 Архитектура Web приложения Многослойная архитектура (2- слойная, 3-слойная), ONION архитектура, CLEAN архитектура

Обзор Независимость данных в реляционной СУБД N-уровневая архитектура

НЕЗАВИСИМОСТЬ ДАННЫХ В РЕЛЯЦИОННОЙ СУБД Отделение представления от логики и данных

Архитектура базы данных и представления Жестки диск User 1User 2 View 2View 1 Схема БД Система хранения То что видит пользователь Таблицы и ссылки Файлы на диске Каждый уровень не зависит от уровня ниже

Логический и физический уровни Жестки диск User 1User 2 View 2View 1 Схема БД Система хранения Логический уровень Физический уровень

Независимость данных Логическая независимость: возможность изменять логическую схему без изменения внешнего вида (кода приложения) – Можно добавлять новые поля, новые таблицы без изменения представления – Можно изменять структуру таблиц без изменения представления Физическая независимость: возможность изменять физическую схему без изменения логической схемы – Можно изменять размер пространства хранения – Тип данных может быть изменён по причинам оптимизации Нужно всегда отделять представление - VIEW (то что видит пользователь) от модели- MODEL (данных сервиса)

N-УРОВНЕВАЯ АРХИТЕКТУРА

Необходимость слоёв N-уровневая архитектура имеет следующие компоненты: – Уровень представления – Уровень бизнес логики – Данные N-уровневая архитектура пытается разделить компоненты на разные уровни (слои): – Физическое разделение – Логическое разделение

1-уровневая архитектура Все 3 слоя на одной машине – Весь код и все процессы происходят на одной машине Представление, логика, уровень данных сильно связаны – Расширяемость (Scalability): на одном процессоре можно запустить только ограниченное число процессов – Переносимость (Portability): для перемещения на новую машину придётся переписать весь код – Поддержка (Maintenance): при изменении одного слоя придётся изменять остальные Data Base Presentation Logic Business Logic Data Access Logic Приложение

2-уровневая архитектура База данных работает на сервере – Отделение данных от клиента – Клиент может переключить базу данных Отделение представления от данных – Высокая нагрузка на сервер – Потенциальные проблемы с задержкой в сети – Уровень представления и логики остаются сильно связанны Business Logic Presentation Logic Data Access Logic Data Base Data Layer Client Server

3-уровневая архитектура Каждый слой может быть потенциально запущен на отдельной машине Представление логика и данные разделены Presentation Layer Содержит Presentation Logic Business Layer Содержит Business Logic Data Layer Содржит Data Access Logic Data Base ClientServerDB Server

Любое приложение Взять общее число клиентов Число клиентов 4 Взять список клиентов за последний год Сложить всех клиентов вместе Запрос Клиент 1 Клиент 2 Клиент 3 клиент 4 База данных Хранилище Уровень представления Уровень логики Уровень данных Принципы архитектуры: Клиент-серверная архитектура Каждый слой (данные, представление и логика) не зависит от остальных и не зависит от реализации Несоединённые слои вообще никогда не взаимодействуют Изменение платформы влияет только на тот уровень который на ней находится

Любое приложение Взять общее число клиентов Число клиентов 4 Взять список клиентов за последний год Сложить всех клиентов вместе Запрос Клиент 1 Клиент 2 Клиент 3 клиент 4 База данных Хранилище Уровень представления Уровень логики Уровень данных Предоставляет графический интерфейс Обрабатывает пользовательские события Иногда называют GUI или client view of front- end

Любое приложение Взять общее число клиентов Число клиентов 4 Взять список клиентов за последний год Сложить всех клиентов вместе Запрос Клиент 1 Клиент 2 Клиент 3 клиент 4 База данных Хранилище Уровень представления Уровень логики Уровень данных Набор правил для работы с данными Может обрабатывать запросы нескольких пользователей Иногда называют middleware или back- end Недолжен содержать пользовательских форм или непосредственно обращаться к данным

Любое приложение Взять общее число клиентов Число клиентов 4 Взять список клиентов за последний год Сложить всех клиентов вместе Запрос Клиент 1 Клиент 2 Клиент 3 клиент 4 База данных Хранилище Уровень представления Уровень логики Уровень данных Физическое хранилище для данных (persistence) Управляет доступом к БД или файловой системе Иногда называется back- end Недолжен содержать пользовательских форм или бизнес логики

