МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическая работа № 1/2 ч.

Разработка мобильных приложений

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень: бакалавриат
(бакалавриат, магистратура, специалитет)

Форма обучения: очная
(очная, очно-заочная, заочная)

Направление(-я) подготовки: 09.03.02 «Информационные системы и технологии»
(код(-ы) и наименование(-я))

Институт: кибербезопасности и цифровых технологий
(полное и краткое наименование)

Кафедра: КБ-14 «Цифровые технологии обработки данных»
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2024/25
(учебный год цифрами)

Проверено и согласовано «____» _____20___г. _____ _____
(подпись директора Института/Филиала с расшифровкой)

Москва 2024 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Структурирование исходного кода заключается в организации программы таким образом, чтобы он был логически упорядочен, легко читаем и понятен для разработчиков. Это важнейший аспект разработки, особенно в крупных проектах, где работают большие команды и существует высокая степень сложности. Основными задачами структурирования кода и его преимущества заключаются в:

1. Улучшение читаемости и понимания кода. Когда код структурирован, он становится легче для чтения и понимания. Разработчики, работающие над проектом, могут быстро понять, что делает тот или иной модуль, класс или функция, даже если они впервые сталкиваются с кодом. Это особенно важно в командах, где люди часто меняются или работают с разными частями проекта.

2. Облегчение поддержки и расширения. Код, который хорошо структурирован, проще поддерживать и расширять. Внесение изменений в такой код требует меньших усилий, поскольку четкое разделение на модули и слои позволяет модифицировать одну часть системы, не затрагивая другие. Это снижает вероятность возникновения ошибок и ускоряет процесс разработки.

3. Обеспечение модульности. Структурирование кода подразумевает разделение системы на отдельные модули с четко определенными обязанностями. Модульность позволяет разрабатывать, тестировать и изменять части системы независимо друг от друга. Это повышает гибкость системы и делает её более адаптируемой к изменениям требований или технологий.

4. Улучшение тестируемости. Когда код структурирован, его легче тестировать. Модули, отвечающие за разные аспекты системы (например, бизнес-логику, работу с базой данных, взаимодействие с пользователем), могут быть протестированы в изоляции. Это уменьшает вероятность багов и упрощает отладку.

5. Повышение производительности команды. Хорошо структурированный код способствует улучшению командной работы. Когда каждый член команды понимает, как организован код, и следует единым принципам структурирования, процессы разработки и ревью кода становятся более эффективными. Это позволяет команде быстрее разрабатывать новые функции и исправлять баги.

6. Упрощение повторного использования кода. Структурированный код проще переиспользовать. Например, если в проекте существует модуль, отвечающий за определенную функциональность, его можно использовать в других проектах без значительных изменений. Это снижает затраты на разработку и способствует созданию библиотек и общих решений, которые можно использовать многократно.

7. Снижение технического долга. Технический долг — это термин, обозначающий затраты на исправление плохо написанного или неструктурированного кода в будущем. Чем хуже структурирован код, тем больше времени и ресурсов потребуется для его поддержания и модернизации. Хорошее структурирование помогает снизить технический долг, делая код более устойчивым к изменениям и улучшая его качество.

8. Обеспечение гибкости и масштабируемости. В условиях растущего проекта структурированный код позволяет легко масштабировать систему. Когда добавляются новые функции или увеличивается количество пользователей, хорошо структурированная система способна адаптироваться к этим изменениям без значительных усилий. Это достигается благодаря ясной иерархии компонентов и модулей, которые легко интегрируются друг с другом.

9. Повышение безопасности. Код, который структурирован и разделен на отдельные слои и модули, позволяет лучше контролировать доступ к данным и функциональности системы. Например, бизнес-логика может быть отделена от пользовательского интерфейса, что предотвращает несанкционированный доступ к важным данным. Это делает систему более безопасной и устойчивой к атакам.

10. Создание долгосрочной архитектуры. Хорошая структура кода помогает создать архитектуру, которая может быть устойчивой в долгосрочной перспективе. Даже если технология изменяется или требования к системе эволюционируют, хорошо структурированный код позволяет легко адаптироваться к этим изменениям, минимизирую необходимость переписывания значительной части системы.

Структурирование кода — это не просто хорошая практика, а необходимый шаг для создания качественного, поддерживаемого и масштабируемого

программного обеспечения. Оно улучшает читаемость, снижает риск ошибок, упрощает тестирование и поддержку, а также позволяет команде работать более эффективно. В конечном счете, хорошо структурированный код — это залог успешного и устойчивого проекта, способного быстро адаптироваться к изменениям и новым вызовам.

Понятие чистой архитектуры пошло из одноименной статьи Роберта Мартина 2012 года. Чистая архитектура является подходом к проектированию программного обеспечения, который помогает организовать код таким образом, чтобы он был легко поддерживаемым, масштабируемым и тестируемым.

Основная идея заключается в разделении системы на слои с четко определенными обязанностями, где зависимости направлены только внутрь, от внешних слоев к центральным.

1 **Entities** (Сущности) — центральный слой, содержащий бизнес-логику и правила предметной области. Сущности являются независимыми от внешних изменений, таких как изменения в базе данных или пользовательском интерфейсе.

2 **Use Cases** (Сценарии использования) — слой содержит логику приложения, которая определяет, как должны использоваться сущности для достижения конкретных целей. Сценарии использования описывают взаимодействие между сущностями.

3 **Interface Adapters** (Адаптеры интерфейсов) — слой преобразует данные из форматов, удобных для использования (например, API или базы данных), в форматы, понятные для внутренней логики приложения, и наоборот. Примеры включают преобразование данных из объектов базы данных в бизнес-объекты и обратно.

4 **Frameworks and Drivers** — внешний слой, содержащий фреймворки, инструменты, базы данных и так далее. В этом слое код должен связываться с предыдущим слоем, но не влиять в значительной степени на внутренние слои.

