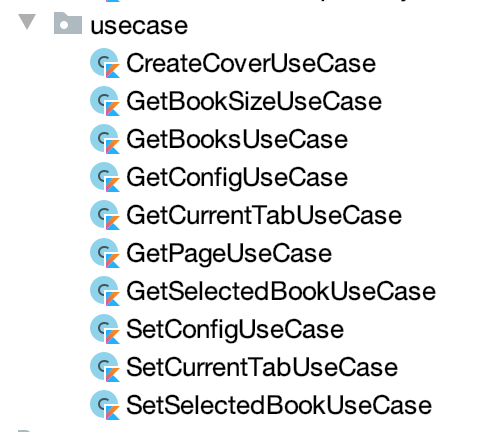
Entity - пакет, в котором находятся те самые бизнес сущности, которыми оперируют пользовательские сценарии для работы с пользователем. Эти сущности отображают смысл происходящего на экране, представляя собой все данные, показанные пользователю. В пакете находятся только классы-хранилища



Repository – пакет протоколов, являющихся мостом domain-слоя с data-слоем. Отображают только функционал, который domain сущности могут получить от data – слоя. Благодаря им, на слое пользовательских сценариев слой данных кажется черным ящиком из которого можно достать все тебе необходимое. Имплементации данных протоколов находятся на data-слое

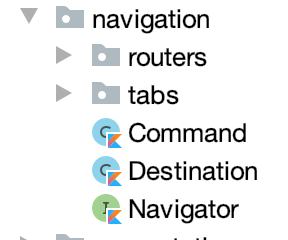


UseCase – пакет, в котором находится совокупность атомарных пользовательских сценариев. Зачастую представляют собой класс, с зависимостями в виде протоколов репозиториев и с одним атомарным действием. Несут самую большую бизнес важность, являясь кирпичиками для строительства поведений экранов. Название таких классов отражают явный бизнес-сценарий, тем самым составляя агрегацию пользовательских действий на каком-либо экране



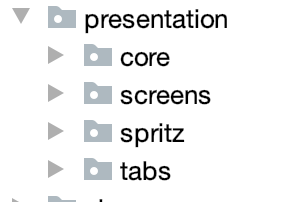
**4.2.4 Navigation**

Пакет, в котором лежат классы навигации. Навигация платформа зависимая, но она также подлежит шаблонизации при помощи общих протоколов. Это может быть необходимо, когда разработчики используют несколько реализаций навигаций, например, на базе Activity или Fragment.

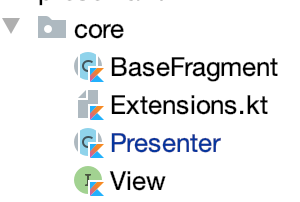


**4.2.5 Presentation**

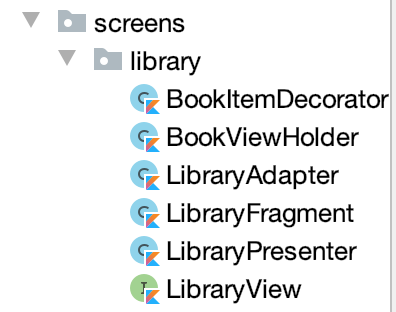
Пакет классов отвечающий за слой приложения, на котором спроектирована презентационная логика приложения. В данном пакете находятся имплементации всех контрактов View-Presenter и соответствующие им реализации из фреймворка Android (Fragment и Activity). В пакете находятся классы, отвечающие за логику взаимодействия пользователя и данных, реагирующие на пользовательские ввод, и прочие пользовательские события. Также эти классы отвечают за отображение данных бизнес-логики пользователю. Необходимо рассмотреть этот пакет детальней, чтобы разобраться в сути происходящего.



Пакет core представляет собой модуль, который сохраняет в себе самый базовый функционал, связанный с Android фреймворком. Тут находятся базовые классы, упрощающие работу с Android-компонентами. BaseFragment и Presenter являются абстракциями инкапсулирующими в себе функционал инициализации верстки на экране и передачи контракта (View) в управляющий класс Presenter`а. Помимо этого, класс содержит в себе так называемые функции расширения (extensions). Суть таких функций в том, что они способны расширить функционал класса, которому они принадлежат. Расширения — это часть языка Kotlin – основного языка разработки под платформу Android.



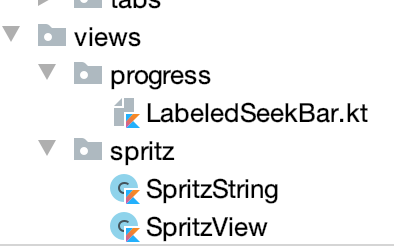
Пакет screens содержит в себе классы, отвечающие за определенное отображение данных (экраны). В этом пакете лежат все основные экраны пользовательских сценариев. У каждого экрана свой пакет, в котором лежат классы конкретного экрана. Рассмотрим пакет экрана списка книг



В данном пакете прежде всего выделяются архитектурные составляющие, а именно класс, отвечающий за реагирование на пользовательские события (LibraryPresenter), класс-контракт между отображением и логикой реагирования на пользовательские события (LibraryView) и класс имплементация этого контракта, обычно платформа зависимый (LibraryFragment). Остальные классы являются вспомогательными для класса-имплементации, поскольку необходимы различным визуальным компонентам: прокручивающиеся списки, кнопки и т.д.

**4.2.6 Views**

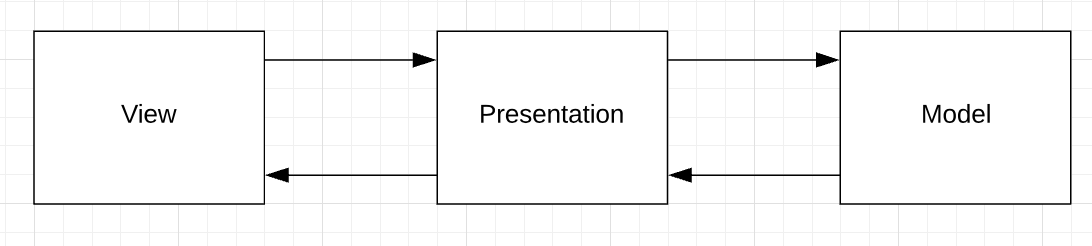
В пакете views находятся все несистемные Android-компоненты. В число таких компонентов могут входить различные кнопки, поля ввода, списки, имеющие логику отличную от системных. В данном проекте были имплиментированны два таких компонента. Это LabeledSeekBar – ползунок с зависящим от его положения текстом и SpritzView – компонент, отображающий текст согласно принципам RSVP и ORP.



**4.2.7 Модель архитектуры SpritzReader**

Архитектура моего приложения, если смотреть горизонтально состоит из поведенческой части (Model-View-Presenter) и части, ответственной за обеспечение корректной работы с данными (Domain-Data). И в данном случае одна архитектура является подархитектурой другой, иными словами Model – это domain – слой приложения, который инкапсулирует всю остальную логику приложения. Важно отметить, для поддержания атомарности и независимости всех модулей следует сохранять последовательные зависимости, то есть View может зависеть только от Presenter, а Presenter только от Model. В противном же случае приложение превратится в так называемый “спагетти-код”, поскольку модули будут зависеть от других слоев непоследовательно, что приведет к образованию багов, большого количества технического долга, и невозможности расширения функционала приложения, путем внедрения новых фич.

