GeekBrains

Архитектура ПО

Методичка к уроку 6

Принципы построения приложений «чистая архитектура»

Оглавление

Введение 3
На этом уроке 4
1. Некоторые виды архитектур 5
1.1. Слоёная архитектура 5
1.2. DCI (Data Context Interaction) 6
1.3. Boundary-Control-Entity 7
1.4. Гексагональная архитектура 10
2. Принципы чистой архитектуры 13
2.1. Независимость от фреймворков 13
2.2. Независимость от базы данных 14
2.3. Независимость от пользовательского интерфейса 16
2.4. Независимость от любых внешних агентов 17
2.5. Простота тестирования 17
2.6. Чистая архитектура 19
2.6.1. Правило зависимостей 19
2.6.2. Сущности 22
2.6.3. Варианты использования 24
2.6.4. Адаптеры интерфейсов 27
2.6.5. Фреймворки и драйверы 27
2.6.6. Пересечение границ между «кругами» архитектуры 28
2.7. Принципы построения чистой архитектуры 29
Глоссарий 35
Дополнительные материалы 36
Используемые источники 37

Введение

Развитие компонентного подхода за последние десятилетия породило несколько идей об организации архитектур.

Целью традиционной многоуровневой архитектуры является разделение приложения на различные уровни, где каждый уровень содержит модули и классы, которые имеют общие или схожие обязанности и работают вместе для выполнения определённых задач.

Существуют различные варианты многоуровневых архитектур, и нет правила, определяющего, сколько уровней должно существовать. Наиболее распространённым шаблоном является трёхуровневая архитектура, в которой приложение делится на презентационный слой, логический слой и слой данных.

Следование многоуровневой архитектуре выгодно во многих отношениях, одним из самых важных является разделение проблем. Однако всегда существует риск. Поскольку не существует естественного механизма для обнаружения утечки логики между слоями, можно — и, скорее всего, так и произойдёт — получить в итоге брызги бизнес-логики в пользовательском интерфейсе или проблемы инфраструктуры, смешанные с бизнес-логикой.

В 2005 году Алистер Кокберн понял, что нет особой разницы между тем, как пользовательский интерфейс и база данных взаимодействуют с приложением, поскольку они оба являются внешними действующими лицами, которые взаимозаменяемы с аналогичными компонентами, которые эквивалентными способами взаимодействуют с приложением. Рассматривая ситуацию таким образом, можно сосредоточиться на том, чтобы приложение оставалось независимым от этих «внешних» участников, позволяя им взаимодействовать через порты и адаптеры, тем самым избегая запутывания и утечки логики между бизнес-логикой и внешними компонентами.

На этом уроке

1. Рассмотрим некоторые подходы организации многокомпонентных архитектур:

- **DCI** (Data Context Interaction)
- **BCE** (Boundary-Control-Entity)

- **Гексагональная архитектура**

2. Разберём понятие и критерии «чистой архитектуры».

3. Изучим правило зависимостей.

4. Разберём пересечение границ «кругов» архитектуры.

1. Некоторые виды архитектур

1.1. Слоёная архитектура

Практически любую систему, взаимодействующую с пользователем, можно разделить на логически слои и выделить сценарии-срезы связывания такой системы для выполнения конкретных задач.

Слои и срезы – это следующий уровень абстракции после компонентов.

1. **Горизонтальные слои** связаны с назначением слоя в коде системы, например, пользовательский интерфейс, слой бизнес-логики и слой базы данных.

2. **Вертикальные слои** связаны с тем, что делает система, с её функциями (сценарии использования). Сценарии использования меняются по разным причинам, независимым друг от друга. Поэтому важно защищать их друг от друга.

Такие принципы деления системы применяются также для объединения всего, что изменяется по одной причине.

Систему всегда можно разделить на горизонтальные уровни:

- пользовательский интерфейс;
- бизнес-правила, характерные для приложения;
- бизнес-правила, не зависящие от приложения;
- слой базы данных.

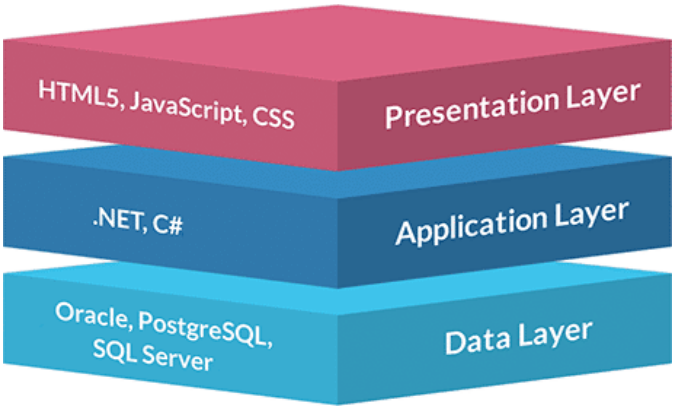
Слои обеспечивают логический уровень разделения в приложении. Если логика приложения при этом физически распределяется между несколькими серверами или процессами, такие раздельные физические целевые объекты развёртывания называются **уровнями**.

Пользовательский интерфейс — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее — область человеческой деятельности, которую поддерживает система.

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Программные системы, разбитые на ряд слоёв, рисуются в виде горизонтальных прямоугольников, например, на диаграмме, показанной на рисунке 1, и представляют различные высокоуровневые логические компоненты системы.



Пример многослойной архитектуры

1.2. DCI (Data Context Interaction)

DCI (Data Context Interaction) предложили Джеймс Коплиен и Трюгве Реенскауг.

Этот подход впервые применил Трюгве Реенскауг, автор знаменитого паттерна MVC. Немногие знают, что MVC сначала назывался MVC-U, где U обозначало пользователя (user), управляющего моделью предметной области через View с Controller. Но в какой-то момент о юзере забыли.

Данные, контекст и взаимодействие (DCI) — это подход в создания систем содержащих взаимодействующие объекты в понятиях предметной области.

Её целями являются:

- Улучшить наглядность объектно-ориентированного кода.
- Предложить чистое разделение кода для быстро меняющегося поведения системы (то, что делает система) и медленно меняющегося домена (что представляет собой система), вместо объединения обоих в одном интерфейсе класса.
- Помочь разработчикам программного обеспечения рассуждать о состоянии и поведении на уровне системы, а не только о состоянии и поведении объектов.
- Поддерживать объектный стиль мышления, который близок к ментальным моделям человека, а не строгий абстрактный объектно-ориентированный подход на основе классов.
- Парадигма отделяет модель домена (данные) от сценариев использования (контекст) и ролей, которые играют объекты (взаимодействие). DCI дополняет модель-вид-контроллер (MVC).

DCI — это попытка вернуть фокус разработчиков на взаимодействие пользователя с системой. Применяя эту парадигму, программист начинает думать, а главное, писать код так, как пользователь представляет себе модель предметной области, а именно в терминах объектов данных (Data). Эти данные в разных ситуациях (Context) по-разному взаимодействуют (Interaction) между собой, исполняя разные роли.

В DCI чтобы узнать, как реализуется какой-либо сценарий использования, потребуется открыть всего лишь один файл.

Контекст содержит только методы — часть сценария выполнения, которую он представляет. Поэтому нам не придётся сканировать десятки, а то и сотни методов, не имеющих ничего общего с нашей проблемой.

1.3. Boundary-Control-Entity

Подход BCE (Boundary-Control-Entity – граница-управление-сущность) представляет собой подход к объектному моделированию, основанный на трехфакторном представлении классов.

BCE (Boundary-Control-Entity) предложил Ивар Якобсон в книге Object-Oriented Software Engineering: A Use-Case Driven Approach.

Архитектура Boundary-Control-Entity (BCE) — шаблон структуры каталога (пакета) для упаковки классов. Это альтернатива многоуровневой архитектуре. Такая архитектура — простой и естественный способ структурирования классов, который хорошо сочетается с модульной архитектурой.

В правильно спроектированной иерархии пакетов актор может взаимодействовать только с пограничными объектами из пакета BoundaryPackage, объекты-сущности из пакета EntityPackage могут взаимодействовать только с управляющими объектами из ControlPackage и управляющие объекты из ControlPackage могут взаимодействовать с объектами любого типа.

Основным преимуществом подхода BCE является группирование классов в виде иерархических уровней. Это способствует лучшему пониманию модели и уменьшает её сложность.

В языке UML для классов определены 3 основных стереотипа:

1. Boundary. Пограничные классы, классы, которые представляют интерфейс между субъектом и системой.

2. Control — управляющие классы, описывают объект, который перехватывает входные события, инициированные пользователем, и контролирует выполнение бизнес-процессов.

3. Entity — сущности, которые описывают семантику сущностей. Для каждого класса-сущности создают таблицу в БД, каждый атрибут становится полем БД.

Шаблон организует обязанности классов в соответствии с их ролью в реализации сценария использования:

1. Сущность представляет долгоживущую информацию, значимую для заинтересованных сторон (т.е. в основном полученную из объектов домена, обычно постоянную).

2. Граница инкапсулирует взаимодействие с внешними субъектами (пользователями или внешними системами).

3. Элемент управления обеспечивает обработку, необходимую для выполнения сценария использования и его бизнес-логики, а также координирует, упорядочивает и контролирует другие объекты, участвующие в сценарии использования.

