

DDD

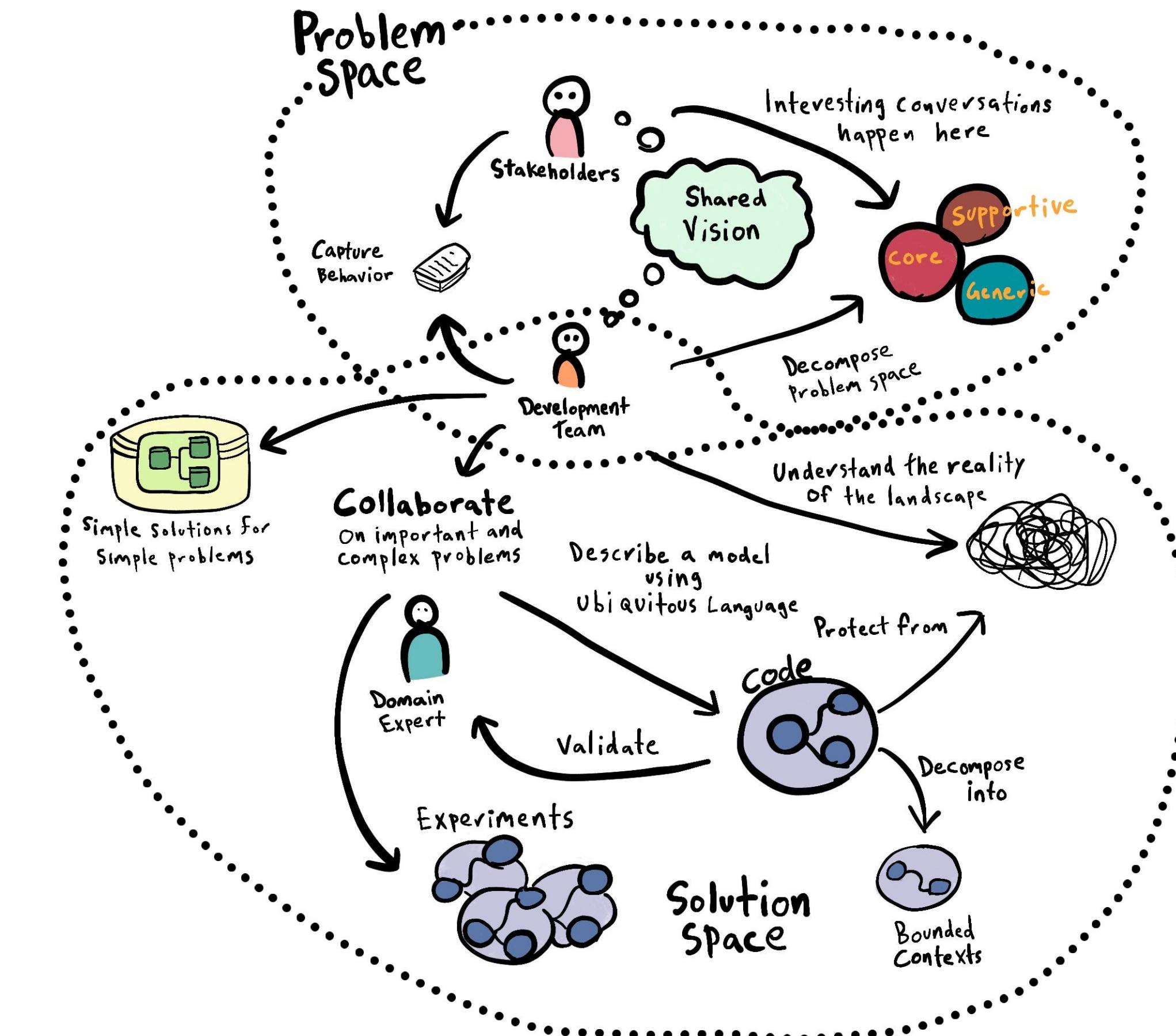
Предметно-ориентированное проектирование

Неделя 8

Определение

DDD (Domain Driven Design) – подход к проектированию, основанный на модели предметной области.

