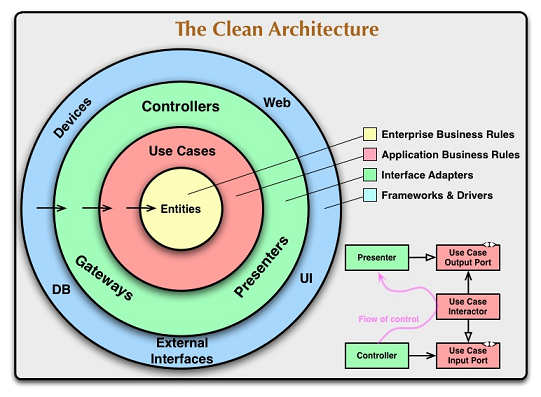
**Clean Architecture**

Существует достаточно много подходов для построения сложных систем с хорошей архитектурой. Несмотря на небольшие различия этих подходов, у них много общего. Разумеется, они все задают способы разбиения приложения на отдельные модули. При этом в каждой системе как минимум есть модули, содержащие бизнес-логику приложения, и модули для отображения данных. И каждый подход в итоге позволяет построить систему, которая удовлетворяет следующим принципам:

1. Архитектура должна быть независима от различных фреймворков. Разумеется, в современном мире мы не можем обходиться без каких-то библиотек, которые позволяют решать задачи намного быстрее и чаще эффективнее, чем это сделали бы мы в случае самостоятельной реализации. Но здесь важно понимать, что библиотека должна встраиваться в вашу архитектуру, а не архитектура должна подстраиваться под выбранную библиотеку. При использовании библиотеки, которая вынуждает вас переделывать архитектуру приложения, вы всегда будете сталкиваться с определенными ограничениями этой библиотеки и не сможете перестраивать архитектуру нужным вам образом. Нужно использовать библиотеки только в качестве вспомогательных инструментов.
2. Система должна быть протестирована. При этом вы должны иметь возможность тестировать как модули системы по отдельности, так и тестировать взаимодействие этих модулей между собой и интеграцию их в системе. Кроме того необходимо тестировать систему без UI, реального сервера и работы с базой данных, то есть архитектура должна быть независима от окружения.
3. Из предыдущих пунктов плавно вытекают и следующие принципы, которые говорят о том, что ваше приложение должно быть независимо от всего: от интерфейса пользователя, от работы с базой данных, от работы сервера и от других элементов окружения. Независимость архитектуры от окружения очень важна, так как это позволяет менять различные компоненты окружения без изменения самой архитектуры. Что подразумевается под изменением компонентов архитектуры? Это может быть изменение в выборе базы данных (или же вообще отказ от нее) или же изменение в UI-части приложения (к примеру, нужно изменить внешний вид экрана). Если для внесения таких изменений требует от вас изменения архитектуры, то, возможно, вам стоит пересмотреть архитектуру.

Для построения архитектуры системы предлагается рассмотреть следующую схему, где каждая концентрическая окружность является определенным компонентом системы:



Как видно из схемы, система состоит из бизнес-объектов, объектов для управления данными и бизнес-логикой, из слоя представления и слоя высокоуровневых фреймворков. Можно сказать, что в общем случае это все такое же разбиение на представление данных и логику работы с ними.

Такое разделение системы на слои является весьма логичным и понятным. И пока оно не привносит ничего нового. Главным является правило зависимостей – ни один внутренний слой не должен знать ничего о внешнем. Именно это и позволяет строить независимую архитектуру, принципы которой были описаны выше.

Рассмотрим элементы системы более детально, чтобы понять, в чем заключается роль каждого элемента и как он должен взаимодействовать с другими элементами системы.

**Бизнес-объекты**

С точки зрения работы с системой в итоге мы всегда работаем с определенными сущностями, которые определяются требованиями системы. По сути бизнес-объекты – это классы моделей с определенными методами или же набор каких-то структур данных. Эти классы отвечают логике вашего приложения, и они должны определять самые общие правила поведения.

Поскольку это самый внутренний слой, то он будет меняться только в крайнем случае, когда вы решите изменить саму суть системы. И, разумеется, он останется нетронутым, когда будет меняться, к примеру, способ работы с данными или интерфейс пользователя.