Соответствующие классы затем группируются в пакеты, представляющие собой неделимый набор связанных классов, которые могут быть использованы в качестве единиц поставки программного обеспечения.

Классы **все** впервые выявляются при анализе вариантов использования (Use cases):

- Каждый вариант использования представляется как класс управления.
- Каждое различное отношение между сценарием использования и действующим лицом представляется как класс границы.
- Сущности выводятся из описания сценария использования.

Затем классы уточняются и переструктурируются или реорганизуются, как это необходимо для проектирования. Например:

Выделение общих моделей поведения в различных контрольных примерах использования

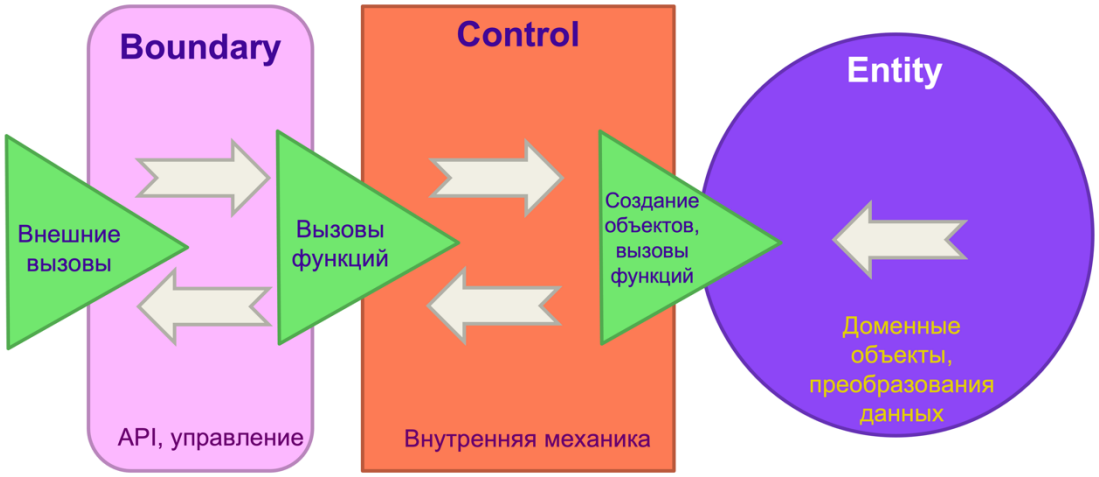
определение центрального пограничного класса для каждого типа человеческого действующего лица и для каждой внешней системы, который обеспечит согласованный интерфейс с внешним миром.

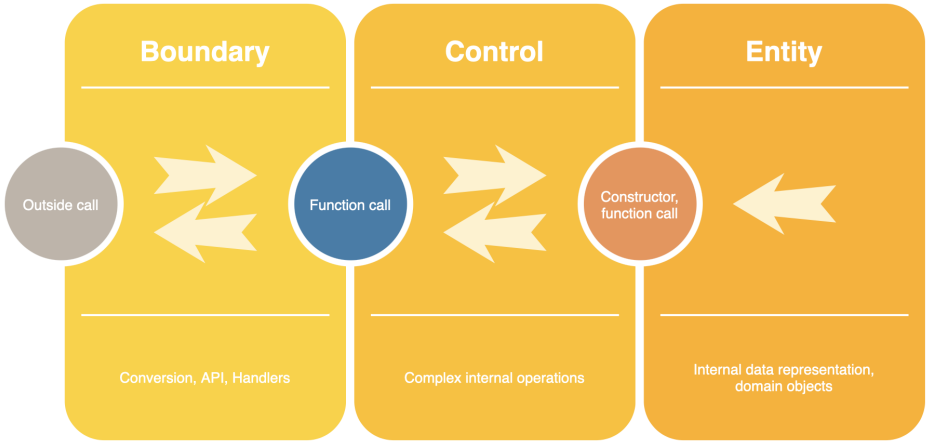
Шаблон ECB предполагает, что обязанности классов также отражаются в отношениях и взаимодействиях между различными категориями классов для обеспечения надёжности дизайна.

Структура пакета часто визуализируется как древовидные луковые слои, где внешний слой — граница, центральный — контроль, а внутренний — сущность.

Пограничный слой содержит классы, ответственные за все коммуникации с системами вне среды выполнения приложения. Контроль представляет собой

всю логику, которая плохо вписывается в границы. Сущность содержит структуры данных, которым разрешается иметь некоторое поведение.





Boundary Control Entity

При различиях в деталях все эти архитектуры похожи. Они преследуют одну цель — разделение задач и жёсткий контроль зон ответственности. Все архитектуры достигают этой цели путём деления программного обеспечения на уровни.

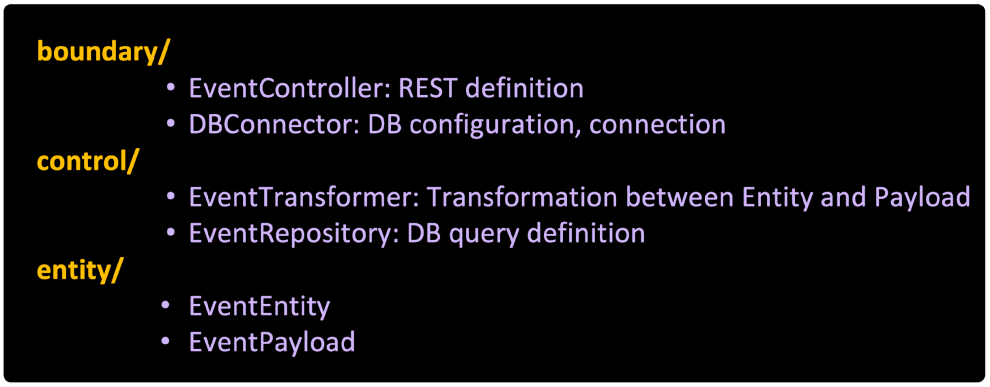
Каждая имеет хотя бы один уровень для бизнес-правил и ещё один для пользовательского и системного интерфейсов. Эти архитектуры способствуют созданию систем, обладающих нижеперечисленными характеристиками.

Структура пакета часто визуализируется как слои лука или как шестиугольные слои, где самый внешний слой — Boundary, центральный — Control и внутренний — Entity.

Boundary — это интерфейс к внешнему миру. Он содержит классы, отвечающие за все коммуникации с системами вне времени выполнения приложения. Он содержит конфигурацию и установление внешних соединений.

Control представляет всю логику, которая не вписывается в Boundary. Она содержит алгоритмы, SQL-запросы.

Entity содержит структуры данных, которым разрешено определённое поведение. Она содержит объекты домена с базовыми функциональными возможностями.
Пример структурирования кода для сервиса событий (Event):



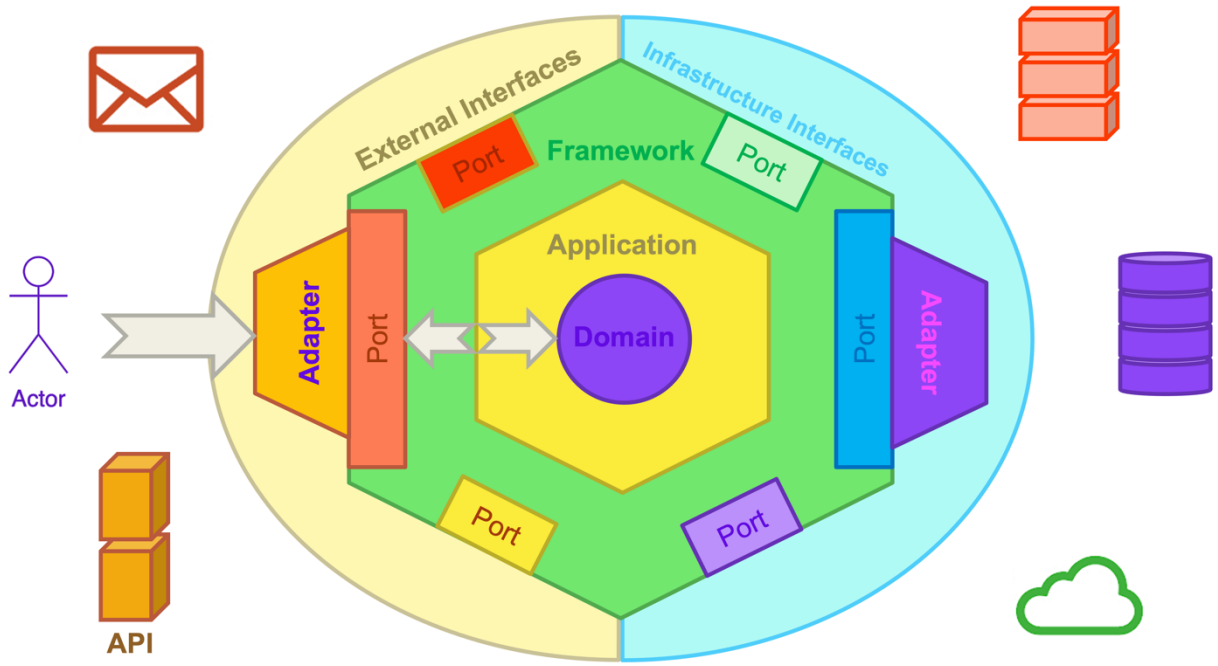
1.4. Гексагональная архитектура

Hexagonal Architecture — также известна как архитектура портов и адаптеров. Её разработал Алистер Кокбёрн, а описали Стив Фриман и Нат Прайс в книге Growing Object Oriented Software with Tests.

