Chapter 9

複雑な生成処理を行う「ファクトリ」

ファクトリは作る知識に特化したオブジェクトです。

オブジェクトの生成はときに複雑な手順を必要とします。そういった手順は、モデルを表現するオブジェクトに無理やり実装するよりも、オブジェクトの生成それ自体を独立したオブジェクトとする方がコードの意図を明確にすることに繋がります。
道具を作ることと道具を使うことはまったく別の知識であるのと同様に、オブジェクト生成の責務はモデルを表現するオブジェクトには相応しくないのです。
本章で解説するファクトリはオブジェクトを生成する責務をもったオブジェクトです。

DDD 9.1 ファクトリの目的

あたりを少し見渡せば、視界には多くの道具が飛び込んできます。机、椅子、紙、ペン……人間は道具に囲まれて暮らしています。世の中には道具が満ち溢れていますが、いまもなお新しい道具が増え続けています。

道具が人間の想像の赴くままに創造されている理由は、道具の扱い方を知っていれば内部構造に詳しくなくとも恩恵を受けることができる便利さにあるでしょう。この便利さは大きな力です。そしてプログラムにおいてもこれは同じです。

オブジェクト指向プログラミングにおけるクラスはさながら道具です。メソッドの扱い方さえ知っていれば、クラスの内部構造を意識せずとも扱うことができます。これは開発者を大きく支援する力です。

ところで、道具は便利なものですが、その便利さに比例して複雑な機構をもつことがあります。

たとえば、コンピュータが便利な道具であることは技術者であればよくご存知のことでしょう。そしてその内部構造が複雑だということもまた熟知しているはずです。ここで争点としたいのは内部構造が複雑であることではなく、「複雑な道具はその生成過程も得てして複雑である」ことです。

皆さんはコンピュータの製造過程を知っているでしょうか。

複雑な道具はその生成過程も複雑です。ともすれば生成過程がある種の知識となります。プログラムにおいてもこれは同じで、複雑なオブジェクトはその生成過程も複雑な処理になることがあります。そうした処理はモデルを表現するドメインオブジェクトの趣旨をぼやけさせます。かといって、その生成をクライアントに押し付けるのはよい方策ではありません。生成処理自体がドメインにおいて意味をもたなかったとしても、ドメインを表現する層の責務であることには変わりないのです。

求められることは複雑なオブジェクトの生成処理をオブジェクトとして定義することです。この生成を責務とするオブジェクトのことを、道具を作る工場になぞらえて「ファクトリ」といいます。ファクトリはオブジェクトの生成に関わる知識がまとめられたオブジェクトです。

DDD 9.2 採番処理をファクトリに実装した例の確認

ファクトリが活躍するわかりやすい例として挙げられるものに採番処理があります。

これまでUserのインスタンスを生成する際、その識別子はGUID（Globally Unique Identifer）を利用していました（**リスト9.1**）。

リスト9.1：ユーザの識別子はコンストラクタで生成される

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

  // ユーザを新規作成するときのコンストラクタ
  public User(UserName name)
  {
    if (name == null) throw new ArgumentNullException(nameof(➡
name));

    // GUIDを利用して識別子を生成している
    id = new UserId(Guid.NewGuid().ToString());
    this.name = name;
  }

  // ユーザを再構築するときのコンストラクタ
  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;
```