Все слои связаны правилом зависимости — Dependency Rule, которое означает, что в исходном коде все зависимости могут указывать только вовнутрь. Например, ничто из внешнего круга не может быть упомянуто кодом из внутреннего

круга. Это относится к функциям, классам, переменным или любым другим сущностям. Эту схему возможно изменять: добавлять или убирать слои, но основным правилом в архитектуре приложения должно всегда оставаться Dependency Rule. Чтобы зависимость была направлена в сторону обратную потоку данных, применяется принцип инверсии зависимостей

Переходы между слоями осуществляются через Boundaries, то есть через два интерфейса: один для запроса и один для ответа (Input/OutputPort). Интерфейсы необходимы, чтобы внутренний слой не зависел от внешнего (следуя Dependency Rule), но при этом мог передать ему данные.

На представленном примере 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.

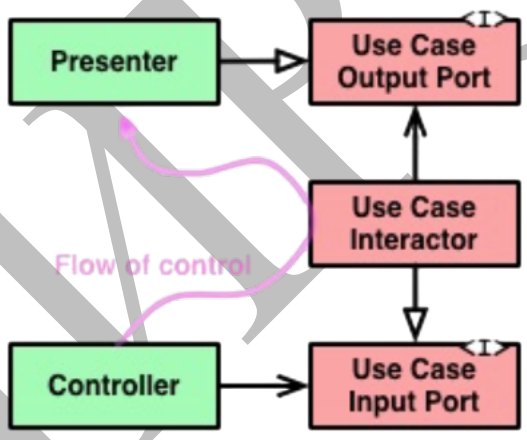


Рисунок 1 - Механизм Boundaries

В чистой архитектуре зависимости направлены от внешних слоев к внутренним. Внутренние слои не зависят от внешних слоев, а наоборот, внешние слои зависят от интерфейсов, определенных во внутренних слоях. Это достигается с помощью Dependency Injection, что позволяет легко подменять реализации зависимостей (например, для тестирования).

За счет строгого соблюдения принципов разделения слоев и инверсии зависимостей, чистая архитектура делает код легко тестируемым. Отдельные слои и модули возможно тестировать независимо друг от друга, используя заглушки (mock

objects) и другие техники.

Проект не должен зависить от конкретных реализаций пользовательского интерфейса. UI является просто еще одним внешним слоем, который может быть заменен или модифицирован без необходимости изменять бизнес-логику. База данных также считается деталью реализации и не должна влиять на внутреннюю бизнес-логику. Это позволяет легко заменять одну базу данных на другую или даже поддерживать несколько баз данных.

Внешние системы, такие как веб-сервисы, библиотеки или другие модули, также должны быть отделены от бизнес-логики. Это позволяет взаимодействовать с этими системами через адаптеры, которые можно легко заменить или обновить.

Важно, чтобы вся система следовала единому стилю и подходу, что способствует лучшей читаемости, поддерживаемости и обучению новых членов команды.

Эти основные понятия чистой архитектуры помогают разработчикам строить устойчивые, гибкие и масштабируемые программные системы, которые легко адаптируются к изменениям в требованиях или технологиях.

1 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

1.1 Диаграмма вариантов использования

Задание. Требуется установить приложение draw.io и нарисовать use-case диаграмму приложения

Проектирование сценариев использования (Use Case) в чистой архитектуре играет ключевую роль, так как данный слой описывает конкретные бизнес-процессы и взаимодействие с сущностями (Entities). Сценарии использования управляют потоком данных между различными слоями системы, обеспечивая выполнение бизнес-логики. Важно правильно спроектировать данный слой, чтобы код был легко поддерживаемым и соответствовал требованиям к проекту.

1. Определение бизнес-логики

Первым шагом в проектировании Use Case является четкое понимание и определение бизнес-логики или задачи, которую необходимо решить. Это могут быть сценарии работы с пользователем, обработка запросов или любые другие действия, связанные с бизнес-процессами системы.

Какие задачи должен выполнять пользователь?

Каковы требования к функциональности?

Какие данные используются или изменяются?

Пример: в интернет-магазине сценарий использования может быть оформлен как «Оформление заказа», где пользователь выбирает товары, заполняет информацию о доставке и производит оплату.

2. Идентификация сущностей (Entities)

Сценарии использования взаимодействуют с сущностями (Entities), которые содержат бизнес-правила и логику. Поэтому следующим шагом является определение, какие сущности будут участвовать в этом сценарии. Какие объекты домена будут задействованы?

Какие правила или условия должны соблюдаться?

Пример: для сценария «Оформление заказа» сущностями могут быть «Пользователь», «Товар», «Корзина» и «Заказ».

3. Определение границ (Boundaries)

Важной частью проектирования Use Case является разделение на внешние и внутренние границы. Данный принцип подразумевает четкое разграничение между бизнес-логикой (сущностями и сценариями) и внешними источниками данных (база данных, интерфейсы). Это позволяет избегать зависимости бизнес-логики от инфраструктурных компонентов. Какие данные должны быть получены из внешних систем (например, API, базы данных)? Как будет происходить обмен данными между слоями?

Пример: В сценарии «Оформление заказа» данные о товарах могут быть получены из внешней базы данных через интерфейсы, а обработка платежей через внешний платежный сервис.

4. Создание сценариев (Use Case Flow)

После определения сущностей и границ важно описать, как будет выполняться бизнес-логика в сценарии использования. Это может включать последовательность шагов или действий, которые должны быть выполнены для достижения результата. Какой основной поток операций? Какие условия или проверки должны выполняться? Какие альтернативные или исключительные сценарии могут возникнуть? Пример: для «Оформления заказа» шаги могут быть следующими:

Пользователь добавляет товар в корзину.

Система проверяет наличие товара.

Пользователь вводит данные о доставке и оплачивает заказ.