**Сценарии взаимодействия**

Этот слой содержит реализацию основных методов для работы системы и организует работу с данными и бизнес-объектами. Он использует бизнес-объекты и их логику для того, чтобы выполнить свои задачи. Этот слой можно рассматривать как некоторый посредник между бизнес-объектами и непосредственно слоем представления данных (такое определение немного условно, так как по правилу зависимостей этот посредник не должен знать ничего о слое представления).

Разумеется, в силу правила зависимостей этот слой также не будет меняться при изменении какого-то из элементов окружения. И изменения на этом слое в свою очередь не будут затрагивать бизнес-объекты.

**Слой представления**

На этом слое осуществляется преобразование данных из формата, который используют бизнес-объекты или сценарии взаимодействия, в формат, необходимый для работы системы. Под работой системы в данном случае подразумевается передача данных для отображения пользователю или другой службе.

На этом слое реализуются такие архитектурные паттерны как Model-View-Controller или Model-View-Presenter. К примеру, Controller может передавать данные в сценарии взаимодействия, получать результат и передавать его для отображения во View.

**Фреймворки**

Изначально мы говорили, что система не должна зависеть ни от каких фреймворков. Именно поэтому слой фреймворков на схеме является самым внешним. Поскольку именно на этом уровне пропадает вся абстракция, и мы используем конкретные средства для решения определенных задач. Под конкретными средствами здесь подразумевается база данных, фреймворк для UI и другие службы.

Поскольку мы соблюдаем правило зависимостей, внутренние слои ничего не знают о конкретных используемых фреймворках, что позволяет легко изменить любой из них без изменения внутренних слоев.

В описанной схеме указано 4 слоя, и этого достаточно для большинства систем. Но, разумеется, при необходимости эту систему можно масштабировать и увеличивать количество слоев. Самое главное при этом – сохранять правило зависимостей.

Использование такого подхода позволяет создать архитектуру, принципы которой были описаны в начале раздела. В силу правила зависимостей мы получаем соответствие принципам 1 и 3, а такое разбиение системы на модули позволяет тестировать ее на разных уровнях.

Однако нужно понимать, что это теоретическое изложение, которое является достаточно общим и должно удовлетворять любой системе. На практике всегда нужно адаптировать архитектуру под конкретную задачу или систему. В нашем случае такой задачей являются клиент-серверные Android-приложения, поэтому перейдем к тому, как применить рассмотренные принципы к их архитектуре.

**Clean Architecture Android**

Нужно сказать, что архитектура Android-приложений обычно не бывает слишком сложной, поэтому, возможно, схему из Clean Architecture можно упростить без потери качества. И эту схему в любом случае нужно как-то адаптировать для конкретного использования в Android.

Такая адаптация была изложена в статье Фернандо Цехаса “Architecting Android… The clean way?”, которую мы и разберем детально.

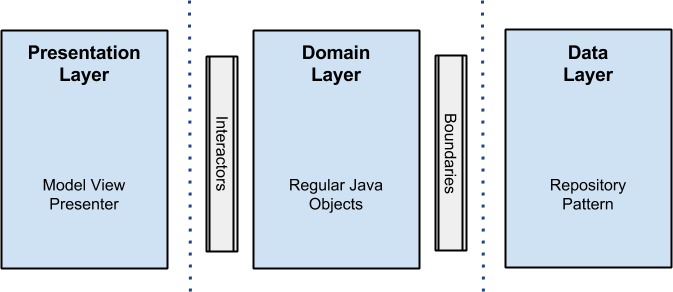
Наиболее критичной с точки зрения тестирования приложения является бизнес-логика или бизнес-правила, определяющие суть работы приложения. И они должны быть в первую очередь независимы от других элементов и протестированы.

Чтобы достичь независимости и возможности тестирования, предлагается разбить приложение на 3 ключевых слоя:

* Слой данных (Data Layer)
* Слой бизнес-логики (Domain Layer)
* Слой представления (Presentation Layer)

При этом чтобы обеспечить максимальную независимость этих слоев, на каждом из них используется своя модель данных, которая конвертируется при взаимодействии между слоями.

