**Overview**

**YAGNI** (**«You aren't gonna need it»**) — процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности, — то есть отказ добавления функциональности, в которой нет непосредственной надобности.

**KISS** (акронимдля «**Keep it simple, stupid**») Принцип KISS утверждает, что большинство систем работают лучше всего, если они остаются простыми, а не усложняются. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать ненужной сложности.

**Don’t repeat yourself**, **DRY** (рус.не повторяйся) — это принципразработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

Когда принцип DRY применяется успешно, изменение единственного элемента системы не требует внесения изменений в другие, логически не связанные элементы. Те элементы, которые логически связаны, изменяются предсказуемо и единообразно.

**SOLID** (сокр. от англ. single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion) - пять основных принципов объектно-ориентированного программирования и проектирования.

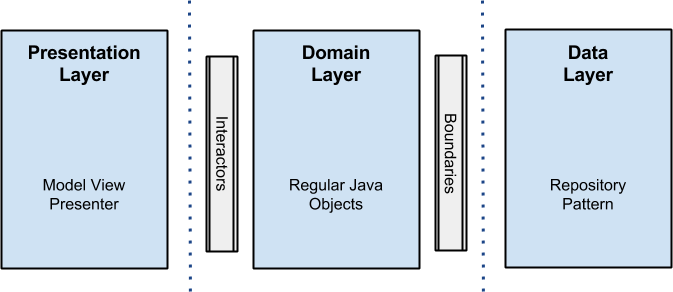
**S** (SRP *The Single Responsibility Principle*) - Принцип единственной ответственности - Каждый класс выполняет лишь одну задачу.

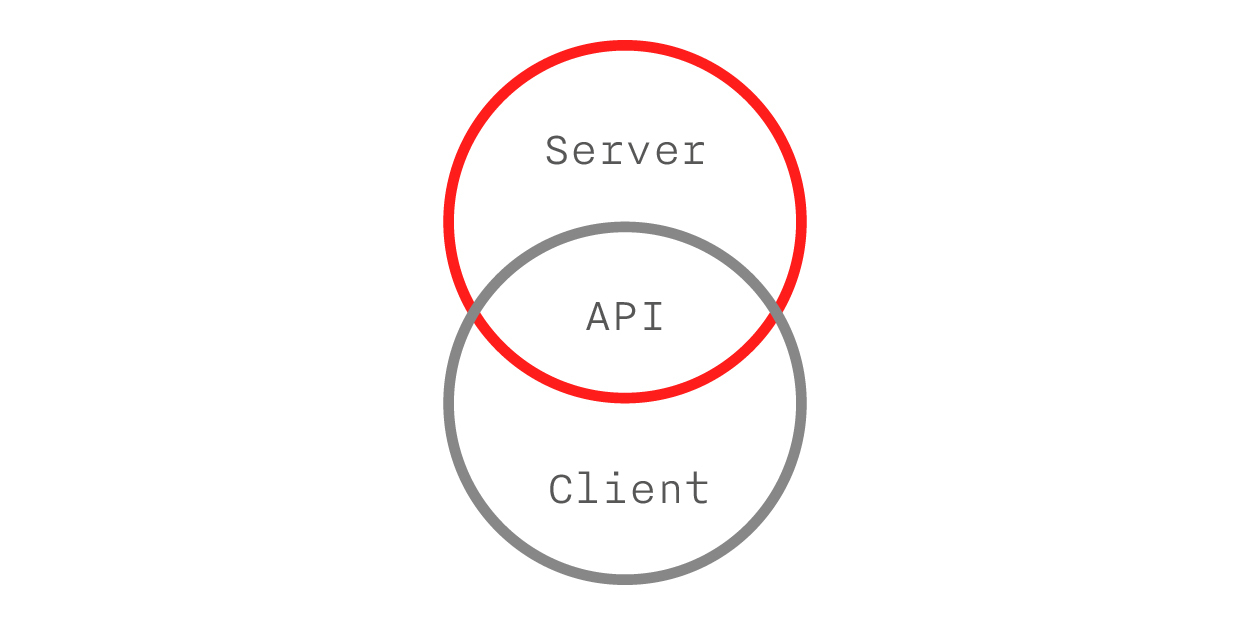
**O (**OCP *The Open Closed Principle***)** - Принцип открытости/закрытости - программные сущности … должны быть открыты для расширения, но закрыты для модификации.