Предметная область – область человеческой деятельности, определяемая общим набором понятий и отношений, используемых совместно



Основные концепции DDD

Единый язык (Ubiquitous Language) – это общий язык, используемый как разработчиками, так и экспертами в предметной области. Он помогает устранить недопонимание между техническими и бизнес-специалистами.

Пример:

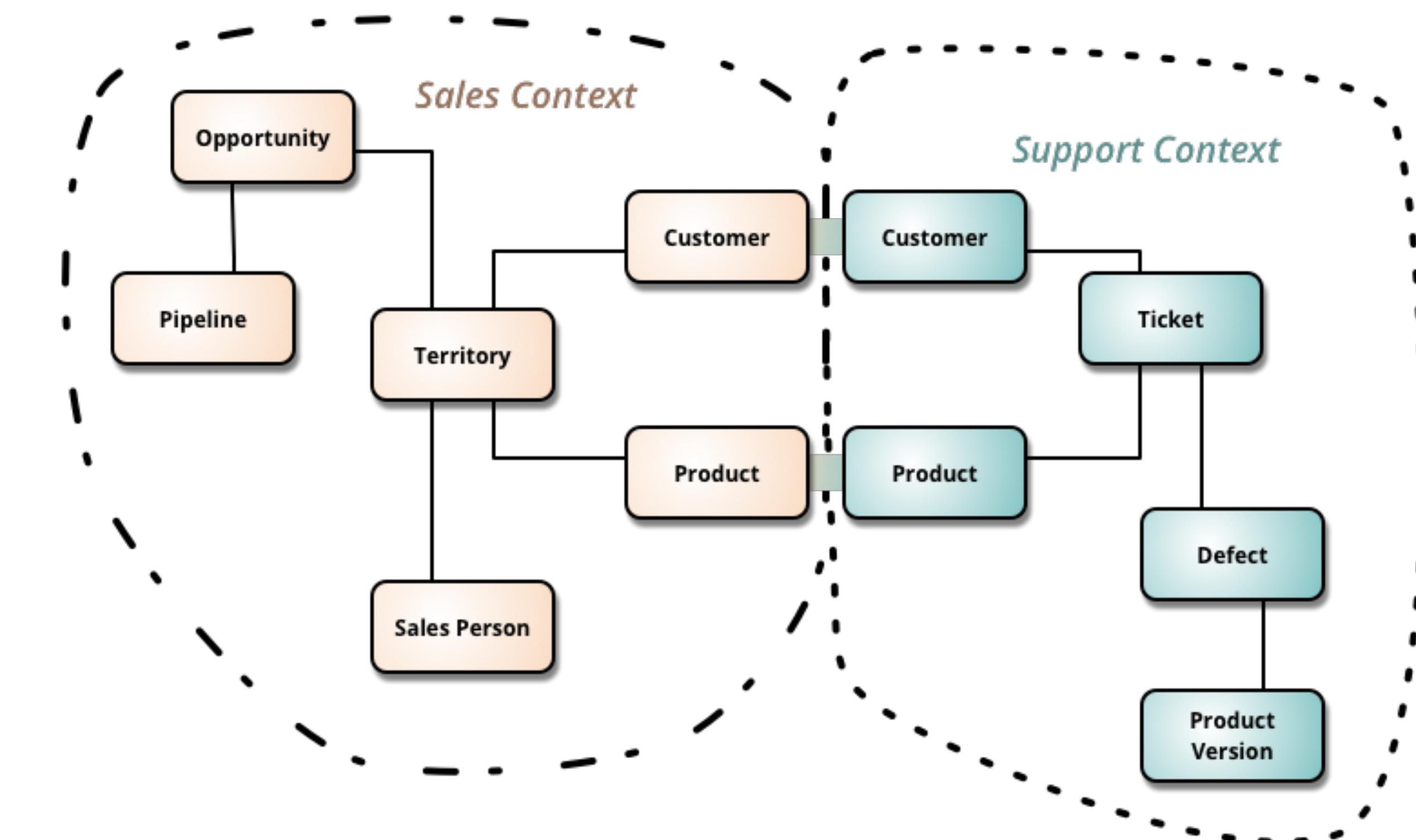
Вместо использования технических терминов вроде “Entity” или “Repository”, мы используем термины из бизнес-домена, такие как “Покупатель”, “Автомобиль”, “Продажа”.

