Chapter 4

不自然さを解決する「ドメインサービス」

ドメインサービスは不自然さを解消します。

ドメインの概念を知識として落とし込み、それをコードで表現しようとしたとき、値オブジェクトやエンティティのふるまいとして定義すると違和感が生じるものが存在します。この違和感はドメインのものを表現しようとしたときよりも、ドメインの活動を表現しようとするときに見られる傾向があります。

違和感のあるふるまいを値オブジェクトなどに無理やり実装しようとすると、オブジェクトの責務を歪なものに変えてしまいます。このようなときに求められることは、そのふるまいをまた別のオブジェクトとして定義することです。本章で解説するドメインサービスはまさにそのオブジェクトです。

DDD 4.1 サービスが指し示すもの

サービスとは何でしょうか。

たとえばサービス業と呼ばれる業種を聞いたことがあるでしょう。ソフトウェアシステムをサービスとして提供するという言い回しもよく聞きます。他には「サービスする」という動詞として使われることもあります。それぞれ同じサービスという言葉でありながら、意味合いは明確に異なるものです。サービスという言葉に慣れ親しんではいても、いざ「サービスとは何か」と問われると案外答えづらいものです。

ソフトウェア開発の文脈で語られるサービスはクライアントのために何かを行うオブジェクトです。その定義域は幅広く、実にさまざまなことをこなします。さらに紛らわしいことにドメイン駆動設計だけに焦点を絞ってみても、同じサービスという言葉が付く用語でありながら、その意味合いの異なるものが存在します。これは大きな混乱を引き起こしているように感じます。

ドメイン駆動設計で取りざたされるサービスは大きく分けて2つです。ひとつがドメインのためのサービスで、もうひとつがアプリケーションのためのサービスです。この2つを混同することは混乱のもとです。その区分けをしっかりとするために前者をドメインサービスと呼び、後者をアプリケーションサービスと呼びます。

この章で解説するのはドメインのサービスであるドメインサービスです。アプリケーションサービスの解説は第6章『ユースケースを実現する「アプリケーションサービス」』で行います。数多くあるサービスという言葉に惑わされないように読み進めていきましょう。

DDD 4.2 ドメインサービスとは

値オブジェクトやエンティティなどのドメインオブジェクトにはふるまいが記述されます。たとえば、ユーザ名に文字数や利用できる文字種に制限があるのであれば、その知識はユーザ名の値オブジェクトに記述されてしかるべきでしょう。

しかし、システムには値オブジェクトやエンティティに記述すると不自然になっ

てしまうふるまいが存在します。ドメインサービスはそういった不自然さを解決するオブジェクトです。

まずは値オブジェクトやエンティティに記述されると不自然なふるまいがどういったものなのかを確認し、その後ドメインサービスによる解決策を確認しましょう。

4.2.1 不自然なふるまいを確認する

現実において同姓同名は起こりえますが、システムにおいてはユーザ名の重複を許可しないことはありえます。ユーザ名の重複を許さないというのはドメインのルールであり、ドメインオブジェクトのふるまいとして定義すべきものです。さて、このふるまいは具体的にどのオブジェクトに記述されるべきでしょうか。

ユーザに関することはユーザを表すオブジェクトに、という健全な論理的思考からまずはUserクラスに重複確認のふるまいを追加してみます（**リスト4.1**）。

リスト4.1：重複確認のふるまいをUserクラスに追加

```
class User
{
  private readonly UserId id;
  private UserName name;

  public User(UserId id, UserName name)
  {
    if (id == null) throw new ArgumentNullException(nameof(id));
    if (name == null) throw new ArgumentNullException(nameof(➡
name));

    this.id = id;
    this.name = name;
  }

  // 追加した重複確認のふるまい
  public bool Exists(User user)
  {
```

```
    // 重複を確認するコード
    (…略…)
  }
}
```

現時点では重複確認の具体的な処理についてを論じたいわけではありません。いま認識すべきことは重複の確認を行う手段がUserクラスのふるまいとして定義されているということです。

このオブジェクトの定義を確認する限りでは問題がないように見えますが、実はこれは不自然さを生み出すコードです。実際にこのメソッドを利用して重複確認をしてみましょう（**リスト4.2**）。