3-уровневая архитектура Уровень представления – Статический или динамически сгенерированный контент отображаемый через браузер (front-end) Уровень логики – Уровень подготовки данных для динамически генерируемого контента, уровень сервера приложений (application server). Middleware платформы: JavaEE, ASP.NET, PHP, ColdFusion Уровень данных – База данных включающая в себя данный и систему управления над ними или же готовая RDBMS система, предоставляющая доступ к данным и методы управления (back-end)

Преимущества Независимость уровней – Лёгкость в поддержке – Отдельные компоненты можно использовать в других задачах – Задача разработки хорошо делится и поэтому может быть быстрее решена (уровни можно разрабатывать параллельно) Web дизайнер делает уровень представления Инженер (Software Engineer) делает логику Администратор БД делает модель данных

ONION архитектура

Большинство традиционных архитектур поднимают фундаментальные проблемы тесной связи и разделения задач. Архитектура Onion была представлена ​​Джеффри Палермо, чтобы предоставить лучший способ создания приложений с точки зрения лучшей тестируемости, ремонтопригодности и надежности. Onion Architecture решает проблемы, с которыми сталкиваются трехуровневые и многоуровневые архитектуры, и предоставляет решение общих проблем. Уровни луковой архитектуры взаимодействуют друг с другом с помощью интерфейсов.

Принципы

Луковая архитектура основана на принципе инверсии управления. Луковая архитектура состоит из нескольких концентрических слоев, взаимодействующих друг с другом по направлению к ядру, представляющему домен. Архитектура зависит не от уровня данных, как в классических многоуровневых архитектурах, а от реальных моделей предметной области.

Проблема и решение

В соответствии с традиционной архитектурой уровень пользовательского интерфейса взаимодействует с бизнес-логикой, а бизнес-логика взаимодействует с уровнем данных, и все уровни перемешаны и сильно зависят друг от друга. В трехуровневых и многоуровневых архитектурах ни один из уровней не является независимым; этот факт вызывает опасения. Такие системы очень сложно понять и поддерживать. Недостатком этой традиционной архитектуры является ненужное сопряжение.

Слои луковой архитектуры

Уровни луковой архитектуры

Onion Architecture решила эту проблему, определив уровни от ядра до инфраструктуры. Он применяет основное правило, перемещая все соединения к центру. Эта архитектура, несомненно, ориентирована на объектно-ориентированное программирование и ставит объекты выше всех остальных.

В центре Onion Architecture находится модель предметной области, которая представляет бизнес-объекты и объекты поведения. Вокруг уровня домена находятся другие слои с большим количеством поведений.

Слои луковой архитектуры

В Onion Architecture используется концепция слоев, но они отличаются от трехуровневых и многоуровневых архитектурных слоев. Давайте посмотрим, что представляет и должен содержать каждый из этих слоев.

Уровень домена

В центральной части луковой архитектуры существует уровень домена; этот слой представляет объекты бизнеса и поведения. Идея состоит в том, чтобы все ваши доменные объекты находились в этом ядре. Он содержит все объекты домена приложения. Помимо объектов домена, у вас также могут быть интерфейсы домена. У этих доменных сущностей нет никаких зависимостей. Объекты домена также плоские, какими они должны быть, без какого-либо тяжелого кода или зависимостей.

Уровень репозитория

Этот уровень создает абстракцию между сущностями предметной области и бизнес-логикой приложения. На этом уровне мы обычно добавляем интерфейсы, которые обеспечивают поведение при сохранении и извлечении объектов, обычно с использованием базы данных. Этот уровень состоит из шаблона доступа к данным, который представляет собой более слабосвязанный подход к доступу к данным. Мы также создаем общий репозиторий и добавляем запросы для извлечения данных из источника, сопоставляем данные из источника данных с бизнес-объектом и сохраняем изменения бизнес-объекта в источнике данных.

Уровень услуг

Уровень сервиса содержит интерфейсы с общими операциями, такими как добавление, сохранение, редактирование и удаление. Кроме того, этот уровень используется для связи между уровнем пользовательского интерфейса и уровнем репозитория. Уровень услуг также может содержать бизнес-логику для объекта. На этом уровне сервисные интерфейсы отделены от его реализации, с учетом слабой связи и разделения задач.

Уровень пользовательского интерфейса

Это самый внешний уровень, который отвечает за периферийные аспекты, такие как пользовательский интерфейс и тесты. Для веб-приложения он представляет проект веб-API или модульного теста. На этом уровне реализован принцип внедрения зависимостей, так что приложение создает слабосвязанную структуру и может взаимодействовать с внутренним уровнем через интерфейсы.

Реализация луковой архитектуры

Руководящие принципы Onion Architecture не дают никаких указаний о том, как должны быть реализованы слои. Архитектор должен принять решение о реализации и может выбрать любой уровень класса, пакета, модуля или чего-либо еще, что требуется для добавления в решение.