Основные концепции DDD

Ограниченный контекст (Bounded Context) – это граница, внутри которой определенная модель имеет конкретное значение. Разные контексты могут иметь разные модели для одних и тех же понятий.

Пример:

Понятие “Клиент” в контексте продаж может отличаться от понятия “Клиент” в контексте бухгалтерии.



Строительные блоки DDD. Value-objects

Объекты-значения (Value Objects) – это объекты, которые не имеют идентичности и полностью определяются своими атрибутами.



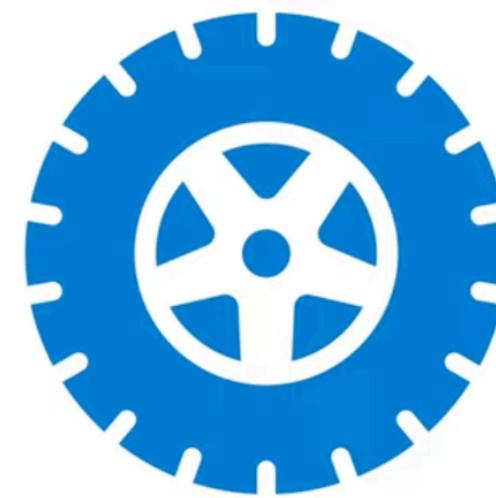
Red: 176
Green: 83
Blue: 167



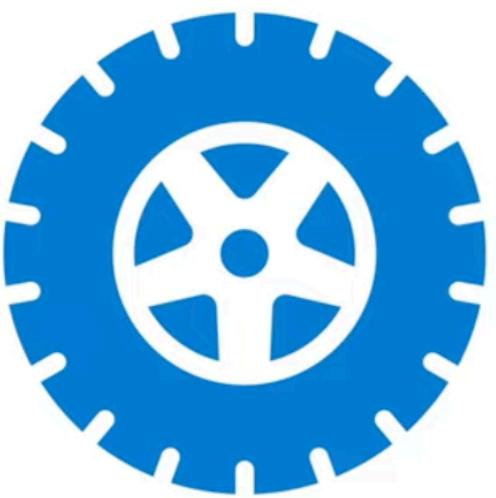
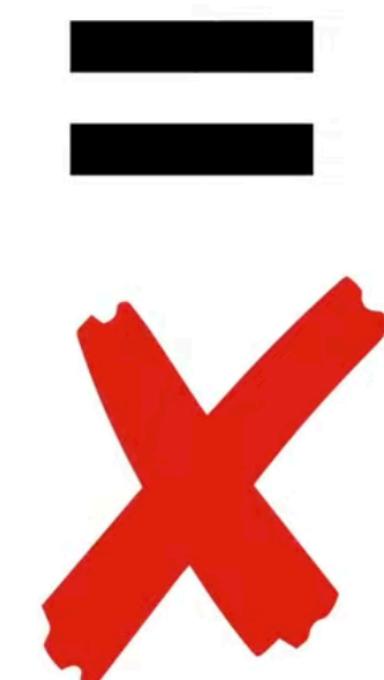
Red: 176
Green: 83
Blue: 167

Строительные блоки DDD. Сущности

Сущности (Entities) – это объекты, которые имеют идентичность, не зависящую от их атрибутов. Две сущности могут быть равны, даже если их атрибуты различаются.



