

у меня есть вот такой проект: <https://github.com/sunday-research/sunday-research-bot>
контрибьюторы, не погружённые в контекст, задают вопросы про структуру и устройство проекта, про назначение сущностей.
пожалуйста, проанализируй проект и опиши:

1. концепцию структуры
2. типы и назначение классов
3. один пример flow

Проект Sunday Research Bot — это Telegram-бот, изначально созданный для отправки .gif в чат каждую пятницу, но с более широкой целью — предоставить пользователям чата дополнительные удобные функции (например, заметки, карма, стикеры, интеграция с ChatGPT и др.)^[1].

1. Концепция структуры проекта

- **Domain** — содержит бизнес-логику. В этой директории располагаются основные классы, управляющие поведением бота.
- **Controller** — предназначен для HTTP API (в том числе веб-хуков), сейчас пуст.
- **Command** — содержит CLI-команды, через которые запускаются процессы, например, обновление бота.
- **Bundle** — модули, которые могут быть переиспользованы в Domain.

Иерархия классов в Domain:

- `<Class>Manager` — верхнеуровневый класс, реализующий бизнес-логику. Вызывает сервисы.
- `<Class>Service` — классы-«кирпичики» для реализации отдельных функций (например, работа с пользователями, подписками). Используют репозитории.
- `<Class>Repository` — классы для работы с внешними ресурсами или хранилищами (БД, Redis, Telegram API и т.д.). Обязательно реализуют интерфейсы.

Вызов бизнес-логики обычно происходит через менеджер, который обеспечивает правильный порядок вызовов:

CLI/Controller → Manager → Service → Repository

2. Типы и назначение классов

Тип класса	Назначение
Manager	Основной класс бизнес-логики, координирует работу сервисов, реализует сценарии и логику.
Service	Выполняет отдельные функции, например, работу с пользователями, подписками, сообщениями.
Repository	Абстракция над источниками данных: БД, API Telegram, Redis и др.
Command	CLI-команды для запуска процессов, например, слушатель обновлений Telegram.
Controller	HTTP API (веб-хуки), пока не реализован.
Bundle	Модули, расширяющие функциональность, могут использоваться в Domain.

3. Пример flow (пример работы отправки .gif в пятницу)

1. Пользователь или система запускает CLI-команду, например, `app:bot-update-listener`, которая запускает цикл опроса Telegram API.
2. Обновления (сообщения, команды) поступают в менеджер, например, `GifManager`.
3. Менеджер вызывает сервисы, которые реализуют логику выбора и отправки gif (например, `GifService`).
4. Сервис через репозиторий (например, `TelegramRepository`) отправляет сообщение с gif в чат.
5. Если команда сложная и многошаговая, менеджер может использовать сессию пользователя, чтобы определить следующий шаг взаимодействия.

Таким образом, flow выглядит так:

CLI command → GifManager (Manager) → GifService (Service) → TelegramRepository (Repository) → Telegram API → Отправка gif

Итог

Sunday Research Bot построен по классической архитектуре с разделением ответственности: менеджеры управляют бизнес-логикой, сервисы — отдельными функциями, репозитории — доступом к данным и API. Это облегчает поддержку и расширение функционала, например, добавление новых команд, интеграций и сценариев взаимодействия.

В ближайших планах — разработка механизма многошаговых сценариев и планировщика событий для автоматических сообщений, что позволит существенно расширить возможности бота^[1].

чуть раньше мы обсуждали DDD: <https://www.plexity.ai/search/rasskazhi-kakie-v-ddd-est-klas-C7ybefVBQgqxCW1JxnFVBA> **рассмотри текущую архитектуру в сравнении с DDD: какие плюсы и минусы ты видишь?**

Сравнивая текущую архитектуру Sunday Research Bot с принципами DDD (Domain Driven Design), можно выделить следующие **плюсы и минусы**.