Схема этих слоев выглядит следующим образом:



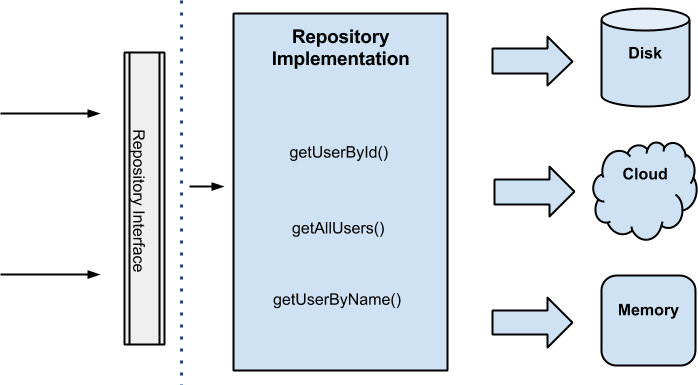
Основная идея всех архитектурных паттернов сохраняется – мы разбиваем код на модули на несколько слоев, среди которых обязательно есть слой, содержащий логику приложения, и слой, отвечающий за представление данных и работу с UI.

В предложенном варианте от Цехаса к этим двум основным слоям добавляется слой для работы с данными. Это позволяет упростить клиент-серверное взаимодействие и избавляет бизнес-логику приложения от необходимости работать с получением данных из различных источников.

Рассмотрим указанные слои подробнее.

**Слой данных**

Данный слой отвечает в первую очередь за получение данных из различных источников и их кэширование. Он реализуется за счет паттерна Repository, и его общую схему можно представить следующим образом:



К примеру, когда необходимо получить какого-то пользователя по имени, реализация репозитория проверяет наличие этого пользователя в локальном хранилище, при отсутствии сохраненного пользователя она отправляет запрос на сервер и получает ответ, который сохраняется в локальное хранилище и возвращается. Или, к примеру, репозиторий может всегда обращаться к серверу, а уже в случае ошибки возвращать сохраненный результат.

Существует несколько плюсов от использования такого подхода. Во-первых, другие слои, которые запрашивают данные, не знают о том, откуда эти данные приходят. Более того, им не нужно этого знать, так как это усложняет логику работы и модуль берет на себя лишнюю ответственность. Во-вторых, слой данных в таком случае выступает единственным источником информации, что, как обсуждалось ранее, очень удобно.

**Слой бизнес-логики**

На слое бизнес-логики содержится, как ни странно, вся бизнес-логика приложения. Этот слой является неким объединением слоев сценариев взаимодействия и бизнес-логики в оригинальной “чистой” архитектуре. Именно к этому слою обращается слои представления для выполнения запросов и получения данных.

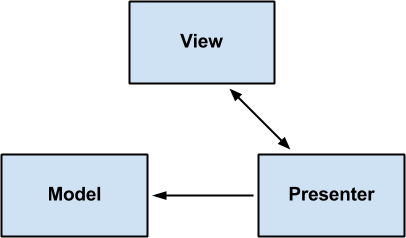
Фернандо Цехас предлагает реализовывать слой бизнес-логики в виде Java-модуля, который не содержит никаких зависимостей от Android-классов. И это хороший подход, так как для реализации бизнес-логики нам нужны только классы моделей и стандартные средства языка Java. Более того, такой подход позволит легко тестировать этот слой с помощью обычных тестов на JUnit, что очень удобно. В таком случае иногда не будет возможности выполнить какой-либо метод или использовать некоторые классы из других слоев. Поэтому для взаимодействия с этим слоем используются интерфейсы.