Упрощенно архитектура выглядит так:



Важно отметить следующее:

1. View получает данные только через Presenter, чтобы сохранить атомарность и независимость от других слоев архитектуры
2. View является так называемой “dumb view”, то есть за состояние view отвечает ее presentation-логика
3. Presentation логика может работать только с domain-cущностями, и при необходимости, трансформировать их во view-модели, согласно состоянию view
4. Model должен представлять собой набор сущностей (Interactor или UseCase), которые должны предоставлять модели из пакета entity, но никак не data (если таковые имеются). Также эти сущности должны отвечать либо single-scenario принципу, когда бизнес-логика привязана к определенному экрану, либо single-responsibility принципу, когда бизнес-логика отвечает за один определенный сценарий. Второе в современной разработке предпочтительней, поскольку позволяет конструировать бизнес-логику детерминировано.
5. Зависимости слоя от слоя также обусловлены Unit-тестирование таких сущностей. Намного удобней тестировать сущности, закрытые абстракциями, чем эмулировать существующий Android-компонент
6. **VIEW И PRESENTER**

**5.1 Проблемы Android разработки**

Android – мобильная операционная система с открытым исходным кодом. Эта система является одной из наиболее устанавливаемых ОС на мобильные устройства. Но одними мобильными устройствами Android не ограничен, он поддерживается на самых разных платформах. От микрокомпьютеров до умных автомобилей. Такой большой спектр устройств несет большие проблемы разработчикам приложений. На данный момент Android выпускает уже 11 версию, но некоторые проблемы тянутся за ним с 3 версии, а может и раньше. Рассмотрим наиболее часто встречающие:

1. Context – глобальное состояние вашего приложения

Сontext в Android приложении, это главная абстракция, связывающая разработчика с операционной системой. С помощью этого класса разработчик способен получить ресурсы приложения, системный функционал и системные сервисы. От Context`а зависят все Android – компоненты вашего приложения, в частности, все компоненты отображения какого-либо контента (View, Fragment, Activity). Однако, архитектура ОС спроектирована так, что любая смена конфигурации (поворот экрана смартфона, выдвигание клавиатуры), уничтожают существующий Context приложения, пересоздавая его. Это значит, что ссылку на Context невозможно сохранить нигде в приложении, что ведет к созданию различных механизмов работы с данной сущностью. Поэтому хорошо спроектированная архитектура делает работу с Context почти бесшовной.

1. Версионированный функционал

Google выпускает в этом году 11 версию Android, каждая из них уникальна в своем роде и приносит новый функционал и новые системные изменения. Разработчики с нетерпением ждут новые обновления, но также и готовятся к новым трудностям с версионированием функционала. Версии Android зачастую приносят несовместимые с предыдущими версиями функции, которые способны работать только с определенного уровня API (программный интерфейс системы). А так как вендоры мобильных устройств достаточно скупы на поддержку обновлений своих устройств, то в мире в равном количестве существуют устройства всех версий, а я напомню, что разработчики любят всех своих пользователей и должны поддерживать максимальное количество версий. Поэтому им приходится придумывать различные архитектурные приемы для своих приложений.

1. Навигация между экранами внутри приложения

За 11 версий Android Google эволюционировал механизм отображения от громоздких и неуклюжих Activity и ListView, до легких в применении Fragment и RecyclerView, но за все это время они не смогли предоставить никакого API для навигации внутри приложения. Да, навигация в Android отсутствует – и это одна из основных задач архитектуры. Каждая компания пытается организовать свою собственную навигацию, и это действительно место для жарких споров.

1. Жизненный цикл Android-компонентов

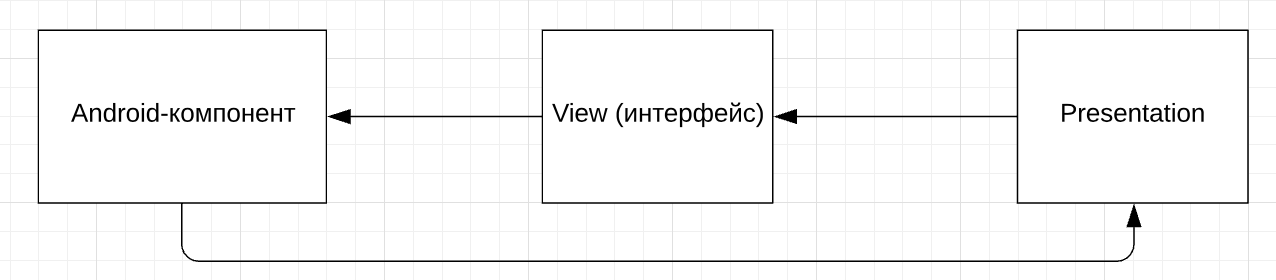
Android – приложения состоят из Android – компонентов, те в свою очередь привязаны к жизненному циклу приложения. Однако каждый вложенный компонент начинает зависеть не только от жизненного цикла приложения, но также становится привязанным к жизненному циклу родителя. Поэтому разработчикам почти всегда необходимо уметь работать с ЖЦ своих компонентов. Если же не соблюдать это будут происходить различные баги, утечки памяти, необъяснимые падения, что в свою очередь отразится на оценках пользователей. Жизненный цикл компонента построен на так называемых замыканиях. У компонента есть ряд функций, и система в нужный момент времени вызывает эти функции, давая понять разработчику, в каком состоянии его приложение на данный момент. Примером может послужить жизненный цикл Fragment`а



Каждое событие помогает понимать разработчику текущее состояние его приложения, тем самым помогая следить за какими – либо ресурсами пользователя (открытые файлы, сокеты, потоки данных). Однако, Android – система с открытым исходным кодом, значит вендорам мобильных устройств можно изменять код системы, что может разниться с тем поведением, которое описано на сайте разработчиков ОС. Это приносит разработчикам приложений некоторую боль, которую тоже призван решить архитектурный подход.

**5.2 Контракты взаимодействия презентационной логики**

Итак, мы выделили основные сущности нашей архитектуры. Теперь необходимо рассмотреть, как эти сущности должны взаимодействовать между собой. Начнем с первого этапа: контрактов презентационной логики. Ранее мы видели, что между Android-компонентом и логикой находится некоторая абстракция в виде View-интерфейса. Этот интерфейс убирает необходимость зависеть presentation-логику от конкретной реализации визуального компонента. Получается, что наша View – это контракт общения между Presenter и конкретной реализации визуального компонента. В условиях паттерна MVP, View – не должна иметь никакой логики внутри себя, иначе мы становимся зависимыми от имплементации. Это значит, что в имплементации должна быть только обработка и передача в Presenter внешних действий пользователя (нажатие, скролл, включение/выключение экрана) и действий операционной системы (малый заряд батареи, появление пуш-уведомления и т.д.). В то же время View, связанная контрактом с Presenter`ом, должна уметь выполнять команды, приходящие от presentation-логики.



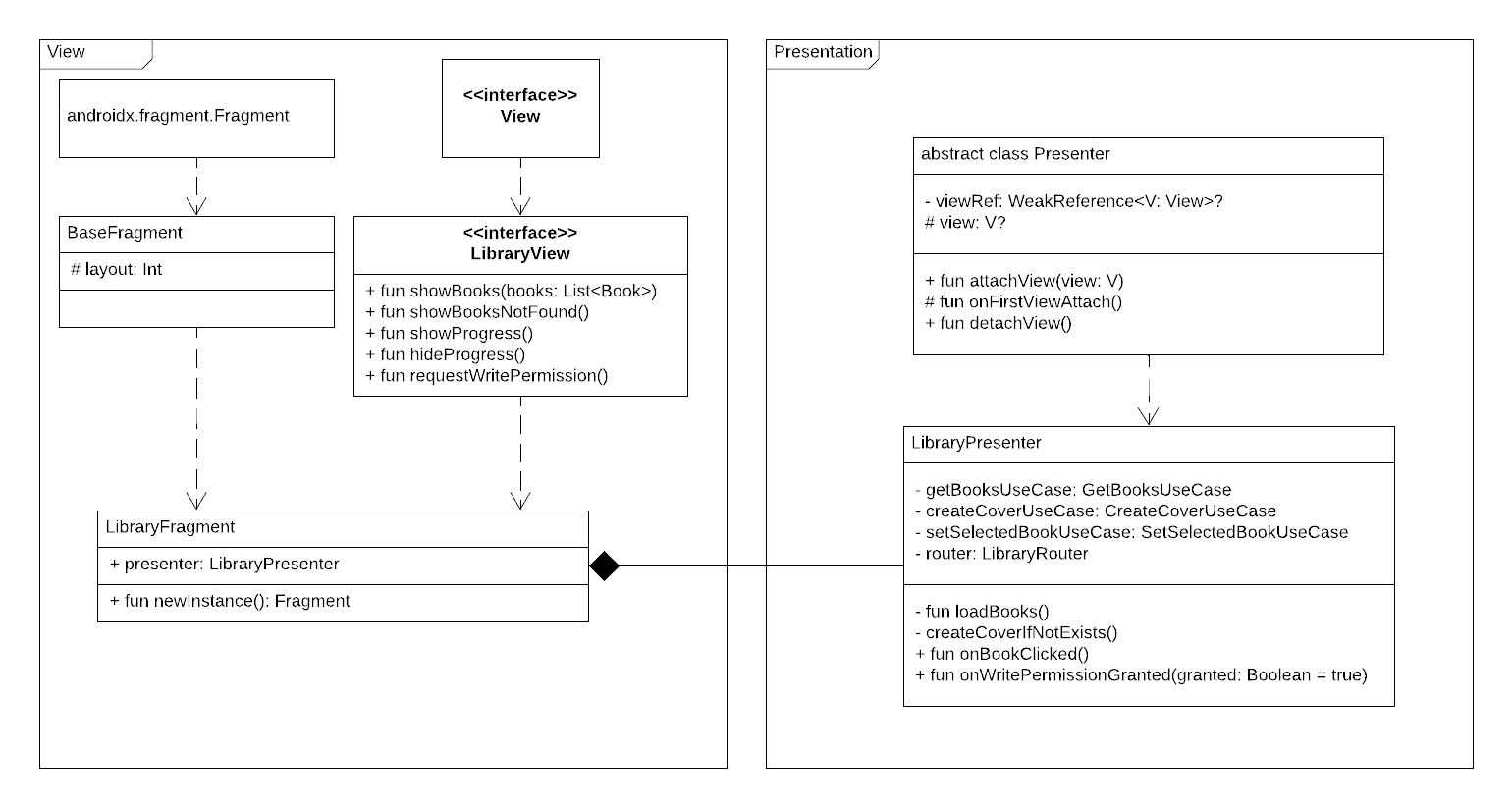
Что это нам дает?