Преимущества и недостатки луковой архитектуры

Ниже приведены преимущества реализации луковой архитектуры: Уровни луковой архитектуры связаны через интерфейсы.

Имплантации предоставляются во время выполнения.

Архитектура приложения построена на модели предметной области.

Все внешние зависимости, такие как доступ к базе данных и вызовы служб, представлены во внешних слоях.

Никаких зависимостей внутреннего слоя с внешними слоями.

Гибкая, устойчивая и портативная архитектура.

Можно быстро протестировать, потому что ядро ​​приложения ни от чего не зависит.

Несколько недостатков луковой архитектуры:

Непросто понять для новичков, требует обучения.

Архитекторы в большинстве случаев путают распределение обязанностей между слоями.

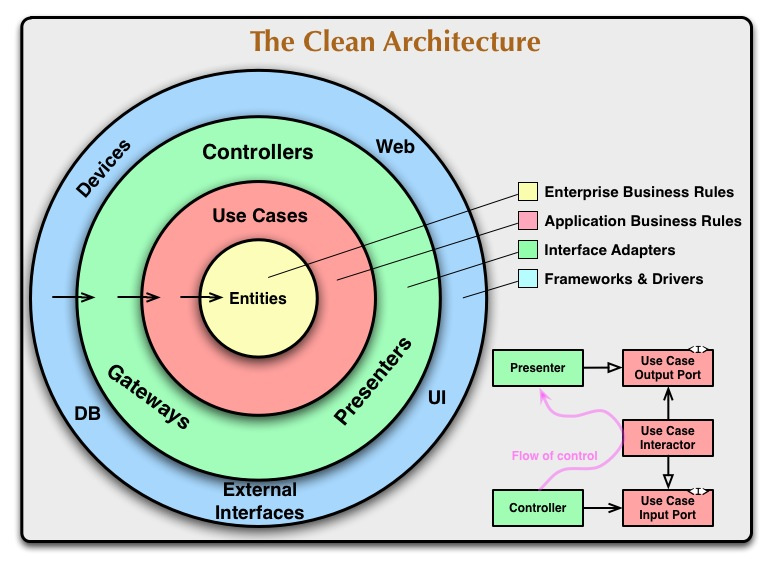
Широко используемые интерфейсы

Clean Architecture

# Истоки

В 2011 году [Robert C. Martin](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82), также известный как Uncle Bob, опубликовал статью [Screaming Architecture](https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html), в которой говорится, что архитектура должна «кричать» о самом приложении, а не о том, какие фреймворки в нем используются. Позже вышла [статья](https://8thlight.com/blog/uncle-bob/2011/11/22/Clean-Architecture.html), в которой Uncle Bob даёт отпор высказывающимся против идей чистой архитектуры. А в 2012 году он опубликовал статью «**[The Clean Architecture](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html)**», которая и является основным описанием этого подхода. Кроме этих статей я также очень рекомендую посмотреть [видео](https://www.youtube.com/watch?v=Nsjsiz2A9mg) выступления Дяди Боба.

Вот **оригинальная схема** из статьи, которая первой всплывает в голове разработчика, когда речь заходит о Clean Architecture:

  
**Сlean Architecture**

Clean Architecture объединила в себе идеи нескольких других архитектурных подходов, которые сходятся в том, что **архитектура должна**:

* быть тестируемой;
* не зависеть от UI;
* не зависеть от БД, внешних фреймворков и библиотек.

Это достигается разделением на слои и следованием Dependency Rule (правилу зависимостей).

### **Dependency Rule**

**Dependency Rule говорит нам, что внутренние слои не должны зависеть от внешних**. То есть наша бизнес-логика и логика приложения не должны зависеть от презентеров, UI, баз данных и т.п. На оригинальной схеме это правило изображено стрелками, указывающими внутрь.

В статье сказано: имена сущностей (классов, функций, переменных, чего угодно), объявленных во внешних слоях, не должны встречаться в коде внутренних слоев.

Это правило позволяет строить системы, которые будет проще поддерживать, потому что изменения во внешних слоях не затронут внутренние слои.

### **Слои**

Uncle Bob выделяет 4 слоя:

* **Entities**. Бизнес-логика общая для многих приложений.
* **Use Cases (Interactors)**. Логика приложения.
* **Interface Adapters**. Адаптеры между Use Cases и внешним миром. Сюда попадают Presenter’ы из MVP, а также Gateways (более популярное название репозитории).
* **Frameworks**. Самый внешний слой, тут лежит все остальное: UI, база данных, http-клиент, и т.п.

