



アプリケーションを 1から組み立てる

理解を深めるためにアプリケーションを最初から組み立てる過程を確認します。

ここまでドメイン駆動設計に登場する多くのパターンとソフトウェアとして成り立たせるために必要な要素の解説をしてきました。最初はその数に圧倒されたとしても、ひとつひとつ紐解いていくことで、その意味や意図を掴み取ることができたのではないかでしょうか。

ものごとを会得するためにはまず把握をし、そしてそれを反復することが大事です。理解をより深いものへと変化させるために、この章ではこれまでのパターンを使って実装の練習をしてみましょう。

DDD 11. 1

アプリケーションを組み立てるフロー

この章ではこれまで登場した要素を使って、新たな機能をアプリケーションに追加します。ここでは実践的な手順に沿って進めています。実装を始める前にどういった手順で進めるのか俯瞰しておきましょう。

まず最初に確認することは、どういった機能が求められているかです。そもそも求められているものを確認しないことには何も始められません。要求にしたがって必要な機能を考えます。

追加する機能が定まつたら、今度はその機能を成り立たせるために必要となるユースケースを洗い出します。機能を実現しようとしたとき単一のユースケースだけでは無理であることも多く、いくつかのユースケースを必要とすることもあります。

ユースケースが揃ったらそれらを実現するにあたって、必要となる概念とそこに存在するルールからアプリケーションが必要とする知識を選び出し、ドメインオブジェクトを準備します。

そして、ドメインオブジェクトを用いてユースケースを実現するアプリケーションサービスを実装していきます。

この手順が唯一のものではありませんが、本章はこの手順にしたがって進めます。

DDD 11. 2

題材とする機能

これまで題材にしてきたものはSNSのユーザ機能でした。この機能だけではユーザ登録をしただけでその先がありません。そこでここではユーザ同士の交流を促すための機能として、サークル機能を作っていくましょう。

サークルは同じ趣味をもつユーザ同士で交流するために作成されるグループです。作成されるグループはたとえばスポーツを行うためのグループであったり、ボードゲームで遊ぶためのグループであったりと、多岐に渡ります。サークルを別の言葉で表すなら、クラブやギルド、あるいはチームといった表現があるでしょう。

11.2.1 サークル機能の分析

サークル機能を実現するにあたって必要とされるユースケースは「サークルの作成」と「サークルへの参加」です（図11.1）。

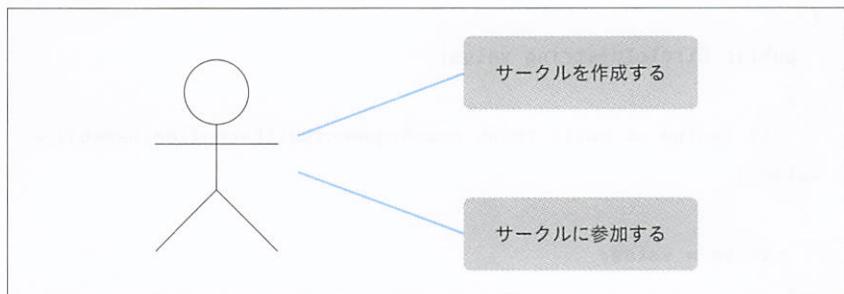


図11.1：サークル機能のユースケース

「サークルからの脱退」や「サークルの削除」といったユースケースも考えられますが、本章では図11.1のユースケースを実装します。

次にサークルの前提条件を確認しておきましょう。サークルには次のルールがあります。

- ・サークル名は3文字以上20文字以下
- ・サークル名はすべてのサークルで重複しない
- ・サークルに所属するユーザの最大数はサークルのオーナーとなるユーザを含めて30名まで

これらのルールを踏まえて2つのユースケースを組み立てていきます。

11.3 サークルの知識やルールをオブジェクトとして準備する

サークルに関する知識やルールが定まったところで、それらをコードとして表現していきます。

まずはサークルを構成する要素です。サークルはライフサイクルがあるオブジェ

クトで、つまりエンティティです。ライフサイクルを表現するには識別子が必要です。識別子は値ですので値オブジェクトとして実装します（[リスト11.1](#)）。

リスト11.1：サークルの識別子となる値オブジェクト

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

        Value = value;
    }

    public string Value { get; }
}
```

またサークルには名前を付けることができます。サークルの名前を表す値オブジェクトも用意します。サークル名に存在するルールにしたがい異常値を検知したら例外を送出するようにします（[リスト11.2](#)）。

リスト11.2：サークルの名前を表す値オブジェクト

```
public class CircleName : IEquatable<CircleName>
{
    public CircleName(string value)
    {
        if (value == null) throw new ArgumentNullException(nameof(value));

        if (value.Length < 3) throw new ArgumentException("サークル名は3文字以上です。", nameof(value));

        if (value.Length > 20) throw new ArgumentException("サークル名は20文字以下です。", nameof(value));