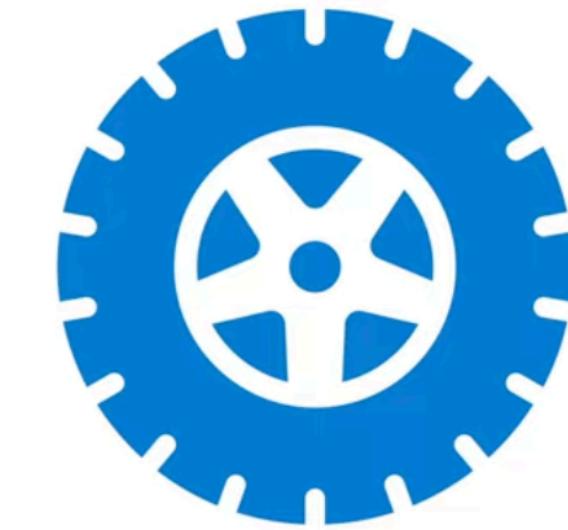
Size: 450mm
Type: Snow
Air: 29psi



Size: 450mm
Type: Snow
Air: 29psi

Строительные блоки DDD. Domain Events

События предметной области (Domain Events) – это объекты, которые фиксируют что-то, что произошло в предметной области.



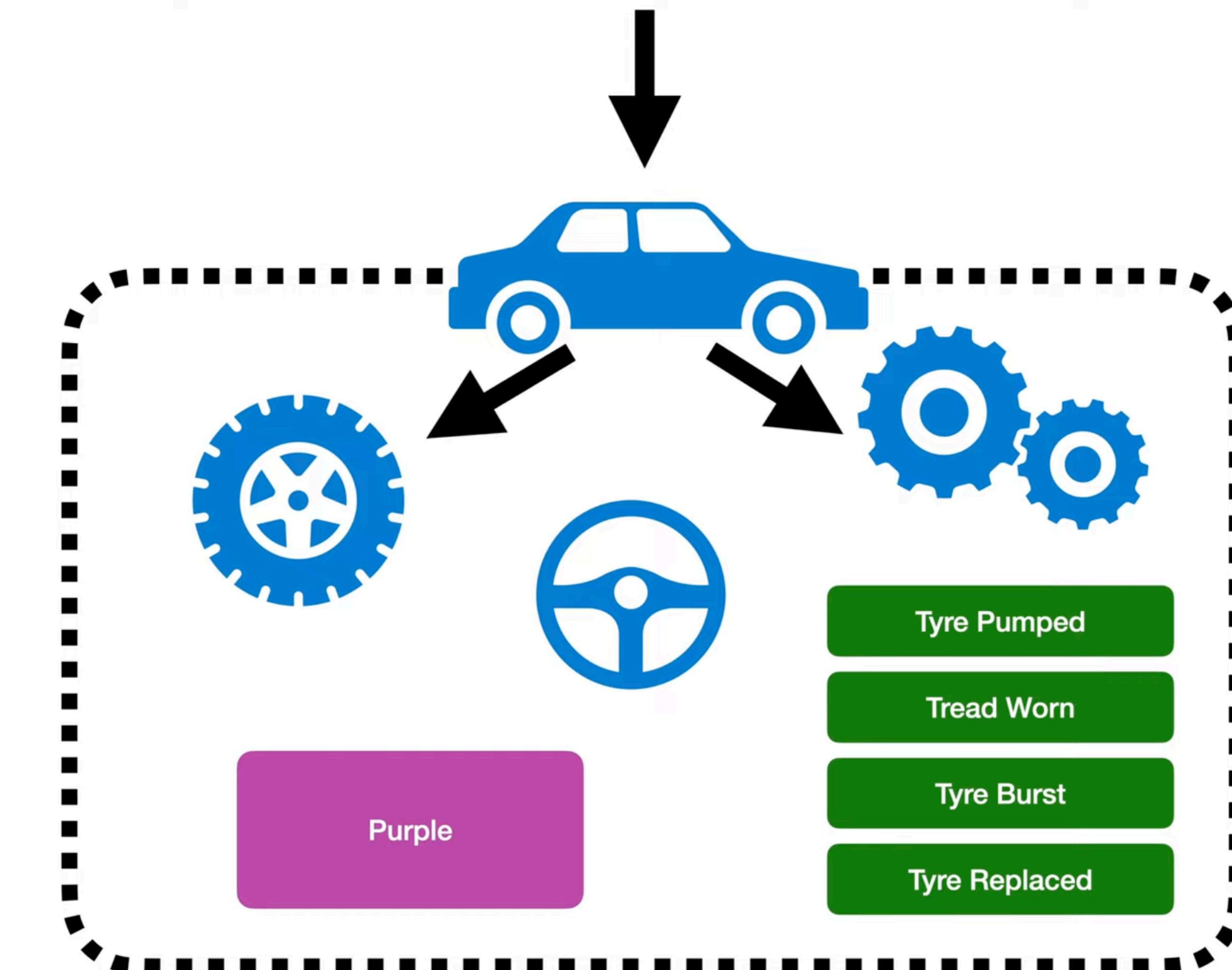
Tyre Pumped

Size: 450mm
Type: Snow
Air: 35psi
Id: 1234

Id: 1234
OldAir: 29psi
NewAir: 35psi

Строительные блоки DDD. Агрегаты