Система подтверждает заказ и отправляет уведомление.

5. Интерфейсы и контракты (Interfaces and Contracts)

Важной частью проектирования Use Case является создание интерфейсов, через которые сценарий использования будет взаимодействовать с внешними слоями (например, с интерфейсом пользователя или базой данных). Интерфейсы позволяют абстрагироваться от конкретной реализации и упрощают тестирование.

Какие методы или функции необходимы для выполнения сценария?

Какие параметры и возвращаемые значения требуются?

Пример: В сценарии «Оформление заказа» может быть интерфейс для

взаимодействия с платежной системой (например, *PaymentGateway*), который будет содержать метод *processPayment()*.

6. Обработка ошибок и исключений

В процессе выполнения сценариев могут возникать различные исключительные ситуации (например, ошибка при обработке платежа, отсутствие товара на складе и т.д.). Необходимо определить, как будут обрабатываться такие ситуации, и что должен делать сценарий в случае ошибки.

Какие возможны исключения?

Как система должна реагировать на исключения?

Какие данные или сообщения нужно вернуть пользователю?

Пример: В сценарии «Оформление заказа» может быть проверка на наличие товара перед подтверждением заказа. Если товар отсутствует, сценарий должен вернуть сообщение об ошибке пользователю.

7. Тестируемость сценария использования

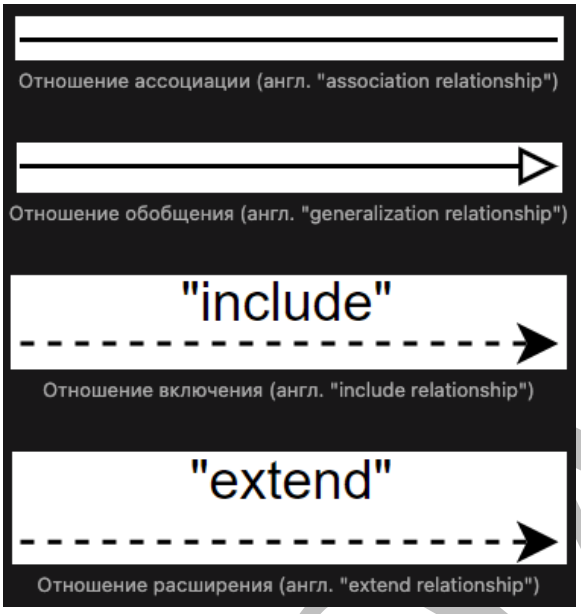
Сценарии использования должны быть легко тестируемыми. Для этого следует проектировать сценарии так, чтобы они не зависели от внешних систем. Это достигается путем использования интерфейсов и заглушек (mock-объектов) для замены внешних сервисов.

Как можно изолировать сценарий для тестирования?

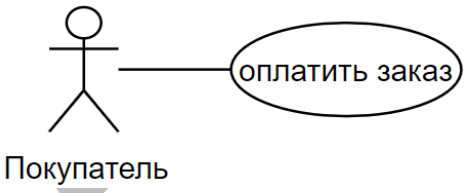
Какие зависимости нужно подменить заглушками?

Диаграмма вариантов использования начинается с размещения пользователя приложения (актор) или несколько возможных акторов в случаях, когда одним и тем же приложением пользуются гость, зарегистрированный пользователь, администратор. В терминологии UML, эллипс называется вариантом использования (англ. «use-case»). В общем случае, вариант использования – набор действий, который может быть использован актёром для взаимодействия с системой.

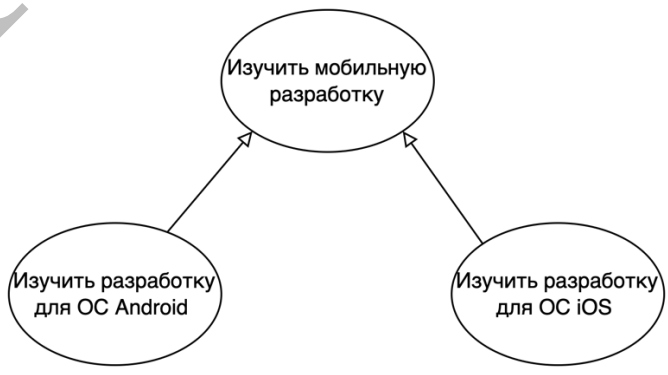
На диаграммах UML для связывания элементов используются различные соединительные линии, которые называются отношениями. Каждое такое отношение имеет собственное название и используется для достижения определённой цели.



Отношение ассоциации предназначено только для соединения актёров и вариантов использования. Нет никакого смысла соединять отношением ассоциации двух актёров или два варианта использования. Если на диаграмме вариантов использования актёр соединен с вариантом использования с помощью отношения ассоциации, это означает, что данный актёр может выполнять действия, описанные вариантом использования.



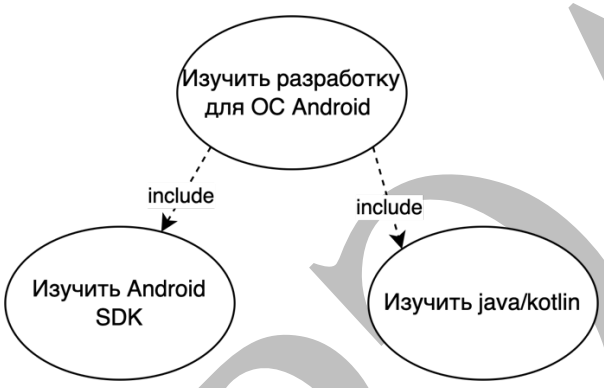
Отношение обобщения означает, что некоторый актёр (вариант использования) может быть обобщён до другого актёра (варианта использования). Стрелка направлена от частного случая(специализации) к общему случаю.



