

## Основные классы и паттерны в Domain-Driven Design (DDD)

В DDD выделяют несколько ключевых типов классов и паттернов, каждый из которых играет свою роль в моделировании бизнес-домена.

### 1. Entity (Сущность)

- **Назначение:** Представляет объект, обладающий уникальной идентичностью, которая сохраняется на протяжении всего жизненного цикла, даже если его свойства меняются.
- **Пример:** Пользователь, Заказ, Продукт.
- **Особенности:** Идентичность обычно реализуется через уникальный идентификатор (ID) <sup>[1]</sup> <sup>[2]</sup> <sup>[3]</sup>.

### 2. Value Object (Объект-значение)

- **Назначение:** Описывает объект, который определяется только своими атрибутами, а не идентичностью. Обычно неизменяемый.
- **Пример:** Деньги (Money), Адрес, Email.
- **Особенности:** Два объекта-значения считаются равными, если их свойства совпадают. Изменять такие объекты нельзя — вместо этого создаются новые экземпляры <sup>[4]</sup> <sup>[5]</sup> <sup>[3]</sup>.

### 3. Aggregate (Агрегат)

- **Назначение:** Группа связанных сущностей и объектов-значений, которые рассматриваются как единое целое для изменений данных и транзакций.
- **Особенности:** Один из объектов внутри агрегата становится корнем (Aggregate Root), через который осуществляется доступ к другим объектам агрегата. Внешние объекты могут ссылаться только на корень агрегата <sup>[6]</sup> <sup>[7]</sup> <sup>[3]</sup>.
- **Пример:** Заказ (Order) как агрегат, содержащий позиции заказа (OrderLine).

### 4. Repository (Репозиторий)

- **Назначение:** Абстракция для хранения и получения агрегатов (или сущностей) из источников данных (например, базы данных).
- **Особенности:** Репозиторий скрывает детали хранения данных и предоставляет коллекцию объектов домена, с которыми работает бизнес-логика <sup>[8]</sup> <sup>[9]</sup> <sup>[3]</sup>.

## 5. Domain Event (Доменное событие)

- **Назначение:** Фиксирует важные события, произошедшие в домене, которые могут быть интересны другим частям системы.
- **Пример:** Заказ оплачен, Пользователь зарегистрирован<sup>[3]</sup>.

## 6. Factory (Фабрика)

- **Назначение:** Инкапсулирует процесс создания сложных объектов или агрегатов, чтобы избежать дублирования логики и ошибок при создании<sup>[3]</sup>.

## 7. Service (Сервис)

- **Назначение:** Описывает операции, которые не подходят ни для одной конкретной сущности или объекта-значения, но важны для предметной области.
- **Пример:** Сервис расчёта стоимости доставки.

**Таблица: Сравнение основных классов DDD**

Класс	Идентичность	Изменяемость	Назначение	Пример
Entity	Да	Да	Представляет уникальный объект	Пользователь, Заказ
Value Object	Нет	Нет	Описывает значение	Деньги, Адрес
Aggregate	Да	Да	Группа связанных объектов	Заказ с позициями
Repository	-	-	Доступ к сущностям/агрегатам	OrderRepository
Domain Event	-	Нет	Фиксирует значимые события	OrderPaidEvent
Factory	-	-	Создаёт сложные объекты	OrderFactory
Service	-	-	Выполняет операции домена	ShippingCalculator

Эти паттерны помогают структурировать код, сделать его ближе к бизнес-домену и упростить сопровождение сложных систем<sup>[1] [3] [9]</sup>.

✱✱

**Взаимодействие классов и паттернов DDD: где живёт бизнес-логика и как всё связано**

## 1. Где находится бизнес-логика

- **Бизнес-логика** в DDD преимущественно размещается внутри доменных сущностей (Entity), объектов-значений (Value Object), агрегатов (Aggregate) и доменных сервисов (Domain Service).
- **Агрегаты** инкапсулируют инварианты и правила целостности для группы связанных объектов.
- **Value Object** содержит логику, связанную с их значением (например, валидация Email, арифметика Money).
- **Domain Service** реализует бизнес-операции, которые не подходят ни одной конкретной сущности или объекту-значению.

## 2. Use Case (Application Service, Сценарий использования)

- **Use Case** (или Application Service) — это слой приложения, который координирует выполнение бизнес-сценариев, используя доменные объекты.
- Use Case не содержит бизнес-правил, а только управляет последовательностью действий, делегируя бизнес-логику домену.
- Примеры: оформление заказа, регистрация пользователя, начисление бонусов.

## 3. Схема взаимодействия

1. **Внешний слой** (например, контроллер, API) принимает запрос пользователя и вызывает соответствующий Use Case.
2. **Use Case** (Application Service) выполняет:
  - Валидацию входных данных (на уровне приложения).
  - Получение нужных агрегатов или сущностей через Repository.
  - Вызов методов агрегата или доменного сервиса для выполнения бизнес-логики.
  - Сохранение изменений через Repository.
  - Публикацию Domain Events, если необходимо.
3. **Repository** абстрагирует работу с хранилищем данных и возвращает агрегаты/сущности.
4. **Aggregate/Entity/Value Object** содержат бизнес-правила и изменяют своё состояние согласно этим правилам.
5. **Domain Event** регистрируется, если в домене произошло значимое событие.
6. **Factory** используется для создания сложных объектов или агрегатов, если процесс создания нетривиален.