Агрегаты (Aggregates) – это кластер объектов, которые рассматриваются как единое целое с точки зрения изменения данных. У каждого агрегата есть корень (Aggregate Root), через который осуществляется доступ к другим объектам агрегата.



Строительные блоки DDD. Репозитории

Репозитории (*Repositories*) – это механизм для инкапсуляции хранения, поиска и извлечения объектов модели.

Для чего?

1. Инкапсуляция доступа к данным

Репозиторий скрывает детали работы с базой данных (SQL-запросы, ORM, кеширование) и предоставляет удобные методы для работы с сущностями.

2. Отделение доменной логики от инфраструктуры

Позволяет доменному слою не зависеть от конкретной технологии хранения данных (EF Core, Dapper, MongoDB и т. д.).

3. Обеспечение работы с агрегатами

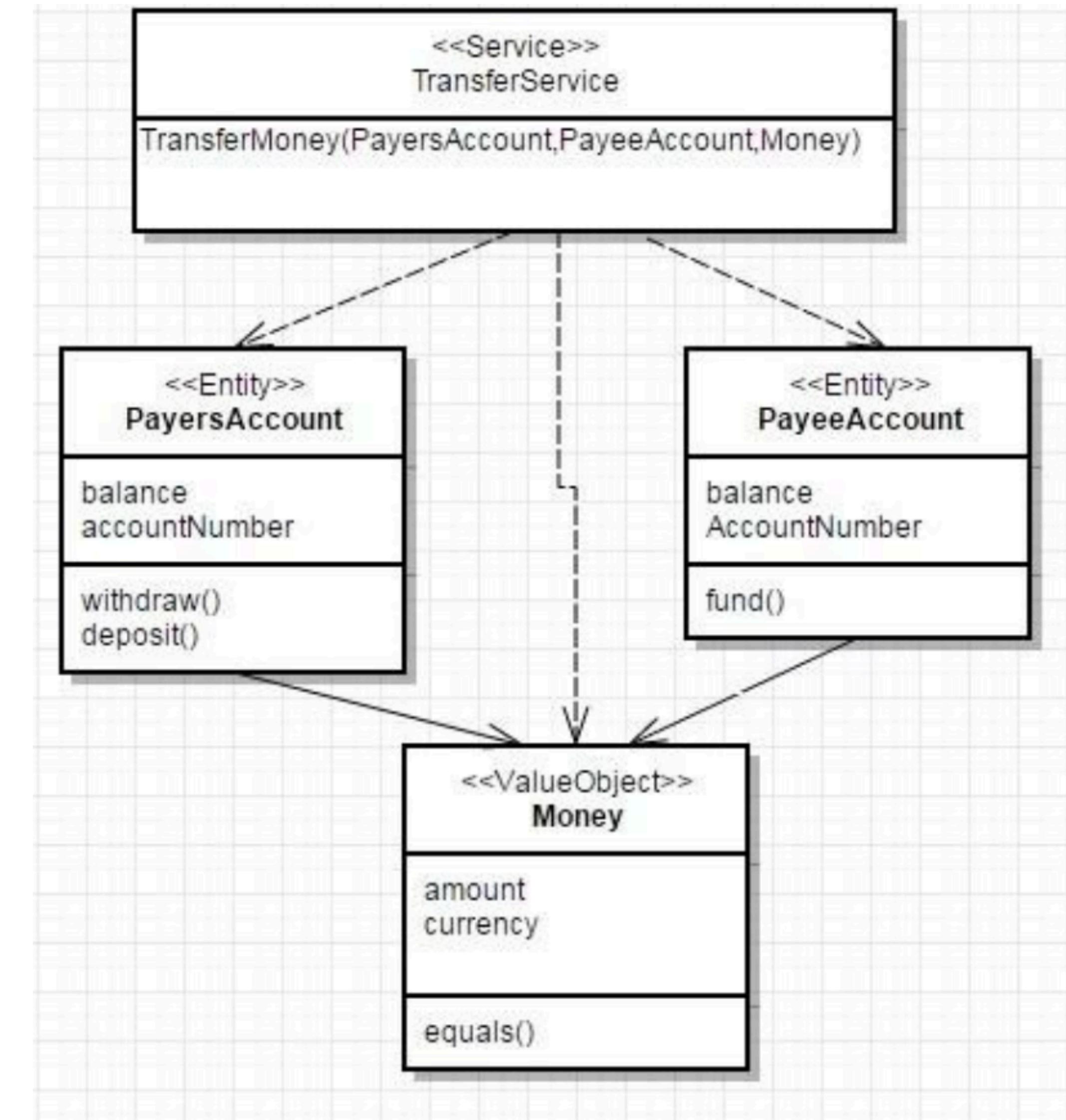
В DDD есть понятие **агрегатов (Aggregates)** — групп объектов, которые изменяются и сохраняются как единое целое. Репозиторий управляет агрегатами, гарантируя целостность данных.