Во-первых, атомарность. Мы можем переопределить наш View – интерфейс в Java-десктопном приложении, и не переписывать никакой логики внутри Presenter`а.

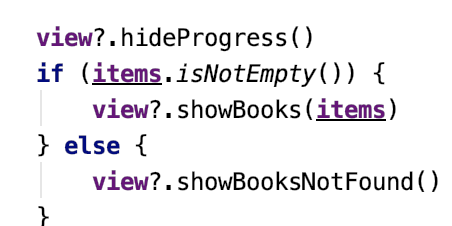
Во-вторых, реагирование на изменение presentation-логики. Если мы меняем поведение presentation-логики расширением контракта, то Android-компонент не сможет проигнорировать такое изменение, поскольку является имплементацией данного контракта

В-третьих, тестирование. Пропадает необходимость эмулировать Android-компонент для того, чтобы протестировать какую-либо presentation-логику, поскольку мы можем передать в наш Presenter, наивную реализацию этого контракта, ведь в большинстве случаев необходимо проверить, что в результате выполнения кода View выполнила необходимую команду.

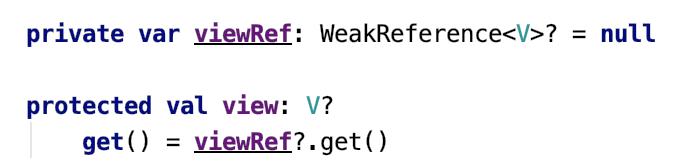
**5.3 Пример взаимодействия View и Presenter**

****

На диаграмме выше показаны отношения и зависимости между View и Presentation модулями на примере экрана книг пользователя (LibraryFragment). Точкой соприкосновения этих модулей является поле presenter в LibraryFragment. Для написания более универсального кода, и выделения повторяющейся логики был создан в presentation – модуле класс Presenter параметризованный View из модуля View. Тем самым мы можем написать немного платформозависимой логики внутри Presenter. Так как Context Android-компонента уничтожается при каждой смене конфигурации, то разработчику необходимо контролировать этот момент, чтобы не получить случайные падения приложения. Для этого в базовом презентере существуют два метода attachView и detachView, они привязаны к жизненному циклу компонента и сохраняют слабую ссылку на интерфейс, поскольку ОС не всегда может удалить ваш контекст, и это лишь некоторая оптимизация по кешированию протокола. Поскольку мы имеем ссылку на нашу View удобно ее сделать видимой для наследников в виде исчисляемой переменной для того, чтобы писать лаконичный код



Это фрагмент кода, который выполняется после загрузки книг пользователя из локального хранилища. Легко заметить, что данный код легко читается и понимается, потому что благодаря языку программирования Kotlin, мы можем писать доступные для наследников исчисляемые переменные, коей является view.

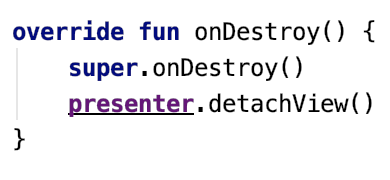


Презентер узнает о том, что необходимо начать работу с помощью метода attachView, который принимает в себя имплементацию протокола LibraryView. В этот момент можно считать, что Android-компонент проинициализирован системой, тем самым начать загрузку соответствующих данных о книгах. Если взглянуть на схему жизненного цикла Fragment, то можно понять, что конец инициализации происходит в методе onViewCreated, а значит процесс прикрепления View к Presenter необходимо делать именно в этот момент.



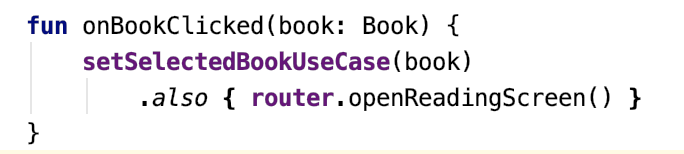
Помимо инициализации презентера, в этом методе жизненного цикла также происходит инициализация дочерних Android-компонентов, необходимых для отображения нужной пользователю информации. Так, например, в данном методе инициализируется компонент отображения списка элементов.

Но как понять, что Android-компонент будет уничтожен? Для этого к нам на помощь снова приходит жизненный цикл компонента, а если быть конкретнее метод onDestroy.

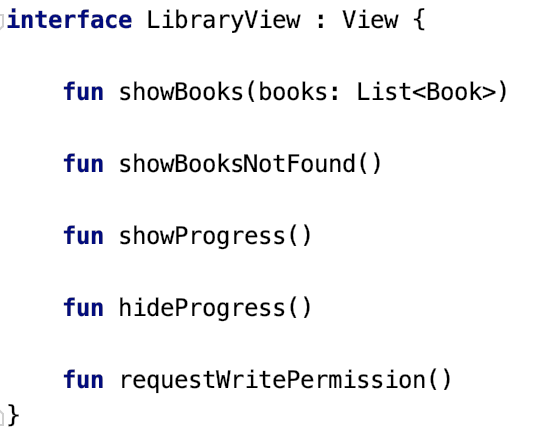


В данном методе мы можем освободить наш презентер от ссылки на view, просто пометив ее как null, так как она была обернута в WeakReference. Сборщик мусора в DalvikVM соберет ее при следующем проходе обязательно, а это может быть важно для ОЗУ смартфона, поскольку будет освобождена память. Если же не оборачивать компонент в слабую ссылку, то можно спровоцировать retain-цикл, когда две сильные ссылки держат друг друга. В этом случае стадия Mark алгоритма сборки мусора пометит их как false и GC не сможет их собрать на стадии Sweep, тем самым создав утечку памяти. // Может быть показать алгоритм Mark&Sweep