リスト4.2：リスト4.1を利用して重複確認を行う

```
var userId = new UserId("id");
var userName = new UserName("nrs");
var user = new User(userId, userName);

// 生成したオブジェクト自身に問い合わせをすることになる
var duplicateCheckResult = user.Exists(user);
Console.WriteLine(duplicateCheckResult); // true? false?
```

重複を確認するふるまいはUserクラスに定義されているので、重複の有無を自身に対して問い合わせることになります。これは多くの開発者を混乱させる不自然な記述です。自身が重複しているかどうかの確認を自身に依頼したとき、果たして問い合わせの結果は真を返すべきでしょうか。それとも偽を返すべきでしょうか。

重複確認を行うときに、生成したオブジェクト自身に問い合わせを行うというのは開発者を惑わせるようです。異なるアプローチを考えてみましょう。たとえば重複を確認するために専用のインスタンスを用意するのはどうでしょうか（**リスト4.3**）。

リスト4.3：重複確認専用のインスタンスを用意する

```
var checkId = new UserId("check");
var checkName = new UserName("checker");
var checkObject = new User(checkId, checkName);
```

```
var userId = new UserId("id");
var userName = new UserName("nrs");
var user = new User(userId, userName);

// 重複確認専用インスタンスに問い合わせ
var duplicateCheckResult = checkObject.Exists(user);
Console.WriteLine(duplicateCheckResult);
```

リスト4.3は自身に問い合わせをせずに済む点では、不自然さがいくばくか軽減されています。しかし、重複確認のために作成されたcheckObjectはユーザを表すオブジェクトでありながらユーザではない不自然なオブジェクトです。このような不自然な存在を容認するのが正しいコードであるとは思えません。

どうやら重複確認はエンティティであるユーザオブジェクトに記述すると不自然になるふるまいの典型のようです。こういった不自然さを解決するのに活躍するのがドメインサービスです。

4.2.2 不自然さを解決するオブジェクト

ドメインサービスは通常のオブジェクトと何ら変わりはありません。ユーザのドメインサービスは**リスト4.4**のように定義します。

リスト4.4：ユーザのドメインサービスの定義

```
class UserService
{
  public bool Exists(User user)
  {
    // 重複を確認する処理
    (…略…)
  }
}
```

ドメインサービスは値オブジェクトやエンティティと異なり、自身のふるまいを変更するようなインスタンス特有の状態をもたないオブジェクトです。

重複を確認するための具体的な実装についてはこの後に解説を行います。いまは重複確認を行うメソッドがUserServiceクラスに定義されていることだけを確認してください。

このユーザのドメインサービスを利用して、実際にユーザの重複確認を行ってみましょう（**リスト4.5**）。

リスト4.5：リスト4.4を利用して重複確認を行う

```
var userService = new UserService();

var userId = new UserId("id");
var userName = new UserName("naruse");
var user = new User(userId, userName);

// ドメインサービスに問い合わせ
var duplicateCheckResult = userService.Exists(user);
Console.WriteLine(duplicateCheckResult);
```

ドメインサービスを用意することで、自身に重複を問い合わせたり、チェック専用のインスタンスを用意したりする必要がなくなりました。**リスト4.5**のコードは開発者を困惑させない自然なものでしょう。

値オブジェクトやエンティティに定義すると不自然に感じる操作はドメインサービスに定義することで、そこに存在する不自然さは解消されます。

DDD 4.3 ドメインサービスの濫用が行き着く先

エンティティや値オブジェクトに記述すると不自然なふるまいはドメインサービスに記述します。ここで重要なのは「不自然なふるまい」に限定することです。実をいうとすべてのふるまいはドメインサービスに記述できてしまいます。

たとえばユーザ名変更のふるまいをエンティティではなくドメインサービスに記述すると、コードは**リスト4.6**のようになります。

リスト4.6：ドメインサービスにユーザ名変更のふるまいを記述する

```
class UserService
{
  public void ChangeName(User user, UserName name)
  {
    if (user == null) throw new ArgumentNullException(nameof(➡
user));
    if (name == null) throw new ArgumentNullException(nameof(➡
name));

    user.Name = name;
  }
}
```

リスト4.6は意図したとおり、ユーザ名の変更をこなします。そういった意味では正しいコードですが、このときUserクラスの記述はどのようなものになっているでしょうか（**リスト4.7**）。