Плюсы текущей архитектуры с точки зрения DDD

- **Чёткое разделение слоёв:**
В проекте выделены менеджеры (Manager) как верхний слой бизнес-логики, сервисы (Service) для реализации функциональных блоков и репозитории (Repository) для доступа к данным и внешним API. Это соответствует идее DDD о разделении домена, приложения и инфраструктуры^[2].
- **Менеджеры как агрегаты бизнес-логики:**
Менеджеры концентрируют сценарии и бизнес-правила, что похоже на роль агрегатов в DDD — они управляют состоянием и обеспечивают целостность бизнес-процессов.
- **Репозитории изолируют доступ к данным:**
В проекте репозитории реализуют интерфейсы и скрывают детали работы с БД, Telegram API, Redis и т.п., что соответствует DDD-принципу изоляции инфраструктуры от домена^[2] ^[3].
- **Domain слой изолирован от фреймворка и внешних зависимостей:**
Бизнес-логика реализована в Domain (Manager, Service), которая не зависит напрямую от контроллеров или CLI, что соответствует DDD-идее о независимости домена от технических деталей^[2].

Минусы и области для улучшения с точки зрения DDD

- **Отсутствие явных сущностей (Entities) и Value Objects:**
В описании проекта и структуре не выделены отдельные доменные сущности и объекты-значения, которые являются ключевыми строительными блоками DDD для моделирования предметной области^[4] ^[2]. Менеджеры и сервисы скорее реализуют логику, но не моделируют предметную область через богатые объекты.
- **Неявное использование агрегатов и ограниченных контекстов (Bounded Contexts):**
В проекте нет явного разделения на ограниченные контексты, что в DDD помогает управлять сложностью и согласованностью модели. Все бизнес-логика сосредоточена в менеджерах и сервисах без разграничения по контекстам.
- **Отсутствие слоёв Application и Domain в классическом понимании DDD:**
Менеджеры ближе к Application Service, а сервисы — к Domain Service, но нет чёткого

выделения доменных сущностей с поведением. Это может снижать выразительность модели и усложнять поддержку бизнес-правил.

- **Валидация и бизнес-правила не централизованы в домене:**
В DDD важна централизованная валидация и инварианты в сущностях и агрегатах, чтобы бизнес-правила были защищены от нарушения. В текущем проекте скорее бизнес-логика распределена по менеджерам и сервисам, что может привести к дублированию и рассеиванию правил.

Итог

Аспект	Текущая архитектура	DDD идеал	Комментарий
Разделение слоёв	Есть Manager → Service → Repository	Domain, Application, Infrastructure	Хорошее разделение, близко к DDD
Моделирование домена	Логика в менеджерах и сервисах	Богатые сущности и Value Objects	Нет явных сущностей — упрощение
Ограниченные контексты	Отсутствуют	Обязательны для масштабируемости	Можно улучшить
Валидация и инварианты	В сервисах/менеджерах	В доменных сущностях	Требует централизации
Зависимости	Domain изолирован от инфраструктуры	Domain полностью независим	Соответствует

Таким образом, архитектура Sunday Research Bot **близка к DDD по структуре слоёв и изоляции инфраструктуры**, что является сильным плюсом. Однако для полного соответствия DDD стоит **ввести явные доменные сущности и объекты-значения, разграничить ограниченные контексты и централизовать бизнес-правила и валидацию в домене**. Это повысит выразительность модели, упростит поддержку и масштабирование проекта в будущем.



<q>Валидация и бизнес-правила не централизованы в домене:

В DDD важна централизованная валидация и инварианты в сущностях и агрегатах, чтобы бизнес-правила были защищены от нарушения. В текущем проекте скорее бизнес-логика распределена по менеджерам и сервисам, что может привести к дублированию и рассеиванию правил.</q> меня очень заинтересовало это замечание. расскажи на примере моего проекта, что это значит? с приведением реального кода.

Замечание о **централизованной валидации и бизнес-правилах в домене** означает, что в DDD (Domain-Driven Design) основная ответственность за поддержание корректного состояния данных и соблюдение всех бизнес-правил лежит на самих **доменных сущностях (Entities)** и **агрегатах (Aggregates)**, а не распределена по менеджерам или

сервисам. Это гарантирует, что объект всегда находится в валидном состоянии, и никакое действие не может нарушить его инварианты (неизменные бизнес-правила) ^[5] ^[6].