## 4. Пример потока: оформление заказа

1. Контроллер вызывает `Use Case PlaceOrder`.
2. `Use Case` через `Repository` получает необходимые данные (например, пользователя, товары).
3. `Use Case` создаёт агрегат `Order` (через `Factory`, если нужно) и вызывает на нём метод `place()`.
4. Внутри агрегата `Order` происходит проверка бизнес-правил (например, проверка наличия товара, расчёт суммы).
5. Если всё успешно, агрегат генерирует `Domain Event OrderPlaced`.
6. `Use Case` сохраняет агрегат через `Repository`.
7. `Use Case` может инициировать обработку `Domain Event` (например, отправку письма клиенту).

## 5. Таблица: кто что вызывает

Класс/Паттерн	Кто вызывает	Что вызывает/делает
Controller/API	Use Case	Запускает бизнес-сценарий
Use Case	Repository, Aggregate, Service	Получает/сохраняет данные, делегирует бизнес-логику
Repository	Use Case	Загружает/сохраняет агрегаты
Aggregate/Entity	Use Case, Repository	Инкапсулирует бизнес-правила, изменяет состояние
Value Object	Aggregate, Entity, Service	Валидирует и реализует операции над значениями
Domain Service	Use Case, Aggregate	Реализует доменные операции вне сущностей
Factory	Use Case, Aggregate	Создаёт сложные объекты
Domain Event	Aggregate, Use Case	Фиксирует важные события домена

## 6. Кратко: роль Use Case

- **Use Case** — это дирижёр, который координирует работу доменных объектов, но не реализует бизнес-логику сам.
- Вся сложная бизнес-логика должна быть инкапсулирована в доменных классах (`Aggregate`, `Entity`, `Value Object`, `Domain Service`).

Такой подход позволяет отделить сценарии использования от бизнес-правил, повысить тестируемость и гибкость архитектуры.

## Пример структуры проекта DDD: от каталога `src` до инфраструктурного слоя

Ниже приведён типовой пример структуры папок для приложения, реализующего принципы Domain-Driven Design. Такая схема помогает разделить бизнес-логику, сценарии использования и инфраструктурные детали, делая проект масштабируемым и поддерживаемым <sup>[10] [11] [12]</sup>.

```
src/
├── modules/
│   ├── users/
│   │   ├── application/      # Use Cases, Application Services
│   │   ├── domain/          # Entities, Value Objects, Aggregates, Domain Services
│   │   ├── infrastructure/   # Репозитории, интеграции, адаптеры (реализации интерфейсов)
│   │   ├── presentation/     # Контроллеры, API, обработчики запросов (если требуется)
│   │   └── users.module.ts    # Точка входа модуля
│   ├── orders/
│   │   ├── application/
│   │   ├── domain/
│   │   ├── infrastructure/
│   │   ├── presentation/
│   │   └── orders.module.ts
│   └── ... (другие модули)
├── shared/                   # Общие компоненты для всех модулей
│   ├── kernel/               # Общие Value Objects, базовые классы (например, BaseEntity)
│   ├── providers/            # Общие сервисы (например, почта, логирование)
│   ├── errors/               # Общие ошибки и исключения
│   └── utils/                # Вспомогательные функции
├── config/                   # Конфигурация приложения
├── database/                 # Миграции, схемы, конфиги подключения
└── main.ts                   # Точка входа приложения
```

### Описание слоёв и папок

- **modules/** — каждый модуль отражает отдельный бизнес-контекст (bounded context).  
Внутри модуля:
  - **domain/** — бизнес-логика: сущности, объекты-значения, агрегаты, доменные сервисы, доменные события.
  - **application/** — сценарии использования (Use Cases), application-сервисы, DTO, мапперы.
  - **infrastructure/** — реализация репозиторий, интеграция с внешними сервисами, работа с БД, адаптеры.
  - **presentation/** — контроллеры, обработчики HTTP/REST/gRPC/GraphQL-запросов.
- **shared/** — общие компоненты, используемые несколькими модулями.
- **config/** — файлы конфигурации приложения.
- **database/** — миграции, схемы БД, скрипты инициализации.
- **main.ts** — основной файл запуска приложения.

## Как связаны слои

- **Слой domain** не зависит ни от каких внешних слоёв — только от себя и shared.
- **Слой application** использует domain и orchestrates бизнес-сценарии, но не реализует бизнес-логику.
- **Слой infrastructure** реализует технические детали (работа с БД, сетью, файлами и т.д.), инкапсулируя их за интерфейсами domain/application.
- **Слой presentation** (если есть) отвечает за взаимодействие с внешним миром (API, UI), вызывая use cases из application.