```
  }

  (…略…)
}
```

Userクラスにはコンストラクタが2つあります。引数としてUserIdを渡すコンストラクタが再構築用で、UserIdを渡さないコンストラクタが新規作成用となっています。ユーザを新規作成するときに生成しているGUIDは衝突しない識別子として扱えるランダムな文字列ですので、コンストラクタで生成してもユニークであることが保証されます。

しかし、システムによってはこの採番処理をコントロールしたいことがあります。そういった採番処理はどのように実装するのがよいでしょうか。

伝統的な採番処理の手法にシーケンスや採番テーブルを利用したものがあります。Userクラスの採番処理をシーケンスを利用するように書き換えてみましょう(**リスト9.2**)。

リスト9.2：採番テーブルを利用するように変更

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

  public User(UserName name)
  {
    string seqId;
    // データベースの接続設定からコネクションを作成して
    var connectionString = ConfigurationManager.➡
ConnectionStrings["DefaultConnection"].ConnectionString;
    using (var connection = new SqlConnection(connectionString))
    using (var command = connection.CreateCommand())
    {
      connection.Open();
      // 採番テーブルを利用し採番処理を行っている
      command.CommandText = "SELECT seq = (NEXT VALUE FOR ➡
UserSeq)";
```

```
      using (var reader = command.ExecuteReader())
      {
        if (reader.Read())
        {
          var rawSeqId = reader["seq"];
          seqId = rawSeqId.ToString();
        }
        else
        {
          throw new Exception();
        }
      }
    }

    id = new UserId(seqId);
    this.name = name;
  }

    (…略…)
}
```

リスト9.2はあまり好ましいコードではありません。高レベルな概念であるUserにデータベースの操作という低レベルな処理が記述されてしまっています。このようなコードが引き起こす弊害は目もあてられません。Userクラスをただインスタンス化するだけでもデータベースや採番テーブルの準備が必要です。それはとても面倒な作業で直感的ではありません。

可能であればテスト用に気軽にインスタンスを生成したいときは適当なIDを振り、さもなければデータベース接続して採番を行えるようにしたいところです。こういったとき、ファクトリが役に立ちます。

採番処理を切り替えるような仕組みが必要なときには**リスト9.3**のようなファクトリのインターフェースを用意します。

リスト9.3：ファクトリのインターフェース

```
public interface IUserFactory
{
  User Create(UserName name);
}
```

ファクトリに定義されているUserNameを引数に取りUserのインスタンスを返却するメソッドはUserを新規作成する際にコンストラクタの代わりとして利用されます。

Userを生成する処理は**リスト9.3**を実装したクラスが取り持ちます。**リスト9.4**はシーケンスを利用して採番処理を行うファクトリの実装クラスです。

リスト9.4：シーケンスを利用したファクトリ

```
public class UserFactory : IUserFactory
{
  public User Create(UserName name)
  {
    string seqId;

    var connectionString = ConfigurationManager.➡
ConnectionStrings["DefaultConnection"].ConnectionString;
    using (var connection = new SqlConnection(connectionString))
    using (var command = connection.CreateCommand())
    {
      connection.Open();
      command.CommandText = "SELECT seq = (NEXT VALUE FOR ➡
UserSeq)";
      using (var reader = command.ExecuteReader())
      {
        if (reader.Read())
        {
          var rawSeqId = reader["seq"];
          seqId = rawSeqId.ToString();
        }
```

```
        else
        {
          throw new Exception();
        }
      }
    }
    var id = new UserId(seqId);
    return new User(id, name);
  }
}
```

インスタンス生成の処理がファクトリに移設されたことでUserクラスをインスタンス化する際には必ず外部からUserIdが引き渡されることになります。その変化を受けてUserクラスのUserIdを採番していたコンストラクタが不要になります（**リスト9.5**）。

リスト9.5：Userクラスのコンストラクタはひとつになる

```
public 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;
  }

  (…略…)
}
```

これでUserクラスのコンストラクタにおいてデータベースに接続するコードを記述しなくて済むようになります。

なお、ファクトリを利用するようになるとUserApplicationServiceのユーザ登録処理ではUserのインスタンスの生成をファクトリ経由で行うようになります(**リスト9.6**)。

リスト9.6：ファクトリを経由してインスタンスを生成する

```
public class UserApplicationService
{
  private readonly IUserFactory userFactory;
  private readonly IUserRepository userRepository;
  private readonly UserService userService;

  (…略…)

  public void Register(UserRegisterCommand command)
  {
    var userName = new UserName(command.Name);
    // ファクトリによってインスタンスを生成する
    var user = userFactory.Create(userName);

    if (userService.Exists(user))
    {
      throw new CanNotRegisterUserException(user);
    }

    userRepository.Save(user);
  }
}
```