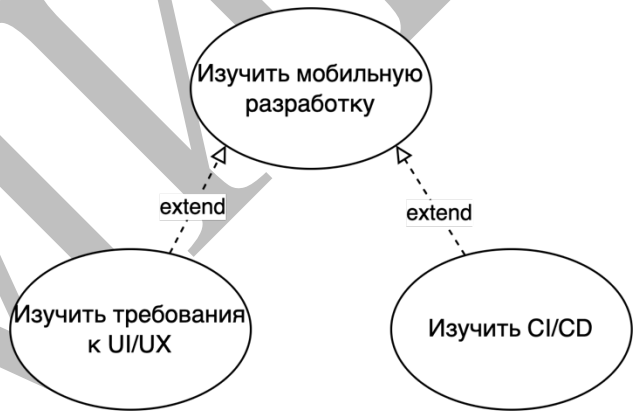
Отношение обобщения используется, чтобы показать, что одно действие

является частным случаем другого действия или что одну группу людей можно обобщить до другой группы

Отношение включения используется, чтобы показать, что некоторый вариант использования включает в себя другой вариант использования в качестве составной части. Когда мы используем отношение включения, мы подразумеваем, что составные варианты использования ОБЯЗАТЕЛЬНО входят в состав общего варианта использования.



Отношение расширения – выборочное отношение включения. Если отношение включения обозначает, что элемент обязательно включается в состав другого элемента, то в случае отношения расширения это включение *необязательно*.



Два нижних варианта использования описывают возможные «расширения» для базового варианта использования.

Общими рекомендациями при проектировании приложения являются:

– избегайте слишком детализированных диаграмм – важно сосредоточиться на ключевых взаимодействиях и сценариях;

– используйте правильные обозначения. Актор — человечек, вариант использования — эллипс с названием, система — прямоугольник, внутри которого

расположены эллипсы (варианты использования), взаимодействие — простая линия между актором и сценарием, Include/Extend — пунктирные линии с соответствующими надписями;

– рассмотрите модульность - для сложных систем имеет смысл разделить диаграммы на несколько модулей или контекстов, чтобы каждая диаграмма охватывала свою часть системы и взаимодействий;

– убедитесь, что диаграмма структурирована логически: основные действия должны быть центральными, второстепенные — на периферии. Размещайте актора, который инициирует взаимодействие, ближе к соответствующему варианту использования;

– определите обобщение (наследование) для акторов и вариантов использования. Если несколько акторов имеют общие черты или могут выполнять схожие действия, можно использовать обобщение. Это помогает избежать дублирования. То же самое возможно сделать и для вариантов использования, когда разные сценарии имеют общее поведение.

На рисунке 2 представлен пример диаграммы вариантов использования для типового интернет-магазина

Рисунок 2 – Пример Use Case диаграммы типового интернет-магазина

Задание: каждый студент продумывает функционал своего *собственного* приложения и формирует диаграмму вариантов использования.

В дальнейших практических занятиях, будет реализовываться функционал, указанный в диаграмме. Приложение должно содержать следующие функциональные возможности:

1. Авторизация в приложении.
2. Взаимодействия с каким-либо внешним сервисом, возвращающим данные о стоимости валюты, погоды и т.д. в виде json-формата. Возможно использовать сервисы типа https://www.wiremock.io/, https://mockapi.io/, https://github.com/typicode/json-server, https://thecatapi.com/ и т.д.
3. Сохранение данных в БД.
4. Отображение списка каких-либо сущностей с изображениями.
5. Отображение страницы сущности.
6. Различные возможности у авторизованного и гостевого пользователя.
7. Использование обученной модели для распознавания изображений, аудио, маски и т.д. (нужно произвести анализ существующих обученных моделей TenzorFlow Lite, которые желаете использовать в приложении).

Допускается использование собственных use case в приложении.

1.2 Разделение на слои

Приложение должно быть разбито на слои, у каждого слоя своя зона ответственности. Обычно выделяют следующие уровни: *presentation, domain, data.* **Presentation**: отвечает за отображение пользовательского интерфейса и реагирование на его события. **Domain**: бизнес-логика, изолированная от деталей реализации, определяет правила и операции, как приложение должно взаимодействовать с данными. **Data**: хранилище данных.

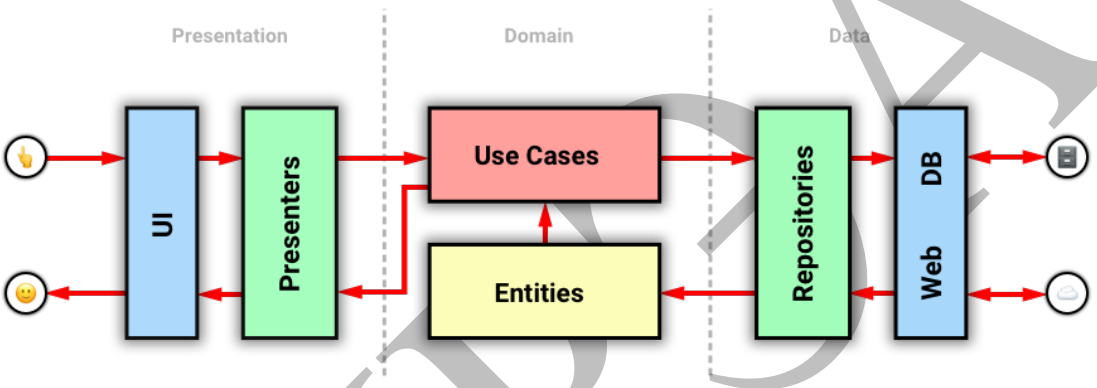


Рисунок 3 - Схема разложения слоев по категориям

Определимся с основными компонентами:

Entity: представляет объекты данных, с которыми работает бизнес-логика

Interactor или *UseCase*: содержит функциональность или операции

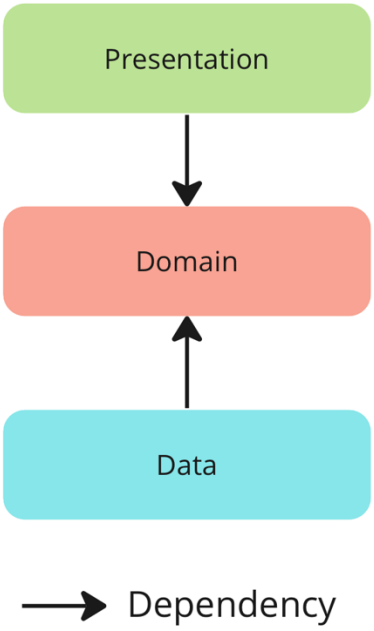
Repository: обеспечивает контракт для доступа к данным

Красными стрелками показано течение данных. Событие пользовательского интерфейса 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. В соответствии с бизнес-логикой, формируется результат, который передается обратно в Presenter. А тот, в свою очередь, отображает результат в UI. На переходах между слоями (не категориями, а слоями, отмеченными разным цветом.

При организации передачи данных стоит использовать общий контракт, такой как интерфейс или абстрактный класс, вместо прямой зависимости слоя верхнего уровня от компонентов слоя нижнего. Таким образом, каждый слой использует этот

контракт, что обеспечивает изоляцию изменений в верхнем слое.

В Clean Architecture центральным слоем является domain, тогда возможно схематично представить подобную связь зависимостей:



Domain слой содержит бизнес-логику, должен быть независим от деталей реализации приложения и внешних библиотек (можно делать исключения, пример RxJava, DI Framework). Это делает его высокоуровневым слоем.

Data слой отвечает за доступ к данным (локальным и удаленным) и их управлением. Модуль является низкоуровневым, допустимы любые зависимости. Базовыми компонентами данного модуля являются:

Repository Implementation: реализованные контракты репозиториев, которые определены в domain модуле;

Data Sources: конкретные реализации, обеспечивающие доступ к данным из различных источников (Remote: обращения к внешнему API для получения данных, Memory: взаимодействие с данными, хранящимися в оперативной памяти, Local: чтение данных из локальной базы данных)

Data Models: объекты данных, представляющие хранимую информацию. Включает в себя DTO (Data Transfer Objects), объекты для операций CRUD (Create, Read, Update, Delete), а также объекты запросов (Query) и т.д

Mappers: преобразование объектов данных, к примеру в Entity

Presentation отвечает за отрисовку пользовательского интерфейса и

управление его состояния на основе бизнес-сущностей. Слой является низкоуровневым, допустимы любые зависимости. Основные компоненты:

View: сам пользовательский UI в виде Fragment, Activity, Composable

Presenter / ViewModel: посредник между UI и бизнес-логикой в модельном виде MVVM, MVP, MVI

Следующим этапом проектирования приложения выделение различных слоев. Экран имеет право обращаться только к domain - слою. На рисунке 4 представлен пример декомпозиции на слои типового приложения категории «Фильмы». Прямоугольник movie указывает на слой data.