4. Упрощение тестирования

Благодаря репозиториям можно легко подменять реальное хранилище мок-объектами (например, `InMemoryRepository`), что делает тесты независимыми от БД.

Строительные блоки DDD. Сервисы

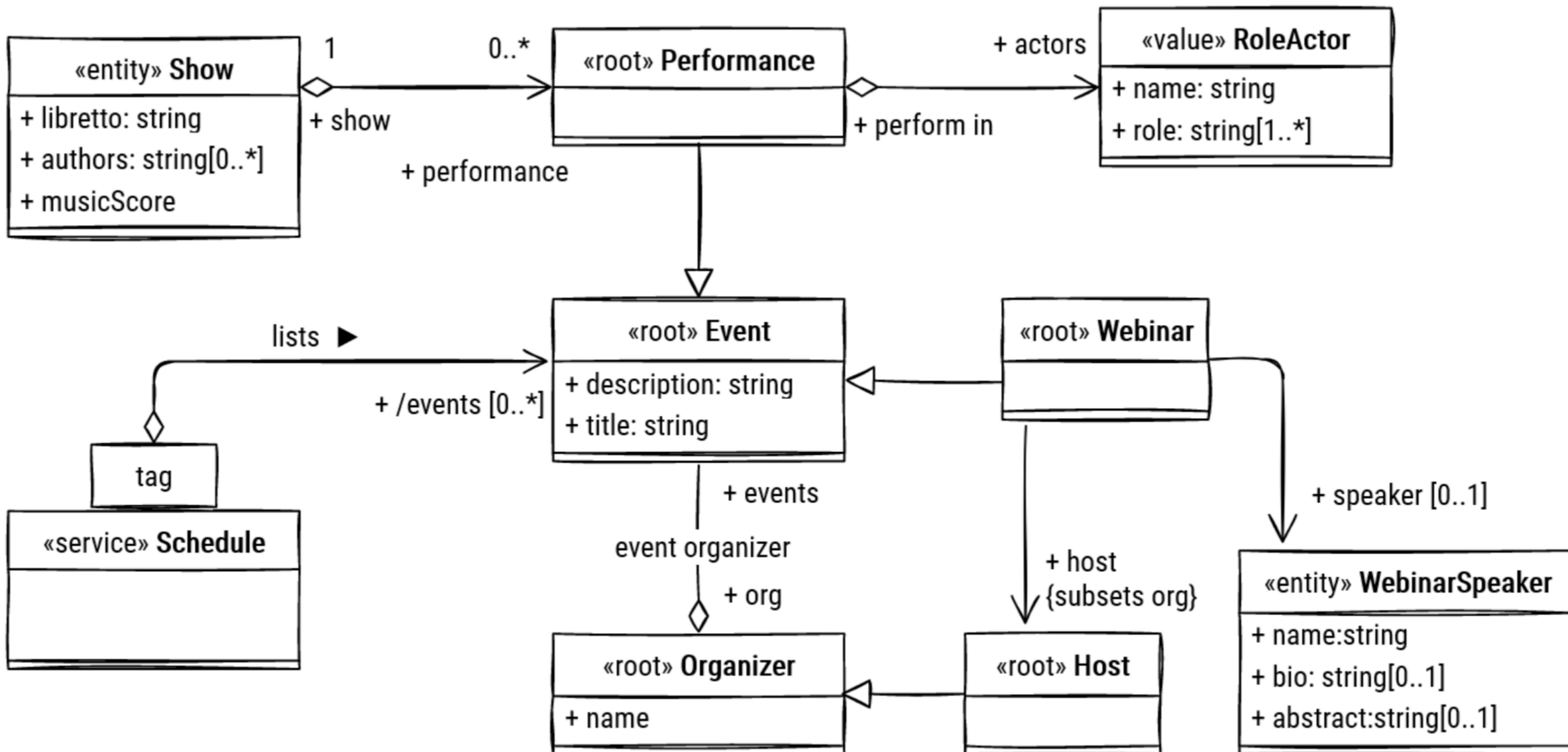
Сервисы предметной области (Domain Services) – это операции, которые не принадлежат ни к одной сущности или объекту-значению, но являются частью модели предметной области.



Строительные блоки DDD. Фабрики

Фабрики (Factories) – это объекты, отвечающие за создание сложных объектов или агрегатов.

Пример для билетной кассы с DDD стереотипами



Анализ текущего проекта

Где мы соответствуем принципам DDD?

1. Использование единого языка:

- Наши классы и методы используют термины из предметной области: `Car`, `Customer`, `CarService`, `HseCarService`.
- Комментарии к коду также используют термины из предметной области.

2. Четкое разделение на модули:

- Проект разделен на модули по функциональности: `Cars`, `Customers`, `Sales`, `Reports`, `Accounting`.
- Каждый модуль отвечает за свою часть функциональности.

3. Использование фабрик:

- `PedalCarFactory` и `HandCarFactory` отвечают за создание автомобилей разных типов.

4. Использование сервисов предметной области:

- `HseCarService` отвечает за бизнес-логику продажи автомобилей.

5. Инкапсуляция:

- Сущности `Car` и `Customer` инкапсулируют свое состояние и поведение.
- Некоторые свойства доступны только для чтения, что защищает инварианты.

Где мы противоречим принципам DDD?

1. Отсутствие явных агрегатов:

- В проекте нет явного определения агрегатов и их корней.
- Отношения между сущностями не всегда четко определены.

2. Смешение ответственостей:

- **CarService** выполняет функции как репозитория, так и сервиса предметной области.
- **CustomersStorage** также смешивает ответственности.

3. Отсутствие объектов-значений:

- В проекте не используются объекты-значения, хотя они могли бы упростить модель.

4. Недостаточное использование инвариантов:

- Бизнес-правила не всегда явно выражены в коде.
- Некоторые инварианты могут быть нарушены из-за отсутствия проверок.

5. Отсутствие событий предметной области:

- В проекте не используются события предметной области, что затрудняет отслеживание изменений.

6. Фасад как антипаттерн:

- **CarShop** является фасадом, который скрывает сложность системы, но также скрывает и модель предметной области.
- Фасад может противоречить принципам DDD, если он скрывает важные бизнес-концепции.