        Value = value;
    }

    public string Value { get; }
}
```

```
}

public string Value { get; }

public bool Equals(CircleName other)
{
    if (ReferenceEquals(null, other)) return false;
    if (ReferenceEquals(this, other)) return true;
    return string.Equals(Value, other.Value);
}

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((CircleName) obj);
}

public override int GetHashCode()
{
    return (Value != null ? Value.GetHashCode() : 0);
}
```

サークル名クラスには「サークル名は3文字以上20文字以下」というルールが記述されています。また「サークル名はすべてのサークルで重複しない」というルールに対応するため、サークル名同士を比較するふるまいが定義されています。

これらの値オブジェクトを利用してライフサイクルをもったオブジェクトであるサークルエンティティを用意します（[リスト11.3](#)）。

リスト11.3：サークルを表すエンティティ

```
1 public class Circle
2 {
3     public Circle(CircleId id, CircleName name, User owner, ↪
4         List<User> members)
5     {
6         if (id == null) throw new ArgumentNullException⇒
7             (nameof(id));
8         if (name == null) throw new ArgumentNullException⇒
9             (nameof(name));
10        if (owner == null) throw new ArgumentNullException⇒
11            (nameof(owner));
12        if (members == null) throw new ArgumentNullException⇒
13            (nameof(members));
14
15        Id = id;
APP      Name = name;
ア       Owner = owner;
ブリケ   Members = members;
シヨンを
1       }
から組
立てる
}
}

public CircleId Id { get; }
public CircleName Name { get; private set; }
public User Owner { get; private set; }
public List<User> Members { get; private set; }
}
```

サークルにはサークルのオーナーになるユーザを表すOwnerと所属しているユーザの一覧を表すMembersが定義されています。

次にサークルの永続化を行うために必要となるリポジトリも用意します（リスト11.4）。

リスト11.4：サークルのリポジトリ

```
public interface ICircleRepository
{
    void Save(Circle circle);
    Circle Find(CircleId id);
    Circle Find(CircleName name);
}
```

ユースケースのロジックを組み立てる分には、このリポジトリを実装したクラスを定義することはまだ不要です。まずはロジックを組み立てることに集中します。

サークルを生成するファクトリも同じように準備します（[リスト11.5](#)）。

リスト11.5：サークルのファクトリ

```
public interface ICircleFactory
{
    Circle Create(CircleName name, User owner);
}
```

またサークルはサークル名が重複していないかを確認する必要があります。重複に関するふるまいを[リスト11.3](#)のCircleクラスに定義すると違和感が生じます。これまでサンプルにしてきたユーザと同様に、重複を確認するふるまいをドメインサービスとして定義しましょう（[リスト11.6](#)）。

リスト11.6：サークルの重複確認を行うドメインサービス

```
public class CircleService
{
    private readonly ICircleRepository circleRepository;

    public CircleService(ICircleRepository circleRepository)
    {
        this.circleRepository = circleRepository;
    }

    public bool Exists(Circle circle)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
APP
ア
プリケ
ーシ
ョンを
1
か
ら
組
み
立
て
る
{
    var duplicated = circleRepository.Find(circle.Name);
    return duplicated != null;
}
}
```

以上で値オブジェクトからドメインサービスまでひととおりのオブジェクトを用意が終わり、必要最低限の準備は整いました。これらを取りまとめてユースケースを実現していきます。

DDD 11.4

ユースケースを組み立てる

いよいよユースケースを組み立てていきましょう。最初はサークルを作成する処理を実装していきます。

まずはコマンドオブジェクトを準備します（リスト11.7）。

リスト11.7：サークル作成処理のコマンドオブジェクト

```
public class CircleCreateCommand
{
    public CircleCreateCommand(string userId, string name)
    {
        UserId = userId;
        Name = name;
    }

    public string UserId { get; }
    public string Name { get; }
}
```

クライアントではこのコマンドオブジェクトを使ってサークルを作成するユーザ（サークルのオーナー）のIDと作成しようとしているサークルの名前を指定します。

リスト11.7を受け取って実際に処理を行うサークル作成処理はリスト11.8です。

リスト11.8：アプリケーションサービスにサークル作成処理を追加する

```
public class CircleApplicationService
{
    private readonly ICircleFactory circleFactory;
    private readonly ICircleRepository circleRepository;
    private readonly CircleService circleService;
    private readonly IUserRepository userRepository;

    public CircleApplicationService(
        ICircleFactory circleFactory,
        ICircleRepository circleRepository,
        CircleService circleService,
        IUserRepository userRepository)
    {
        this.circleFactory = circleFactory;
        this.circleRepository = circleRepository;
        this.circleService = circleService;
        this.userRepository = userRepository;
    }

    public void Create(CircleCreateCommand command)
    {
        using (var transaction = new TransactionScope())
        {
            var ownerId = new UserId(command.UserId);
            var owner = userRepository.Find(ownerId);
            if (owner == null)
            {
                throw new UserNotFoundException(ownerId, ➔
                    "サークルのオーナーとなるユーザが見つかりませんでした。");
            }
        }
    }
}
```