В текущем проекте Sunday Research Bot бизнес-логика и валидация сосредоточены в слоях Manager и Service. Это может привести к следующим проблемам:

- **Рассеивание правил:** Одни и те же правила могут проверяться в разных местах или быть неявно связаны с вызовами методов, что затрудняет их изменение и поддержку ^[5].
- **Возможность создания невалидных объектов:** Если объекты данных (например, User, Subscription) не инкапсулируют свои правила, они могут быть созданы в невалидном состоянии, а затем переданы по системе, что может привести к ошибкам в дальнейшем ^[6].
- **Дублирование логики:** Одно и то же бизнес-правило может быть реализовано в нескольких сервисах или менеджерах, что приводит к избыточности кода и усложняет его поддержку.

Что это значит на примере Sunday Research Bot?

Предположим, у нас есть сущность User (пользователь) и правило, что **имя пользователя не может быть пустым и должно быть уникальным**.

Текущий подход (предполагаемый):

В вашей архитектуре валидация может происходить на разных уровнях:

1. **В контроллере/команде (на входе):** CLI-команда или будущий контроллер получает данные и делает первичную проверку на наличие имени, например, перед тем как передать его в менеджер.
2. **В UserManager или UserService:** Допустим, при создании нового пользователя, UserManager или UserService проверяет уникальность имени, прежде чем сохранить его через UserRepository.

```
// Пример упрощенного кода в UserManager
class UserManager
{
    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function createUser(string $username, string $telegramId): User
    {
        // Валидация 1: Проверка на пустоту имени
        if (empty($username)) {
            throw new \InvalidArgumentException("Username cannot be empty.");
        }
    }
}
```

```

        // Валидация 2: Проверка уникальности имени (бизнес-правило)
        if ($this->userRepository->findByUsername($username) !== null) {
            throw new \InvalidArgumentException("Username already exists.");
        }

        $user = new User($username, $telegramId); // Предполагаем, что User — это просто структура данных
        $this->userRepository->save($user);
        return $user;
    }
}

// Пример User - это просто структура данных
class User
{
    public string $username;
    public string $telegramId;

    public function __construct(string $username, string $telegramId)
    {
        $this->username = $username;
        $this->telegramId = $telegramId;
    }
}

```

В этом примере:

- Валидация на пустоту и уникальность имени пользователя выполняется в `UserManager`. Если бы существовал другой способ создания пользователя (например, через другую команду или сервис), эту валидацию пришлось бы дублировать или обеспечить, чтобы все пути проходили через `UserManager`.
- `User` — это "анемичная" сущность, она не содержит логики, а только данные.

Подход DDD с централизованной валидацией в домене:

В DDD эти правила инкапсулируются внутри самой доменной сущности `User` или в **Value Object** для имени пользователя. Цель — гарантировать, что объект `User` всегда находится в валидном состоянии с момента его создания^[6].

1. **Использование Value Object для имени:** Если имя имеет свои правила (не пустое, возможно, определённый формат), его можно выделить в отдельный Value Object `Username`.

```

// Value Object для имени пользователя
class Username
{
    private string $value;

    public function __construct(string $value)
    {
        if (empty($value)) {
            throw new \InvalidArgumentException("Username cannot be empty.");
        }
        // Здесь могут быть другие правила форматирования имени, например, длина, раз
        $this->value = $value;
    }
}

```

```

    }

    public function getValue(): string
    {
        return $this->value;
    }

    public function equals(Username $other): bool
    {
        return $this->value === $other->value;
    }
}

```

2. **Сущность User с инкапсулированными правилами:** Сущность User теперь содержит инварианты. Валидация на уникальность имени пользователя — это бизнес-правило, которое касается коллекции пользователей, а не отдельного пользователя, поэтому она может быть выполнена до создания User в "Фабрике" или в доменном сервисе, но сам User не позволит создать себя с невалидным именем.

```

// Доменная сущность User
class User
{
    private Username $username; // Использование Value Object
    private string $telegramId;
    // ... другие свойства

    // Конструктор, обеспечивающий валидность сущности при создании
    public function __construct(Username $username, string $telegramId)
    {
        // Уникальность имени - это скорее правило агрегата/репозитория,
        // но сам UserName уже гарантирует, что он не пуст
        $this->username = $username;
        $this->telegramId = $telegramId;
        // Здесь могут быть другие инварианты, касающиеся внутренней консистентности
    }

    public function getUsername(): Username
    {
        return $this->username;
    }

    public function getTelegramId(): string
    {
        return $this->telegramId;
    }

    // Методы, изменяющие состояние User, также должны гарантировать инварианты
    // public function changeUsername(Username $newUsername): void
    // {
    //     $this->username = $newUsername;
    // }
}

// Пример использования в Domain Service или Application Service (например, UserManager
class UserManager