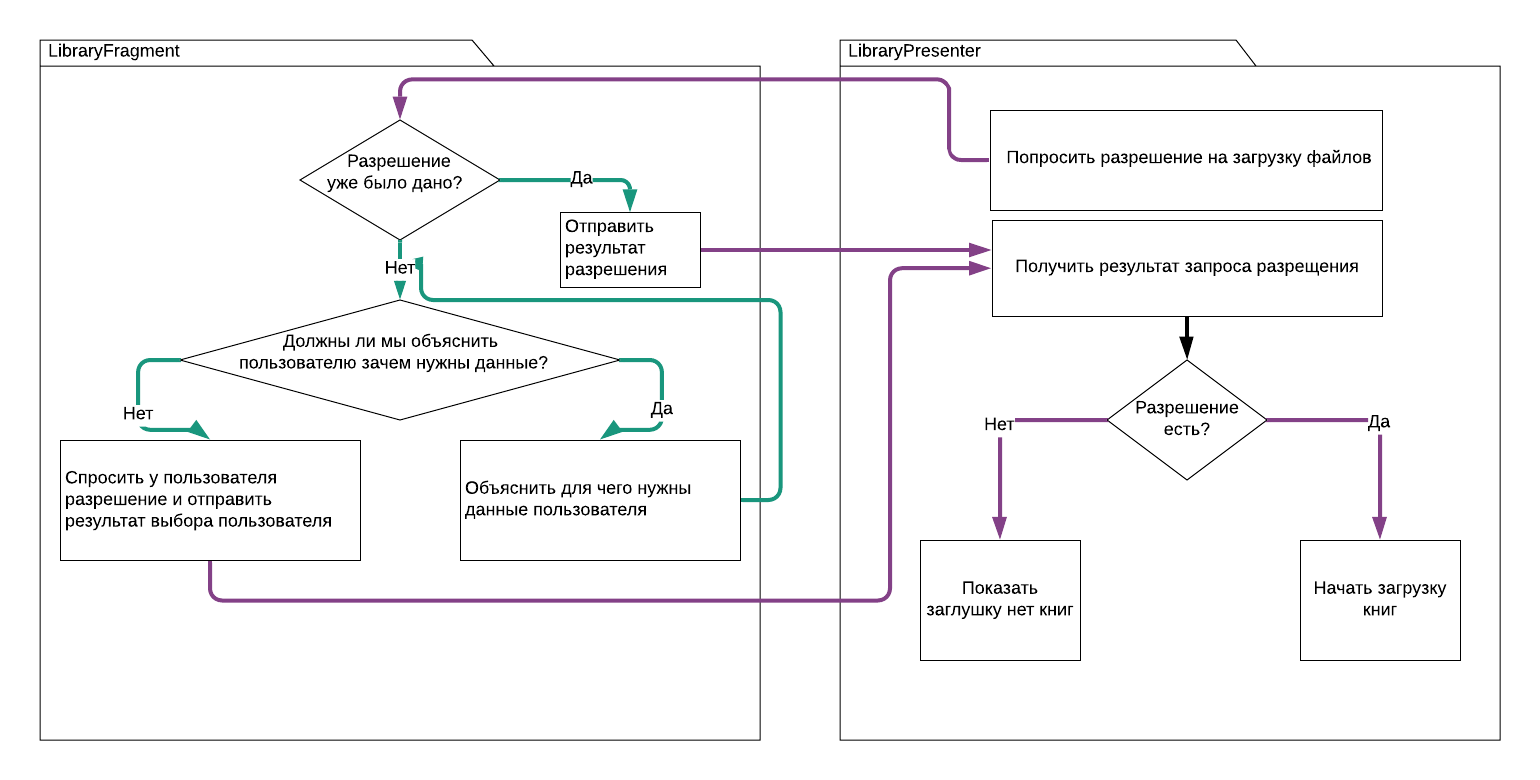
Необходимо еще отметить, как строится нейминг методов контракта и презентера. Методы презентера должны быть обработкой какого-либо события, как реакция на определенное событие. К примеру, если пользователю отображены список книг, и он жмет на одни из них, то презентеру должен поступить запрос о нажатии, выражающийся как коллбэк (callback):



Правила относительно нейминга методов View немного другие. View протоколы должны иметь методы, которые заставляют их что-то делать, и не должны содержать методов-коллбэков:



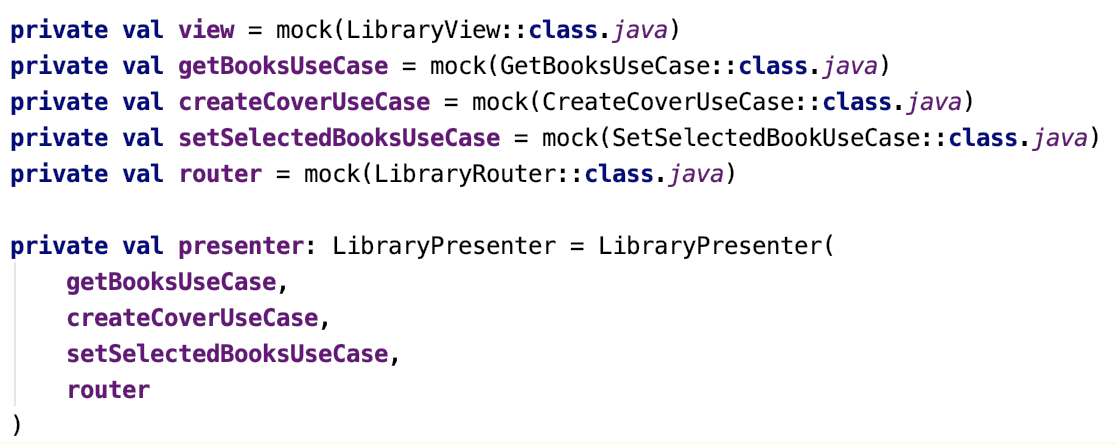
Однако, существуют исключения относительно системы разрешений в ОС. Так как все современные операционные системы должны предоставлять пользователю полный контроль над его данными, в Android введена система разрешений (Permissions API). Поэтому, прежде чем работать с какими – либо областями данных пользователя или ресурсами системы, контролируемых пользователем необходимо спросить разрешения у пользователя. Этот механизм построен на замыканиях в Android-компонентах и из-за этого приходится играть в так называемый ping-pong.



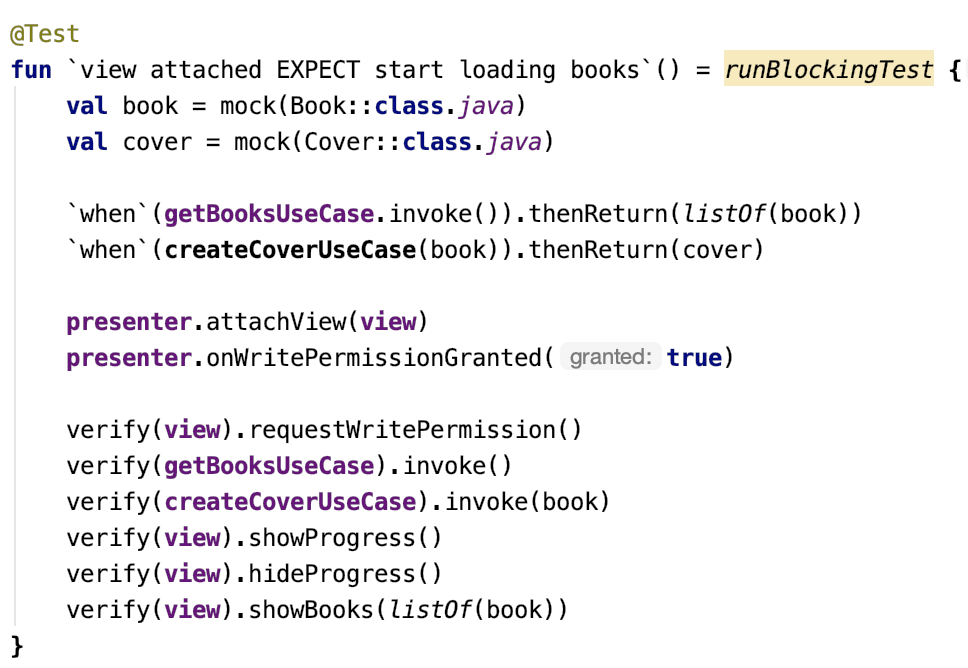
Процесс получение или отклонение на чтение локального хранилища в ОС Android. Фиолетовыми стрелками отмечены клиентские действия, написанные внутри презентационной логики, зелеными стрелками отмечены действия, совершаемые операционной системой без участия разработчика.