Представляя собой один из вариантов слоёной архитектуры, гексагональная архитектура подразумевает разделение приложения на отдельные концептуальные слои, имеющие разные зоны ответственности. Архитектура также регламентирует, каким образом слои связаны друг с другом.

Гексагональная архитектура позволяет взаимодействовать с приложением как пользователю, так и программам, автоматическим тестам, скриптам пакетной обработки. Архитектура применяется в разработке и тестировании приложений без каких-либо дополнительных устройств или баз данных.

Хотя эта архитектура и называется гексагональной, что указывает на фигуру с определённым количеством граней, основной мыслью считается то, что граней у этой фигуры много. Каждая грань представляет собой «порт» доступа к приложению или его связь с внешним миром. Порт — это некий проводник входящих запросов или данных к приложению. Например, через HTTP-порт (запросы браузера, API) приходят HTTP-запросы, которые конвертируются в команды для приложения.



Гексагональная архитектура

Основа данной архитектуры — порты и адаптеры.

Порты — это интерфейсы нашего приложения,

Адаптеры — реализация наших портов.

Преимущества перед традиционной многоуровневой архитектурой

Гексагональная архитектура была отходом от традиционной многоуровневой архитектуры. Одним из основных отличий гексагональной архитектуры является возможность замены пользовательского интерфейса.

Использование гексагональной архитектуры вместо многоуровневой даёт множество преимуществ. Давайте посмотрим на некоторые плюсы, минусы и варианты использования:

Плюсы	**Минусы**
• Ремонтопригодность: наши приложения обладают высокой ремонтопригодностью, поскольку изменения в одной области нашего приложения не влияют на другие области. • Простое тестирование: поскольку наш код отделён от внешних деталей реализации, мы можем тестировать изолированно. • Независимость: возможность не зацикливаться на бизнес-логике. Можно задекларировать, описать схему работы нашего приложения до создания внешних сервисов, использовать mock данные (заглушки) в реализации адаптеров. • Гибкость: использование любых фреймворков, перенос доменов адаптеров в другие проекты, добавление новых адаптеров без изменения исходного кода. • Лёгкая изменчивость: изменения в одной области нашего приложения не влияют на другие области.	• Развязка: на производительность нашего приложения могут повлиять промежуточные классы. • Отладка: иногда бывает трудно понять и отладить адаптеры. • Сложность: гексагональная архитектура иногда может сбивать с толку, потому что не всегда очевидно, что мы должны учитывать снаружи. • Погружение: многим разработчикам бывает сложно освоиться, особенно при недостатке знаний и опыта. • Также могут возникнуть сложности реализации с технологией graphql.

2. Принципы чистой архитектуры

2.1. Независимость от фреймворков

Архитектура не зависит от какой-либо библиотеки. Это позволяет рассматривать фреймворки как инструменты, вместо того чтобы стараться втиснуть систему в их рамки.

Любая программная система делится на два основных элемента: политику и детали.

1) Политика воплощает все бизнес-правила и процедуры, это истинная ценность системы.

2) Детали — это остальное, что позволяет людям, другим системам и программистам взаимодействовать с политикой, не влияя на её поведение. К ним относятся устройства ввода и вывода, базы данных, веб-системы, серверы, фреймворки, протоколы обмена данными и т. д.

Цель архитектора — создать такую форму для системы, которая сделает политику самым важным элементом, а детали — не относящимися к политике. Это позволит откладывать и задерживать принятие решений о деталях.

Представленные архитектуры не связаны и не должны быть связаны с фреймворками. Архитектура не определяется фреймворками. Фреймворки — это инструменты, а не аспекты, определяющие черты архитектуры.

Хорошая архитектура позволяет отложить решение о выборе фреймворков, баз данных, веб-серверов, а также других инструментов и элементов окружения. Фреймворки относятся к открытым возможностям. Такая архитектура даёт с лёгкостью менять подобные решения. Фреймворки не определяют архитектуру, хотя некоторые пытаются.

Используя фреймворк, нельзя к нему привязываться. Стоит рассматривайть фреймворк как деталь, принадлежащую одному из внешних кругов архитектуры. Не впускаем его во внутренние круги.

Важно понимать, что, заключив союз между фреймворком и своим приложением, нам придётся придерживаться этого фреймворка в течение всего жизненного цикла этого приложения. Отказавшись от любых других, мы станем использовать этот фреймворк. Этот шаг нельзя делать по легкомыслию.

Рассмотрим, например, действия архитектора, работающего над системой S. Он пожелал включить в систему некоторый фреймворк F. Теперь представим, что авторы F связали его с поддержкой конкретной базы данных D. То есть S зависит от F, который подчиняется D.

Проблемная архитектура

Представим, что D включает функции, которые не используются фреймворком F и, соответственно, не применяются системой S. Изменения в этих функциях внутри D могут вынудить повторно развернуть F и, соответственно, повторно развернуть S. Хуже того, ошибка в одной из таких функций внутри D может спровоцировать появление ошибок в F и S.

2.2. Независимость от базы данных

Можно поменять Oracle или SQL Server на MongoDB, BigTable, CouchDB или что-то ещё. Бизнес-правила не привязаны к базе данных.

База данных, язык запросов и даже схема — технические детали, которые имеют мало общего с бизнес-правилами и пользовательским интерфейсом. Они будут изменяться со скоростью и по причинам, не зависящим от других аспектов системы. Следовательно, в архитектуре они отделяются от остальной системы, чтобы изменяться независимо.

С архитектурной точки зрения, базы данных — не сущности. Это деталь, которая не поднимается до уровня архитектурного элемента. Её отношение к архитектуре программной системы сопоставимо с отношением дверной ручки к архитектуре здания.

Структура, которую мы придаём данным в своём приложении, важна для архитектуры системы. Но база данных — это не модель данных. База данных — это часть программного обеспечения, а также утилита, обеспечивающая доступ к данным. С архитектурной точки зрения, эта утилита не имеет никакого значения, это низкоуровневая деталь — механизм.

Хороший архитектор не позволяет низкоуровневым механизмам просачиваться в архитектуру системы.

Организационная структура и модель данных считаются архитектурно значимыми, а технологии и системы, перемещающие данные на вращающуюся магнитную поверхность, — нет. То есть, с архитектурной точки зрения, мы не должны беспокоиться о том, какую форму принимают данные, пока хранятся на поверхности вращающегося магнитного диска.

Нас также не касается, есть этот диск или нет.

Не стоит на ранних этапах разработки выбирать тип базы данных. Высокоуровневая политика не зависит от этого выбора.

У мудрого архитектора высокоуровневая политика не зависит от того, какая база данных будет выбрана впоследствии: реляционная, распределённая, иерархическая или просто набор файлов.

Сумев разработать высокоуровневую политику, не связывая её с окружающими деталями, мы сможем откладывать и задерживать принятие решений об этих деталях на более поздний срок. И чем дольше получится откладывать такие решения, тем большим объёмом информации мы будем обладать, чтобы сделать правильный выбор.

Такой подход даёт возможность экспериментировать. Если есть действующая часть высокоуровневой политики, не зависящая от типа базы данных, можно попытаться связать её с разными базами данных, чтобы проверить их применимость и производительность.

Например, есть задача создать простую вики-страницу — проект FitTrack, для управления своими тренировками.

Сначала появился собственный веб-сервер, отвечающий потребностям FitTrack. На ранней стадии была идея использовать MySQL, но конечное решение о базе данных было отложено. Все методы доступа к данным вошли в интерфейс с именем FitPage, методы которые обеспечивали всё необходимое для поиска и извлечения результатов.

В результате получилась первая версия программы FitTrack. Появилась возможность создавать страницы, ссылаться на другие страницы, применять любое форматирование и строить отчёты FIT. Нельзя только сохранять результаты.

Когда пришло время реализовать долговременное хранение отчётов, разработчики снова подумали о MySQL, но решили, что в краткосрочной перспективе это необязательно, так как хеш-таблицы было очень легко записывать в простые файлы. Как результат, сформировался класс FileSystemFitPage.

В результате разработчики пришли к заключению, что решение на основе простых файлов достаточно для выполняемой задачи, и решили вообще отказаться от идеи использовать MySQL.

Таким образом, только начиная работу над FitTrack, появилась граница между бизнес-правилами и базами данных. Граница дала реализовать бизнес-правила так, что они вообще никак не зависели от выбора базы данных, им требовались только методы доступа к данным. Это позволило опробовать вариант с файловой системой и изменить направление, когда появилось оптимальное решение. Такое решение не препятствовало и даже не мешало движению в первоначальном направлении к MySQL, если бы это потребовалось.

2.3. Независимость от пользовательского интерфейса

Пользовательский интерфейс можно легко изменять, не затрагивая остальной системы. Например, веб-интерфейс заменяется консольным интерфейсом, не изменяя бизнес-правил.

Пользовательский интерфейс изменяется по причинам, не имеющим ничего общего с бизнес-правилами. Все варианты использования включают оба этих элемента. Поэтому хороший архитектор отделит часть, отвечающую за пользовательский интерфейс, от части, реализующей бизнес-правила. Таким образом, их можно будет изменять независимо друг от друга, сохраняя ясную видимость вариантов использования.