Подробнее, что из себя представляют эти слои, мы рассмотрим по ходу статьи. А пока остановимся на передаче данных между ними.

### **Переходы**

**Переходы между слоями осуществляются через Boundaries**, то есть через два интерфейса: один для запроса и один для ответа. Их можно увидеть справа на оригинальной схеме (Input/OutputPort). Они нужны, чтобы внутренний слой не зависел от внешнего (следуя Dependency Rule), но при этом мог передать ему данные.

Оба интерфейса относятся к внутреннему слою (обратите внимание на их цвет на картинке).

Смотрите, Controller вызывает метод у InputPort, его реализует UseCase, а затем UseCase отдает ответ интерфейсу OutputPort, который реализует Presenter. То есть данные пересекли границу между слоями, но при этом все зависимости указывают внутрь на слой UseCase’ов.

Чтобы зависимость была направлена в сторону обратную потоку данных, применяется [*принцип инверсии зависимостей*](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B8_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9) (буква D из аббревиатуры [SOLID](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))). То есть, вместо того чтобы UseCase напрямую зависел от Presenter’a (что нарушало бы Dependency Rule), он зависит от интерфейса в своём слое, а Presenter должен этот интерфейс реализовать.

Точно та же схема работает и в других местах, например, при обращении UseCase к Gateway/Repository. Чтобы не зависеть от репозитория, выделяется интерфейс и кладется в слой UseCases.

Что же касается данных, которые пересекают границы, то это должны быть **простые структуры**. Они могут передаваться как [DTO](https://ru.wikipedia.org/wiki/DTO) или быть завернуты в HashMap, или просто быть аргументами при вызове метода. Но они обязательно должны быть в форме более удобной для внутреннего слоя (лежать во внутреннем слое).

**Особенности мобильных приложений**

Надо отметить, что **Clean Architecture была придумана с немного иным типом приложений на уме**. Большие серверные приложения для крупного бизнеса, а не мобильные клиент-серверные приложения средней сложности, которые не нуждаются в дальнейшем развитии (конечно, бывают разные приложения, но согласитесь, в большей массе они именно такие). Непонимание этого может привести к [overengineering](https://en.wikipedia.org/wiki/Overengineering)’у.

На оригинальной схеме есть слово Controllers. Оно появилось на схеме из-за frontend’a, в частности из Ruby On Rails. Там зачастую разделяют Controller, который обрабатывает запрос и отдает результат, и Presenter, который выводит этот результат на View. Многие не сразу догадываются, но в android-приложениях Controllers не нужны.

Ещё в статье Uncle Bob говорит, что слоёв не обязательно должно быть 4. Может быть любое количество, но Dependency Rule должен всегда применяться.

.**Заблуждение: Слои и линейность**

Сравнивая оригинальную схему от Uncle Bob’a и cхему Fernando Cejas’a многие начинают путаться. Линейная схема воспринимается проще, и люди начинают неверно понимать оригинальную. А не понимая оригинальную, начинают неверно толковать и линейную. Кто-то думает, что расположение надписей в кругах имеет сакральное значение, или что надо использовать Controller, или пытаются соотнести названия слоёв на двух схемах. **Смешно и грустно, но основные схемы стали основными источниками заблуждения!**

Постараемся это исправить. Для начала давайте очистим основную схему, убрав из нее лишнее для нас. И переименуем Gateways в Repositories, т.к. это более распространенное название этой сущности.

Стало немного понятнее. Теперь мы сделаем вот что: разрежем слои на части и превратим эту схему в блочную, где цвет будет по-прежнему обозначать принадлежность к слою.

Как я уже сказал выше, цвета обозначают слои. А стрелка внизу обозначает Dependency Rule.

На получившейся схеме уже проще представить себе течение данных от UI к БД или серверу и обратно. Но давайте сделаем еще один шаг к линейности, расположив слои по категориям:

Я намеренно не называю это разделение слоями. Потому что мы и так делим слои. Я называю это категориями или частями. Можно назвать как угодно, но повторно использовать слово «слои» не стоит.

Надеюсь теперь вcё начало вставать на свои места. Думаю теперь это тоже стало понятнее. В Domain части у нас находятся и UseCases и Entities.

Такая схема воспринимается проще. Ведь обычно события и данные в наших приложениях ходят от UI к backend’у или базе данных и обратно. Давайте изобразим этот процесс:

Красными стрелками показано течение данных.