* 1. **Тестирование Presentation-логики**

Чтобы снизить риск ошибок и появления багов. Необходимо большую часть своего кода покрывать тестами. Тесты – показатель роста и усложнения кодовой базы. Если продукт очень быстро развивается, то появляется очень высокая вероятность что-нибудь сломать. В больших проектах тесты скорее являются инструментов контроля непредвиденных поломок, когда, например меняется программный интерфейс какого-либо класса, или меняется количество параметров у определенной функции. В нашем архитектурном подходе, когда мы можем передавать зависимости через конструктор, можно с легкостью тестировать любой функционал. Это связано с тем, что все зависимости можно подменить на так называемые заглушки (stub). Заглушки – это пустые реализации интерфейсов или объекты классов, которые мы можем настраивать в зависимости от условий тестов. Однако, в условиях быстрого роста кода и внедрения нового функционала нет времени заниматься написанием пустых реализаций интерфейсов и классов. Для таких тестов сообщество Android-разработчиков уже давно придумало и написало библиотеки тестирования. Одна из них называется – Mockito. Именно ее я использовал в своем проекте для реализации юнит-тестирования своего функционала. Для того, чтобы в этом разобраться необходимо детальнее рассмотреть тест.



Первое, что необходимо сделать – это создать заглушки для наших бизнес-сущностей и view-зависимости. Используя функцию mock из библиотеки Mockito сделать это не составит труда, в противном случае мне бы пришлось писать руками реализации всех сущностей и протоколов. Далее необходимо рассмотреть сам тест. Наш тест будет проверять логику того, что если у нас создался презентер и мы получили разрешение на чтение и запись пользовательских данных в локальном хранилище, то мы начнем загружать книги и создавать обложки для этих книг, а после отобразим их непосредственно пользователю. Также, чтобы показать пользователю процесс загрузки мы должны сопроводить индикатор загрузки, а как получим необходимую информацию скрыть его, иначе пользователь может не понять, что приложение выполняет какую-либо тяжелую работу и просто зарыть его.

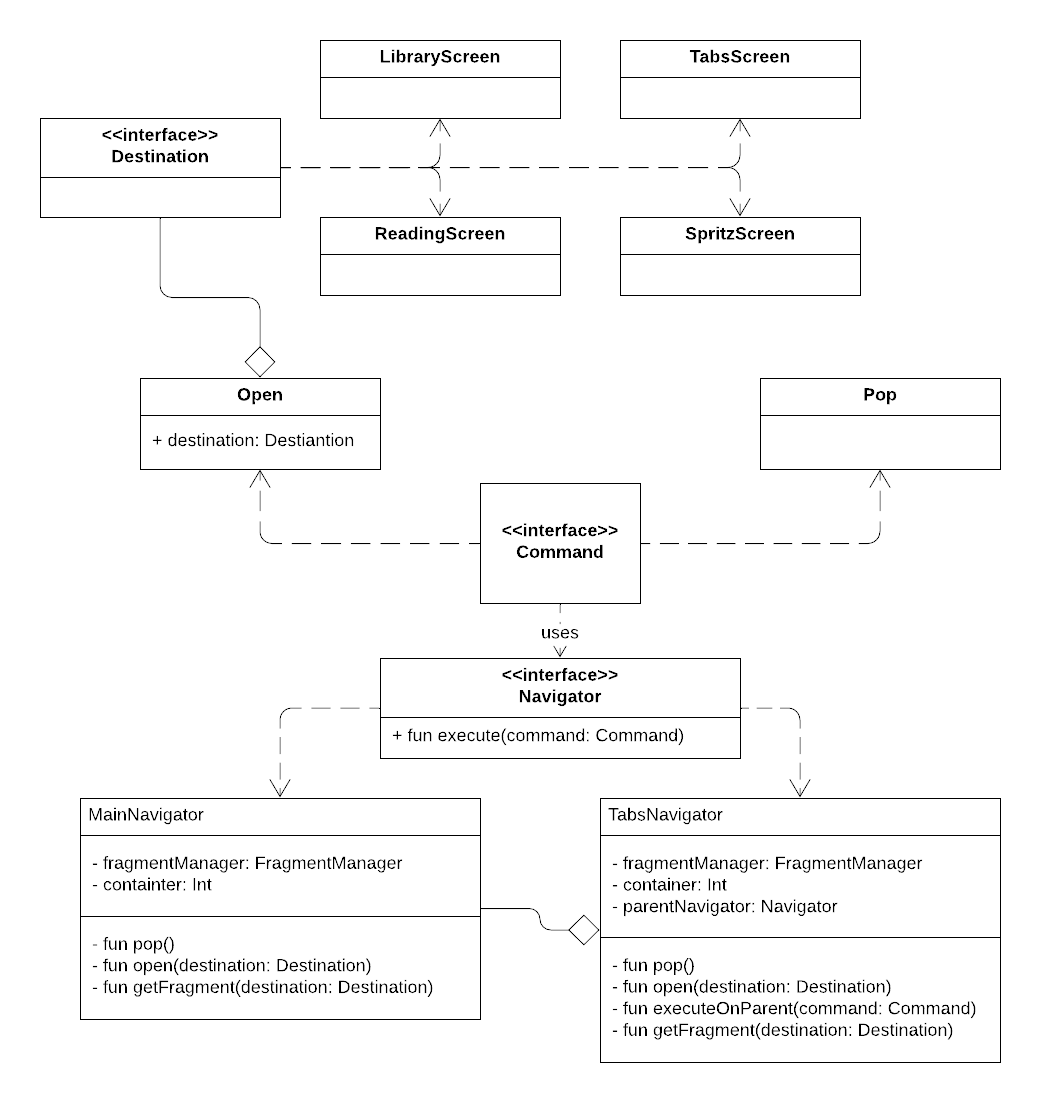


Типичный тест состоит из названия, которое кратко отображает суть происходящего в тесте. Обычно оно состоит из условий теста и ожидаемых результатов. В данном случае в названии теста отражено: “В момент, когда view прикрепится к презентеру ожидается загрузка книг”. Важно отметить, что тестовые функции должны быть помечены специальной аннотацией @Test, чтобы сборщик тестов смог понять и запустить написанный вами тест. Следующий шаг – это подготовка заглушек для наших бизнес-сущностей. В рассматриваемом тесте мы подменяем реализацию класса Book и класса Cover, поскольку нам неважно содержимое сущностей, нас интересует факт того, что, вызвав пользовательский сценарий GetBookUseCase, мы получим список книг. Тоже самое касается CreateCoverUseCase – факт того, что мы создали обложку не так важен, относительно того, что он действительно вызвался. Все это гарантирует нам тестирование на уровне пользовательского сценария, значит наши тесты – продукто - ориентированы и могут покрыть большинство пользовательских сценариев. После того, как мы проинициализировали заглушки, нам необходимо написать тестируемый код. Мы эмулируем прикрепление View к нашей презентационной логике и даем разрешение на чтение и запись данных так, как бы это сделал реальный пользователь.

После этого идет стадия проверки выполненных команд, дабы убедиться, что пользовательский сценарий действительно состоит из этих последовательных действий. Опять проверки касаются только самих вызовов, но не контента, который передается в функции, поскольку тестирование presentation-логики опирается на сценарии, а верификация различных данных уже область тестирования бизнес-логики и data-тестирования. Unit – тестирование - это хороший, гибкий инструмент поддержки актуального функционала в кодовой базе, однако оно не является “серебряной пулей” и все же не может покрыть интеграционные сценарии работы приложения. К примеру, для того чтобы написать тест перехода с одного экрана на другой необходимо написать большое количество кода, которое будет очень сложно поддерживать. Для решение этой проблемы коммьюнити создала библиотеку для написания интеграционных тестов (Espresso и UIAutomator). Если в Unit-тестировании мы эмулировали условия выполнения теста, то в рамках интеграционных тестов нам необходимо описывать сценарии нажатий и ввода пользовательских данных в само приложение по средством эмулирования нажатий на экран смартфона. Такие тесты требуют больших вычислительных ресурсов, поскольку необходимо содержать ферму эмуляторов или реальных девайсов. В рамках этого исследования интеграционное тестирование не затрагивалось.