Registerメソッドをテストする際にはリレーショナルデータベースに接続しないインメモリで動作させたいと考えるでしょう。その際には**リスト9.7**に示すファクトリを用意します。

リスト9.7：インメモリで動作するファクトリ

```
class InMemoryUserFactory : IUserFactory
{
  // 現在のID
  private int currentId;

  public User Create(UserName name)
  {
    // ユーザが生成されるたびにインクリメントする
    currentId++;

    return new User(
      new UserId(currentId.ToString()),
      name
    );
  }
}
```

このインメモリで動作するオブジェクトを依存解決の対象として設定すればテストを行うことが可能になります。

COLUMN

ファクトリの存在に気づかせる

ファクトリを準備したとき、オブジェクトのインスタンス化はファクトリを経由して実施されることが期待されます。しかし、Userクラスを見ても、ファクトリの存在に気づくことはできません（リスト9.8）。

リスト9.8：Userクラスの定義を見てもファクトリの存在に気づけない

```
public class User
{
  // コンストラクタがあることがわかるのみ
  public User(UserId id, UserName name);
   (…略…)
}
```

ファクトリの存在に気づかせるための仕掛けとして挙げられるのは、次のようなパッケージによるグループ分けです。

- SnsDomain.Models.Users.User
- SnsDomain.Models.Users.IUserFactory

後続の開発者がSnsDomain.Models.Usersパッケージを俯瞰してみたとき、UserとIUserFactoryが同居していることがわかります。

9.2.1 自動採番機能の活用

採番処理といえばデータベースの機能として存在する自動採番機能を無視することはできません。

たとえばSQL ServerではIDENTITYをカラムに設定するとレコードが挿入された際に自動で採番が行われます（図9.1）。

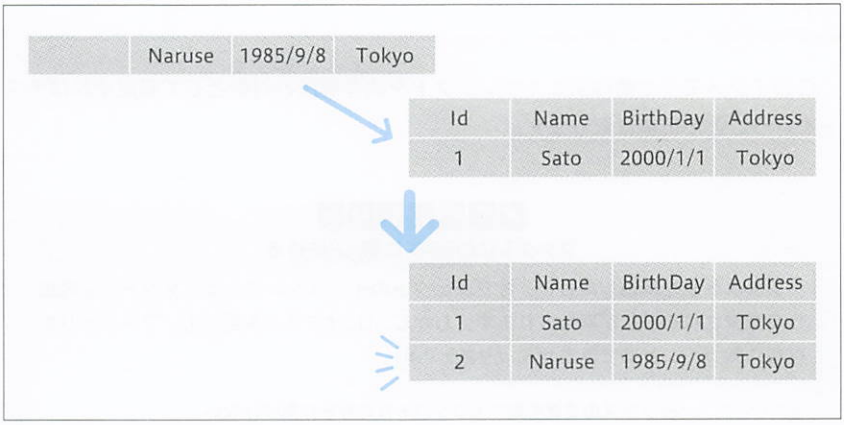

図9.1：自動採番機能

この機能は強力です。もしこの自動採番処理を取り入れたならば、コードにはどのような変化が現れるでしょうか。

自動採番処理はデータベースに対しての永続化を行うことでIDが割り振られます。必然的にインスタンスが初めて作られたときにはIDが存在していないオブジェクトとして生成されます。またIDを永続化の際に設定するため、セッターを用意する必要ができます（リスト9.9）。これらはオブジェクトを不安定にさせる要素

です。

リスト9.9：オブジェクトにセッターを用意する

```
public class User
{
  private UserName name;

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

  public UserId Id { get; set; }
}
```

エンティティは識別子により識別されるオブジェクトです。その識別子が永続化を行うまで存在しないというのは不自然で強烈な制限事項です。誤って識別子が設定されないうちに操作してしまったら、意図しない挙動になるでしょう。開発者は永続化されるまで識別子が生成されないことを常に意識し、細心の注意を払う他ありません。