リスト4.7：**リスト4.6**で利用されているUserクラスの定義

```
class User
{
  private readonly UserId id;

  public User(UserId id, UserName name)
  {
    this.id = id;
    Name = name;
  }

  public UserName Name { get; set; }
}
```

ドメインサービスにすべてのふるまいを記述するとエンティティにはゲッターとセッターだけが残ります。いかに熟練した開発者であっても、このクラスの定義を

見ただけでユーザにどのようなふるまいやルールが存在するのかを読み取ることは不可能です。

無思慮にドメインサービスへふるまいを移設することは、ドメインオブジェクトをただデータを保持するだけの無口なオブジェクトに変容させる結果を招きます。ドメインオブジェクトに本来記述されるべき知識やふるまいが、ドメインサービスやアプリケーションサービスに記述され、語るべきことを何も語っていないドメインオブジェクトの状態をドメインモデル貧血症といいます。これはオブジェクト指向設計のデータとふるまいをまとめるという基本的な戦略の真逆をいくものです。

ユーザ名を変更するふるまいは本来であればUserクラスに定義するべきものです（**リスト4.8**）。

リスト4.8：Userクラスにふるまいを定義する

```
class User
{
  private readonly UserId id;
  private UserName name;

  public User(UserId id, UserName name)
  {
    this.id = id;
    this.name = name;
  }

  public void ChangeUserName(UserName name)
  {
    if (name == null) throw new ArgumentNullException(nameof(➡
name));
    this.name = name;
  }
}
```

4.3.1 可能な限りドメインサービスを避ける

先の例からわかるとおり、すべてのふるまいはドメインサービスに移設できま

す。やろうと思えばいくらでもドメインモデル貧血症を引き起こせてしまいます。

もちろんふるまいの中にはドメインサービスとして抽出しないと違和感のあるものは存在します。ふるまいをエンティティや値オブジェクトに定義するべきか、それともドメインサービスに定義するべきか、迷いが生じたらまずはエンティティや値オブジェクトに定義してください。可能な限りドメインサービスは利用しないでください。

ドメインサービスの濫用はデータとふるまいを断絶させ、ロジックの点在を促す行為です。ロジックの点在はソフトウェアの変化を阻害し、深刻に停滞させます。ソフトウェアの変更容易性を担保するためにも、コードを一元的に管理することを早々に諦めることは絶対にしてはいけません。

DDD 4.4 エンティティや値オブジェクトと共にユースケースを組み立てる

ドメインサービスは値オブジェクトやエンティティと組み合わせて利用されます。ドメインサービスの扱い方を確認するために、ここで実際にユースケースを組み立ててみましょう。ここで組み立てるユースケースはこれまで題材にしてきたユーザを作成する処理です。

ユーザ作成処理の仕様は単純です。クライアントはユーザ名を指定してユーザ作成処理を呼び出します。そのときユーザ名が重複しないようであればユーザを作成し保存します。今回取り扱うデータストアは一般的なリレーショナルデータベースを対象にします。

4.4.1 ユーザエンティティの確認

まずはユーザを表現するUserクラスを定義します（**リスト4.9**）。

リスト4.9：Userクラスの定義

```
class User
{
  public User(UserName name)
  {
```

```
    if (name == null) throw new ArgumentNullException(nameof(➡
name));

    Id = new UserId(Guid.NewGuid().ToString());
    Name = name;
  }

  public UserId Id { get; }
  public UserName Name { get; }
}
```

ユーザはIdにより識別されるオブジェクトであるエンティティです。なおユーザ作成処理においてはUserクラスにふるまいは不要なので、主だったメソッドは定義されていません。

Userクラスを構成するオブジェクトについても確認しておきましょう。Userクラスには UserId型の識別子が属性として定義されています。またユーザ名を表すUserName型のプロパティも定義されています。これらの実装は**リスト4.10**です。

リスト4.10： UserIdクラスとUserNameクラスの定義

```
class UserId
{
  public UserId(string value)
  {
    if (value == null) throw new ArgumentNullException(nameof(➡
value));

    Value = value;
  }

  public string Value { get; }
}

class UserName
{
```

```
  public UserName(string value)
  {
    if (value == null) throw new ArgumentNullException(nameof(➡
value));
    if (value.Length < 3) throw new ArgumentException("ユーザ名は➡
3文字以上です", nameof(value));

    Value = value;
  }

  public string Value { get; }
}
```