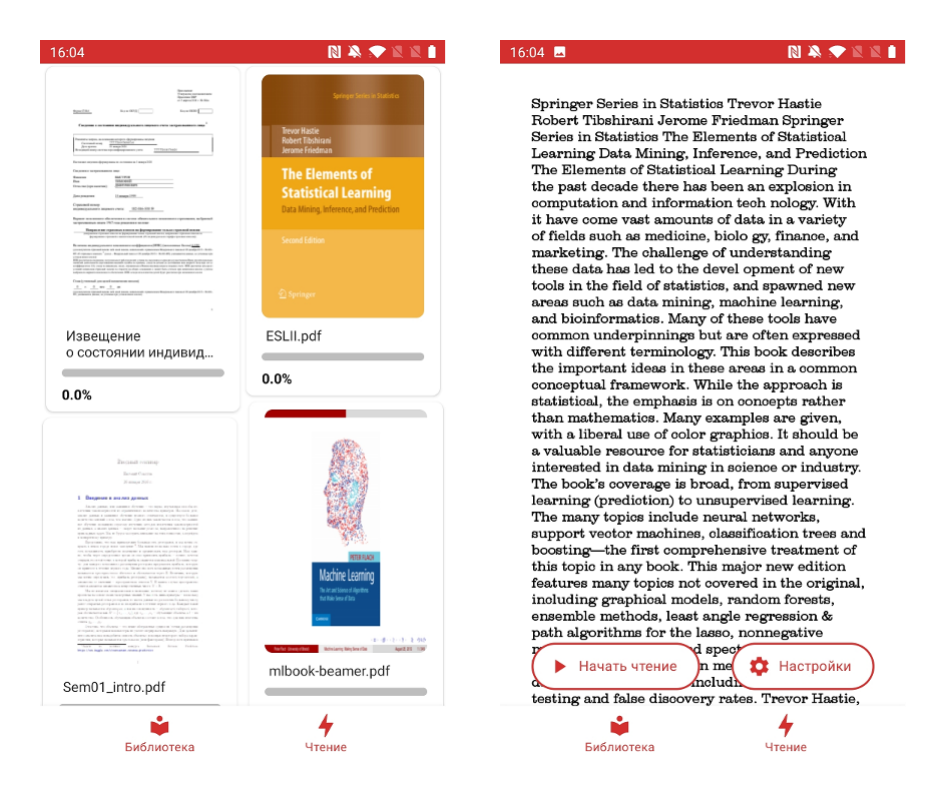
1. **Navigation – View или Presentation?**

Следующий не самый очевидный модуль, связанный с View и Presentation – модуль навигации. В упомянутых выше проблемах Android-разработки я рассказал о проблемах навигации. У разработчика есть возможность менять экраны местами, переходить с одного экрана на другой, однако, все эти механизмы не универсальны и от проекта к проекту разработчикам приходится снова воспроизводить свои наработки, связанные с проектированием навигации в приложении. В свое приложении я посчитал, что навигация – это все-таки часть архитектуры поэтому постарался шаблонизировать свою логику и постарался максимально абстрагироваться от Android-фреймворка.

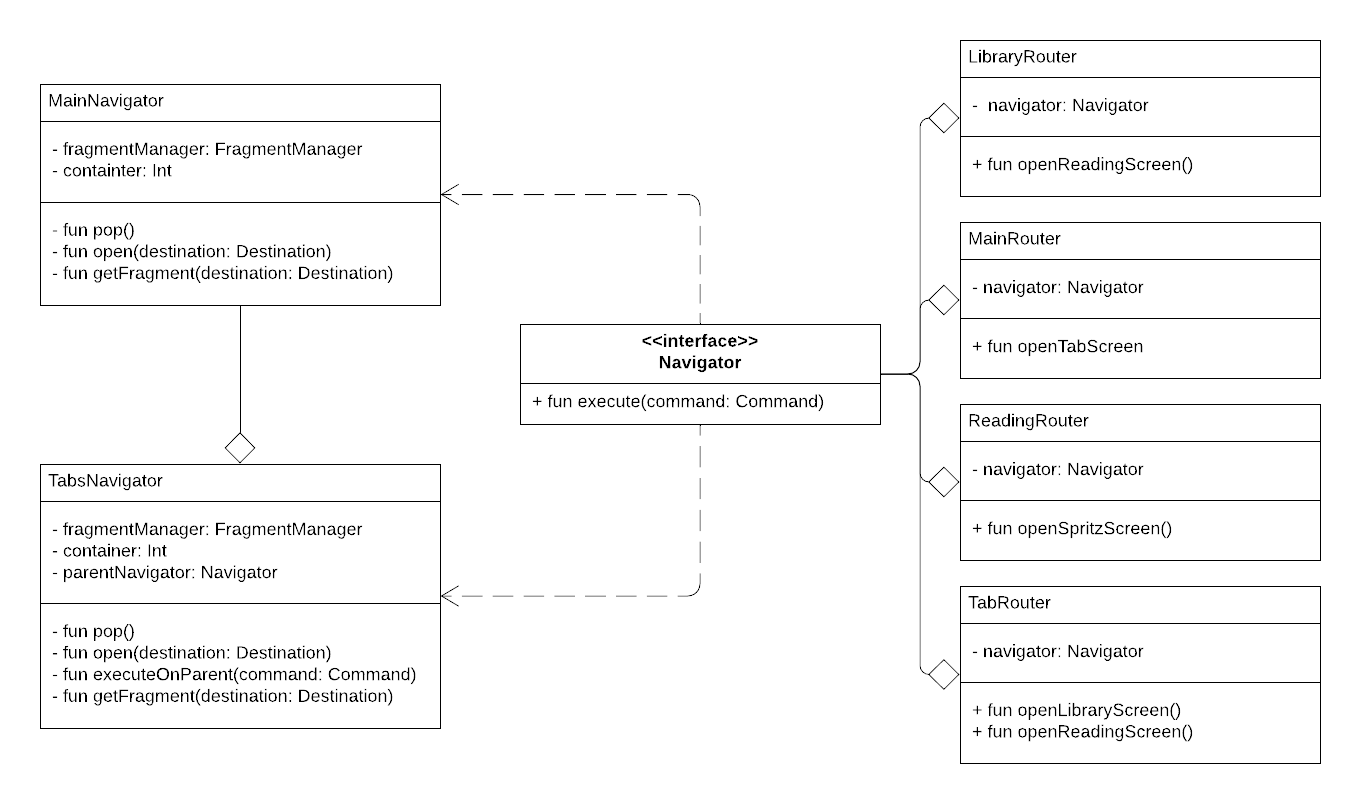


На схеме выше представлена архитектура навигации в приложении SpritzReader. Главной сущностью в ней является интерфейс Navigator, который умеет совершать только одно действие – запускать команду. Следующей главной сущностью является интерфейс Command, наследники которой представляют собой физическую команду по типу открыть, выкинуть и т.д. Навигатор использует команды чтобы получить из них информацию и соответственно применяя функционал Android-фреймворка совершает открытие, удаление экрана с девайса пользователя. Так, например, команда Открыть (Open) имеет поле типа Destination, что в свою очередь является меткой экрана, который необходимо открыть. Тем самым мы полностью абстрагируемся от понятий и сущностей Android-фреймворка, скрывая его за реализацией конкретного навигатора. А на presentation-логике оперируем только нашими сущностями, что определенно дает нам гибкость и расширяемость в разработке мобильного ПО.

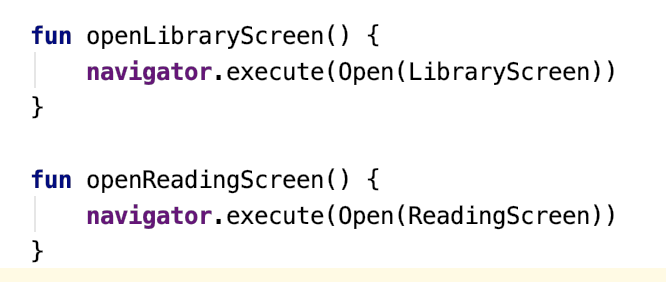
Из-за ситуации вложенности некоторых экранов друг в друга появляется необходимость создания вложенных навигаторов. Эта ситуация вызвана тем, что некоторые экраны используют общие элементы пользовательского интерфейса, как например сделано на экране пользовательской библиотеки и экране чтения страницы книги.