```

```

{
    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function createUser(string $usernameValue, string $telegramId): User
    {
        // Шаг 1: Создаем Value Object. Здесь происходит первая валидация (не пустое)
        $username = new Username($usernameValue);

        // Шаг 2: Бизнес-правило уникальности имени пользователя
        if ($this->userRepository->findByUsername($username) !== null) {
            throw new \InvalidArgumentException("Username already exists.");
        }

        // Шаг 3: Создаем сущность User. Она уже будет валидной.
        $user = new User($username, $telegramId);
        $this->userRepository->save($user);
        return $user;
    }
}

```

Преимущества DDD подхода в данном случае:

- **Гарантия инвариантов:** Username всегда содержит непустое значение. User всегда создаётся с валидным Username и telegramId. Невозможно создать невалидный User с точки зрения его внутренних правил.
- **Устранение дублирования:** Правило "имя не пустое" находится в одном месте — в конструкторе Username. Любое использование Username будет гарантировать его валидность.
- **Чистота доменной модели:** Бизнес-правила инкапсулированы там, где они логически принадлежат — в доменных объектах.
- **Более понятная бизнес-логика:** Методы UserManager теперь фокусируются на оркестрации бизнес-процессов (например, проверка уникальности перед созданием), а не на деталях валидации каждого поля.

Это не означает, что внешняя валидация (например, в контроллерах или CLI-командах) не нужна. Она по-прежнему важна для быстрой отбраковки некорректных запросов и предоставления обратной связи пользователю до того, как они достигнут доменного слоя^[5]. Однако доменный слой является последней линией защиты, гарантирующей, что бизнес-правила не будут нарушены^{[5] [6]}.

а можешь теперь в контексте примера выше описать, как бы ты сюда добавил агрегаты? мне нужно понять на примере кода их назначение и структуру

В контексте вашего проекта Sunday Research Bot и примера с сущностью `User` добавление **агрегатов** поможет структурировать доменную модель так, чтобы обеспечить целостность и согласованность данных, а также централизовать бизнес-правила.

Что такое агрегат и зачем он нужен?

Агрегат — это **кластер сущностей и объектов-значений**, который воспринимается системой как единое целое. В агрегате есть **корень агрегата (Aggregate Root)** — единственная точка входа для изменения состояния всего агрегата. Вся бизнес-логика и инварианты агрегата реализуются в корне агрегата.

Это гарантирует, что изменения внутри агрегата происходят **только через корень агрегата**, что обеспечивает согласованность и защиту от нарушения правил.

Как это применить к вашему примеру с `User`

1. Определяем агрегат

- В вашем случае агрегатом может быть сущность `User` вместе с сопутствующими объектами (например, настройками пользователя, ролями, подписками и т.п.).
- `User` — это корень агрегата, он контролирует все изменения внутри себя.

2. Структура кода с агрегатом

```
// Value Object для имени пользователя
class Username
{
    private string $value;

    public function __construct(string $value)
    {
        if (empty($value)) {
            throw new \InvalidArgumentException("Username cannot be empty.");
        }
        $this->value = $value;
    }

    public function getValue(): string
    {
        return $this->value;
    }

    public function equals(Username $other): bool
    {

