Chapter 3 ライフサイクルのあるオブジェクト「エンティティ」

エンティティは値オブジェクトと対を為すドメインオブジェクトです。

エンティティという言葉を見聞きしたことがあるでしょうか。
ソフトウェア開発の文脈でエンティティという単語はよく出てくる用語です。たとえばデータベースのテーブル設計などで用いられる実体関連図（ER図）にはエンティティが登場します（そもそもその名前からしてentity-relationship diagramです）。またオブジェクト関係マッピング（Object-relational mapping、ORM）では永続化対象のデータをエンティティと呼びます。
しかし、ドメイン駆動設計におけるエンティティはそれらとは異なるものです。もしもあなたの知るエンティティがドメイン駆動設計の文脈をもたないのであれば、いったんその知識はしまい込みましょう。ここで解説をしようとしているのは「ドメイン駆動設計の」エンティティです。

DDD 3.1 エンティティとは

ドメイン駆動設計におけるエンティティはドメインモデルを実装したドメインオブジェクトです。第2章『システム固有の値を表現する「値オブジェクト」』で解説した値オブジェクトもドメインモデルを実装したドメインオブジェクトです。両者の違いは同一性によって識別されるか否かです。同一性という言葉はあまり耳慣れないことでしょう。まずは同一性がどういったものなのかを確認します。

人間には名前や身長、体重、趣味などさまざまな属性があります。これらの属性は固定ではなく、さまざまな要因によって変化します。たとえば年齢は誕生日を迎えると年を重ねる可変な属性です。ここで考えるべきは誕生日を迎えた当事者は、誕生日以前と以後で別人になりうるかということです。

当然ながら年齢を重ねたからといって、その人がまったくの別人になってしまうことはありません。同じように身長や体重が増減したところで、別人になり変わることはないでしょう。その人がその人たる所以は属性とはまったく別の無関係なところにあり、同一性を担保する何かが存在することを示唆しています。

ソフトウェアシステムにおいても同じように、属性で区別されないオブジェクトは存在します。たとえばシステムのユーザという概念はその典型です。

システムの利用者は最初にユーザ登録を行い、利用者個人の情報をユーザ情報として登録します。ユーザ情報はたいていの場合任意で変更可能です。このとき、ユーザ情報として登録されているデータが変更されたからといって、まったく別のユーザになってしまうことはありません。ユーザはその名前が変更されたとしても、ユーザ情報が変更されただけであって、ユーザ自体が変更されたわけではありません。ユーザは属性ではなく同一性（identity）により識別されています。

ソフトウェアシステムにはエンティティが多く登場します。まさにソフトウェア開発を行う上では切っても切れない関係です。この章では値オブジェクトと並んでドメイン駆動設計の中核を担うドメインオブジェクトであるエンティティについて学んでいきましょう。

DDD 3.2 エンティティの性質について

冒頭でも解説したとおり、エンティティは属性ではなく同一性によって識別されるオブジェクトです。これとは反対に同一性ではなく属性によって識別されるオブジェクトも存在します。

たとえば姓と名の属性からなる氏名は、そのいずれかの属性が変更されたらまったく異なるものになってしまいます。反対に属性が同じであった場合はまったく同じものであるとみなされます。まさしく氏名はその属性によって識別されるオブジェクトです。そのようなオブジェクトのことを何と呼ぶか、皆さんは既に学んでいます。氏名はまさに「値オブジェクト」です。

エンティティと値オブジェクトは共にドメインモデルの実装であるドメインオブジェクトとして似通っていますが、その性質は異なります。エンティティの性質は次のとおりです。

- 可変である
- 同じ属性であっても区別される
- 同一性により区別される

エンティティの性質には値オブジェクトの性質を真逆にしたような性質もあります。この先の解説は、値オブジェクトとの違いを意識しながら読み進めるとより理解しやすい内容です。もしも値オブジェクトに対する理解に不安が残っているようであれば、第2章『システム固有の値を表現する「値オブジェクト」』の解説を確認しに戻ってみるとよいでしょう。