Поэтому проще для таких экранов создать “Экран - контейнер” с общими элементами пользовательского интерфейса. Разработчик ничего этого не замечает, когда пишет код, потому что эта иерархия скрывается за грамотно спроектированным API. По сути, разработчику необходимо знать список экранов и какую имплементацию навигатора получить в свой роутер. На данном этапе мы знакомимся с еще одним понятием как Роутер (Router). Роутеры нужны для того, чтобы ограничить возможности Навигатора, поскольку он похож на анти-паттерн God Object. Суть этого паттерна заключается в том, что его повсеместное использование объекта делает его неявной зависимостью в проекте, что потом, при очередном рефакторинге, приводит к большему количеству времени на разработку, появлению багов и ошибок. Чтобы избежать такой ситуации я внес в модуль навигации такое понятие как Роутер. Он представляет собой узкоспециализированную, зависимую от бизнес-логики, обертку над главным навигатором, тем самым снижая шанс того, что разработчик напишет неправильный навигационный код.



Роутерам не нужны аргументы в методах, поскольку они знают на какие экраны и как нужно навигировать, тем самым облегчая разработчику написание кода, связанных с навигацией.



Как показано выше, логика открытия новых экранов тривиально и не обременена различными зависимостями из Android-фреймворка, тем самым она становится переиспользуемоей и скалируемой на другие проекты с другой архитектурой. В заключение, стоит сказать, что навигация скорее находится на presentation – уровне приложения, также как и в случае с View, за протоколом Навигатора скрывается конкретная реализация сильно зависящая от платформы.

Model как реализация подхода Clean Architecture

Путь модели от запроса до экрана

Работа с данными внутри приложения. Слои приложения

Data-слой

Domain-слой

Presentation-слой

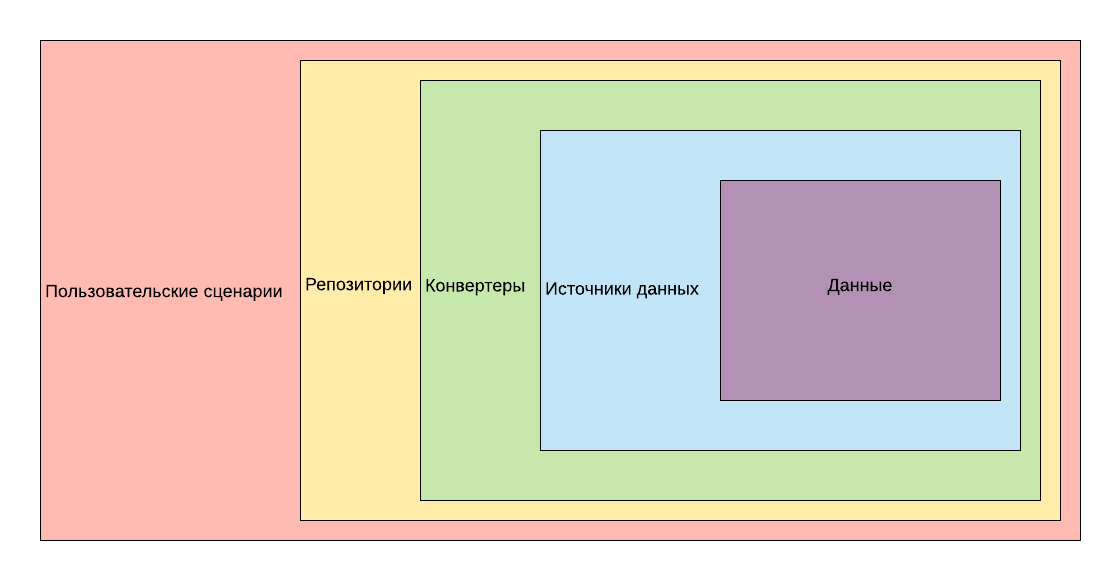
Паттерн Repository и паттерн DataSource

Interactor vs. UseCase или как работать с бизнес логикой

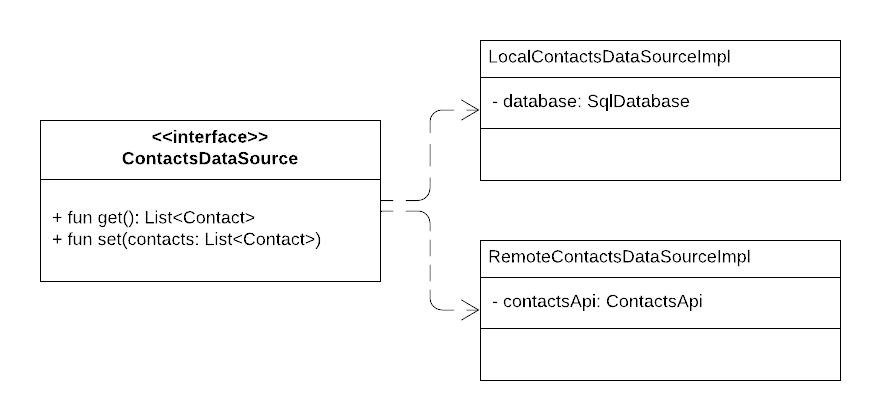
Тестирование бизнес логики

1. Model как реализация подхода Clean Architecture

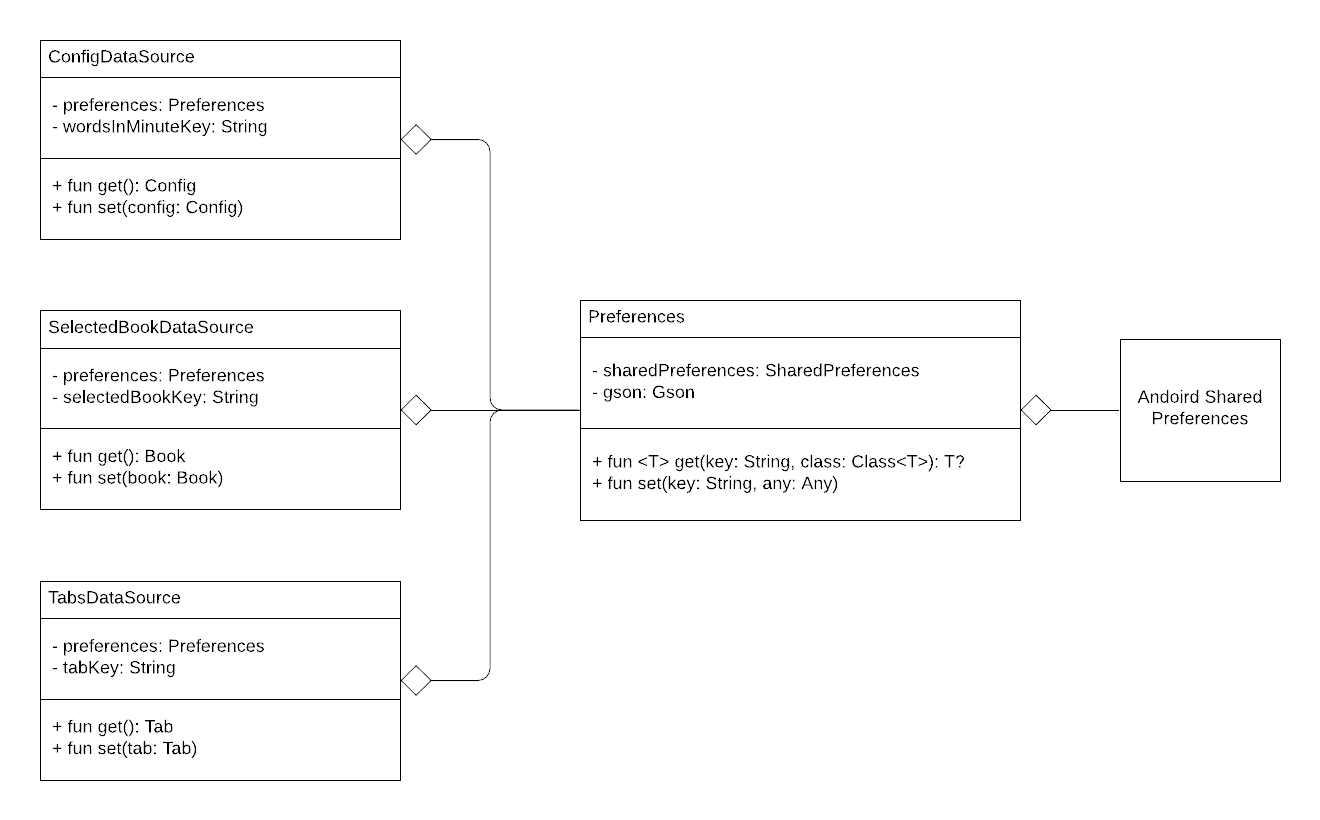
Принцип Clean Architecture представляет собой инкапсуляцию простых правил, соблюдая которые можно писать легко расширяемый и тестируемый код. Чистая архитектура является сборников других подходов и принципов, о некоторых из них я упоминал ранее в разделе “Какие проблемы решает архитектура?”. Теперь стоит рассказать о подходах, используемых применительно к domain-data структуре приложения. Первое, что необходимо отметить, так это структуру отношений domain-data слоев в чистой архитектуре. В центре всего стоит такой компонент как сущности. Сущность представляет собой обычно контейнер данных, который ничего не умеет как переносить информацию со слоя на слой. На каждом уровне данных должна быть своя модель для того, чтобы сохранять предметную область своего класса источника. В больших проектах со сложными бизнес-сценариями принято создавать специальные конвертеры для перевода моделей одного уровня (data) в модели уровня выше (domain) и наоборот. Необходимо это в первую очередь для того, чтобы слои были максимально друг от друга отвязаны через принцип инверсии зависимостей, тем самым спокойно обезопасить себя от ненужных изменений. В моем проекте модели достаточно тривиальны, поэтому от слоя конвертеров я отказался и начал формировать модели для domain-слоя прямо внутри репозиториев, но о них немного позже. Для начала рассмотрим архитектуру самых низших уровней моего приложения, а именно data – слой приложения.