## Пример для одного модуля

```
src/modules/orders/
├── application/
│   ├── use-cases/
│   │   └── place-order.usecase.ts
│   ├── dto/
│   │   └── place-order.dto.ts
│   └── domain/
│       ├── entities/
│       │   └── order.entity.ts
│       ├── value-objects/
│       │   └── order-id.vo.ts
│       ├── services/
│       │   └── order-policy.service.ts
│       └── events/
│           └── order-placed.event.ts
├── infrastructure/
│   ├── repositories/
│   │   └── order.repository.ts
│   ├── mappers/
│   │   └── order.mapper.ts
├── presentation/
│   └── order.controller.ts
└── orders.module.ts
```

## Инфраструктурный слой

Инфраструктурный слой реализует:

- Репозитории (доступ к данным: БД, кеши, файловая система)
- Интеграцию с внешними сервисами (API, очереди, email и т.д.)
- Адаптеры и мапперы данных
- Конфигурацию и DI-контейнеры [\[12\]](#) [\[13\]](#) [\[14\]](#)

Такой подход облегчает масштабирование, тестирование и сопровождаемость кода, а также позволяет изолировать бизнес-логику от технических деталей реализации.

**Примечание:** Структура может незначительно различаться в зависимости от языка, фреймворка и специфики проекта, но общие принципы разделения домена, приложения и инфраструктуры сохраняются [\[10\]](#) [\[11\]](#) [\[12\]](#).

✱

**теперь приведи пример взаимодействия модулей, основываясь на концепции onion architecture + DDD. мне важно понять, как сделать правильный flow и при этом не сделать лишних зависимостей (чтобы модули были изолированы друг от друга).**

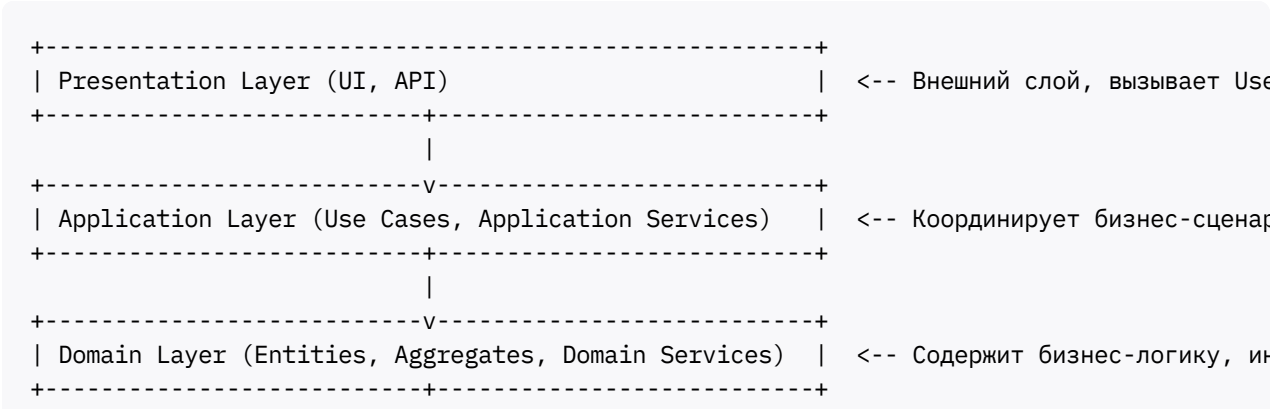
Для правильного построения взаимодействия модулей в архитектуре, основанной на **Onion Architecture + DDD**, важно обеспечить:

- **Изоляцию модулей друг от друга** (минимум зависимостей между модулями)
- **Направление зависимостей внутрь** — от внешних слоёв к ядру (домена)
- **Чёткое разграничение ответственности слоёв**

**Ключевые принципы взаимодействия модулей в Onion Architecture + DDD**

Принцип	Описание
Зависимости направлены внутрь	Внешние слои (инфраструктура, UI) зависят от внутренних (application, domain), а домен — только от себя
Модули изолированы друг от друга	Каждый модуль — отдельный Bounded Context с собственным доменом, приложением и инфраструктурой
Взаимодействие через интерфейсы (порты)	Модули общаются друг с другом через интерфейсы (Application Services, Domain Events, API), а не напрямую
Инверсия зависимостей	Доменные интерфейсы определяются в ядре, реализации — во внешних слоях, которые внедряются через DI

**Пример схемы взаимодействия модулей**



```

|
+-----v-----+
| Infrastructure Layer (Реализация репозитория, адаптеры) | <-- Реализует интерфейсы до
+-----+

```

## Как модули взаимодействуют между собой

- Каждый модуль содержит **собственные слои Domain, Application, Infrastructure**.
- Внутри модуля:
  - **Domain** — бизнес-логика и интерфейсы (например, интерфейс репозитория).
  - **Application** — сценарии использования, которые вызывают доменные объекты.
  - **Infrastructure** — конкретные реализации интерфейсов (репозитории, интеграции).
- **Модули не зависят друг от друга напрямую**. Если нужно взаимодействие — оно происходит через:
  - **События домена (Domain Events)** — один модуль публикует событие, другой подписывается и реагирует.
  - **Внешние API/сервисы** — модули могут обмениваться данными через API, а не через внутренние классы.
  - **Интерфейсы (порты)** — если один модуль должен использовать функционал другого, он зависит только от интерфейса, а реализация внедряется через DI.