Событие пользователя идет в Presenter, тот передает в Use Case. Use Case делает запрос в Repository. Repository получает данные где-то, создает Entity, передает его в UseCase. Так Use Case получает все нужные ему Entity. Затем, применив их и свою логику, получает результат, который передает обратно в Presenter. А тот, в свою очередь, отображает результат в UI.

На переходах между слоями (не категориями, а слоями, отмеченными разным цветом) используются Boundaries, описанные ранее.

Теперь, когда мы **поняли, как соотносятся две схемы**, давайте рассмотрим следующее заблуждение.

# Заблуждение: Слои, а не сущности

Как понятно из заголовка, кто-то думает, что на схемах изображены сущности (особенно это затрагивает UseCases и Entities). Но это не так.

**На схемах изображены слои, в них может находиться много сущностей**. В них будут находиться интерфейсы для переходов между слоями (Boundaries), различные DTO, основные классы слоя (Interactors для слоя UseCases, например).

Не будет лишним взглянуть на схему, собранную из частей, показанных в видео выступления Uncle Bob’a. На ней изображены классы и зависимости:

Видите двойные линии? Это границы между слоями. Разделение между слоями Entities и UseCases не показаны, так как в видео основной упор делался на том, что вся логика (приложения и бизнеса) отгорожена от внешнего мира.

C Boundaries мы уже знакомы, интерфейс Gateway – это то же самое. Request/ResponseModel – просто DTO для передачи данных между слоями. По правилу зависимости они должны лежать во внутреннем слое, что мы и видим на картинке.

Про Controller мы тоже уже говорили, он нас не интересует. Его функцию у нас выполняет Presenter.

А ViewModel на картинке – это не ViewModel из [MVVM](https://ru.wikipedia.org/wiki/Model-View-ViewModel) и не ViewModel из [Architecture Components](https://developer.android.com/topic/libraries/architecture/viewmodel.html). Это просто DTO для передачи данных View, чтобы View была тупой и просто сетила свои поля. Но это уже детали реализации и будет зависеть от выбора презентационного паттерна и личных подходов.

В слое UseCases находятся не только Interactor’ы, но также и Boundaries для работы с презентером, интерфейс для работы с репозиторием, DTO для запроса и ответа. Отсюда можно сделать вывод, что на оригинальной схеме отражены всё же слои.

# Заблуждение: Entities

Entities по праву занимают первое место по непониманию.

Мало того, что почти никто (включая меня до недавнего времени) не осознает, что же это такое на самом деле, так их ещё и путают с DTO.

Постараемся хорошо разобраться, чтобы таких заблуждений больше не было ни у кого.

Что же такое Entities?

Чаще всего они воспринимаются как POJO-классы, с которыми работают Interactor’ы. Но это не так. По крайней мере не совсем.

В статье Uncle Bob говорит, что **Entities инкапсулируют логику бизнеса**, то есть всё то, что не зависит от конкретного приложения, а будет общим для многих. Но если у вас отдельное приложение и оно не заточено под какой-то существующий бизнес, то Entities будут являться бизнес-объектами приложения, содержащими самые общие и высокоуровневые правила.

Я думаю, что именно фраза: «Entities это бизнес объекты», – запутывает больше всего. Кроме того, на приведенной выше схеме из видео Interactor получает Entity из Gateway. Это также подкрепляет ощущение, что это просто POJO объекты.

Но в статье также говорится, что Entity может быть объектом с методами или набором структур и функций. То есть упор делается на то, что важны методы, а не данные.

Это также подтверждается в [разъяснении](https://groups.google.com/forum/#!topic/clean-code-discussion/mvP_NR2MUPc) от Uncle Bob’а, которое я нашел недавно:  
Uncle Bob говорит, что для него Entities содержат бизнес-правила, независимые от приложения. И они не просто объекты с данными. Entities могут содержать ссылки на объекты с данными, но основное их назначение в том, чтобы реализовать методы бизнес-логики, которые могут использоваться в различных приложениях.

Итак, **слой Entities содержит**:

* Entities – функции или объекты с методами, которые реализуют логику бизнеса, общую для многих приложений (а если бизнеса нет, то самую высокоуровневую логику приложения);
* DTO, необходимые для работы и перехода между слоями.

Кроме того, когда приложение отдельное, то надо стараться находить и выделять в Entities высокоуровневую логику из слоя UseCases, где зачастую она оседает по ошибке.

Преимущества CLEAN архитектуры:

* Модельное разделение
* Dependency Rule
* Легко тестировать
* Domain слой
* Легко поддерживать

Недостатки:

* Громоздкость
* Порог вхождения