もうひとつ気になることがあります。それはセッターの存在です。**リスト9.9**のUserクラスのIdプロパティのセッターはリポジトリから操作されるということを前提としています。しかしクラスの定義を見ただけでは、それをうかがい知ることが叶いません。事情を知らない開発者が不意にIDを付け替える記述をしてしまう可能性を残します。

いずれにせよ共通する問題は開発者に対して暗黙の了解を課すことです。暗黙の了解は開発者に強力な自制心を求めます。すなわち「やりすぎないように」と。

自動採番処理を利用することに決めるといくつかの懸念事項が発生します。しかし、その上であえて自動採番機能によってIDを割り振ることを受け入れる選択肢はもちろんあります。自動採番機能を採用する場合には開発上のルールをよく周知することが必要です。チームの合意として受け入れられているのであれば、問題を引き起こすことは稀でしょう。

9.2.2 リポジトリに採番用メソッドを用意する

ファクトリとは少し外れますが、リポジトリに採番を行うメソッドを用意するパターンもあります（**リスト9.10**）。

リスト9.10：リポジトリに採番処理を定義する

```
public interface IUserRepository
{
  User Find(UserId id);
  void Save(User user);
  UserId NextIdentity();
}
```

NextIdentityメソッドは採番を行い、新しいUserIdを生成します。この採番処理を利用するとコードは**リスト9.11**のように変更されます。

リスト9.11：採番処理を利用してユーザを登録する

```
public class UserApplicationService
{
  private readonly IUserRepository userRepository;

  (…略…)

  public void Register(UserRegisterCommand command)
  {
    var userName = new UserName(command.Name);
    var user = new User(
      userRepository.NextIdentity(),
      userName
    );

    (…略…)
  }
}
```

リポジトリに採番処理のメソッドをもたせるのはとても気楽な選択肢です。ファクトリを用意するほど手間ではなく、かといってIDが存在しない不安定なエンティティの存在を許容するわけでもありません。

ただし**リスト9.12**のように採番処理と永続化処理の具体的な技術が異なっていた場合は少し事情が異なってきます。

リスト9.12：採番処理と永続化で利用される技術が異なる

```
public class UserRepository : IUserRepository
{
  private readonly NumberingApi numberingApi;

  (…略…)

  // リレーショナルデータベースを利用しているが
  public User Find(UserId id)
  {
    var connectionString = ConfigurationManager.➡
ConnectionStrings["DefaultConnection"].ConnectionString;
    using (var connection = new SqlConnection(connectionString))
    using (var command = connection.CreateCommand())
    {
      connection.Open();
      command.CommandText = "SELECT * FROM users WHERE id = ➡
@id";
      command.Parameters.Add(new SqlParameter("@id", id.Value));
      using (var reader = command.ExecuteReader())
      {
        if(reader.Read())
        {
          var name = reader["name"] as string;
          return new User(
            id,
            new UserName(name)
          );
```

```
        } else {
          return null;
        }
      }
    }
  }

  // 採番処理はリレーショナルデータベースを利用していない
  public UserId NextIdentity()
  {
    var response = numberingApi.Request();
    return new UserId(response.NextId);
  }
}
```

ひとつのクラス定義の中に複数の技術基盤に基づく操作が記述されています。これを歪に感じる方もいるでしょう。気にするレベルではないという意見もあります。

このパターンはその手軽さからして受け入れられやすいものであることも確かです。開発チームでの合意が取れているのであればこのパターンを採用することは問題ではありません。

筆者の個人的な感覚では、そもそもリポジトリはデータの永続化と再構築を行うオブジェクトです。採番処理にまで手を伸ばすのは少し責務を広げ過ぎているように感じるため推奨していません。

DDD 9.3 ファクトリとして機能するメソッド

クラスそれ自体がファクトリとなる以外に、メソッドがファクトリとして機能することもあります。これはオブジェクトの内部データを利用してインスタンスを生成する必要があるときに利用されます。