**стройная архитектура**основывается на группе методов, реализующих системы, которые являются:

* **Независимыми от фреймворков.**
* **Тестируемыми.**
* **Независимыми от UI.**
* **Независимыми от БД.**
* **Независимыми от любой внешней службы.**

Наша цель — **разделение задач**таким образом, чтобы бизнес-логика ничего не знала о внешнем мире, так, чтобы можно было тестировать её без никаких зависимостей и внешних элементов. Для достижения этого я предлагаю **разбить проект на 3 слоя**, каждый из которых имеет свою цель и может работать независимо от остальных. Стоит отметить, что каждый слой использует свою модель данных, таким образом можно достигнуть необходимой независимости (вы можете увидеть в коде, для выполнения трансформации данных необходим преобразователь данных (data mapper). Это вынужденная плата за то, чтобы модели внутри приложения не пересекались). Вот схема, как это выглядит:





Дано:

- сервер с документированным API из нескольких **web-сервисов**;

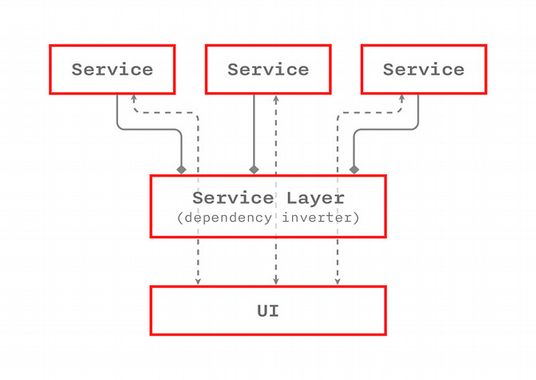
- приложение-клиент и несколько типов устройств, которые это приложение должно поддерживать: планшеты, телефоны, часы и прочее;

- макеты того, как должен выглядеть интерфейс приложения;

- набор пользовательских историй, декларирующих, как приложение должно себя вести;

- список нефункциональных требований, которым система должна соответствовать.

Слои приложения должны общаться между собой посредством ***модельных объектов***— классов с «прозрачной» реализацией, состоящей исключительно из акцессоров и мутаторов (get- и set-методы; фактически, класс будет включать в себя только свойства). Подобные классы не несут с собой никакой логики, и предназначены лишь для хранения и перемещения данных.



Центральной сущностью приложения — «горлышком» песочных часов — будет выступать объект, иначе называемый «инвертор зависимостей». Он же станет *единственным* singleton-классом, и на нём будет лежать единственная ответственность — предоставлять «верхним» слоям приложения *сервисы*.

Как делаются нормальные серверные приложения? Очень просто. Всегда есть какая-то база данных (или их может быть несколько), в которой есть таблицы. Таблицы состоят из записей. Каждая запись содержит набор полей, в которых хранится информация.

Записи представляются в виде *модельных объектов с полями*, тем самым в приложении возникает логическое разграничение по типам данных: одна таблица — один тип данных. Таблица «Пользователи» — тип данных «Пользователь» (поля: ID, ФИО, адрес). Таблица «Сообщения» — тип данных «Сообщение» (поля: ID, Тема сообщения, Тело сообщения, Кому, От кого, Дата). И так далее.

Таким образом, вся логика крутится вокруг таблиц и типов данных:  
  
1. *Модельный объект*, представляющий тип данных.  
2. Сериализация *модельного объекта* для сохранения его в таблицу.  
3. Десериализация *модельного объекта* из базы данных.  
4. Преобразования *модельного объекта*: подсчёт какой-то статистики,  
сортировка данных, поиск данных и т.д.  
5. Сериализация *модельного объекта* для передачи его по сети.  
6. Десериализация *модельных объектов*, приходящих по сети.  
7. Выставленный в сеть web-сервис, соответствующий данному типу  
данных.

Всё это выстраивается в своего рода*стек*, а само серверное приложение состоит из нескольких таких независимых *стеков*, каждый из которых соответствует какому-то типу данных.  
  
Грубо говоря, если по спецификации API у вас имеется web-сервис [api.domain.com/1-0/messages](http://api.domain.com/1-0/messages) с классическим CRUD-интерфейсом — это означает, что за ним стоит вышеупомянутый «стек» для типа данных «сообщение».  
  
**C**reate = POST;  
**R**ead = GET;  
**U**pdate = PUT;  
**D**elete = DELETE.

*Существуют различные интерпретации операторов POST и PUT. В одних реализациях POST отвечает за создание сущностей, в других — за обновление. Канонического трактования не существует.*

Что нужно знать, разрабатывая приложения под Android — это уровень инфраструктуры приложения.

Разные части Android приложения могут вызывать друг друга и взаимодействовать между собой только формально; ни один из них не может обратиться к другому напрямую.

Для каждого состояния жизненного цикла существуют методы, каждый из которых может быть переопределён разработчиком. Эти методы вызываются фреймворком в заранее определённые ключевые моменты, например, когда пользовательский интерфейс показывается на экране, прячется и т.п. В этих методах разработчик может реализовать логику для хранения и восстановления состояния объектов.

В Android-устройствах существует стека пользовательских интерфейсов, в котором текущий видимый интерфейс помещается на вершину, а все остальные сдвигаются вниз (стековая операция «push»). Нажатие «назад» удаляет интерфейс с вершины стека и показывает элемент, который был за ним (стековая операция «pop»). Подобный стек существует в Android. В документации он называется «activity stack» или иногда «back stack».

В плане обработки взаимодействия между пользовательским интерфейсом и его логикой Android следует архитектурному шаблону «Model-View-ViewModel» (MVVM). Для разработчиков это хорошая новость, поскольку MVVM — самая лучшая архитектура GUI-приложений на настоящий момент.

Основной задачей при декомпозиции системы является осуществление двух условий: (1) максимизация связей внутри компонентов (*высокое внутреннее сцепление*, tight internal cohesion) и (2) минимизация связей между компонентами (*низкая внешняя связанность*, loose external coupling).