UserIdとUserNameはいずれもデータをラップしているだけの単純な値オブジェクトです。UserNameは特に3文字未満のユーザ名は例外を送出することで、ユーザ名が3文字以上であることを強制しています。

4.4.2 ユーザ作成処理の実装

ユーザエンティティとそれを構成するオブジェクトの実装について確認したところで、いよいよ具体的なユーザ作成処理に移ります。**リスト4.11**はユーザ作成処理の具体的な実装です。まずはコードを俯瞰しましょう。

リスト4.11：ユーザ作成処理の実装

```
class Program
{
  public void CreateUser(string userName)
  {
    var user = new User(
      new UserName(userName)
    );
```

```
    var userService = new UserService();
    if (userService.Exists(user))
    {
      throw new Exception($"{userName}は既に存在しています");
    }

    var connectionString = ConfigurationManager.➡
ConnectionStrings["FooConnection"].ConnectionString;
    using (var connection = new SqlConnection(connectionString))
    using (var command = connection.CreateCommand())
    {
      connection.Open();
      command.CommandText = "INSERT INTO users (id, name) ➡
VALUES(@id, @name)";
      command.Parameters.Add(new SqlParameter("@id", ➡
user.Id.Value));
      command.Parameters.Add(new SqlParameter("@name", ➡
user.Name.Value));
      command.ExecuteNonQuery();
    }
  }
}
```

コードをあまり深く読み込むことをしなくても、まず最初にユーザを作成し、次に重複確認を行っているところまでは読み取れます。しかし、その後に続く処理についてはどうでしょうか。

後半のコードはそれまでのコードと異なり、眺めるだけでは意図を掴めません。処理を注意深く読み込むとリレーショナルデータベースに接続するための接続文字列を用いてデータストアに接続し、SQLを発行してユーザ情報の保存を行っているのが読み取れます。それまでのユーザ作成や重複確認に比べて、コードの大部分はデータストアに対する具体的な操作が多く記述されています。コード自体はさほど難易度が高くないものですが、作成されたユーザを保存する意図を読み取るにはコードを読み込む必要があります。

ドメインサービスであるUserServiceの実装はどうでしょうか（**リスト4.12**）。

リスト4.12：ドメインサービスの実装

```
class UserService
{
  public bool Exists(User user)
  {
    var connectionString = ConfigurationManager.➡
ConnectionStrings["FooConnection"].ConnectionString;
    using (var connection = new SqlConnection(connectionString))
    using (var command = connection.CreateCommand())
    {
      connection.Open();
      command.CommandText = "SELECT * FROM users WHERE name = ➡
@name";
      command.Parameters.Add(new SqlParameter("@name", ➡
user.Name.Value));
      using (var reader = command.ExecuteReader())
      {
        var exist = reader.Read();
        return exist;
      }
    }
  }
}
```

ユーザ名の重複を確認するにはデータストアへの問い合わせが必要です。そのため、UserServiceの重複確認処理はデータストアの操作に終始しています。

これらのコードはいずれも正しく動作しますが、柔軟性に乏しいコードです。たとえば、もしもデータストアがリレーショナルデータベースではなくNoSQLデータベースへ変更する必要に迫られたとしたら、どのようなことが起きるでしょうか。ユーザ作成処理の本質は何も変わっていないにもかかわらず、そのコードの大半を変更する必要があるでしょう。UserServiceクラスにいたっては、すべてのコードをNoSQLデータベースを操作するコードに置き換える必要があります。

データを取り扱う以上、データの保存や読み取りにまつわる処理を記述することは避けられません。しかしユーザ作成処理において、コードの大半はデータストア

に対する操作処理が占めるべきでしょうか。特定のデータストアに依存することが正しい道でしょうか。

もちろんそんなことはありません。ユーザ作成処理の本質は「ユーザが作成される」ことと「重複確認が行われる」こと、そして「生成されたユーザが保存される」ことです。コードで表現すべきはこういった本質的なことです。決して特定のデータストアにまつわるアレやコレやではありません。