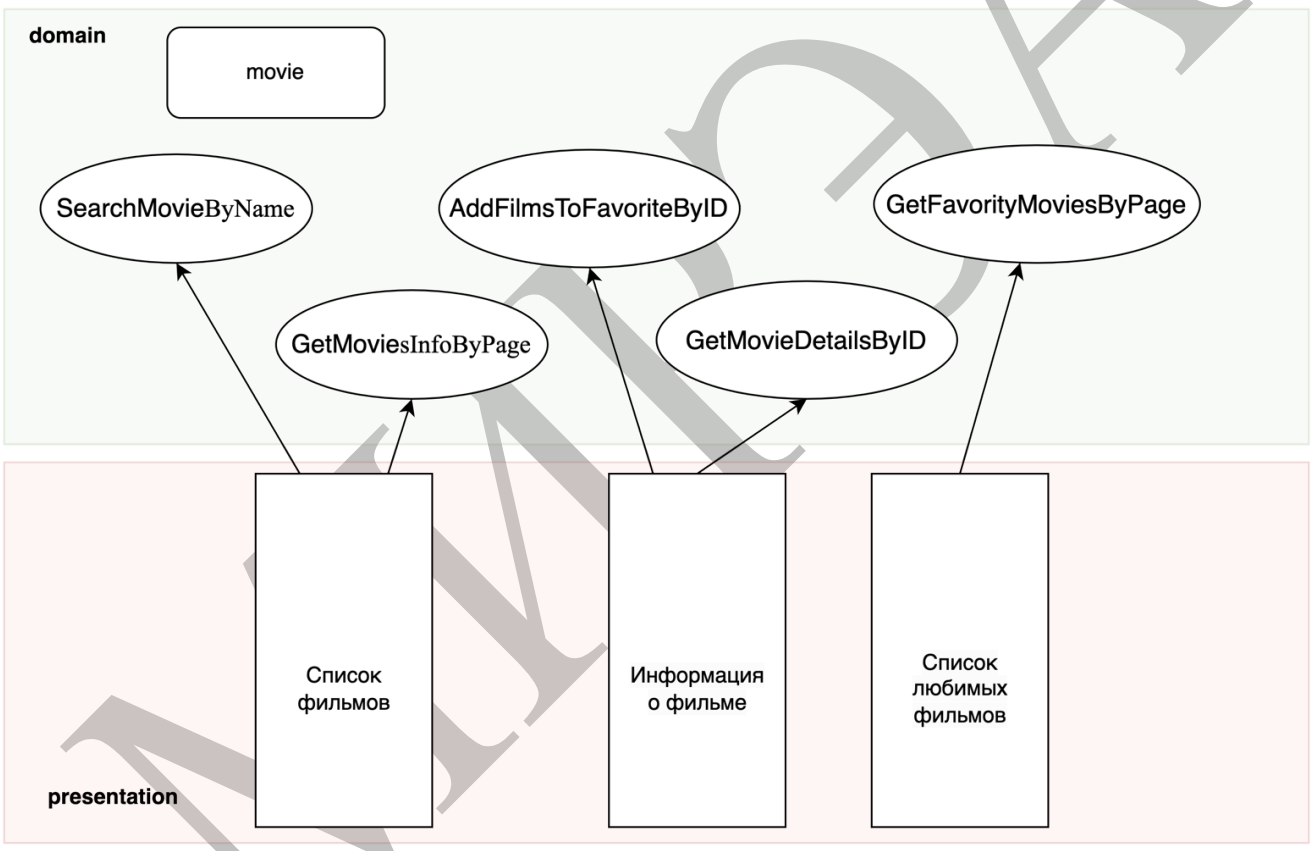


Рисунок 4 – Декомпозиция на слои типового приложения категории «Фильмы»

Задание: на основе сформированных use case необходимо спроектировать экраны приложения с указанием их зоны ответственности.

Далее рассмотрен будет слой ***data*** – это слой, который предназначен для хранения данных, получения данных и их отправки. То есть данный слой не имеет никакой бизнес логики, он никаким образом не манипулирует данными. И вот здесь должна

быть какая-то структура или какие-то классы, которые непосредственно будут отвечать за то, чтобы сохранить и извлечь информацию. Имеется много подходов, различных паттернов, но на текущий момент одно из самых распространённых решений является паттерн «репозиторий». Все данные, необходимые для приложения, поставляются из этого слоя через реализацию Repository (интерфейс находится в domain layer — слое бизнес-логики), который использует Repository Pattern со стратегией, которая, через фабрику, выбирает различные источники данных, в зависимости от определенных условий (рис.5).

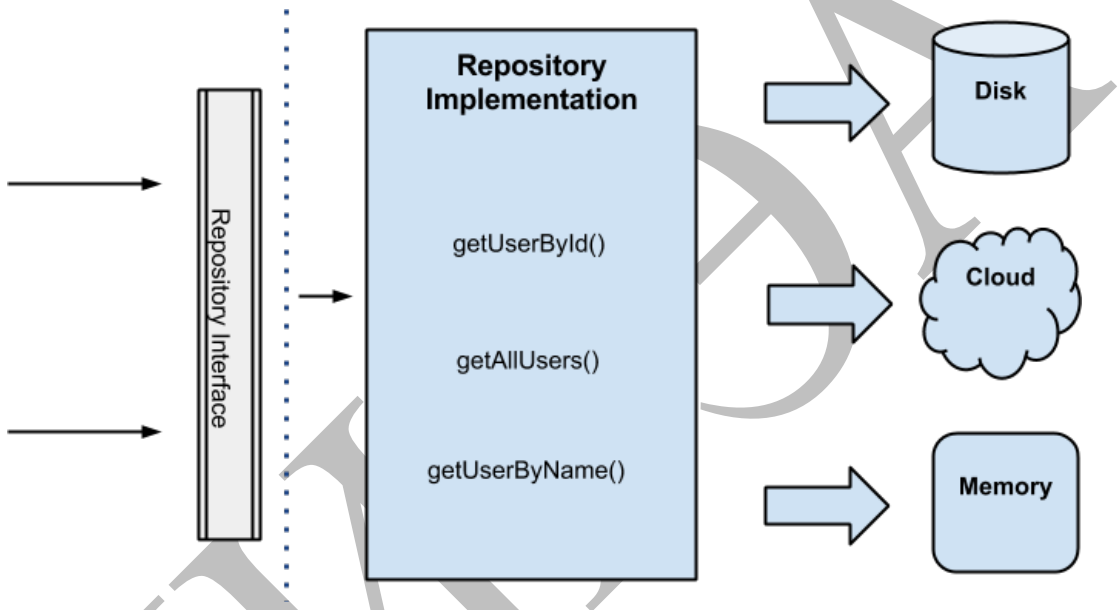


Рисунок 5 - Паттерн Repository

Например, для получения конкретного пользователя по id источником данных выбирается дисковый кэш, если пользователь уже загружен в него, в противном случае запрос отправляется в облако на получение данных для дальнейшего сохранения в тот же кэш. Идея всего этого заключается в том, что происхождение данных является понятным для клиента, которого не волнует, поступают данные из памяти, кэша или облака, ему важно только то, что данные будут получены и доступны.

Обычно, если имеется какая-та сущность, то к ней делается один репозиторий. В рассматриваемом примере сущность User. Соответственно, в проекте будет UserRepository, в котором будут методы для того, чтобы сохранить какую-то информацию и для того, чтобы получить какую-то информацию.

На рисунке 5 представлен пример декомпозиции на слои типового приложения категории «Фильмы».