Так как в таком дизайне пользовательский интерфейс считается плагином, у нас появляется возможность подключать разные пользовательские интерфейсы:

- веб-интерфейсы;
- интерфейсы «клиент — сервер»;
- интерфейсы служб;
- консольные интерфейсы;

- интерфейсы, основанные на других способах взаимодействия с пользователем.

Однако такие замены осуществляются не всегда просто. Если первоначально система опиралась на веб-интерфейс, создание плагина для интерфейса «клиент — сервер» может оказаться сложной задачей. Вполне вероятно, что придётся переделать какие-то взаимодействия между бизнес-правилами и новым пользовательским интерфейсом.

Варианты использования также не описывают пользовательский интерфейс, они лишь неформально определяют входные и выходные данные, поступающие и возвращаемые через интерфейс. По вариантам использования нельзя определить, считается ли конкретная система веб-приложением, «толстым» клиентом, утилитой командной строки или чистой службой.

Это очень важно. Варианты использования не описывают, как выглядит система для пользователя. Они отображают только конкретные правила работы приложения, определяющие порядок взаимодействий между пользователями и сущностями. Для вариантов использования абсолютно неважно, как система осуществляет ввод или вывод данных.

Например, есть компания Q. Компания Q создала очень популярную систему управления персональными финансами. Это приложение для настольного компьютера с удобным графическим интерфейсом. В следующей версии системы компания Q изменила пользовательский интерфейс — он стал выглядеть и действовать подобно веб-интерфейсу в браузере.

Теперь рассмотрим пример с компанией A, выпускающей замечательные смартфоны. Недавно она выпустила обновлённую версию своей «операционной системы». Модернизация кардинально изменила внешний вид всех приложений.

Для удобства архитекторам в компаниях A и Q следовало изолировать бизнес-правила и пользовательский интерфейс друг от друга, чтобы в дальнейшем производить изменения в интерфейсе.

2.4. Независимость от любых внешних агентов

Наши бизнес-правила ничего не знают об интерфейсах, ведущих во внешний мир.

Задача правил бизнес-логики — заботиться только о своих задачах и ни о чём больше, что входит в приложение.

Бизнес-логика имеет дело только с наиболее удобным для неё форматом данных, так же как и внешние агенты, например, база данных или пользовательский интерфейс. Но этот формат обычно отличается. По этой причине адаптер интерфейсов отвечает за преобразование данных.

Этот слой конвертирует данные в формат, подходящий для внешних слоёв, а также превращает внешние данные в формат для внутренних.

2.5. Простота тестирования

Бизнес-правила тестируются без пользовательского интерфейса, базы данных, веб-сервера и любых других внешних элементов.

Тесты — часть системы, они занимают своё место в архитектуре, как любые другие части системы.

Тесты следуют правилу зависимостей, они очень детальны и конкретны и всегда зависят от тестируемого кода. Фактически тесты считаются внешним кругом архитектуры.

Ничто в системе не зависит от тестов, но тесты всегда зависят от внутренних компонентов системы. А также тесты — наиболее изолированные компоненты системы. Они не нужны системе для нормального функционирования. Пользователи не зависят от них. Их задача — поддерживать разработку, а не работу.

Тесты, тесно связанные с системой, изменяются вместе с системой. Даже самые безобидные изменения в системном компоненте могут нарушить нормальную работу многих связанных тестов или потребовать их изменения.

Изменения в общих системных компонентах способны нарушить работу сотен и даже тысяч тестов. Нетрудно понять, как это может произойти.

Представим набор тестов, использующих графический интерфейс для проверки бизнес-правил. Такие тесты начинают работу на странице авторизации и затем последовательно переходят от страницы к странице, проверяя конкретные бизнес-правила. Любое изменение в странице авторизации или в структуре навигации в силах нарушить работу большинства тестов.

Первое правило проектирования программного обеспечения — идёт ли речь о тестируемости или о чём-то ещё — всегда одно: **не зависеть ни от чего, что может часто меняться.**

Пользовательские интерфейсы переменчивы. Наборы тестов, осуществляющие проверки посредством пользовательского интерфейса, хрупкие. Поэтому система и тесты проектируются так, чтобы работа бизнес-правил проверялась без пользовательского интерфейса.

Чистая встраиваемая архитектура позволяет выполнять послойное тестирование, потому что модули взаимодействуют посредством интерфейсов. Каждый интерфейс обеспечивает шов или точку подстановки для тестирования вне целевого окружения.

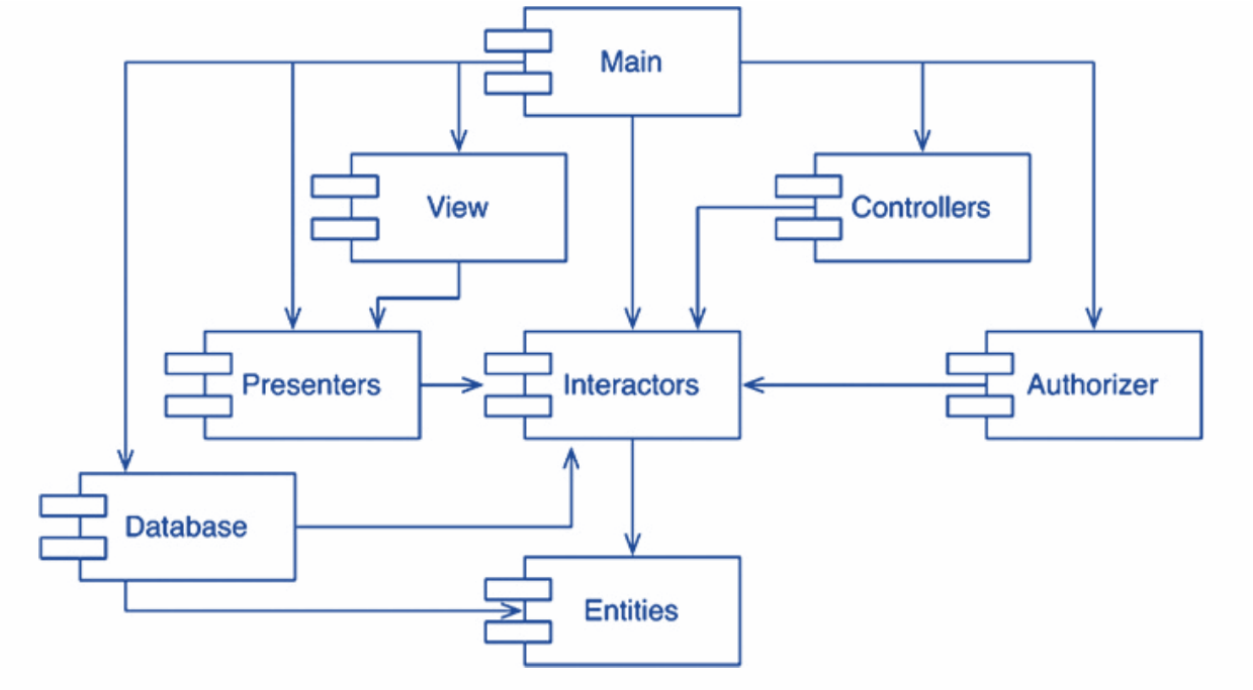
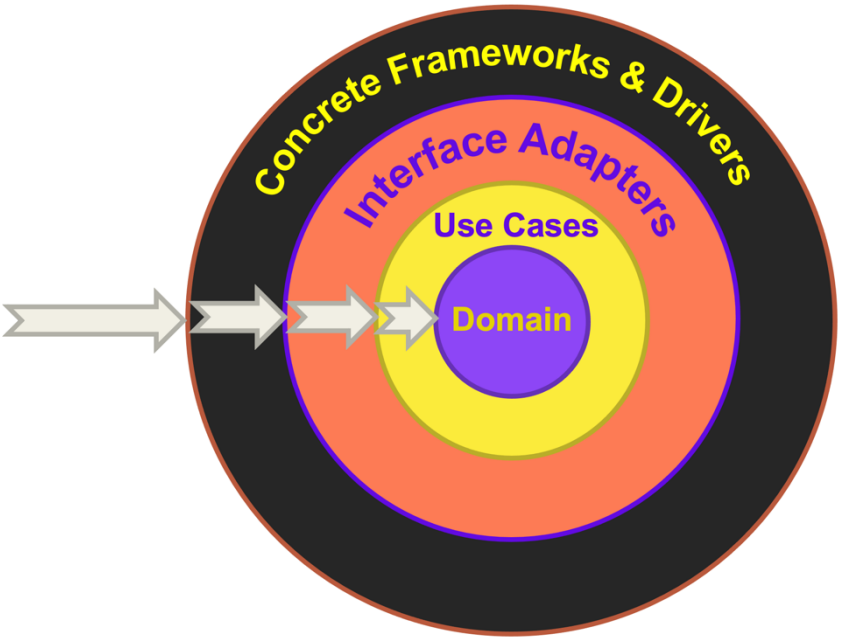


Диаграмма компонентов

Например, когда разработчики компонента Presenters пожелают протестировать его, им просто потребуется собрать собственную версию Presenters с версиями компонентов Interactors и Entities, используемыми в конкретный момент. Никакой другой компонент в системе им для этого не потребуется. Это означает, что разработчикам Presenters не придётся прилагать значительных усилий для подготовки к тестированию, им достаточно учесть небольшое число переменных.