Мое приложение работает с различными информационными устройствами операционной системы. Для работы и сохранения персистентных данных были необходимо различные места ОС. Так как в Android память разделена на память приложения (кэш), которая является не общей, и внешней памяти (локальное хранилище), которое в свою очередь является общей, работа с этими видами памяти была инкапсулирована в отдельные классы, для того, чтобы сохранить принцип единой отвествнности, а позже они стали зависимостями, так называемых источников данных (datasource). DataSource – это источник данных, инкапсулирующий в себе логику получения каких-либо данных, скрытую под определенным протоколом. Изюминка такого подхода в том, что вы можете легко менять имплементацию своего датасорса, меняя непосредственно источник получения данных. Например, вы хотите получить какие-нибудь данные о пользователе, который хранится в локальном хранилище и на веб-сервере, тогда проверив Интернет-соединение клиента можно предоставить имплементацию, которая ходит за данными на сервер, в противном случае вы можете сходить в локальное хранилище для получение тех же данных. Кейс достаточно распространённый, поэтому подменой имплементаций приходится часто пользоваться.

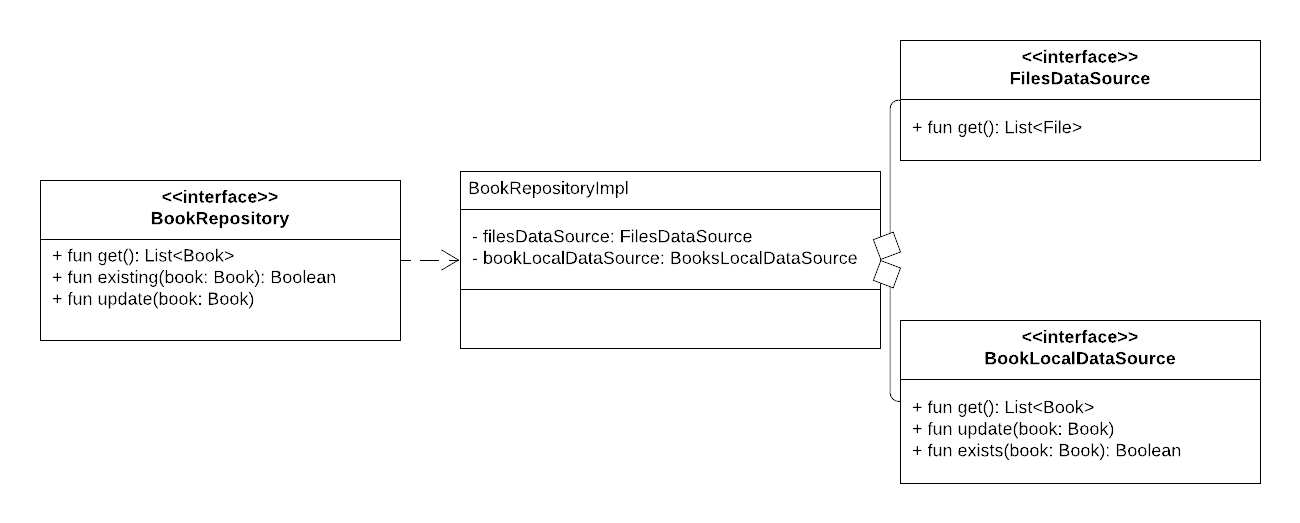


Также датасорсы полезны тем, что скрывают детали реализации той технологии, или библиотеки, которую вы используете. К примеру, вы с легкостью можете заменить SQL базу данных, на модный Realm, достаточно имплементировать необходимые методы датасорса. Также нужно учитывать, что датасорсы в большинстве своем количестве должны удовлетворять принципу единой ответственности, чтобы из них можно было строить логические компоненты, и генерировать domain-модели, не обремененные лишней логикой. Поэтому не стоит бояться создавать однотипные источники данных с одним или двумя методами только для того, чтобы единожды использовать их в репозиториях. С таким подходом, мы сможем легко передать их в любое место и обеспечить прозрачную тестируемость. Инкапсулирование одного источника данных можно рассмотреть на примере кэширования настроек внутри моего приложения



Мы можем видеть, что в данном случае, мы избавляемся от зависимости в виде Android-фрейморка, скрыв Android механизм кэширования данных за классом Preferences, а также мы можем переиспользовать класс с настройками локального кэша в источниках данных, у которых своя определенная ответственность. Плюс такого подхода, что во время написания тестов нам нет необходимости эмулировать поведение Android-фреймворка в виде его механизма кэширования, достаточно подменить результаты вызовов класса Preferences.

Следующим слоем является слой репозиториев, который является мостом между, data и domain слоями. Во всех репозиториях происходит конвертация данных из data-моделей в domain-модели. Поэтому имплементации репозиториев принадлежат data-слою из-за тесноты работы с сущностями data – слоя, и напротив протоколы этих репозиториев уже находятся на domain-уровне. Данные протоколы знают только о domain-сущностях. В рамках чистой архитектуры нет возможности перехода data-модели на domain-уровень, однако обратная ситуация возможно, поскольку иногда модели уровня data бывают идентичными моделям domain, либо же настолько тривиальными, что поддержка таких моделей лишена смысла. Так случилось и в моем проекте. Модель Book, принадлежащая domain-слою, является достаточно тривиальной, поэтому формируется на data уровне без использования лишних конвертеров.



Важно отметить, как например формируются данные в BookRepositoryImpl. На данном примере видно, насколько удобна концепция источников данных. Она позволяет строить логику из отдельных источников данных, давая тем самым возможность писать раздельный, независимый код. Также отдельные абстракции для источников данных дают нагляднее понять, что происходит в коде. В данном случае мы видем, что сначала получаем список файлов, затем уже имеющиеся книги, и в случае совпадения книги и файла в локальном хранилище (базе данных), возвращаем книгу, иначе создаем новую.