Превращаем проект в DDD-compliant

1. Ввести объекты-значения

- **EngineSpecification**: Объект-значение для хранения характеристик двигателя.
- **CustomerCapabilities**: Объект-значение для хранения возможностей покупателя (сила ног, сила рук).

Создание объекта-значения CustomerCapabilities

```
// Создаем новый файл CustomerCapabilities.cs в папке Customers
namespace UniversalCarShop.Customers;

public sealed record CustomerCapabilities
{
    public int LegPower { get; }
    public int HandPower { get; }

    public CustomerCapabilities(int legPower, int handPower)
    {
        if (legPower < 0)
            throw new ArgumentException("Сила ног не может быть отрицательной", nameof(legPower));

        if (handPower < 0)
            throw new ArgumentException("Сила рук не может быть отрицательной", nameof(handPower));

        LegPower = legPower;
        HandPower = handPower;
    }
}
```

Создание объекта-значения для двигателя

```
// Создаем новый файл EngineSpecification.cs в папке Engines
namespace UniversalCarShop.Energies;

public sealed record EngineSpecification
{
    public EngineSpecification(int requiredLegPower, int requiredHandPower)
    {
        if (requiredLegPower < 0)
            throw new ArgumentException("Требуемая сила ног не может быть отрицательной", nameof(requiredLegPower));

        if (requiredHandPower < 0)
            throw new ArgumentException("Требуемая сила рук не может быть отрицательной", nameof(requiredHandPower));

        RequiredLegPower = requiredLegPower;
        RequiredHandPower = requiredHandPower;
    }

    public int RequiredLegPower { get; }
    public int RequiredHandPower { get; }

    public bool IsCompatibleWith(CustomerCapabilities capabilities)
    {
        if (capabilities == null)
            throw new ArgumentNullException(nameof(capabilities));

        return capabilities.LegPower >= RequiredLegPower &&
               capabilities.HandPower >= RequiredHandPower;
    }
}
```

Рефакторинг классов для использования объектов-значений

```
// Модифицируем класс Customer
public sealed class Customer
{
    public Customer(string name, CustomerCapabilities capabilities)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Имя не может быть пустым", nameof(name));

        Name = name;
        Capabilities = capabilities ?? throw new ArgumentNullException(nameof(capabilities));
    }

    public string Name { get; }
    public CustomerCapabilities Capabilities { get; }
    public Car? Car { get; private set; }

    // Метод для назначения автомобиля покупателю
    public void AssignCar(Car car)
    {
        Car = car;
    }

    // Переопределяем ToString
    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append($"Имя: {Name}. Сила ног: {Capabilities.LegPower}. Сила рук: {Capabilities.HandPower}. ");
        if (Car is null)
        {
            builder.Append("Автомобиль: { Нет }");
        }
        else
        {
            builder.Append($"Автомобиль: {{ {Car} }}");
        }
        return builder.ToString();
    }
}
```

Выделение агрегатов

```
// Модифицируем класс Car
public sealed class Car
{
    private readonly IEngine _engine;

    public Car(IEngine engine, int number)
    {
        Number = number;
        _engine = engine;
    }

    public int Number { get; }
    public bool IsSold { get; private set; }

    public bool IsCompatible(Customer customer) => _engine.IsCompatible(customer);

    public void MarkAsSold() => IsSold = true;

    public override string ToString() => $"Номер: {Number}. Двигатель: {_engine}";
}
```

Разделение репозиториев и сервисов

```
// Модифицируем класс Car
public sealed class Car
{
    private readonly IEngine _engine;

    public Car(IEngine engine, int number)
    {
        Number = number;
        _engine = engine;
    }

    public int Number { get; }
    public bool IsSold { get; private set; }

    public bool IsCompatible(Customer customer) => _engine.IsCompatible(customer);

    public void MarkAsSold() => IsSold = true;

    public override string ToString() => $"Номер: {Number}. Двигатель: {_engine}";
}
```