**СОВЕТ**  
Используйте принцип «тестируемости» класса в качестве «лакмусовой бумажки» хорошего дизайна класса.

Для тестирования всего приложения необходимо использовать интеграционные тесты его целых подсистем.

Работа с сетью в стандартном приложении сводится к решению нескольких задач:

* авторизация
* запрос и отправка данных
* хранение данных
* работа с картинками

При большом количестве различных вкладок в приложении(сообщения, друзья, профили, check-ins, активности пользователей, новости, симпатии) каждый новый контролл требует подгрузки данных. Также есть фоновые запросы к серверу, которые по таймеру запрашивают данные все время работы приложения(это могут быть запросы на кол-во новых сообщений, инвайты в друзья, изменение баланса валюты внутри приложения и тд). Также контроллы необходимо оповещать об обновлениях синхронно или асинхронно, в зависимости от типа запроса.

Как видно картинка может быть использована одна и та же. Если сервер присылает url только на изображение большого разрешения, то необходимо его, после загрузки, кэшировать, ужимать, делать круглым или округлять углы. Все, что потребуется для дизайна приложения.  
Сохраняем оригинал локально и при запросе картинок ищем среди кэшированных файлов. Если оригинал есть, но разрешение нужно меньше или другой формы: уменьшаем картинку и результат сохраняем с названием вида имя\_файла\_разрешение если есть — показываем, нет грузим.

Основное решение — организовать класс, управляющий запросами. Singleton класс(или набор классов) отвечающий за работу с сетью.   
Его задачи:

* Создавать очередь вызовов, автоматически упорядочивать ее в зависимости от верных/не верных ответов сервера
* Посылать уведомления о получении данных подписанным на те или иные методы контроллам (в ios через nsnotification)
* Посылать задания менеджеру загрузки картинок
* Управлять фоновым поток загрузки активностей пользователя(уведомление о новые сообщения, лайке и тд)

В iOS класс реализовался на базе стандартного NSConnection.  
Запросы добавляются в очередь. Будет ли это динамический массив с приоритетами или NSOperationQueue(как в iOS) решать вам. Каждый запрос в очереди имеет приоритет и если возвращается error с потерянной авторизацией при последнем запросе максимальный приоритет получает сформированный запрос на реконнет. Это позволит избежать ситуации, когда пользователь тапает по кнопке, сессия потеряна и возвращая error подключения ничего не происходит.

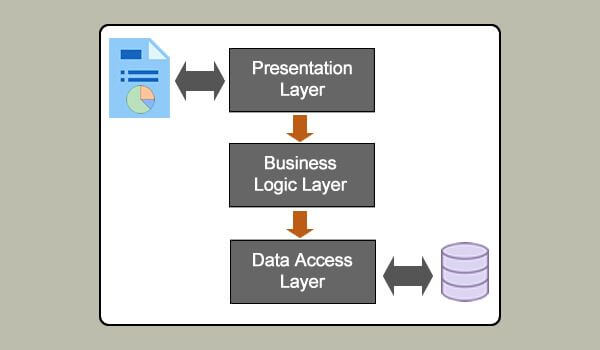
Предположим, что с будущей функциональностью Вы определились. Тогда Вы точно знаете, кто или что будет поставлять данные, кто/что будет их потреблять и какие преобразования данных должна выполнять сама система.

Mobile application architecture usually comprises the following layers:

- **Presentational layer** – this layer comprises of the UI components as well as the UI process components.

- **Business layer** – this layer is formed of business entities, workflows and business component as well.

- **Data layer** – data access components, data utilities and service agents together form this layer.



**What I need to consider (keep in mind):**

- determine the device type – screen resolution, size of screen, CPU characteristics, memory, availability of development framework, storage space.

- internet speed fluctuation.

- power consumption and speed while selecting protocols and hardware for mobile app.

- cache mechanism, data access mechanism, and state management such that it can adjust to slow and intermittent internet connection.

**Presentational layer (UI)**

While selecting navigation layer bear in mind app requirements. Most popular are

- Stacked navigation bar

- tab controller

- modal controller

- single view

- gesture-based navigation

- scroll views

- search driven navigation.

The correct client type must be compliant with the infrastructure. Select the correct data format and put in place a strong data validation technique so that your app can be protected from invalid data input.

Decouple business login from the presentation layer.

**Business layer (UI)**

Logging, caching, validation, security (applying policies), data transformation and exception management are various aspects that the business layer is concerned with. Tasks must be separated into various categories in order to reduce the complexity of the this layer. Using a separate business layer wherever permissible is the ideal approach of designing this layer.

**Data access layer (UI)**

This layer meets with the application requirements and facilitates secure data transactions. Therefore, you must design this layer such that it could scale in the future as the business requirements change down the road.

All the data access functionalities are encapsulated inside this layer and it manages all the data connections needed by the mobile application. Moreover, it handles all CRUD (Create, Read, Update and Delete) operations and data sources.

Implement the least privilege approach in order to protect any attempt to steal or corrupt data by safeguarding the data access mechanism.