ソフトウェアシステムにおいてデータの保存処理はなくてはならないものです。しかし、保存処理にまつわるコードをそのまま記述すると処理の趣旨がぼやけてしまいます。この問題を解決するには、次の章で解説するリポジトリというパターンが役立ちます。

COLUMN

ドメインサービスの基準

ドメインサービスはドメインモデルのコード上の表現であり、括りとしては値オブジェクトやエンティティと同一です。そのためドメインサービスは入出力を伴う処理を取り扱わないようにすべきという考えもあります。それに照らし合わせると本文のように「ユーザの重複」に関する確認をドメインサービスとして実装することは間違っています。

本来データストアが存在しないドメインにとって、入出力操作はアプリケーションを構築する上で追加されたもので、アプリケーションの関心事です。そのためドメインの概念や知識のコード上の表現であるドメインオブジェクトが、データストア操作を取り扱うことは好ましくありません。ドメインオブジェクトはドメインモデルを表現することに徹するべきです。にもかかわらず題材を、それに反するものにしたのかというと、筆者の見解と異なるからです。

その処理がドメインサービスかどうかを見極める際に筆者が重要視していることは、ドメインに基づくものかそうでないかという点です。「ユーザの重複」という考えがドメインに基づくものであれば、それを実現するサービスはドメインサービスです。それをコードとして表現するためにインフラストラクチャのサービスの協力を得ることは問題ないと考えています。反対に、もしもアプリケーションを作成するにあたって必要になったのであれば、それはドメインサービスでありません。それはアプリケーションのサービス（第6章）として定義されるものでしょう。

もちろん、可能な限り入出力はドメインサービスで取り扱わないようにするという方針には賛成です。それを考慮した上で、必要とあらば入出力の伴う操作をドメインサービスとすることも厭いません。

DDD 4.5 物流システムに見るドメインサービスの例

ドメインサービスにはデータストアといったインフラストラクチャが絡まないドメインオブジェクトの操作に徹したものも存在します。というよりむしろこちらの方が本流でしょう。少し脇道に逸れることになりますが、ここでユーザの重複確認以外のドメインサービスの例を確認します。

ここで題材とするのは物流システムです。

物流システムでは、荷物を拠点から直接配送するのではなく、拠点から配送先の近くの拠点に輸送してから配送をします（図4.1）。

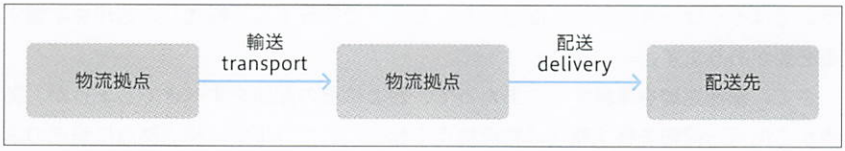

図4.1：配送のイメージ

この輸送の概念をコードに落とし込んでみましょう。

4.5.1 物流拠点のふるまいとして定義する

図4.1には物流拠点という用語があります。これはドメインの重要な概念で、エンティティとして定義されています（リスト4.13）。

リスト4.13：物流拠点エンティティ

```
// 処理の具体的な内容は主題ではないので省略
class PhysicalDistributionBase
{
    (…略…)

  public Baggage Ship(Baggage baggage)
  {
      (…略…)
  }
```

```
  public void Receive(Baggage baggage)
  {
     (…略…)
  }
}
```

物流拠点には出庫（Ship）と入庫（Receive）のふるまいがあります。出庫と入庫はセットで取り扱われるべき活動です。誤って出庫していない架空の荷物を入庫してしまったり、出庫したまま荷物をほったらかすということは起きてはいけません。現実では物理法則に従い出庫と入庫は確実にセットで実施されますが、プログラムではそうはいきません。間違いなくセットで実行する「輸送」の処理を準備する必要があります。

さて、輸送処理を準備するにあたって、どこにその記述をすべきでしょうか。拠点から拠点へ荷物を移す輸送は物流拠点を起点にしています。物流拠点に輸送のふるまいを定義してみましょう（**リスト4.14**）。

リスト4.14：物流拠点に輸送のふるまいを定義する

```
class PhysicalDistributionBase
{
   (…略…)

  public void Transport(PhysicalDistributionBase to, ➡
Baggage baggage)
  {
    var shippedBaggage = Ship(baggage);
    to.Receive(shippedBaggage);

    // たとえば配送の記録は必要だろうか
  }
}
```