3.2.1 可変である

値オブジェクトは不変なオブジェクトでした。それに比べてエンティティは可変なオブジェクトです。人々がもつ年齢や身長といった属性が変化するのと同じように、エンティティの属性は変化することが許容されています。

人生において名前を変更するケースはそれほど頻繁には発生しませんが、システム上のユーザ名を変更したいケースは存在します。ユーザ名の変更を例に「可変である」性質がどういったものか確認してみましょう。

リスト3.1はユーザを表すUserクラスですが、現在のところユーザ名の変更を行うことができません。

リスト3.1：ユーザを表すクラス

```
class User
{
  private string name;

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

    this.name = name;
  }
}
```

最初はこれでよいと思ったユーザ名であっても、システムを利用しているうちに素敵なユーザ名を後から思いつくこともあるでしょう。そのとき、せっかく思いついた素敵なユーザ名を使うことができないと、とても残念な体験になってしまいます。素敵なユーザ名を利用できるようにUserオブジェクトを可変なオブジェクトにしてみましょう（**リスト3.2**）。

リスト3.2：可変なオブジェクトに変化させる

```
class User
{
  private string name;

  public User(string name)
  {
    ChangeName(name);
  }
```

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

    this.name = name;
  }
}
```

UserオブジェクトはChangeNameメソッドを通じて名前を表す属性を変更できます。無味無色のセッターによってユーザ名の交換を行わないことで、メソッド名によりそのふるまいが何であるかが語られ、またガード節により異常な値が設定されることはありません。

値オブジェクトは不変の性質が存在するため交換（代入）によって変更を表現していましたが、エンティティは交換により変更を行いません。エンティティの属性を変化させたいときには、そのふるまいを通じて属性を変更することになります（図3.1）。

図3.1：可変なオブジェクト

但し、すべての属性を必ず可変にする必要はありません。エンティティはあくまでも、必要に応じて属性を可変にすることが許可されているに過ぎません。可変なオブジェクトは基本的には厄介な存在です。可能な限り不変にしておくことはよい習慣です。

COLUMN

セーフティネットとしての確認

モデルを表現したオブジェクトの値がドメインのルールに適合しているかどうかは重要な問題です。したがってドメインのルールに違反するようなことは排除する必要があります。本文に登場したUserオブジェクトはまさにそれを行っていて、異常な値（nullや短すぎる名前）が引き渡されるとオブジェクトは例外を送出し、プログラムは終了します。

この例外はあくまでもセーフティとして機能する例外です。

したがって、例外が起こりうることを前提にするのではなく、その検査は事前に行うべきです。たとえばユーザの名前を変更するときに、新たなユーザ名が異常な値を取りうるのであれば、クライアント側で事前に検査をします。この検査を行うと「新たなユーザ名に異常な値が混ざりうる」という意図を明確にできます（リスト3.3）。

リスト3.3：クライアントで事前に検査する