2.6. Чистая архитектура

2.6.1. Правило зависимостей

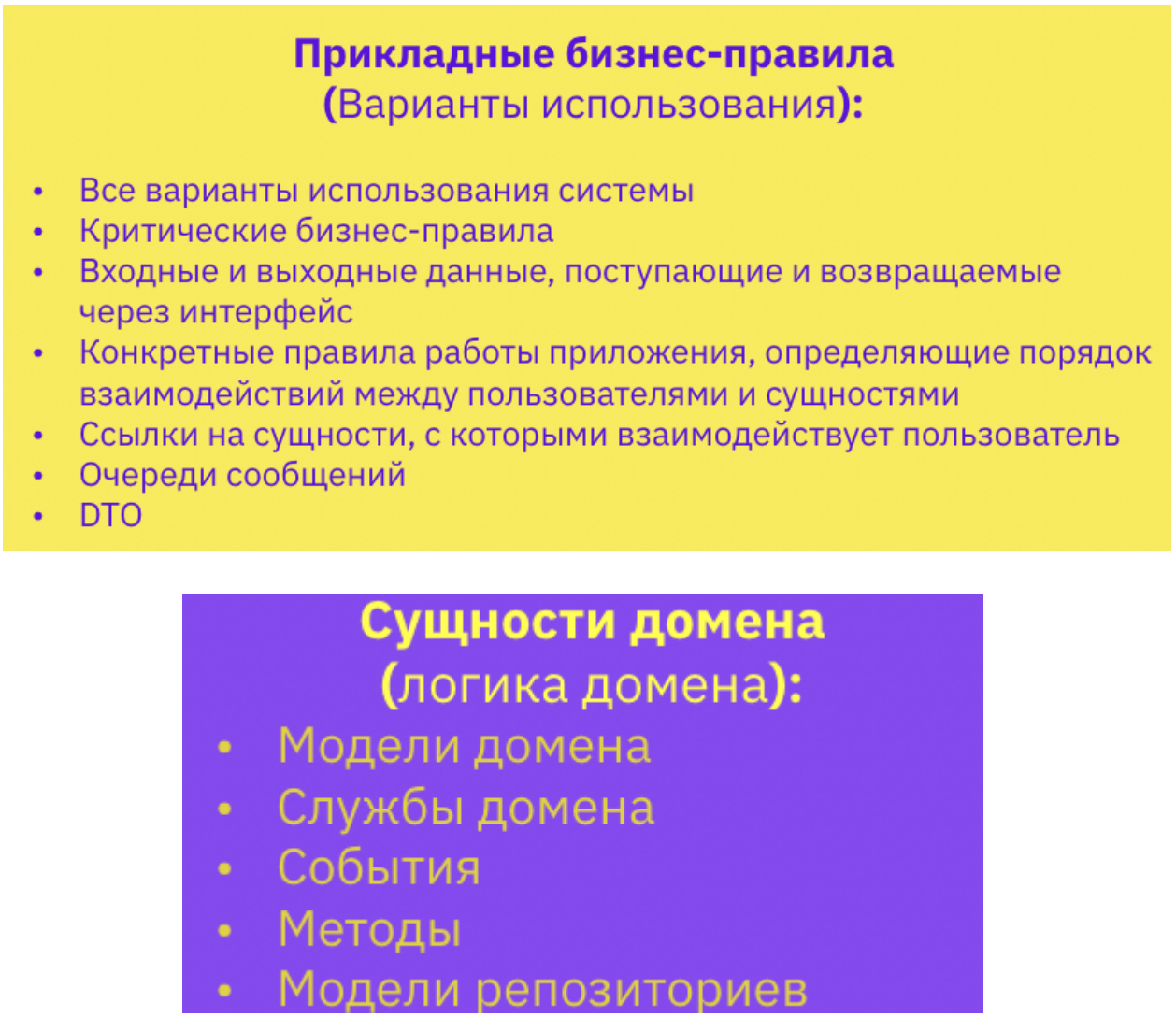


Внешние зависимости
(интерфейсы к внешним системам):

- Пользовательский интерфейс
- Базы данных
- Фреймворки
- Драйверы
- Другие приложения
- Устройства

Адаптеры интерфейсов (Инфраструктура)**:**

- API Шлюзы
- Контроллеры
- Валидаторы
- Формы
- Презентаторы
- Подключения к репозиториям



Чистая архитектура

Концентрические круги на рисунке представляют разные уровни программной архитектуры. Чем ближе к центру, тем выше уровень.

Внешние круги — это механизмы.

Внутренние — политики.

В некотором смысле обозначенные круги — это продолжение идеи горизонтальных слоёв, но скомпонованных по принципу устойчивости.

Главным правилом, приводящим эту архитектуру в действие, считается правило зависимостей (Dependency Rule) Согласно ему, зависимости в исходном коде должны быть направлены внутрь, в сторону высокоуровневых политик.

Ничто во внутреннем круге ничего не знает о внешних кругах. Например, имена, объявленные во внешних кругах, не упоминаются в коде, находящемся во внутренних кругах. Сюда относятся функции, классы, переменные и любые другие именованные элементы программы.

Точно так же форматы данных, объявленные во внешних кругах, не используются во внутренних, особенно если эти форматы генерируются фреймворком во внешнем круге. Ничто во внешнем круге не должно влиять на внутренние круги.

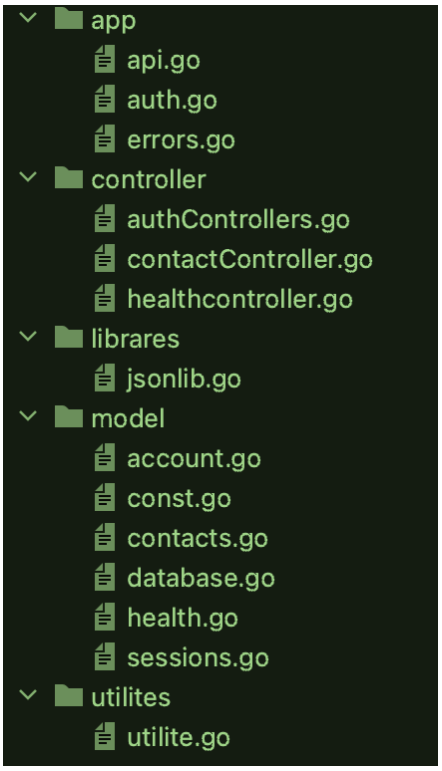
Круги на рисунке лишь схематически изображают основную идею: иногда требуется больше четырёх кругов. Фактически нет никакого правила, утверждающего, что кругов должно быть именно четыре. Но всегда действует правило зависимостей. Продвигаясь внутрь, уровень абстракции и политик увеличивается.

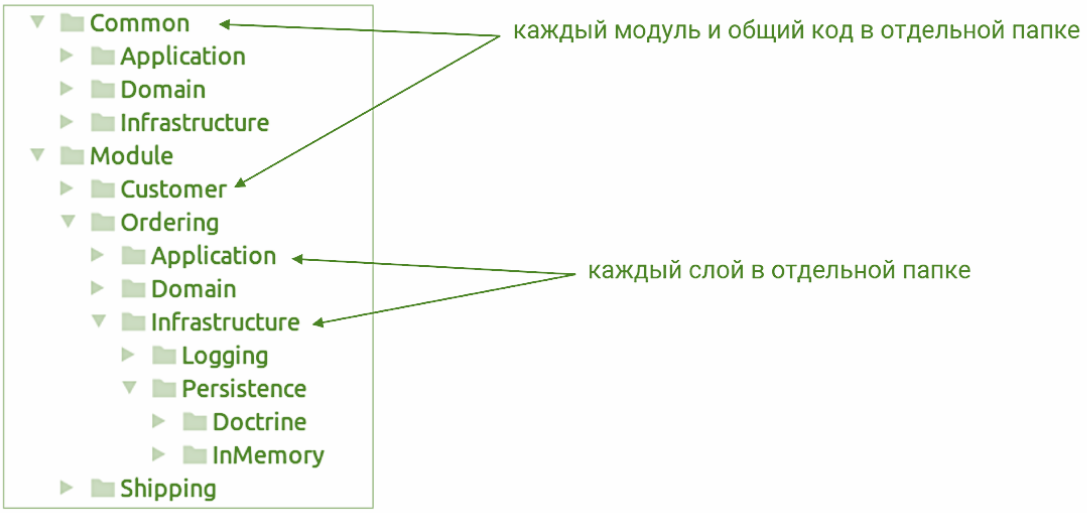
Внешний круг включает низкоуровневые конкретные детали. По мере приближения к центру программное обеспечение становится всё более абстрактным и инкапсулирует более высокоуровневые политики.

Самый внутренний круг — самый обобщённый, он находится на самом высоком уровне.

Разделив программное обеспечение на уровни и соблюдая правило зависимостей, мы создаём систему, которую легко протестировать, со всеми вытекающими из этого преимуществами. Когда какой-либо из внешних элементов системы, например, база данных или веб-фреймворк, устареет, мы сможем легко его заменить.

Чистая архитектура — это способ разделения ответственностей и частей функциональности по степени их близости к предметной области приложения.