```

1     var name = new CircleName(command.Name);
2     var circle = circleFactory.Create(name, owner);
3     if (circleService.Exists(circle))
4     {
5         throw new CanNotRegisterCircleException(circle, ➔
6         "サークルは既に存在しています。");
7     }
8
9     circleRepository.Save(circle);
10
11    transaction.Complete();
12 }
13 }
14 }
15 }
```

サークルを作成するためにまず最初にサークルのオーナーとなるユーザを検索しています。ユーザの存在を確認できたらサークルを生成し、重複確認を行っています。重複しないことの確認が取れたらリポジトリに永続化を依頼し、処理は完了です。この処理はトランザクションスコープによりデータの整合性が維持されています。

次に、このCircleApplicationServiceにユーザがサークルに参加するための処理を追加してみましょう。まずはコマンドオブジェクトの実装です（リスト11.9）。

リスト11.9：サークル参加処理のコマンドオブジェクト

```

public class CircleJoinCommand
{
    public CircleJoinCommand(string userId, string circleId)
    {
        UserId = userId;
        CircleId = circleId;
    }
}
```

```
public string UserId { get; }
public string CircleId { get; }
}
```

サークルに参加するユーザのIDと参加先のサークルのIDを指定することで、どのユーザがどのサークルに参加するかを指定します。

このオブジェクトを利用したサークル参加処理は[リスト11.10](#)です。

リスト11.10：アプリケーションサービスにサークル参加処理を追加する

```
public class CircleApplicationService
{
    (...略...)

    public void Join(CircleJoinCommand command)
    {
        using (var transaction = new TransactionScope())
        {
            var memberId = new UserId(command.UserId);
            var member = userRepository.Find(memberId);
            if (member == null)
            {
                throw new UserNotFoundException(memberId, ⇒
                    "ユーザが見つかりませんでした。");
            }

            var id = new CircleId(command.CircleId);
            var circle = circleRepository.Find(id);
            if (circle == null)
            {
                throw new CircleNotFoundException(id, ⇒
                    "サークルが見つかりませんでした。");
            }
        }
    }
}
```

```

1 // サークルのオーナーを含めて30名か確認
2 if (circle.Members.Count >= 29)
3 {
4     throw new CircleFullException(id);
5 }
6
7 // メンバーを追加する
8 circle.Members.Add(member);
9 circleRepository.Save(circle);
10
11 transaction.Complete();
12 }
13 }
14
15 }
```

サークル参加処理ではサークルに参加しようとしているユーザを検索し、参加先のサークルを検索します。そして「サークルに所属するユーザの最大数はサークルのオーナーとなるユーザを含めて30名まで」というルールに適合しているかを確認し、サークルのメンバーとしてユーザを追加しています。トランザクションスコープにより整合性を維持するようになっているのはサークル作成処理と同様です。

リスト11.10はユーザがサークルに参加するユースケースを実現している点では評価できますが、違和感があります。それは"if (circle.Members.Count) >= 29"という記述です。ここに潜む違和感を紐解きましょう。

11.4.1 言葉との齟齬が引き起こす事態

サークルのルールである「サークルに所属するユーザの最大数はサークルのオーナーとなるユーザを含めて30名まで」は**リスト11.10**では"if (circle.Members.Count) >= 29"として実装されていますが、言葉とコードで表現に齟齬が生まれています。かたや30としているのに対し、もう一方では29と表現しているのです。

コード上の表現が29となっているのは、Circleクラスの内部でオーナーとなるユーザとメンバーにあたるユーザが別で管理されているからです（**リスト11.11**）。

リスト11.11：オーナーとメンバーが別管理になっている