```
if (string.IsNullOrEmpty(request.Name))
{
  throw new ArgumentException("リクエストのNameがnullまたは➡
空です。");
}
user.ChangeName(request.Name);
```

3.2.2 同じ属性であっても区別される

値オブジェクトは同じ属性であれば同じものとして扱われました。エンティティはそれと異なり、たとえ同じ属性であっても区別されます。この性質を理解するために、値オブジェクトとの違いを確認していきましょう。

ここで例に挙げる氏名の値オブジェクトは姓と名の2つの属性で構成されています。値オブジェクトは等価性によって比較されるため、姓の値と名の値がそれぞれ同じである氏名オブジェクト同士はまったく同じものとして扱われます（図3.2）。

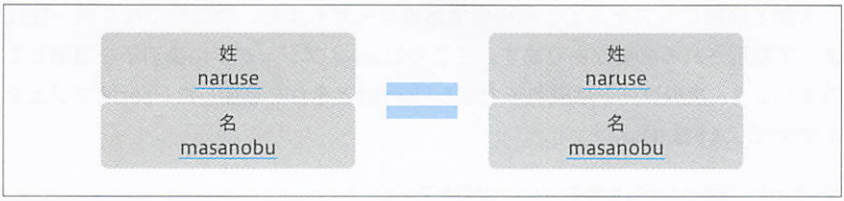

図3.2：等価であれば同一とみなされる

この値がもつ性質は、たとえば人間にはあてはめることはできません。もしこの性質を人間にあてはめたならば、氏名がまったく同じ人間同士は同一人物であるということになってしまいます（図3.3）。

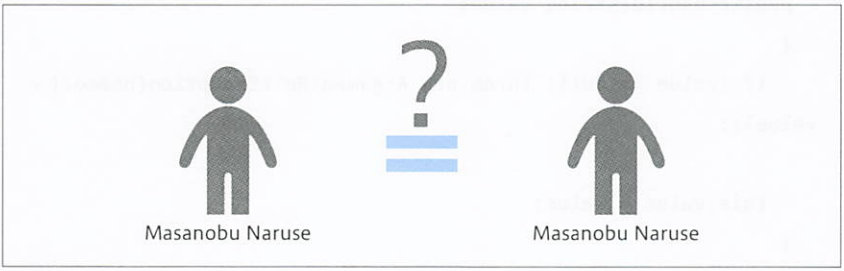

図3.3：同姓同名は同一人物

もちろんそんなことはありえません。同姓同名という言葉があるとおり、氏名が一致したからといって必ずしも同じ人物のことを指していると断定はできません（図3.4）。

図3.4：同姓同名は同一人物でない

私たち人間は属性だけでは区別されないのです。人間が区別されるのはもっと別のところにあります。ここで例にしている人間はまさしくエンティティとして表現されるものです。

人間が何をもってして区別されるのか、というのは哲学的な問いになってしまいますが、エンティティ同士を区別するためには識別子（Identity）が利用されます。

人間と同様にシステム上の利用者であるユーザもまた、等価性でなく同一性によって識別される必要があります。そこでUserオブジェクトに識別子を追加してみましょう。次のコードは識別子であるUserIdとそれを追加したUserオブジェクトです（**リスト3.4**）。

リスト3.4：識別子とそれを利用したユーザのオブジェクト

```
class UserId
{
  private string value;

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

    this.value = value;
  }
}

class User
{
  private readonly UserId id;
  private string name;

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

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

まったく同じ名前のユーザがいたとき、それが同一のユーザかそれとも別のユーザかどうかはこの識別子によって区別されます。

3.2.3 同一性をもつ

たとえばユーザ名を変更したときを考えてみましょう（図3.5）。

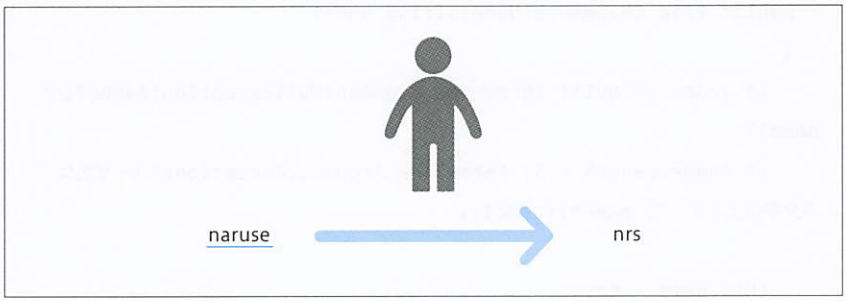

図3.5：ユーザ名変更のケース

ユーザ名を変更する前のユーザとユーザ名を変更した後のユーザは同一のユーザと判定されるべきでしょうか。それとも別のユーザと判定されるべきでしょうか。

たいていのシステムではユーザ名が異なったとしても、変更前と変更後のユーザを同一のユーザとして認識してほしいでしょう。ユーザには同一性があります。

オブジェクトには属性が異なっていたとしても同じものとしてみなす必要があるものが存在します。それらはみな同一性により識別されるオブジェクトです。

もちろんプログラムはユーザが同一かどうかを判断できませんから、同一性を判断するために何らかの手段が必要です。プログラムでは同一性の判断を実現するために識別子を利用します（リスト3.5）。

リスト3.5：同一性の判断するために識別子を追加

```
class User
{
  private readonly UserId id; // 識別子
  private string name;

  public User(UserId id, string name)
  {
```

```
    if (id == null) throw new ArgumentNullException(nameof(id));

    this.id = id;
    ChangeUserName(name);
  }

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

    this.name = name;
  }
}
```

識別子は同一性の実体です。その性質からして可変にする必要はありません。C#ではreadonly修飾子を付けて再代入を不可能にすることで、インスタンスが実体化している間もIDが変化しないことを保証できます。

こうして定義された識別子は、フィールドとして保持するだけでは意味がありません。同一性の比較を行うためのふるまいが必要です。**リスト3.6**のEqualsメソッドは比較手段の典型的な実装です。

リスト3.6：比較手段の実装

```
class User : IEquatable<User>
{
  private readonly UserId id;
  private string name;

   (…略…)

  public bool Equals(User other)
```

```
  {
    if (ReferenceEquals(null, other)) return false;
    if (ReferenceEquals(this, other)) return true;
    return Equals(id, other.id); // 比較は id 同士で行われる
  }

  public override bool Equals(object obj)
  {
    if (ReferenceEquals(null, obj)) return false;
    if (ReferenceEquals(this, obj)) return true;
    if (obj.GetType() != this.GetType()) return false;
    return Equals((User) obj);
  }

  // 言語によりGetHashCodeの実装が不要な場合もある
  public override int GetHashCode() {
    return (id != null ? id.GetHashCode() : 0);
  }
}
```

第2章『システム固有の値を表現する「値オブジェクト」』で紹介した値オブジェクトの比較処理ではすべての属性が比較の対象となっていましたが、エンティティの比較処理では同一性を表す識別子（id）だけが比較の対象となります。これにより、エンティティは属性の違いにとらわれることなく同一性の比較が可能になります（**リスト3.7**）。

リスト3.7：エンティティの比較を行う

```
void Check(User leftUser, User rightUser)
{
  if (leftUser.Equals(rightUser))
  {
    Console.WriteLine("同一のユーザです");
  }
```

```
  else
  {
    Console.WriteLine("別のユーザです");
  }
}
```

DDD 3.3 エンティティの判断基準としてのライフサイクルと連続性

値オブジェクトとエンティティはドメインの概念を表現するオブジェクトとして似通っています。であれば何を値オブジェクトにして、何をエンティティにするかという判断の基準が欲しいところです。ライフサイクルが存在し、そこに連続性が存在するかというのは大きな判断基準になります。

たとえばこれまでサンプルにしてきたユーザという概念にはライフサイクルがあるでしょうか。

ユーザはシステムを利用するために利用者によって作成されます。システムを利用していくうちにユーザ名を変更することもあるでしょう。そうして月日が流れ、あるとき利用者にとってシステムが不要になったとき、残念なことですがユーザは削除されます。

作成されて生を受け、削除されて死を迎える。まさにユーザはライフサイクルをもち、連続性のある概念です。ユーザはエンティティで間違いなさそうです。

もしもライフサイクルをもたない、またはシステムにとってライフサイクルを表現することが無意味である場合には、ひとまずは値オブジェクトとして取り扱うとよいでしょう。ライフサイクルをもつオブジェクトは生まれてから死ぬまで変化をすることがあります。正確さが求められるソフトウェアを構築するにあたって、可変なオブジェクトはその取扱いに慎重さが要求される厄介なものです。不変にしておけるものは可能な限り不変なオブジェクトのままにして取り扱うことは、シンプルなソフトウェアシステムを構築する上で大切なことです。

DDD 3.4 値オブジェクトとエンティティのどちらにもなりうるモデル

ものごとの側面は決してひとつだけとは限りません。それがまったく同じ概念を指していても、システムによっては値オブジェクトにすべきときもあればエンティティにすべきときもあります。

たとえば車にとってタイヤはパーツです。特性に細かい違いはあるものの交換可能でまさに値オブジェクトとして表現可能な概念です。しかし、タイヤを製造する工場にとってはどうでしょうか。タイヤにはロットがあり、それがいつ作られたものであるかという個体を識別することは重要なことです。タイヤはエンティティとして表現する方が相応しいでしょう。

同じものごとを題材にしても、それを取り巻く環境によってモデルに対する捉え方は変わります。値オブジェクトにも、エンティティにもなりえる概念があることを認識し、ソフトウェアにとって最適な表現方法がいずれになるのかは意識しておくとよいでしょう。

DDD 3.5 ドメインオブジェクトを定義するメリット

エンティティと値オブジェクトは異なる性質をもちますが、いずれもドメインモデルの表現であるドメインオブジェクトです。ドメインモデルをドメインオブジェクトとして定義することでどのようなメリットがあるでしょうか。ここで一度ドメインオブジェクトを定義するメリットについて確認しておきましょう。

ここに提示するメリットは次の2つです。

- コードのドキュメント性が高まる
- ドメインにおける変更をコードに伝えやすくする

これらのメリットはソフトウェアを生み出す製造工程よりも、その後の保守開発において際立つものです。少しソフトウェアの未来に思いをはせながら確認していってください。

3.5.1 コードのドキュメント性が高まる

開発者は自身が受けもつソフトウェアについて、必ずしも知識があるとは限りません。プロジェクトに途中から参画したり、前任者の異動により引き継いだりといった理由で、まったく知識のないソフトウェアに取り掛かることがあります。事前知識のない開発者はソフトウェアが満たす要件をどのようにして知るのでしょうか。

多くの場合は仕様書などのドキュメントを手掛かりにするでしょう。しかし、残念ながら仕様書というのはマクロな要件について有効であってもミクロな要件については無力であることが多いです。たちの悪いことにドキュメントはコードと異なって、記載されている内容が誤っていてもソフトウェアが動作しなくなるような問題を引き起こしません。

ソフトウェアが満たす要件を知るのにドキュメントが役に立たないのであれば、開発者はコードに頼ることになります。しかし、たとえばユーザ名に関する仕様を知ろうとしたとき、Userクラスのコードが**リスト3.8**のように記述されていたらどうでしょうか。

リスト3.8：無口なコード

```
class User
{
  public string Name { get; set; }
}
```

コードは自身のことを一切語っていません。この無口なコードを前にしては、開発者はユーザ名に関する一切の手掛かりを得ることもできないのです。

それに比べて**リスト3.9**のコードはどうでしょうか。

リスト3.9：饒舌なコード

```
class UserName
{
  private readonly string value;

  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));

    this.value = value;
  }

  (…略…)
}
```

UserNameクラスを見たときに、ユーザ名は3文字以上でないと動作しないことは自明です。コードを饒舌にする努力を怠らなければ、開発者はコードを手掛かりにして、そこに存在するルールを確認できるのです。

なお、本来ドメイン駆動設計ではドメインについて学びドメインモデルを作り上げるところから始め、それをドメインオブジェクトとして実装します（**図3.6**）。

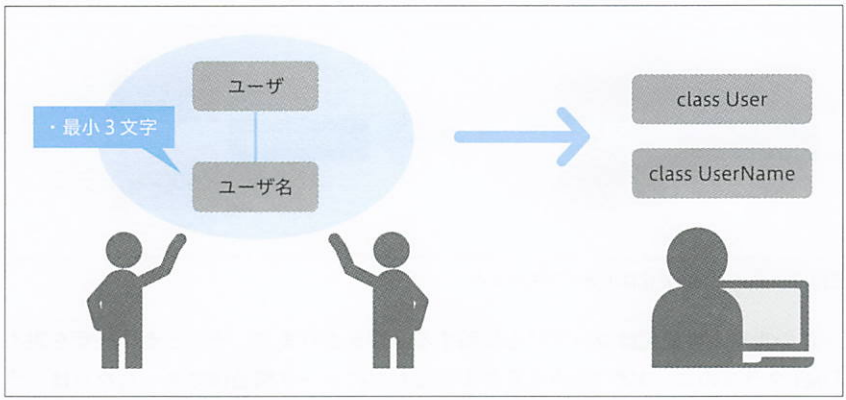

図3.6：ドメインモデルをドメインオブジェクトに

ドメインモデルに渦巻くルールはそのままドメインオブジェクトに記述されることとなります。これはドメインオブジェクトの正当性を担保することに役立ちます。

たとえば**図3.6**に記載されている「ユーザ名は最小3文字」といったルールはドメインオブジェクトとして**リスト3.9**に記述されています。

開発者であれば**リスト3.9**を見て、ユーザ名が最小3文字であるというルールが

守られていることは読み取れることでしょう。ともすればプログラミングに関する知識がない者であったとしても、1行ずつ意味をかみくだいて説明すれば、このドメインオブジェクトの正当性を理解することも可能ではないでしょうか。

もしもこういったルールがオブジェクトに記述されていない**リスト3.8**のような無口なコードの場合、その正当性を主張することは困難になります。そこに存在するルールが守られているかの判断はすべてのコードを洗い出す必要があり、たとえ熟練した開発者であっても多大な労力を求められるでしょう。

3.5.2 ドメインにおける変更をコードに伝えやすくする

ドメインオブジェクトにふるまいやルールを記述することは、ドメインにおける変更をコードに伝えやすくする効果があります。

たとえばドメインのルールに変化が起きたと仮定してみましょう。具体的には**図3.6**にあった「ユーザ名は最小3文字」というルールが「ユーザ名は最小6文字」に変更されたときです。

ドメインモデルはドメインにおける変化を受けて、**図3.7**のように変化されます。

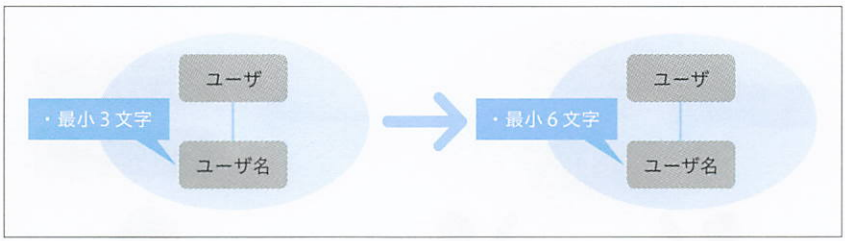

図3.7：ドメインの変化はドメインモデルへ

このルールの変化はコードにも反映する必要があります。そのときユーザを表すUserクラスのコードが**リスト3.8**のようなただのデータ構造体であったならば、その変更は極めて困難な道のりです。プログラムの随所に散らばったコードから変更すべき箇所を探し出す必要があります。

反対に、もしコードが**リスト3.9**のようにそこにあるルールを語っていたらどうでしょうか。ドメインモデルのルールが記載されているところは明白で、その修正もまた容易いものでしょう。

ドメインオブジェクトにルールやふるまいを記述することは、ドメインからドメインモデルへ伝播した変化をドメインオブジェクトまで到達させるために必要なことです。

人の営みは移ろいやすく、ドメインもまた変化するものです。ソフトウェアはドメインに生きる利用者のために存在するものである以上、こうした変化への対応が頻繁に求められます。ソフトウェアが健全に成長していく未来を守るため、コードを饒舌にする努力は常に念頭に置くべき事項でしょう。

DDD 3.6 まとめ

本章では値オブジェクトと並んで重要なモデルを表現するオブジェクトであるエンティティについて解説をしました。

豊かなふるまいをもったオブジェクトはソフトウェアがどのドメインの知識に関心があるか、それをどのように識別しているかということを浮き彫りにします。これはもちろん後続の開発者にとってドメインを理解する有効な手掛かりになります。

ドメインに対する鋭い洞察は実装時にも現れます。これは人が得意とする曖昧さをソフトウェアが受け入れられないことに端を発します。もしもエンティティを実装しようとしてそこに曖昧さを感じたのであれば、それはドメインの捉え方を見つめ直すきっかけです。