Принципы чистой архитектуры:

1. Приложение строится вокруг независимой от других слоёв объектной модели.

2. Внутренние слои определяют интерфейсы, внешние слои содержат реализации интерфейсов.

3. Направление зависимостей — от внешних слоёв ко внутренним.

4. При пересечении границ слоя данные должны преобразовываться.

2.6.2. Сущности

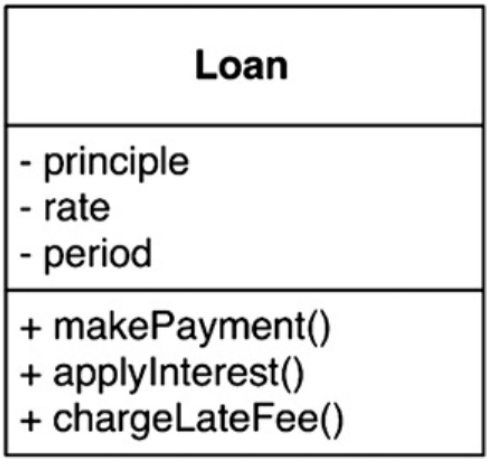
Сущность — это объект в компьютерной системе, который воплощает небольшой набор критических бизнес-правил, оперирующих критическими бизнес-данными. Объект-сущность или содержит критические бизнес-правила в себе, или имеет простой доступ к ним. Интерфейс сущности состоит из функций, реализующих критические бизнес-правила и оперирующих этими данными.

Сущность может быть объектом с методами или набором структур данных и функций. Сама организация не важна, если сущности доступны для использования разными приложениями на предприятии.

Если мы пишем только одно приложение и не для предприятия, эти сущности становятся бизнес-объектами приложения. Они инкапсулируют наиболее общие и высокоуровневые правила. Их изменение маловероятно под влиянием внешних изменений.

Например, трудно представить, что эти объекты изменятся из-за изменения структуры навигации или безопасности страницы. Никакие изменения в работе любого конкретного приложения не влияют на уровень сущностей.

Например, на рисунке показано, как могла бы выглядеть сущность Loan, представляющая банковский кредит, в виде класса на диаграмме UML. Она включает три фрагмента взаимосвязанных критических бизнес-данных и реализует интерфейс, используя три взаимосвязанные критические бизнес-правила.



Сущность Loan

Создавая такой класс, мы объединяем программную реализацию идеи, имеющей решающее значение для бизнеса, и отделяем её от остальных задач в создаваемой нами системе автоматизации.

Этот класс — некий представитель бизнеса. Он не зависит от выбора базы данных, пользовательского интерфейса или сторонних фреймворков.

Класс может служить целям бизнеса в любой системе, независимо от того, какой пользовательский интерфейс она имеет, как хранит данные, или каким образом организованы компьютеры в этой системе.

Сущность — это бизнес в чистом виде и больше ничего.

2.6.3. Варианты использования

Программное обеспечение на уровне вариантов использования содержит бизнес-правила, характерные для приложения. Оно инкапсулирует и реализует все варианты использования системы. Варианты использования организуют поток данных в сущности и из них, а также требуют от этих сущностей использовать их критические бизнес-правила для достижения собственных целей.

Изменения внутри этого уровня не должны влиять на сущности. Аналогично изменения во внешних уровнях, например, в базе данных, пользовательском интерфейсе или в любом из общих фреймворков, не должны влиять на этот уровень. Уровень вариантов использования изолирован от таких проблем.

Но изменения в работе приложения, безусловно, повлияют на варианты использования и, соответственно, на программное обеспечение, находящееся на этом уровне. Изменение деталей вариантов использования определённо затронет некоторый код на этом уровне.

Варианты использования не описывают пользовательский интерфейс, они лишь неформально определяют входные и выходные данные, поступающие и возвращаемые через интерфейс. По вариантам использования нельзя определить, считается ли такая система веб-приложением, «толстым» клиентом, утилитой командной строки или чистой службой.

Такие варианты использования не показывают, как выглядит система для пользователя. Они отображают только конкретные правила работы приложения, определяющие порядок взаимодействий между пользователями и сущностями. Для вариантов использования абсолютно неважно, как система осуществляет ввод или вывод данных.

Класс варианта использования принимает простые структуры данных на входе и возвращает простые структуры данных на выходе. Эти структуры данных ни от чего не зависят. Вариант использования ничего не говорит о способах передачи данных пользователю или другим компонентам.

Вариант использования — это объект. Он имеет одну или несколько функций, реализующих конкретные прикладные бизнес-правила. Такой объект также имеет элементы данных, включая:

- входные данные;
- выходные данные;
- ссылки на соответствующие сущности, с которыми он взаимодействует.

Сущности не знают ничего о вариантах использования, контролирующих их. Это ещё один пример ориентации зависимостей в соответствии с принципом инверсии зависимостей. Высокоуровневые элементы, такие как сущности, ничего не знают о низкоуровневых элементах, например, о вариантах использования. Напротив, низкоуровневые варианты использования знают всё о высокоуровневых сущностях.

Представим приложение, которое используется служащими банка для оформления нового кредита. Банк может решить, что сотрудники, оформляющие кредиты, не должны предлагать график погашения кредита, пока не соберут и не проверят контактную информацию и не убедятся в кредитном балле (500 или выше) кандидата.

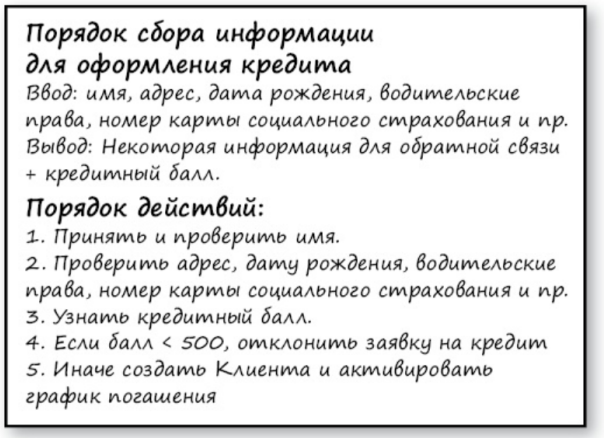
По этой причине есть вероятность, что банк потребует, чтобы система не отображала форму с графиком платежей, пока не будет заполнена и проверена форма с контактной информацией, и не придёт подтверждение о достаточном кредитном балле клиента. Это вариант использования.

Вариант использования описывает способ использования автоматизированной системы и определяет:

- что должен ввести пользователь;
- что будет выведено в ответ;
- какие действия требуются для получения выводимой информации.

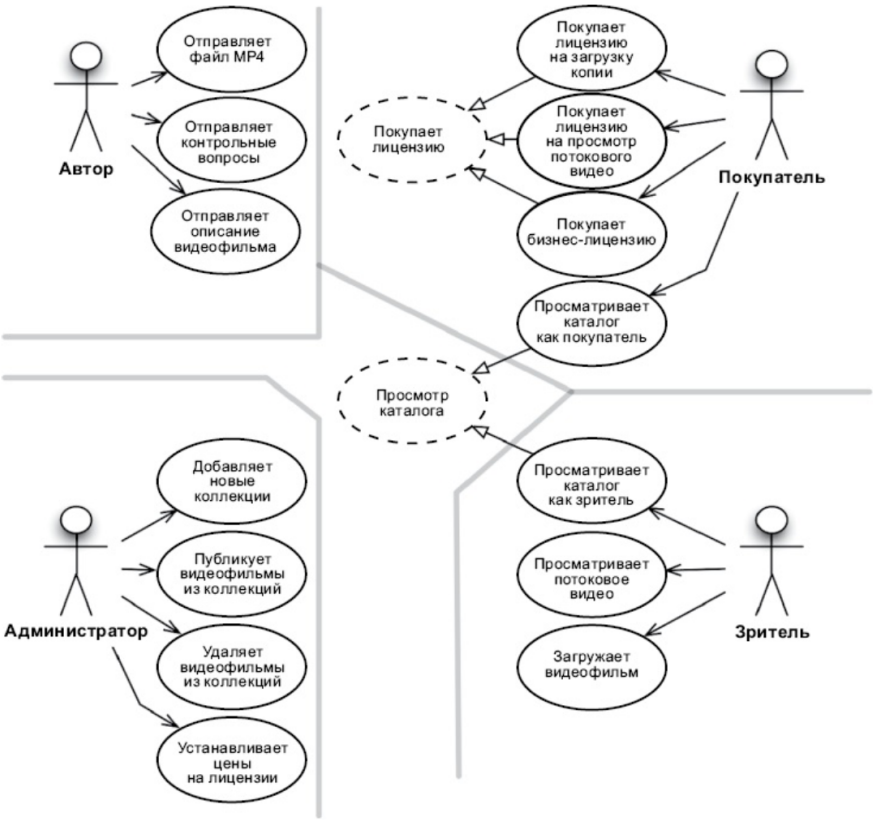
Вариант использования описывает бизнес-правила, характерные для конкретного приложения.

На рисунке — пример варианта использования. В последней строке упоминается клиент. Это ссылка на сущность «Клиент», содержащую критические бизнес-правила, регулирующие отношения между банком и клиентами.



Вариант использования

На следующем рисунке — схема типичного анализа вариантов использования. Но это далеко не все варианты использования.