## Пример потока вызовов между модулями

1. **Presentation** (например, HTTP-контроллер) вызывает Use Case из модуля `Orders`.
2. Use Case из `Orders` вызывает методы доменных агрегатов и сервисов этого же модуля.
3. Для сохранения данных Use Case обращается к интерфейсам репозитория, которые реализованы в инфраструктуре модуля `Orders`.
4. Если в процессе возникает событие, например `OrderPlaced`, оно публикуется как **Domain Event**.
5. Другой модуль, например `Notifications`, подписан на это событие и реагирует, например, отправкой письма.
6. Модуль `Notifications` не зависит от внутренностей `Orders`, а только от интерфейса события.

## Как избежать лишних зависимостей между модулями

- **Не импортировать классы домена одного модуля в другой напрямую**.
- Использовать **Domain Events** для асинхронной коммуникации.
- Если нужна синхронная коммуникация — определить **интерфейс (порт) в одном из модулей** и передавать его реализацию через DI.
- Внешние слои (`Presentation`, `Infrastructure`) могут зависеть от нескольких модулей, но внутренние слои — только от своего домена.

- Общие абстракции (например, базовые классы, общие Value Objects) выносить в отдельный shared модуль.

## Итог: Правильный flow и изоляция

Слой / Модуль	Зависит от	Вызывает / Использует
Presentation	Application (несколько модулей)	Вызывает Use Cases
Application (Orders)	Domain (Orders)	Вызывает доменные агрегаты, сервисы
Domain (Orders)	Только себя, shared	Определяет интерфейсы, бизнес-логику
Infrastructure (Orders)	Domain (Orders)	Реализует интерфейсы репозитория
Другие модули	Свои слои, shared	Через Domain Events или интерфейсы

## Резюме

- **Onion Architecture** гарантирует, что зависимости направлены только внутрь — к домену.
- **DDD** обеспечивает, что домен — это ядро с бизнес-логикой и интерфейсами.
- **Модули изолированы**, общаются через **интерфейсы и события**, что предотвращает жёсткие связи.
- **Use Cases** в Application слое координируют вызовы, но не содержат бизнес-логику.
- **Инфраструктура** реализует детали, инжектируется вовне.

Этот подход позволяет строить масштабируемую, легко поддерживаемую систему с чётким разделением ответственности и минимальными связями между модулями.

Источники:

- [15] [herbertograca.com](http://herbertograca.com) — DDD + Onion Architecture + Ports & Adapters
- [16] [dev.to](http://dev.to) — Onion Architecture в DDD, примеры интерфейсов репозитория
- [17] [itsjavi.com](http://itsjavi.com) — Правила Onion Architecture и инверсия зависимостей
- [18] [stackoverflow.com](http://stackoverflow.com) — Связь DDD и Onion Architecture

✱

**предположим, у меня есть общий Entity, или общий Value Object, или общий DTO, который будет использоваться больше, чем в одном сервисе.**

1. куда лучше всего положить такие классы?
2. стоит ли делать для них интерфейсы?
3. Куда лучше положить общий Entity, Value Object или DTO, используемые в нескольких сервисах?

- Для таких общих классов обычно выделяют отдельный модуль или пакет, который называют **Shared Kernel** или просто **shared** (общий модуль).
- В этом модуле размещают объекты, которые не принадлежат конкретному бизнес-контексту, но при этом используются в нескольких bounded context или сервисах.
- Это может быть папка `src/shared/domain/` для общих сущностей и объектов-значений, а для DTO — `src/shared/application/dto/` или аналогично.
- Главное — чтобы этот модуль был **минимально зависимым**, не содержал бизнес-логики конкретных сервисов и служил именно для переиспользования общих моделей<sup>[19] [20]</sup>.

## 2. Стоит ли делать для них интерфейсы?

- Для **Entity** и **Value Object** в DDD обычно интерфейсы не нужны, поскольку они сами по себе являются конкретными моделями с поведением и состоянием, и их основная задача — инкапсуляция бизнес-логики и данных<sup>[21] [22] [23]</sup>.
- Интерфейсы целесообразно создавать для **сервисов** (Domain Services, Application Services) и для **репозитория**, чтобы отделить абстракции от конкретных реализаций инфраструктуры.
- Для DTO интерфейсы обычно не делают, так как DTO — это простые структуры данных без поведения, они служат для передачи данных между слоями или сервисами.
- Если же общий класс используется как контракт (например, API-интерфейс), то можно определить интерфейс или контракт, но это скорее относится к API или Application слою, а не к домену<sup>[24]</sup>.

## Кратко:

Вопрос	Рекомендация
Где хранить общий Entity/VO/DTO?	В отдельном <b>shared</b> модуле (Shared Kernel), минимально зависимом и доступном для нужных сервисов
Делать ли интерфейсы для Entity/VO?	Обычно <b>нет</b> , они сами по себе конкретные доменные модели с поведением
Делать ли интерфейсы для DTO?	Обычно <b>нет</b> , DTO — простые структуры данных без поведения
Делать ли интерфейсы для сервисов/репозитория?	Да, для отделения абстракций от реализации и удобства тестирования и замены реализаций

## Дополнительно

- Value Object должен быть **неизменяемым** и сравниваться по значению, а не по ссылке<sup>[22] [23]</sup>.
- Если Value Object используется в нескольких агрегатах или модулях, его логично вынести в общий модуль, чтобы избежать дублирования и обеспечить единообразие<sup>[19]</sup>.