リスト4.14の処理自体は問題なく完了するでしょう。Transportメソッドを利用する限り出庫と入庫は1対1で行われます。しかし、物流拠点が他の物流拠点へ

直接荷物を渡すというのは少しぎこちなさを感じます。また現時点のコードはコンテキストによる要素をそぎ落としている極力シンプルなサンプルです。実際には**リスト4.14**にコメントとして記述されているような、配送の記録などの操作が必要となる可能性もあります。それらの操作がすべて物流拠点オブジェクトによって執り行われるというのは、違和感を覚えるのと同時に扱いづらさも感じることでしょう。

4.5.2 輸送ドメインサービスを定義する

どうやら輸送という概念は特定のオブジェクトのふるまいとすると不都合のあるふるまいのようです。そこで今度は物流拠点のふるまいではなく、輸送を執り行うドメインサービスとして定義してみましょう（**リスト4.15**）。

リスト4.15：輸送ドメインサービス

```
class TransportService
{
  public void Transport(PhysicalDistributionBase from, ➡
PhysicalDistributionBase to, Baggage baggage)
  {
    var shippedBaggage = from.Ship(baggage);
    to.Receive(shippedBaggage);

    // 配送の記録を行う
    (…略…)
  }
}
```

リスト4.14に存在したぎこちなさがなくなっています。もし配送の記録を行ったとしても、違和感を感じることはありません。

そのオブジェクトの定義に納まらない操作を無理やり押し込むことになりそうなときは、ドメインサービスとして切り出すことがドメインの概念を自然に表現することに繋がります。

COLUMN

ドメインサービスの命名規則

ドメインサービスの命名規則は次の3つに分けられます。

①ドメインの概念
②ドメインの概念＋Service
③ドメインの概念＋DomainService

サービスはドメインの活動がその対象となりやすく、動詞に基づいて命名されることが多いです。

筆者は②の接尾句にServiceを付けるルールを採用することが多いです。接尾句としてDomainServiceではなくServiceを採用している理由はドメインサービスはそもそもサービスであり、それがドメインサービスかどうかはその由来によって決定づけられるべきと考えているからです。具体的なコードでいえばXxxDomain.Services.XxxServiceという名前空間により、XxxServiceはドメインサービスであることがわかります。

「ユーザの重複確認」といったある特定のドメインオブジェクトと密接に関わるようなサービスは、UserServiceのようにドメインオブジェクト名にServiceを付けた名前にし、そこに処理をまとめます。もしも「ユーザの重複を確認する」ことがそれ単体で独立させる必要があればCheckDuplicateUserServiceといったクラスに仕立てることもあります。

①のドメインの概念単体で定義する方がより表現として適切ですが、それがサービスであることを常に頭の片隅で意識しておく必要があるでしょう。

③はドメインサービスを強調するものです。コード単体で見たときのわかりやすさは他の追随を許しません。

いずれにせよ、ドメインサービスであることがチームの共通の認識となるのであればどれを選択しても構いません。

DDD 4.6 まとめ

本章ではドメインのサービスであるドメインサービスを学びました。

ドメインにはドメインオブジェクトに実装すると不自然になるふるまいが必ず存在します。これは複数のドメインオブジェクトを横断するような操作に多く見られます。そんなときに活用するのがサービスと呼ばれるオブジェクトです。

サービスは便利な存在です。ドメインオブジェクトに記述すべきふるまいは、やろうと思えばすべてサービスに移し替えられてしまいます。ドメインモデル貧血症を起こさないために、そのふるまいがどこに記述されるべきかということに細心の注意を払うようにしてください。ふるまいに乏しいオブジェクトは手続き型プログラミングを助長し、ドメインの知識をオブジェクトのふるまいとして表現するチャンスを奪います。

ここまでの内容で値オブジェクト・エンティティ・ドメインサービスという基本的なドメインの概念を表現する手段が揃ったことになります。さらに本章ではそれらの要素を使い、ついにユースケースを組み立てることを行いました。

しかし、それと同時に1つの課題が浮き出てきました。その課題はユースケースがデータストアの操作に終始してしまっていることです。次章で解説するリポジトリはこの課題を解決する力をもったパターンです。