Типичный анализ вариантов использования

На этой схеме отображаются четыре действующих лица. В соответствии с принципом единственной ответственности, эти четыре действующих лица станут основными источниками изменений для системы. Добавление любых новых возможностей или изменение действующих будет выполняться ради обслуживания одного из действующих лиц. Поэтому мы должны разбить систему так, чтобы изменения, предназначенные для одного действующего лица, не затрагивали других.

Обратим внимание на варианты использования, заключённые в пунктирные рамки. Это абстрактные варианты использования. Абстрактным называется такой вариант использования, который определяет общую политику для других вариантов использования.

2.6.4. Адаптеры интерфейсов

Программное обеспечение на уровне адаптеров интерфейсов — это набор адаптеров. Эти наборы преобразуют данные из наиболее удобного формата для вариантов использования и сущностей в формат, наиболее удобный для некоторых внешних агентов, например, база данных или веб-интерфейс.

Именно на этом уровне находится архитектура MVC графического пользовательского интерфейса. Презентаторы, представления и контроллеры принадлежат уровню адаптеров интерфейсов. Модели часто позиционируются именно как структуры данных, которые передаются из контроллеров в варианты использования, а затем обратно — из вариантов использования в презентаторы и представления.

Аналогично на этом уровне преобразуются данные из формата, наиболее удобного для вариантов использования и сущностей, в формат, наиболее удобный для инфраструктуры хранения данных (например, базы данных). Никакой код, находящийся в других внутренних кругах, не должен ничего знать о базе данных. Если данные хранятся в базе данных SQL, требуется, чтобы весь код на языке SQL находился именно на этом уровне, точнее, в элементах этого уровня, связанных с базой данных.

На этом уровне есть и другие адаптеры, требуемые для преобразования данных из внешнего формата, например, полученных от внешней службы, во внутренний, используемый вариантами использования и сущностями.

2.6.5. Фреймворки и драйверы

Фреймворки — это инструменты, а вовсе не аспекты, определяющие черты архитектуры. Такие инструменты относятся к возможностям, которые остаются открытыми. Хорошая архитектура позволяет отложить выбор Rails, Spring, Hibernate, Tomcat или MySQL на отдалённый срок.

Самый внешний уровень модели (рис. 5) обычно состоит из фреймворков и инструментов, таких как база данных и веб-фреймворк. Как правило, этот код представляет собой связующее звено со следующим внутренним кругом.

На уровне фреймворков и драйверов сосредотачиваются все детали:

- веб-интерфейс;
- база данных.

Всё это мы храним во внешнем круге, где они не смогут причинить большого вреда.

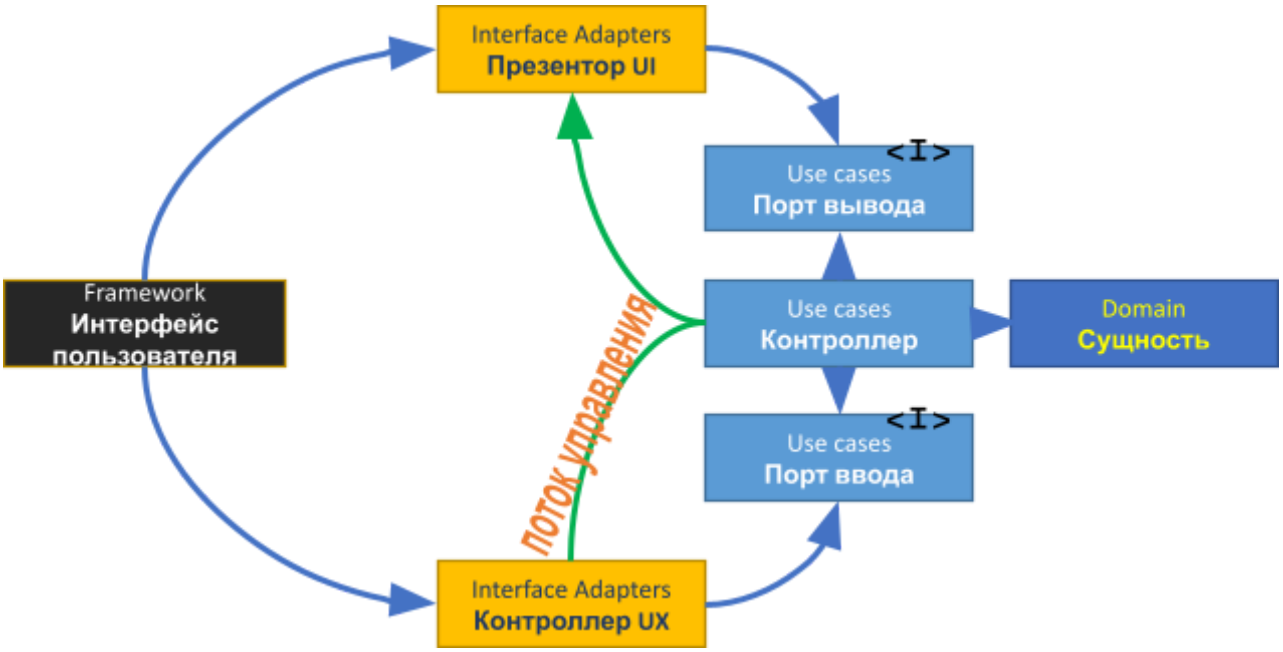
Рассмотрим пример программы copy. Что случится с программой, если создать новое устройство ввода или вывода? Допустим, мы решили использовать программу copy для копирования данных из устройства распознавания

рукописного текста в устройство синтеза речи: что надо изменить в программе copy, чтобы она смогла работать с новыми устройствами?

Никаких изменений не требуется. В действительности, не придётся даже перекомпилировать программу copy. Потому что исходный код программы copy не зависит от исходного кода драйверов ввода или вывода. Пока драйверы реализуют пять стандартных функций, определяемых структурой FILE, программа copy сможет их использовать.

2.6.6. Пересечение границ между «кругами» архитектуры

Справа внизу на рисунке — пример пересечения границ кругов: изображены контроллеры и презентаторы, взаимодействующие с вариантами использования на следующем уровне. Поток управления начинается в контроллере, проходит через вариант использования и завершается в презентаторе. Каждая зависимость указывает внутрь, на варианты использования. По факту этот поток показывает «вертикальный срез» системы.



Поток управления в чистой архитектуре

Обычно мы разрешаем это кажущееся противоречие с использованием принципа инверсии зависимостей (Dependency Inversion Principle). В таких языках, как Java, например, мы можем определять интерфейсы и использовать

механизм наследования, чтобы направление зависимостей в исходном коде было противоположно направлению потока управления на границах справа.

Допустим, что вариант использования должен вызвать презентатора. Такой вызов нельзя выполнить напрямую, потому что иначе нарушится правило зависимостей: никакие имена, объявленные во внешних кругах, не упоминаются во внутренних. Поэтому важно, чтобы Use Case вызывал интерфейс, объявленный во внутреннем круге, а презентатор во внешнем круге его реализовывал.

Тот же приём используется для всех пересечений границ в архитектуре. Мы используем преимущество динамического полиморфизма, чтобы обратить зависимости в исходном коде в направлении, противоположном потоку управления, и тем самым соблюсти правило зависимостей при любом направлении потока управления.

Данные через границы обычно передаются в виде простых структур. При желании можно использовать простейшие структуры или объекты передачи данных: Data Transfer Objects (DTO).

Данные также передаются в вызовы функций через аргументы или упаковываются в ассоциативные массивы или объекты. Важно, чтобы через границы передавались простые, изолированные структуры данных. Не надо хитрить и передавать объекты сущностей или записи из базы данных. Структуры данных не должны нарушать правило зависимостей.

Например, многие фреймворки для работы с базами данных возвращают ответы на запросы в удобном формате. Они называются «представлениями записей». Такие представления записей не передаются через границы внутрь. Это нарушает правило зависимостей, потому что заставляет внутренний круг знать что-то о внешнем круге.

Итак, при передаче через границу данные всегда принимают форму, наиболее удобную и известную для внутреннего круга.

Обычная ситуация, когда в системе есть как минимум 3 варианта одних и тех же сущностей:

- бизнес-сущности;
- сущности уровня базы данных;

- транспортные сущности для передачи в пользовательский интерфейс и внешним системам.

2.7. Принципы построения чистой архитектуры

В решении с чистой архитектурой для каждого проекта чётко определяются обязанности.

Ядро приложения

Ядро приложения содержит бизнес-модель, которая включает в себя сущности, службы и интерфейсы. Такие интерфейсы включают абстракции для операций, которые будут выполняться с использованием архитектуры, включая операции доступа к данным или файловой системе, сетевые вызовы и т. д.

В некоторых случаях службы или интерфейсы, определённые в этом слое, работают с типами, которые не считаются типами сущностей. Эти типы не имеют зависимостей от пользовательского интерфейса или инфраструктуры. Они определяются как простые объекты передачи данных:

- сущности — классы моделей бизнес-логики;
- интерфейсы;
- службы;
- объекты передачи данных.

Инфраструктура

Как правило, проект инфраструктуры включает реализацию доступа к данным. В типовом веб-приложении ASP.NET Core эта реализация включает Entity Framework (EF) DbContext, любые определённые объекты Migration EF Core, а также классы реализации доступа к данным. Наиболее распространённый подход к абстрагированию кода реализации доступа к данным заключается в использовании шаблона репозитория.