たとえばサークル機能を考えてみましょう。サークルはクラブとかチームのよう

なものでユーザが所属して趣味などを語り合うグループです。サークルにはそのオーナーとなるユーザがいます。どのユーザがそのオーナーであるのかの目印としてユーザIDをもつようにしましょう。するとコードはリスト9.13のようになります。

リスト9.13：サークルを生成する

```
var circle = new Circle(
  user.Id, // ゲッターによりユーザのIDを取得
  new CircleName("my circle")
);
```

サークルのオーナーとなるユーザのIDをCircleオブジェクトへ渡すためにゲッターを利用することになります。ゲッターについては既に広く知れ渡っているとおり、安直に使用してよいものではありません（このことについては第12章『ドメインのルールを守る「集約」』にて詳しく解説します）。

内部情報を利用しつつも公開はしないという芸当は、とても単純な手法によって達成可能です。ゲッターを公開するのではなく、メソッドでインスタンスを生成して戻り値として返却すればよいのです（リスト9.14）。

リスト9.14：UserクラスのメソッドでCircleクラスのインスタンスを生成する

```
public class User
{
  // 外部に公開する必要がない
  private readonly UserId id;

  (…略…)

  // ファクトリとして機能するメソッド
  public Circle CreateCircle(CircleName circleName)
  {
    return new Circle (
      id,
      circleName
    );
```

```
    }
}
```

このようにファクトリとして機能するメソッドを用意することでインスタンスの内部情報を引き渡すことができます。

このふるまいが正当なものかどうかはドメインに対する捉え方によります。ユーザがサークルを生成することをドメインオブジェクトのふるまいとして定義するべきであれば正当化されるでしょう。

DDD 9.4 複雑な生成処理をカプセル化しよう

ポリモーフィズムの恩恵に与るためにファクトリを利用する以外に、単純に生成方法が複雑なインスタンスを構築する処理をまとめるためにファクトリを利用するのもよい習慣です。

本来であれば初期化はコンストラクタの役目です。しかしコンストラクタは単純である必要があります。コンストラクタが単純でなくなるときはファクトリを定義します。

「コンストラクタ内で他のオブジェクトを生成するかどうか」はファクトリを作る際の動機付けによい指標となります。もしもコンストラクタが他のオブジェクトを生成するようなことがあれば、そのオブジェクトが変更される際にコンストラクタも変更しなくてはならなくなる恐れがあります。他のオブジェクトをただインスタンス化するだけであったとしても、それは複雑さをはらんでいるのです。

もちろんすべてのインスタンスがファクトリにより生成されるべきと主張しているわけではありません。生成処理が複雑でないのであれば素直にコンストラクタを呼び出す方が好ましいです。ここでの主張は「ただ漫然とインスタンス化をするのではなく、ファクトリを導入すべきか検討する習慣を身に着けるべきである」というものです。

COLUMN

ドメイン設計を完成させるために必要な要素

ファクトリはドメインを由来とするオブジェクトではありません。その点についてはリポジトリもまた同様です。であればファクトリやリポジトリはドメインとは関係ないものであるかというと、それもまた違います。

オブジェクトの生成はドメイン由来ではありませんが、ドメインを表現するために必要なことです。ドメインを表現する手助けをするファクトリやリポジトリといった要素は、ドメインの設計を構成する要素です。

ドメインをモデルへ落とし込み、コードでそれを表現するというドメイン設計を完成させるために、ドメインモデルを表現する以外の要素が存在することを認識しておいてください。

DDD 9.5 まとめ

本章では採番処理に焦点を絞ってファクトリの有用性を解説しました。ファクトリはオブジェクトのライフサイクルの始まりでその役割を果たします。

複雑な処理を伴うオブジェクトの生成にファクトリを使用することでコードの論点が明確になります。同時に、まったく同じ生成処理がそこかしこに記述されることを防ぐことができます。

ファクトリによって生成処理をカプセル化することはロジックの意図を明確にしながら、柔軟性を確保する大切なことです。