- DTO часто создаются отдельно для каждого Use Case, но если структура повторяется, можно вынести общий DTO в shared <sup>[24]</sup>.
- Не стоит делать общий модуль слишком большим — он должен содержать только действительно общие и стабильные компоненты, чтобы не создавать сильных связей между модулями.

Таким образом, **общие Entity, Value Object и DTO лучше всего хранить в отдельном shared-модуле**, а интерфейсы создавать только для сервисов и репозиторий, а не для моделей данных. Это обеспечивает переиспользование без лишних зависимостей и сохраняет чистоту архитектуры.



## если я тебя правильно понял, то зависимость доменного слоя от конкретной реализации в shared - это ОК?

В DDD и Onion Architecture принципиально важно, чтобы **доменный слой не зависел от конкретных реализаций**, включая инфраструктурные детали и даже реализации из shared-модуля. Это ключевая идея, чтобы сохранить чистоту домена и его независимость.

Однако когда речь идёт о **shared kernel** — общем модуле с сущностями, value objects или DTO, используемыми в нескольких bounded contexts, ситуация немного сложнее.

### Ответ на ваш вопрос

**Зависимость доменного слоя от конкретной реализации в shared — не является хорошей практикой.**

- **Shared Kernel — это часть домена, которая действительно может быть общей для нескольких контекстов**, и она должна содержать только **абстрактные, стабильные, бизнес-ориентированные модели и правила**.
- В этом случае **shared kernel — это тоже доменный слой**, только общий для нескольких bounded contexts. Доменные классы из shared kernel — это не инфраструктурные реализации, а именно **доменные модели и абстракции**.
- Доменные слои разных модулей могут зависеть от **shared kernel как от общего доменного ядра**, но не от конкретных реализаций инфраструктуры или технических деталей.

### Что это значит на практике

Ситуация	Рекомендация
Общие Entity, Value Object, DTO в нескольких модулях	Вынести в отдельный <b>shared kernel</b> — это доменный слой, который зависит только от себя и shared

Ситуация	Рекомендация
Domain слой модуля зависит от shared kernel	ОК, это нормальная зависимость от общего доменного ядра
Domain слой зависит от конкретных реализаций shared (например, репозитория, сервисов)	НЕТ, это нарушает принципы DDD и Onion Architecture
Shared kernel содержит инфраструктурные детали	НЕЛЬЗЯ, shared kernel должен быть чистым доменным слоем

## Почему так?

- **DDD и Onion Architecture требуют, чтобы домен был максимально изолирован от технических деталей.**
- Shared kernel — это не инфраструктура, а **общее доменное ядро**, поэтому домен может и должен зависеть от него.
- Если в shared kernel попадут конкретные реализации (например, репозитории, адаптеры), то домен потеряет независимость и чистоту, что приведёт к сильной связанности и усложнению поддержки.

## Подытоживая

- **Зависимость доменного слоя от общего доменного ядра (shared kernel) — это хорошо и правильно.**
- **Зависимость доменного слоя от конкретных реализаций из shared — это плохо и нарушает архитектуру.**
- Shared kernel — это часть домена, содержащая общие сущности, value objects и бизнес-правила, а не технические реализации.

## Дополнительные ссылки из поиска:

- Shared Kernel используется для общих классов доменной модели, когда несколько bounded contexts тесно связаны и должны использовать одни и те же бизнес-правила [\[25\]](#) [\[26\]](#).
- Domain слой не должен зависеть от внешних библиотек и инфраструктуры, включая конкретные реализации [\[27\]](#).
- Shared Kernel — это исключение, которое требует осторожности, чтобы не создавать сильной связанности между контекстами [\[26\]](#).

Если нужны, могу помочь с примером организации shared kernel и зависимостей в проекте.

# под конкретной реализацией я подразумевал классы без интерфейсов

Если под **конкретной реализацией** вы имеете в виду классы **без интерфейсов**, то в контексте DDD и Onion Architecture это важный момент.

## Ключевые моменты:

- Доменные слои (Domain, Application) должны зависеть только от абстракций (интерфейсов или абстрактных классов), а не от конкретных классов. Это позволяет легко менять реализации, упрощает тестирование и снижает связанность <sup>[28]</sup>.
- Если в shared kernel вы помещаете **конкретные классы без интерфейсов**, и доменный слой других модулей напрямую от них зависит, то это создаёт жёсткую связь и нарушает принцип инверсии зависимостей.
- В идеале в shared kernel **должны быть только доменные модели (Entity, Value Object, DTO) и абстракции (интерфейсы)**, а конкретные реализации (например, репозитории, сервисы) — в инфраструктурном слое соответствующего модуля <sup>[29]</sup>.

## Что делать с классами без интерфейсов в shared?

- Если класс в shared — это **чистая модель (Entity, VO, DTO)**, то интерфейсы для них обычно не нужны, потому что они сами по себе абстракция бизнес-логики и данных <sup>[30]</sup>.
- Если класс — **сервис или компонент с поведением**, который будет использоваться в нескольких модулях, то для него нужно определить **интерфейс в shared kernel или в домене**, а конкретную реализацию — в инфраструктуре. Так другие модули будут зависеть от интерфейса, а не от реализации <sup>[28] [29]</sup>.

## Итог:

Вопрос	Рекомендация
Можно ли домену зависеть от конкретных классов без интерфейсов из shared?	<b>Нет</b> , это нарушает принципы инверсии зависимостей и снижает гибкость архитектуры
Нужно ли делать интерфейсы для моделей (Entity, VO, DTO) в shared?	Обычно <b>нет</b> , они уже являются абстракциями бизнес-логики
Нужно ли делать интерфейсы для сервисов или репозиториях в shared?	<b>Да</b> , чтобы отделить абстракции от реализаций и избежать жёсткой связи
Где размещать реализации сервисов и репозиториях?	В инфраструктурном слое соответствующего модуля, не в shared kernel

## Почему это важно?

Инверсия зависимостей — фундаментальный принцип в Onion Architecture и DDD, который требует, чтобы зависимости шли от деталей к абстракциям, а не наоборот<sup>[28]</sup>. Если домен зависит от конкретных классов, то он теряет независимость, усложняется тестирование и сопровождение.

Если кратко: **доменные слои могут и должны зависеть от общих моделей из shared kernel (которые обычно не имеют интерфейсов), но не должны зависеть от конкретных реализаций сервисов или репозиторий без интерфейсов. Для сервисов и репозиторий интерфейсы обязательны, а реализации — в инфраструктуре.**

### Дополнительные ссылки из поиска:

- "Инверсия зависимостей, делая класс зависимым от абстракций (интерфейсов и/или абстрактных классов) вместо конкретных классов" <sup>[28]</sup>
- "Domain layer NEVER implements repository. Domain uses behavior description, repository must be implemented in Infrastructure layer" <sup>[29]</sup>
- "Общее ядро (shared kernel) содержит общий код, но компоненты остаются независимыми друг от друга" <sup>[31]</sup>

Если хотите, могу помочь с примером организации интерфейсов и реализаций в shared kernel и инфраструктуре.



**давай поговорим конкретно про зависимость доменного слоя какого-то модуля от, скажем, Entity в shared. рассмотри разные варианты, с интерфейсов и без (поскольку, ты сам выше заметил, что для таких классов, обычно, интерфейсы не делают), но при этом ты также отметил, что доменный слой не должен зависеть ни от чего. как в этой ситуации лучше всего поступить?**

Вопрос о зависимости доменного слоя конкретного модуля от Entity из **shared kernel** — один из классических в DDD и архитектуре с разделением на bounded contexts. Рассмотрим варианты и лучшие практики с учётом найденных материалов и общих принципов.

## Контекст

- **Shared Kernel** — это часть доменной модели, которую несколько bounded contexts (модулей) разделяют для избежания дублирования и обеспечения согласованности терминологии и правил [\[32\]](#) [\[33\]](#) [\[34\]](#) [\[35\]](#).
- Shared Kernel содержит **общие доменные сущности, value objects, доменные события, базовые типы**, но не должен содержать инфраструктурные детали или технические реализации [\[36\]](#) [\[37\]](#).

### Вариант 1: Domain слой модуля зависит напрямую от Entity из shared kernel (без интерфейсов)

- **Плюсы:**
  - Простая и естественная модель: Entity — это доменные объекты с поведением, их сложно и не нужно абстрагировать интерфейсами.
  - Shared kernel — это тоже доменный слой, поэтому зависимость на уровне домена допустима.
- **Минусы:**
  - Если shared kernel меняется, все модули, зависящие от него, могут сломаться — требуется строгая координация изменений и версионирование [\[36\]](#) [\[33\]](#).
  - Нарушается автономия bounded contexts, так как они связаны общей моделью.
- **Рекомендация:**
  - Использовать этот вариант, если контексты тесно связаны и изменения в shared kernel контролируются централизованно (например, через платформенную команду) [\[33\]](#) [\[38\]](#).
  - В shared kernel помещать только **стабильные, хорошо проработанные модели**, которые редко меняются.

### Вариант 2: Domain слой зависит от интерфейсов, Entity в shared kernel — через интерфейсы

- **Плюсы:**
  - Позволяет инвертировать зависимости, уменьшить связанность и повысить гибкость.
  - Можно менять реализации, не затрагивая домен.
- **Минусы:**
  - Для Entity и Value Object интерфейсы обычно избыточны, так как эти классы несут состояние и поведение, а не только контракт.
  - Усложняет модель, добавляя лишний уровень абстракции там, где это не нужно.
- **Рекомендация:**
  - Интерфейсы полезны для сервисов, репозиторий и других абстракций, но для моделей (Entity, VO) — обычно нет [\[37\]](#) [\[39\]](#).

- Если всё же нужна абстракция, лучше выделить интерфейсы для сервисов, а не для самих сущностей.

### Вариант 3: Дублирование моделей в каждом bounded context (без shared kernel)

- **Плюсы:**
  - Полная автономия контекстов, отсутствие жёстких зависимостей.
  - Каждый контекст может развиваться независимо.
- **Минусы:**
  - Дублирование кода и логики, риск расхождения моделей и терминологии.
  - Трудности с согласованностью данных и интеграцией.
- **Рекомендация:**
  - Использовать, если контексты сильно отличаются и не требуют тесной интеграции<sup>[36]</sup>.

### Как лучше поступить?

Подход	Когда использовать	Рекомендации
Зависимость домена от Entity из shared kernel (без интерфейсов)	Когда контексты тесно связаны и есть централизованное управление shared kernel	Помещать в shared kernel только стабильные, согласованные модели; применять семантическое версионирование и строгую координацию изменений <sup>[36]</sup> <sup>[33]</sup> <sup>[38]</sup>
Использование интерфейсов для Entity	Редко оправдано, усложняет модель	Интерфейсы лучше для сервисов и репозиториев, а не для сущностей и value objects <sup>[37]</sup> <sup>[39]</sup>
Дублирование моделей в модулях	Когда контексты автономны и имеют разные требования	При необходимости синхронизации — использовать интеграционные механизмы (события, API)

### Итоговые рекомендации

- **Shared Kernel — это часть доменного слоя, поэтому зависимость доменного слоя модуля от Entity из shared kernel — это нормально и соответствует архитектуре, если shared kernel — чистый доменный код без инфраструктуры.**
- **Не стоит создавать интерфейсы для Entity и Value Object, они сами по себе абстракции бизнес-логики.**
- **Для сервисов и репозиториев интерфейсы обязательны, реализации — в инфраструктуре.**
- **Важна строгая координация и управление изменениями shared kernel, чтобы избежать проблем с совместимостью.**
- Если контексты слабо связаны — лучше дублировать модели и синхронизировать через интеграцию, чем создавать жёсткие зависимости.

## Дополнительные ссылки из поиска

"Shared Kernel acts as a central repository for ubiquitous language elements, domain logic, and data structures common to multiple bounded contexts" [\[32\]](#)

"Minimize shared domain concepts to preserve autonomy; use semantic versioning for shared libraries" [\[36\]](#)

"Shared Kernel requires strong governance and coordination across teams" [\[33\]](#)

"Domain layer should never depend on infrastructure or concrete implementations" [\[37\]](#)

"Shared Kernel contains stable domain models, not infrastructure" [\[35\]](#)

Если хотите, могу помочь с примером организации shared kernel и зависимостей в конкретном проекте.



“Если shared kernel меняется, все модули, зависящие от него, могут сломаться — требуется строгая координация изменений и версионирование” **расскажи подробнее про это**

**Shared Kernel** — это общая часть доменной модели или кода, которую используют два и более bounded contexts (модуля). Такая общая часть помогает избежать дублирования и обеспечивает согласованность в терминологии и бизнес-правилах между контекстами. Однако при этом возникает риск сильной связанности, и именно поэтому требуется **строгая координация изменений и версионирование**.

## Почему нужна координация и версионирование при использовании Shared Kernel

### 1. Общая модель — общие изменения

Поскольку несколько модулей зависят от одних и тех же классов (Entity, Value Object, сервисов), любое изменение в shared kernel может повлиять на все эти модули. Если изменить структуру или поведение общего Entity, это может привести к поломке кода в модулях, которые от него зависят [\[40\]](#) [\[41\]](#) [\[42\]](#).

### 2. Риск сильной связанности

Shared Kernel создаёт тесную связь между bounded contexts, что снижает их автономность. Без координации изменения в одном контексте могут неожиданно нарушить работу другого [\[40\]](#) [\[42\]](#).

### 3. Необходимость согласованного управления изменениями

Чтобы избежать конфликтов и поломок, команды, работающие над разными модулями, должны:

- Согласовывать изменения в Shared Kernel заранее.
- Проводить ревью и тестирование совместимости.

- Использовать семантическое версионирование (semver) для четкого обозначения изменений (патч, минор, мажор) <sup>[43]</sup>.

#### 4. Версионирование и миграция

При внесении изменений, особенно несовместимых (breaking changes), нужно:

- Вести версии shared kernel и управлять зависимостями модулей на конкретные версии.
- Планировать миграцию модулей на новые версии shared kernel.
- Обеспечивать обратную совместимость, если это возможно, или предоставлять адаптеры <sup>[43]</sup>.

#### 5. Инструменты и процессы для поддержки Shared Kernel

- Автоматические тесты, покрывающие shared kernel и интеграцию с модулями.
- Документация API, описание контрактов и изменений.
- Регулярные встречи и коммуникация между командами, ответственные за shared kernel и потребляющие модули <sup>[43]</sup> <sup>[40]</sup>.

### Альтернативы и рекомендации

- **Использовать Shared Kernel только для действительно стабильных и редко меняющихся моделей и правил.** Если модель часто меняется — лучше рассмотреть другие способы интеграции (например, через события или API) <sup>[40]</sup> <sup>[44]</sup> <sup>[43]</sup>.
- **Если контексты разрабатываются разными командами или имеют разные темпы развития, лучше избегать Shared Kernel,** чтобы не создавать распределённый монолит <sup>[44]</sup>.
- **В микросервисной архитектуре** часто рекомендуют выделять общий функционал в отдельный сервис, а не в библиотеку, чтобы изменения централизованно контролировались и не ломали потребителей <sup>[44]</sup>.