```
public class Circle
{
    (...略...)
    public User Owner { get; private set; }
    public List<User> Members { get; private set; }
}
```

本来であれば表現に即した30という数字をコード上でも用いるようにすべきです。

リスト11.10のコードでは、Circleクラスの内情を知らない開発者が"if (circle.Members.Count) >= 29"を見て、現在のコードが間違っていると考えて"if (circle.Members.Count) >= 30"に変えてしまうことすらあります。クラス内の事情を外部に漏らすことは可能な限り避けるべきことです。

11.4.2 漏れ出したルールがもたらすもの

「サークルに所属するユーザの最大数はサークルのオーナーとなるユーザを含めて30名まで」というルールはドメインのルールとして重要なものです。本来であればこういったルールはドメインオブジェクトに実装されるべきです。もしもそれに反し、ドメインの重要なルールをアプリケーションサービスに記述してしまうと、同じルールが複数箇所に重複して記述されてしまいます。こういった重複はコードを変更する必要に迫られたときに問題となります。

たとえば、サークルに勧誘するユースケースを追加したとしましょう（図11.2、**リスト11.12**）。

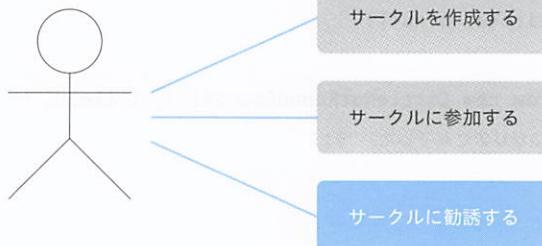


図11.2：サークルに勧誘するユースケースを追加

リスト11.12：メンバー勧誘を行う処理

```
1 public class CircleApplicationService
2 {
3     (...略...)
4
5     public void Invite(CircleInviteCommand command)
6     {
7         using (var transaction = new TransactionScope())
8         {
9             var fromUserId = new UserId(command.FromUserId);
10            var fromUser = userRepository.Find(fromUserId);
11            if (fromUser == null)
12            {
13                throw new UserNotFoundException(fromUserId, ⇒
14                    "招待元ユーザが見つかりませんでした。");
15            }
16
17            var invitedUserId = new UserId(command.InvitedUserId);
18            var invitedUser = userRepository.Find(invitedUserId);
19            if (invitedUser == null)
20            {
21                throw new UserNotFoundException(invitedUserId, ⇒
22                    "招待先ユーザが見つかりませんでした。");
23            }
24
25            var circleId = new CircleId(command.CircleId);
26            var circle = circleRepository.Find(circleId);
27            if (circle == null)
28            {
29                throw new CircleNotFoundException(circleId, ⇒
30                    "サークルが見つかりませんでした。");
31            }
32        }
33    }
34}
```

11

12

13

14

15

APP

アプリケーションを1から組み立てる

```
// サークルのオーナーを含めて30名か確認
if (circle.Members.Count >= 29)
{
    throw new CircleFullException(circleId);
}

var circleInvitation = new CircleInvitation(circle, →
fromUser, invitedUser);
circleInvitationRepository.Save(circleInvitation);
transaction.Complete();
}

}
```

ここでの問題は [リスト11.10](#) にも記述されていた "if (circle.Members.Count >= 29)" というルールが Invite メソッドにも記述されていることです。

もしもサークルのメンバー数の上限数を変更することになったらどのようなことが起きるでしょうか。おそらく Circle オブジェクトの Members プロパティがどのように使われているかをすべて検索し、それがメンバーの上限数に関する処理であつたら修正をする、といった作業を漏れなく行う必要があります。あるいは上限数を表す数字（この場合は 29 や 30）によってコードを検索してもよいでしょう。ただし、数字は他の意味でも利用されている可能性もあります。検索した結果見つけた 29 がサークルの人数を表す 29 のか、それともまったく違う概念の 29 のかを選び分ける作業は神経をすり減らすものです。

ルールがプロダクトのあちこちに点在してしまうと、ルールの変更に対する修正箇所も散らばることになり、修正作業の難易度は上昇していきます。ソフトウェアの改修を任せられた開発者が、おおむね修正したもの的一部修正漏れをしてしまい、バグを引き起こす、などといった話は失敗話としてありふれています。Circle ApplicationService はまさにその危機に晒されています。

この問題の原因は本来1箇所でまとめて管理されるべきルールが複数箇所に記述されていることです。そうなってしまった理由はどこにあるのかは実は単純です。ルールに関わるコードがサービスに記述されてしまっていることです。この問題を解決するために必要な知識は集約という考え方です。

この章ではひとつの機能を作り立たせるために開発する際の流れを意識しながら、ここまでに登場したパターンを実際に利用しました。実際にソフトウェアを開発する際にも、トップダウンで機能を洗い出し、実装はボトムアップにドメインの知識を表現するドメインオブジェクトを定義し、ユースケースの実装に臨む流れを汲むことは多いでしょう。

学んだことは実際に利用することで、理論が実践へと昇華します。テーマとなる機能を決めて、実現するにはどういったユースケースが必要か、登場する知識は何かを考え、コードで実現する。より理解を深めるために、この練習を繰り返すことをお勧めします。

次の章では本章の最後に現れたロジックが点在してしまうといった問題を解決するために「集約」を解説します。「集約」はドメイン駆動設計を構成する要素の中では比較的難しい部類になります。とはいえそう構える必要はありません。オブジェクト指向プログラミングでは当たり前のことを行なうと「集約」の考え方になります。