**Слой представления**

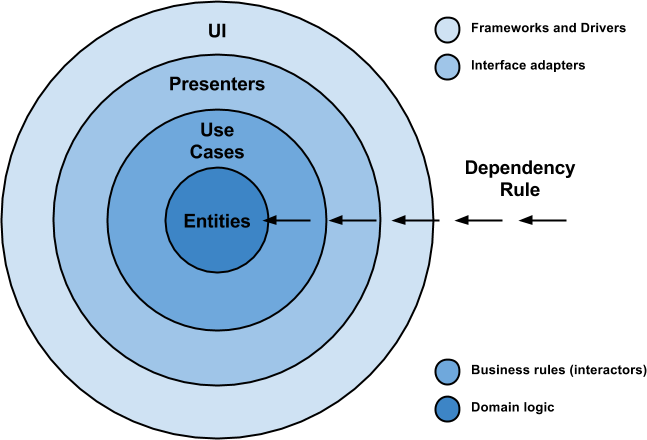
И, разумеется, приложение – это в первую очередь взаимодействие с пользователем. Поэтому нам нужен специальный слой, который будет отвечать за логику отображения данных на экране, за взаимодействие с пользователем и за другие процессы, связанные с UI. Этот слой не должен содержать логику приложения, не связанную с UI.

Именно этот слой привязывается к экранам и помогает организовать взаимодействие со слоем бизнес-логики и работу с данными. Этот слой может быть реализован с использованием любого предпочитаемого паттерна, к примеру, MVC, MVP, MVVM и других.

Конечно, наиболее часто используется паттерн MVP, который мы и будем рассматривать дальше. Подробнее его мы рассмотрим далее, а пока можно сказать, что это достаточно простой паттерн, который позволит нам разделить экран на UI-часть (View), на логику работы с UI (Presenter) и объекты для взаимодействия с UI (Model). В общем виде этот паттерн выглядит следующим образом:



И, разумеется, при разбиении приложения на такие слои нельзя забывать о главном правиле из “чистой” архитектуры – правиле зависимостей:



Теперь мы понимаем, как разбить приложение на модули и сделать эти модули относительно независимыми друг от друга, но нужно прояснить еще три вопроса, которые у нас постоянно возникают при построении архитектуры клиент-серверного приложения.

Во-первых, это вопрос про тестирование. Да, мы сказали, что, если мы строим приложение в виде независимых модулей, то их легко тестировать. Но как именно осуществляется это тестирование на практике? Вопросы тестирования мы будем разбирать в последующих лекциях, но пока приблизительно рассмотрим этот вопрос.

Очевидно, что самый важный слой с точки зрения тестирования – это слой бизнес-логики. А так как мы используем его в виде обычного Java-модуля, то тестировать его очень просто. Для этого достаточно стандартного фреймворка JUnit и, возможно, Mockito. Здесь нет проблем, специфичных для системы Android.

Слой данных также тестируются с помощью Unit-тестов на JUnit, но это уже  требует определенных усилий, так как этот модуль содержит зависимости от классов Android. К примеру, для тестирования этого слоя можно дополнительно использовать фреймворк Robolectric.

И последний слой, слой представления, не содержит критичной логики, зато содержит много UI-элементов, которые также должны быть протестированы. Для этого можно воспользоваться специальными фреймворками, которые позволяют проводить интеграционное и UI-тестирование приложений, к примеру, Espresso.

Второй вопрос связан с обработкой ошибок. Ошибки в такой схеме могут возникать на любом уровне, но больше всего нас интересуют ошибки сервера, которые возникают на уровне данных. Здесь есть несколько возможных решений:

* В случае ошибки при получении данных можно достать данные из локального хранилища. Достаточно часто такая политика вполне оправдана, но при этом все равно стоит как-то уведомлять пользователя о произошедшей ошибке.
* Передавать ошибку через механизм обратного вызова. При получении ошибки в слое данных слой бизнес-логики может выполнить ее первичную обработку и передать ее слою представления, который и отобразит ошибку пользователю. Этот подход можно использовать и в сочетании с первым вариантом.

И последний открытый вопрос – это вопрос об управлении жизненным циклом. Поскольку слой данных никак не относится к проблемам пересоздания Activity и другим проблемам жизненного цикла, а слой бизнес-логики вообще не знает ничего о системе Android, то очевидно, что обработкой жизненного цикла должен заниматься уровень представления. К примеру, хорошим вариантом будет передача в Presenter делегата для управления жизненным циклом, что и будет использоваться далее. Но, разумеется, здесь можно использовать любой из способов, которые были рассмотрены на первой лекции.