### Краткое резюме

Аспект	Описание
Почему нужна координация?	Изменения в Shared Kernel затрагивают все модули, зависящие от него
Что включает координация?	Согласование изменений, ревью, тестирование, версионирование, план миграций
Как управлять изменениями?	Семантическое версионирование, обратная совместимость, адаптеры, документация
Когда использовать Shared Kernel?	При тесной связи контекстов, стабильных общих моделях, тесном сотрудничестве команд
Когда избегать Shared Kernel?	При частых изменениях, разной скорости развития контекстов, разных командах

## Источники из поиска

- Shared Kernel — общая часть модели, требующая координации изменений и версионирования, чтобы избежать проблем с совместимостью<sup>[40] [41] [42]</sup>.
- В микросервисной среде рекомендуют либо выделять общий функционал в отдельный сервис, либо использовать библиотеку с очень стабильным кодом<sup>[44]</sup>.
- Для управления Shared Kernel нужны процессы, тесты, документация и коммуникация между командами<sup>[43]</sup>.

Если хотите, могу помочь с примерами процессов управления изменениями и версионирования Shared Kernel в реальных проектах.

✱

1. [https://en.wikipedia.org/wiki/Domain-Driven\\_Design](https://en.wikipedia.org/wiki/Domain-Driven_Design)
2. <https://stackoverflow.com/questions/57367017/the-meaning-of-entity-in-domain-driven-design>
3. <https://github.com/gayemce/DomainDrivenDesign>
4. <https://www.linkedin.com/pulse/value-objects-ddd-dmitry-neversky-jkijf>
5. <https://www.dremio.com/wiki/value-object/>
6. <https://www.sitepoint.com/ddd-for-rails-developers-part-3-aggregates/>
7. <https://dzone.com/articles/domain-driven-design-aggregate>
8. <https://www.slideshare.net/slideshow/ddd-5-domain-driven-design-repositoriespdf/251899201>
9. <https://blog.stackademic.com/domain-driven-design-and-repository-pattern-f680bf845677?gi=fd3bb628ea84>
10. <https://dev.to/stevescruz/domain-driven-design-ddd-file-structure-4pja>
11. <https://www.ompluscator.com/article/golang/practical-ddd-module/>
12. <https://www.kranio.io/en/blog/de-bueno-a-excelente-en-ddd-comprender-el-patron-de-arquitectura-sugerida-en-domain-driven-design---7-10>
13. <https://itequia.com/en/domain-driven-design-what-is-it-and-how-to-apply-it-in-my-organization/>
14. <https://www.hibit.dev/posts/15/domain-driven-design-layers>
15. <https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/>
16. [https://dev.to/yasmine\\_ddec94f4d4/onion-architecture-in-domain-driven-design-ddd-35gn](https://dev.to/yasmine_ddec94f4d4/onion-architecture-in-domain-driven-design-ddd-35gn)
17. <https://blog.itsjavi.com/target-software-architectures-the-onion-architecture>
18. <https://stackoverflow.com/questions/3399576/what-is-the-relationship-between-ddd-and-the-onion-architecture>
19. <https://softwareengineering.stackexchange.com/questions/406644/where-to-place-the-value-object-code-that-is-shared-by-more-than-one-aggregate-r>
20. <https://deviq.com/domain-driven-design/shared-kernel/>
21. <https://www.kranio.io/en/blog/de-bueno-a-excelente-en-ddd-entendiendo-los-entities-y-value-objects-en-domain-driven-design---2-10>
22. <https://dev.to/ielgohary/domain-driven-design-entities-value-objects-and-services-chapter-51-22cm>

23. <https://github.com/pirocorp/Object-Oriented-Design/blob/main/13. DDD/Elements of a Domain Model Value Objects And Services.md>
24. <https://softwareengineering.stackexchange.com/questions/429703/valueobjects-exposing-outside-do-main-layer>
25. <https://tproger.ru/articles/domain-driven-design-davajte-ne-budem-uslozhnyat>
26. <https://habr.com/ru/articles/892250/>
27. <https://habr.com/ru/articles/718916/>
28. <https://habr.com/ru/articles/427739/>
29. <https://habr.com/ru/articles/718916/>
30. <https://ru.stackoverflow.com/questions/1081115/Где-на-каком-уровне-должны-находиться-перечисления-в-многоуровневой-архитектуре>
31. <https://github.com/Max-Starling/Notes/blob/master/Architecture-Design.md>
32. <https://deviq.com/domain-driven-design/shared-kernel/>
33. <https://buildsimple.substack.com/p/strategic-ddd-the-balancing-act-of>
34. <https://stackoverflow.com/questions/72676223/ddd-how-to-use-shared-kernel-with-multiple-bounded-contexts>
35. <https://herbertograca.com/2018/07/07/more-than-concentric-layers/>
36. <https://dev.to/devaaai/managing-shared-libraries-in-hexagonal-architecture-and-domain-driven-4oi6>
37. <https://stackoverflow.com/questions/4470977/how-to-implement-a-shared-kernel-ddd-in-net-properly>
38. <https://www.youtube.com/watch?v=PiNJ2CaC3Cw>
39. <https://habr.com/ru/articles/718916/>
40. <https://habr.com/ru/articles/892250/>
41. <https://ddd-practitioners.com/home/glossary/bounded-context/bounded-context-relationship/shared-kernel/>
42. <https://tproger.ru/articles/domain-driven-design-davajte-ne-budem-uslozhnyat>
43. <https://buildsimple.substack.com/p/strategic-ddd-the-balancing-act-of>
44. <https://habr.com/ru/companies/oleg-bunin/articles/661129/>