Помимо реализации доступа к данным, проект инфраструктуры также включает реализации служб, которые взаимодействуют с инфраструктурными задачами. Эти службы реализовывают интерфейсы, определённые в ядре приложения. Таким образом, важно, чтобы инфраструктура содержала ссылку на проект ядра приложения:

- типы EF Core: DbContext, Migration;
- типы реализации доступа к данным (репозитории);
- службы, связанные с инфраструктурой: FileLogger или SMTPNotifier.

Уровень пользовательского интерфейса

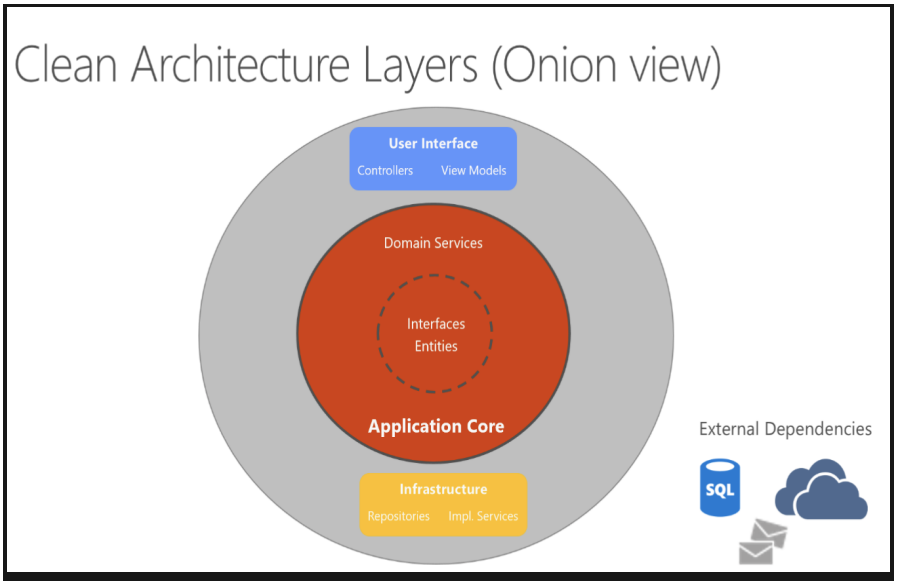
Слой пользовательского интерфейса в приложении MVC ASP.NET Core выступает в качестве точки входа для приложения. Он ссылается на слой ядра приложения, и его (слоя пользовательского интерфейса) типы взаимодействуют с инфраструктурой строго через программные интерфейсы, определённые в ядре приложения. В слое пользовательского интерфейса не разрешается прямое создание экземпляров для типов слоя инфраструктуры, а также их статические вызовы.

1. Контроллеры.
2. Фильтры.
3. Представления.
4. Модели представлений.
5. Запуск. Startup отвечает за настройку приложений и запись типов реализации в интерфейсы, обеспечивая корректную работу по внедрению зависимостей во время выполнения.

Пример хорошо спроектированной системы

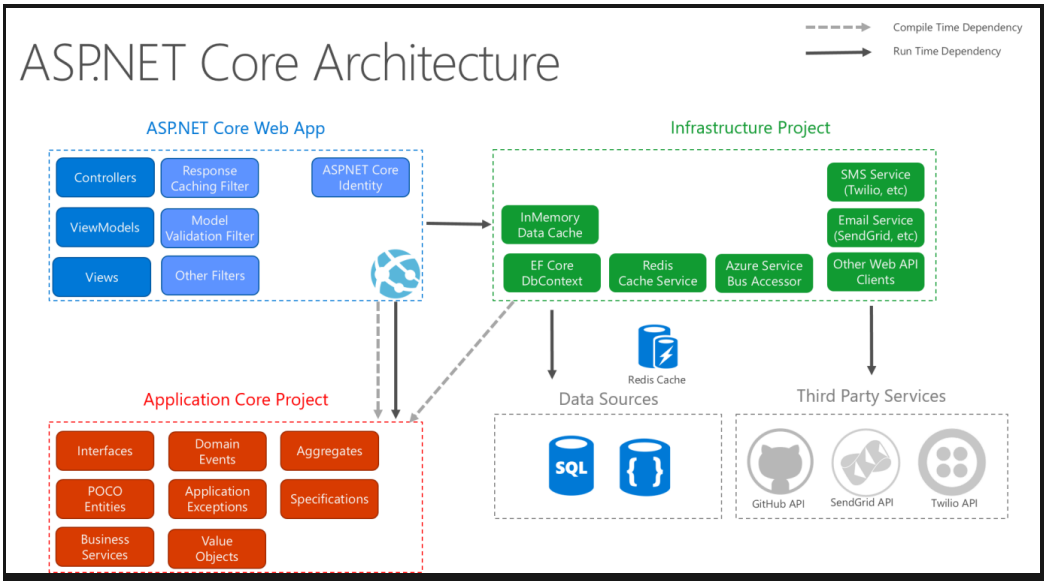
В рамках чистой архитектуры центральными элементами приложения считаются его бизнес-логика и модель. В этом случае бизнес-логика не зависит от доступа к данным или другим инфраструктурам, то есть стандартная зависимость инвертируется: инфраструктура и детали реализации зависят от ядра приложения. Эта функциональность достигается путём определения абстракций или интерфейсов в ядре приложения, которые реализуются типами, определёнными в слое инфраструктуры.

На рисунке — пример такого стиля представления архитектуры.



Пример чистой архитектуры

На рисунке ниже показано более подробное представление архитектуры приложения с фреймворком ASP.NET Core.



Пример чистой архитектуры

Преимущества чистой архитектуры	Недостатки чистой архитектуры
• **Управляемое разделение обязанностей кода** (самое значительное преимущество) и удобная структура пакетов. • **Ясность бизнес-логики**. Сценарии использования бизнес-логики наиболее легко различимы. Чистая архитектура помогает в «проектировании, ориентированном на домен» (DDD). DDD важно для сложных проектов, поскольку основное внимание уделяется основному домену и связанной с ним логике. • **Гибкость**. Архитектура больше не привязана к фреймворкам, поставщикам баз данных или серверов приложений. Можно заменить любого из них без каких-либо негативных последствий. Любой адаптер можно заменить в любое время любой другой реализацией на выбор. • **Тестирование**. Становится проще, поскольку можно установить объём тестирования в соответствии с вашими требованиями. Можно протестировать все интерфейсы и внешние взаимодействия. Модульное тестирование проводится легче, так как зависимости чётко определены. • **Интеграционные тесты**. Можно реализовать любой внешний сервис для использования во время интеграционных тестов. Например, если не хочется платить за	• **Кривая обучения немного крутая.** Может потребоваться некоторое время, чтобы понять, как взаимодействуют все уровни. • **Много компонентов.** Чистая архитектура содержит много дополнительных классов и пакетов, поэтому не подходит для приложений с низким уровнем сложности.

базы данных в облаке, можно использовать реализацию адаптера в памяти. ● **Устойчивость**. Бизнес-логика защищена, и ничто извне не выведет её строя. Код не зависит от внешнего фреймворка, контролируемого кем-то другим. Легко поддерживать проект в рабочем состоянии. ● **Отложенные решения**. Можно построить бизнес-логику, не зная деталей о будущих БД и фреймворках.	

Глоссарий

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее представляет собой область человеческой деятельности, которую система поддерживает.

Компонент — множество классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Пользовательский интерфейс — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Object-Relational Mapping (ORM) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Существуют как проприетарные, так и свободные реализации этой технологии.

Программное обеспечение на уровне адаптеров интерфейсов — это набор адаптеров, преобразующих данные из формата, наиболее удобного для вариантов использования и сущностей, в формат, наиболее удобный для некоторых внешних агентов, таких как база данных или веб-интерфейс.

Презентатор (Presenter) — разновидность шаблона проектирования «Скромный объект» (Humble Object), помогающего выявлять и защищать архитектурные границы. Его задача — получить данные от приложения и преобразовать их так, чтобы представление (View) перемещало их на экран. Например, если приложению потребуется отобразить дату в некотором поле, оно передаст презентатору объект Date. Затем презентатор преобразует дату в строку и поместит её в простую структуру данных, которая называется моделью представления (View Model), где представление сможет её найти.

Дополнительные материалы

1. Статья A starting point for Clean Architecture with ASP.NET Core.
2. Статья Clean Coder Blog
3. Видео «Чистая архитектура на Android».
4. Статья Clean Architecture using Golang.
5. Статья A Guided Tour inside a clean architecture code base.

Используемые источники

1. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения.
2. Статья «Гексагональная архитектура».
3. Статья Data Context Interaction.
4. Статья «Общие архитектуры веб-приложений».
5. The Clean Architecture (дата обращения: 06.06.2021).
6. Мартин, Р. Чистый код: создание, анализ и рефакторинг. / Р. Мартин. – Москва : Юпитер, 2018. – 464 с.
7. BCE — Boundary Control Entity.
8. Фаулер, М. Архитектура корпоративных приложений.: Пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 544 с.
9. Макконнелл, С. (Steve McConnell, Code Complete) Совершенный код.
10. Трюгве Р. — статья в Википедии.