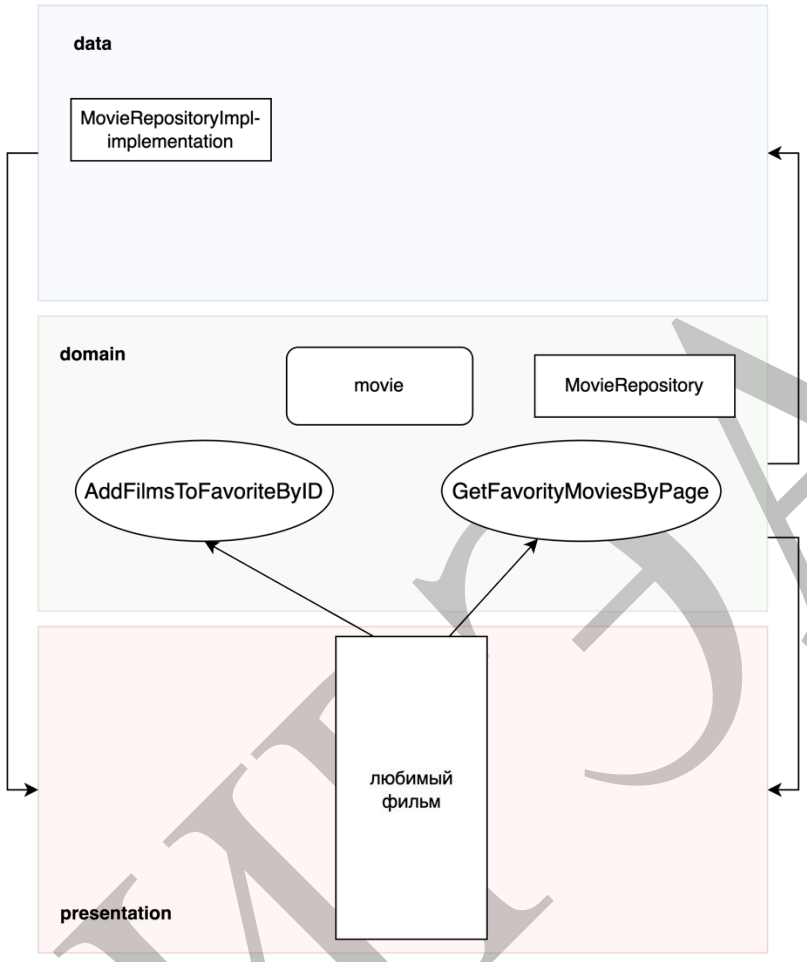


Рисунок 6 – Декомпозиция на слои экрана «Любимый фильм»

Задание: на основе сформированных use case необходимо спроектировать экраны приложения с указанием их зоны ответственности.

2 РЕАЛИЗАЦИЯ КАРКАСА ПРИЛОЖЕНИЯ

Задание. Требуется создать новый проект «ru.mirea.«фамилия».Lesson9». В меню «*File | New Project | Empty Views Activity*». Название модуля «MovieProject»

Реализуем некоторый функционал приложения для просмотра информации о любимом фильме. Необходимо реализовать экран, представленный на рисунке 7.

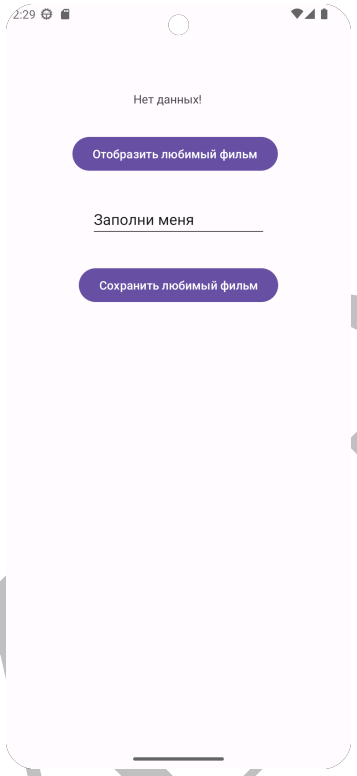


Рисунок 7 - Экран приложения MovieProject

В первую очередь необходимо создать директорию с названием ***domain***. Данная директория предназначена для хранения классов, реализующих бизнес-логику приложения. В данной директории необходимо создать класс Movie:

```
public class Movie {
    private int id;
    private String name;

    public Movie(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

В первую очередь в domain – директории необходимо создать два класса use-case: ***SaveFilmToFavoriteUseCase*** и ***GetFavoriteFilmUseCase***. Стоит учитывать, что

в случае модификации структуры возвращаемого значения лучше использовать не набор примитивов, а создать отдельную сущность. В нашем случае сущность Movie уже создана. Требуется создать директорию ***models*** в директории ***domain*** и перенести файл Movie. Далее реализуем use-case. Внутри данных классов создайте функции, выполняющие соответствующие задачи класса.

```
public class GetFavoriteFilmUseCase {

    public Movie execute(){
        return new Movie(3, "Game of thrones");
    }
}


public class SaveFilmToFavoriteUseCase {

    public boolean execute(Movie movie){
        if (movie.getName().isEmpty()){
            return false;
        }else {
            return true;
        }
    }
}
```

Отображение списка фильмов на главном экране происходит по нажатию соответствующих кнопок:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        EditText text = findViewById(R.id.editTextMovie);
        TextView textView = findViewById(R.id.textViewMovie);
        findViewById(R.id.buttonSaveMovie).setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Boolean result = new SaveFilmToFavoriteUseCase().execute(new
Movie(2, text.getText().toString()));
                textView.setText(String.format("Save result %s", result));
            }
        });
        findViewById(R.id.buttonGetMovie).setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Movie moview = new GetFavoriteFilmUseCase().execute();
                textView.setText(String.format("Save result %s",
moview.getName()));
            }
        });
    }
}
```

Далее рассмотрен будет слой ***data*** – это слой, который предназначен для

хранения данных, получения данных и их отправки. В рассматриваемом примере сущность фильм. Соответственно, в проекте будет MovieRepository. Далее необходимо создать такой репозиторий. Создадим директорию ***data*** и внутри нее папку ***repository***. В созданной директории необходимо сформировать класс ***MovieRepository***.

Далее реализуем базовую функциональность по сохранению и передачи данных.

```
public class MovieRepository {

    public boolean saveMovie(Movie movie){
        return true;
    }

    public Movie getMovie(){
        return new Movie(1,"Doctor Strange");
    }
}
```

Основной вопрос сейчас должен звучать так: «почему используется модель Movie в другом слое». Действительно, не стоит смешивать классы одного слоя с другим слоем, но в нашем случае ситуация немножко другая. Весь секрет того, почему используются модели в Data слое, на самом деле кроется в том, как use case будет связан вот с этим репозиторием. Домен слой он самый-самый главный и ключевой компонент в приложении, он ни от кого не должен зависеть, то есть у нас слой presentaion может зависеть от домена, data тоже может зависеть, но домен ни от кого не зависит, то есть ни от какого слоя. Слой presentaion может к себе подключить домен, дата тоже может к себе подключить домен, но домен к себе никого подключить не может, то есть если к домену начнем подключать другие слоя, то у нас получится такая проблема, как нарушение solid принципов, а именно сформируется круговая зависимость.

Существует хорошее решение, это использование интерфейсов и его реализация. Необходимо создать интерфейс MovieRepository в слое domain, а его реализацию сделать в слое data. Именно поэтому данный интерфейс имеет право использовать модель из своего слоя. Таким образом в домене также необходимо создать директорию repository, то есть имеется два пакета репозиторий, один пакет

репозиторий лежит внутри слоя data, а другой пакет репозиторий лежит внутри домена. Рассмотрим интерфейс из слоя domain.

```
public interface MovieRepository {
    public boolean saveMovie(Movie movie);
    public Movie getMovie();
}
```

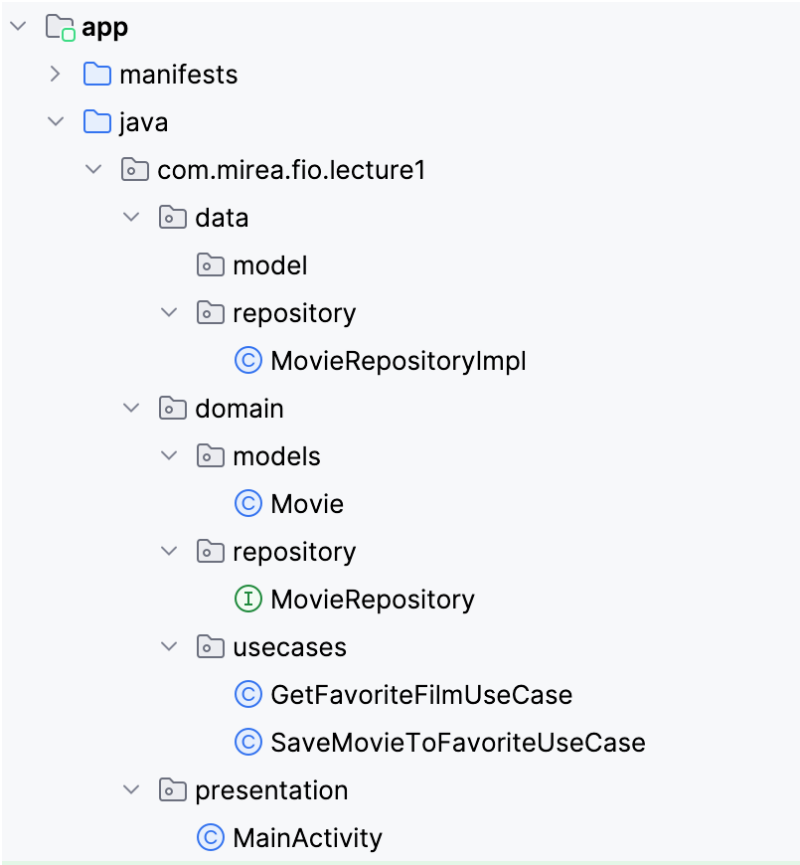
Далее необходимо создать реализацию интерфейса в слое data. Класс, реализующий интерфейс общепринято называть с указанием в конце Impl (сокр. от англ. Implementation)

```
public class MovieRepositoryImpl implements MovieRepository {

    @Override
    public boolean saveMovie(Movie movie){
        return true;
    }

    @Override
    public Movie getMovie(){
        return new Movie(1, "Game of throne");
        }
}
```

Теперь необходимо внести изменения в классы use-case.

```
public class GetFavoriteFilmUseCase {
    private MovieRepository movieRepository;

    public GetFavoriteFilmUseCase(MovieRepository movieRepository) {
        this.movieRepository = movieRepository;
    }

    public Movie execute(){
        return movieRepository.getMovie();
    }
}


public class SaveMovieToFavoriteUseCase {
    private MovieRepository movieRepository;

    public SaveMovieToFavoriteUseCase(MovieRepository movieRepository) {
            this.movieRepository = movieRepository;
    }

    public boolean execute(Movie movie){
        return movieRepository.saveMovie(movie);
    }
}
```

Таким образом структура проекта должны выглядеть следующим образом:



В MainActivity необходимо внести изменения для того, чтобы в слой domain передавалась реализация репозитория для получения данных.

```
MovieRepository movieRepository = new MovieRepositoryImpl(this);

findViewById(R.id.buttonSaveMovie).setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Boolean result = new
SaveMovieToFavoriteUseCase(movieRepository).execute(new Movie(2,
text.getText().toString()));
        textView.setText(String.format("Save result %s", result));
    }
});
findViewById(R.id.buttonGetMovie).setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Movie moview = new GetFavoriteFilmUseCase(movieRepository).execute();
        textView.setText(String.format("Save result %s", moview.getName()));
    }
});
```

Результатом интеграции передачи данных с помощью интерфесов является возможность внесение изменений в репозиторий, которые не повлекут никаких изменений в слое domain.

Задание: добавить механизм сохранения/получения информации о любимом фильме с помощью SharedPreferences. Обратите внимание на необходимость передачи context без использования слоя domain.

3 КОНТРОЛЬНОЕ ЗАДАНИЕ

Задание. Требуется создать новый проект «ru.mirea.«фамилия».«название Вашего проекта»». В меню «*File | New Project | Empty Views Activity*».

Проект на первом этапе представляет из себя болванку для дальнейшего наращивания функционала. Создать классы use-case, указанные на этапе проектирования на уровне слоя domain и data. Репозиторий должен вернуть тестовые данные.