```

```

        return $this->value === $other->value;
    }
}

// Сущность User – корень агрегата
class User
{
    private string $id; // уникальный идентификатор
    private Username $username;
    private string $telegramId;
    private array $subscriptions; // дочерние сущности или объекты значения

    public function __construct(string $id, Username $username, string $telegramId)
    {
        $this->id = $id;
        $this->username = $username;
        $this->telegramId = $telegramId;
        $this->subscriptions = [];
    }

    // Бизнес-правило: смена имени пользователя
    public function changeUsername(Username $newUsername, UserRepository $userRepository)
    {
        // Проверяем уникальность имени в рамках агрегата
        if ($userRepository->findByUsername($newUsername) !== null) {
            throw new \InvalidArgumentException("Username already exists.");
        }
        $this->username = $newUsername;
    }

    // Добавление подписки – пример дочерней сущности
    public function addSubscription(Subscription $subscription): void
    {
        // Здесь можно реализовать инварианты, например, не добавлять дубликаты
        $this->subscriptions[] = $subscription;
    }

    // Геттеры
    public function getId(): string
    {
        return $this->id;
    }

    public function getUsername(): Username
    {
        return $this->username;
    }

    public function getTelegramId(): string
    {
        return $this->telegramId;
    }

    public function getSubscriptions(): array
    {
        return $this->subscriptions;
    }
}

```

```

    }
}

// Пример дочерней сущности Subscription
class Subscription
{
    private string $topic;
    private \DateTimeImmutable $createdAt;

    public function __construct(string $topic)
    {
        $this->topic = $topic;
        $this->createdAt = new \DateTimeImmutable();
    }

    public function getTopic(): string
    {
        return $this->topic;
    }

    public function getCreatedAt(): \DateTimeImmutable
    {
        return $this->createdAt;
    }
}

```

3. Работа с агрегатом через репозиторий и менеджер

```

class UserManager
{
    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function registerUser(string $id, string $usernameValue, string $telegramId):
    {
        $username = new Username($usernameValue);

        if ($this->userRepository->findByUsername($username) !== null) {
            throw new \InvalidArgumentException("Username already exists.");
        }

        $user = new User($id, $username, $telegramId);
        $this->userRepository->save($user);

        return $user;
    }

    public function changeUserUsername(string $userId, string $newUsernameValue): void
    {
        $user = $this->userRepository->findById($userId);
        if (!$user) {

```

```

        throw new \RuntimeException("User not found");
    }

    $newUsername = new Username($newUsernameValue);
    $user->changeUsername($newUsername, $this->userRepository);

    $this->userRepository->save($user);
}

public function addUserSubscription(string $userId, string $topic): void
{
    $user = $this->userRepository->findById($userId);
    if (!$user) {
        throw new \RuntimeException("User not found");
    }

    $subscription = new Subscription($topic);
    $user->addSubscription($subscription);

    $this->userRepository->save($user);
}
}

```

Ключевые моменты и преимущества такого подхода

- **Корень агрегата (User) — единственная точка входа для изменения состояния:** все операции с пользователем (смена имени, добавление подписок и т.п.) происходят через методы User.
- **Инварианты и бизнес-правила инкапсулированы в корне агрегата и связанных сущностях:** например, проверка уникальности имени происходит в методе changeUsername через репозиторий.
- **Вся внутренняя структура агрегата (подписки и другие дочерние сущности) скрыта от внешнего мира:** нельзя напрямую изменить подписки без обращения к корню агрегата.
- **Обеспечение согласованности данных:** изменения в агрегате происходят в рамках одной транзакции (логически), что снижает риск нарушения бизнес-правил.
- **Упрощение поддержки и расширения:** бизнес-правила сосредоточены в одном месте, легче понять и изменить логику.

Итог

В вашем проекте агрегаты помогут:

- Сделать доменную модель более выразительной и защищённой от ошибок.
- Централизовать бизнес-логику и валидацию.
- Обеспечить целостность и согласованность данных.
- Упростить управление сложными сценариями, связанными с пользователями и их состояниями.

Если интересно, могу помочь с примером реализации репозитория и интеграцией агрегатов в текущую архитектуру вашего проекта.

*
**

1. <https://github.com/sunday-research/sunday-research-bot>
2. <https://habr.com/ru/articles/718916/>
3. https://www.reddit.com/r/DomainDrivenDesign/comments/182tk09/clean_archi_vs_ddd_where_the_persistence_logic/?tl=ru
4. <https://www.perplexity.ai/search/rasskazhi-kakie-v-ddd-est-klas-C7ybefVBQgqxCW1JxnFVBA>
5. <https://habr.com/ru/articles/902420/>
6. <https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations>