Chapter 12 ドメインのルールを守る「集約」

集約は変更の単位です。

データを変更するための単位として扱われるオブジェクトの集まりを集約といいます。
集約にはルートとなるオブジェクトが存在し、すべての操作はルート越しに行われます。そのようにして集約内部への操作に制限がかけられ、集約内の不変条件は維持されます。
集約はデータの変更の単位であるため、トランザクションやロックとも密接に関係するものです。

DDD 12.1 集約とは

オブジェクト指向プログラミングでは複数のオブジェクトがまとめられ、ひとつの意味をもったオブジェクトが構築されます。こうしたオブジェクトのグループには維持されるべき不変条件[*1]が存在します。

不変条件は常に維持されることが求められますが、オブジェクトのデータを変更しようとする操作を無制限に受け入れてしまうと、それは難しくなります。オブジェクトの操作には秩序が必要です。

集約は不変条件を維持する単位として切り出され、オブジェクトの操作に秩序をもたらします。

集約には境界とルートが存在します。集約の境界は集約に何が含まれるのかを定義するための境界です。集約のルートは集約に含まれる特定のオブジェクトです。

外部からの集約に対する操作はすべて集約ルートを経由して行われます。集約の境界内に存在するオブジェクトを外部にさらけ出さないことで、集約内の不変条件を維持できるようにしているのです。

集約の定義だけを聞くと難しい概念のように思えますが、実をいうと集約は既に登場しています。UserやCircleといったオブジェクトは集約にあたるものです。

12.1.1 集約の基本的構造

集約は関連するオブジェクト同士を線で囲う境界として定義されます。たとえばユーザの集約を図で表すと図12.1になります。

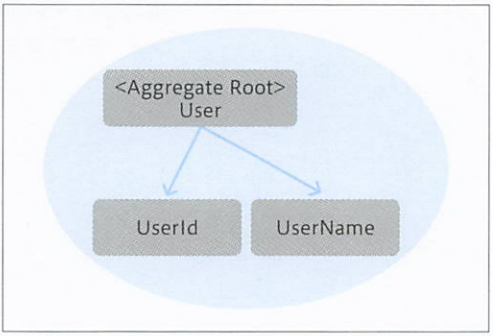

図12.1：ユーザ集約

[*1] ある処理の間、その真理値が真のまま変化しない述語のことです。

集約の外部から境界の内部のオブジェクトを操作してはいけません。集約を操作するための直接のインターフェースとなるオブジェクトは集約ルート（AR: Aggregate Root）と呼ばれるオブジェクトに限定されます。集約内部のオブジェクトに対する変更は、集約ルートが責任をもって行うことで集約内部の不変条件を保ちます。

たとえば**図12.1**でUserNameを直接操作してよいのは集約ルートであるUserのみです。ユーザ名の変更はUserオブジェクトに依頼をする形で変更をしなくてはいけません（**リスト12.1**）。

リスト12.1：ユーザ名の変更はUserオブジェクトに依頼する

```
var userName = new UserName("NewName");

// NG
user.Name = userName;

// OK
user.ChangeName(userName);
```

いずれの操作もその結果に違いはありませんが、ChangeNameといったメソッドを用意することで引き渡された値の確認（nullチェックなど）を行えます。つまり、ユーザ名がないユーザなどの不正なデータの存在を防ぐことが可能です。

次にサークル集約についても確認しましょう。ユーザ集約と同じように図で表すとサークル集約は**図12.2**になります。

サークル集約に含まれるサークル名などを操作してよいのは集約ルートであるCircleです。サークルのメンバーを追加する場合も同様で集約ルート越しに操作をする必要があります。

また**図12.2**にはユーザ集約が描かれています。サークルにはユーザがメンバーとして所属するため、集約同士の関連が表現されています。ユーザ集約はサークル集約に含まれるものではないため、ユーザ集約の情報を変更するような操作はサークル集約からは行いませんが、サークルにメンバーとしてユーザを追加するといった関連を操作する処理はサークル集約が行います。第11章『アプリケーションを1から組み立てる』ではサークルのメンバーを追加する処理を**リスト12.2**のように行っていました。

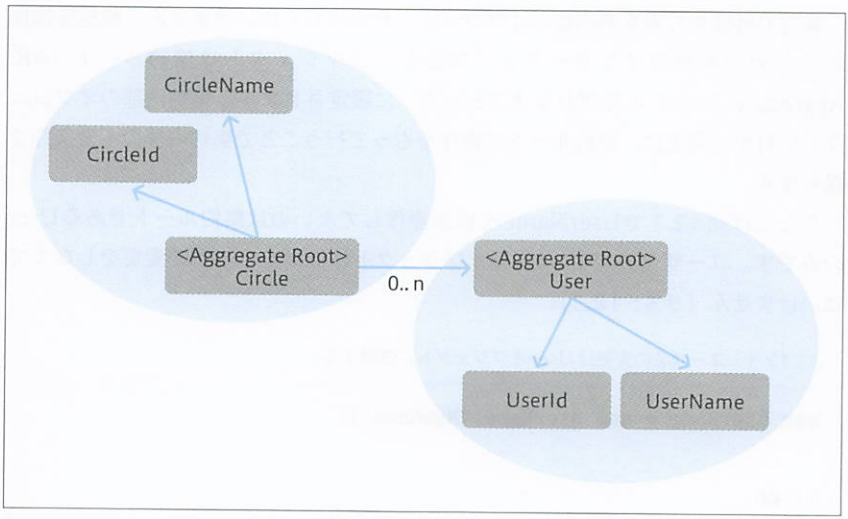

図12.2：サークル集約

リスト12.2：第11章で登場したサークルにメンバーを追加するコード

```
circle.Members.Add(member);
```

これは集約のルールに違反しています。サークル集約の内部に含まれるMembersは集約ルートであるCircleオブジェクトが責任をもって操作すべきです。そのため、本来であれば**リスト12.3**のようにCircleオブジェクトにメソッドを追加することが推奨されます。

リスト12.3：メンバーを追加するコードをエンティティに追加

```
public class Circle
{
  private readonly CircleId id;
  private User owner;
  // メンバーは非公開にできる
  private List<User> members;

  (…略…)
```

```
  public void Join(User member)
  {
    if (member == null) throw new ArgumentNullException➡
(nameof(member));

    if (members.Count >= 29)
    {
      throw new CircleFullException(id);
    }

    members.Add(member);
  }
}
```

Joinメソッドはユーザをメンバーとして追加する際に上限チェックを行います。membersは非公開になったため、サークルのメンバーを追加する際にはJoinメソッドを呼び出す以外に方法がありません。結果として、メンバーを追加する際には常に上限チェックが行われ、「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」という不変条件は常に維持されます。

リスト12.4：メンバー追加のためにCircleのメソッドを呼び出す

```
circle.Join(user);
```

直接プロパティに対してメンバーを追加していたときに比べて、コードの読み方が変化していることに気づくでしょうか。**リスト12.2**は「サークルのメンバーにユーザを追加する」と具体的な処理を読み上げるようになるのに対し、**リスト12.4**は「サークルにユーザを所属させる」とより直感的なものになっています。

オブジェクト指向プログラミングではこのように、外部から内部のオブジェクトに対して直接操作するのではなく、それを保持するオブジェクトに依頼する形を取ります。そうすることで直感的に、かつ不変条件を維持することができるのです。このことは「デメテルの法則」としても知られています。

COLUMN

集約を保持するコレクションを図に表すか

サークル集約を正確に表現しようとして、Userオブジェクトを保持するコレクションの存在を図12.3のように記載しようと考えることもあります。

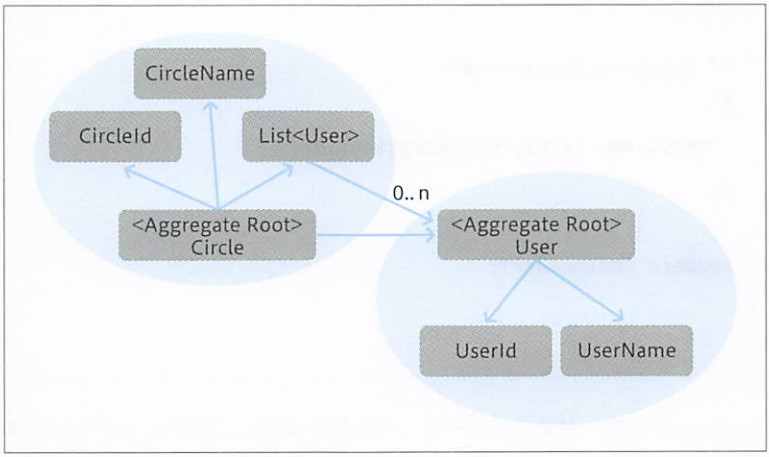

図12.3：Userオブジェクトのコレクションについて言及したサークル集約

しかし、これは必ずしも正しいものではありません。たとえば実際にコレクションをもたず、データストアから直接コレクションを生成することも可能です。

リスト12.5はあまり褒められたコードではないものの、そのことを示すサンプルです。

リスト12.5：データストアを直接操作してコレクションを生成する

```
public class Circle
{
  private readonly CircleId id;

  (…略…)

  public List<User> Members
  {
    get {
      using(var context = new MyDbContext())
      {
```

```
            circle.CircleMembers.Select(x => x.CircleId);
            var circle = context.Circles
              .Include(x => x.CircleMembers)
              .ThenInclude(x => x.Circle)
              .Single(x => x.Id == id.Value);
            var memberIds = circle.CircleMembers.Select➡
(x => x.UserId);
            var members = context.Users.Where➡
(x => memberIds.Contains(x.Id));
            return members.Select(x => new User(
              new UserId(x.Id), new UserName(x.Name))
            ).ToList();
          }
        }
      }
    }
```

集約を表す図はあくまでも集約の境界とそこに含まれるモデルが主題であり、コードに対する正確性を問うものではありません。

12.1.2 オブジェクトの操作に関する基本的な原則

オブジェクト同士が無秩序にメソッドを呼び出し合うと、不変条件を維持することは難しくなります。「デメテルの法則」はオブジェクト同士のメソッド呼び出しに秩序をもたらすガイドラインです。

デメテルの法則によると、メソッドを呼び出すオブジェクトは次の4つに限定されます。

- オブジェクト自身
- インスタンス変数
- 引数として渡されたオブジェクト
- 直接インスタンス化したオブジェクト

たとえば車を運転するときタイヤに対して直接命令しないのと同じように、オブジェクトのフィールドに直接命令をするのではなく、それを保持するオブジェクトに対して命令を行い、フィールドは保持しているオブジェクト自身が管理すべきだということです。

先述したリスト12.2のcircle.Members.Add(member);といったコードはCircleオブジェクトのフィールド（インスタンス変数）であるMembersを操作しているためデメテルの法則に違反しています。それに比べてリスト12.4のcircle.Join(user)というコードはデメテルの法則に則っています。

法則はただただ盲目的にしたがえばそれでよいというものではありません。ルールには必ずそこに至る理由が存在します。その理由まで把握してこそ本当の理解というものです。デメテルの法則が解決したい問題を紐解いてみましょう。

リスト12.6は第11章で登場したサークルにメンバーを追加する際のメンバー上限チェックを行っているコードです。

リスト12.6：メンバーを追加する際の上限チェックを行うコード

```
if (circle.Members.Count >= 29)
{
  throw new CircleFullException(id);
}
```

このコードはサークルに所属するメンバーの数が最大数を超えないように確認をしていますが、Circleオブジェクトのプロパティであるを直接操作し、Countメソッド（プロパティ）を呼び出しています。これはデメテルの法則が提示している「メソッドを呼び出してよいオブジェクト」のいずれにもあてはまりません。まさにデメテルの法則に違反している例です。

このコードの問題はメンバーの最大数に関わるロジックが点在することを助長することです。後続の開発者がメンバーの最大数に関わる処理を記述しようとしたとき、リスト12.6を参考にすると、最大数を確認するロジックが随所に点在してしまうでしょう。そうしてできあがったアプリケーションにおいて、将来サークルのメンバー数の上限に関わる改修をせざるを得なくなったとき、いったい何箇所の修正をすることになるのでしょうか。想像するだけで背筋が凍ります。

ルールがあるべきところから漏れ出し、随所にばらまかれることを見てみぬふりをすることは、自らの首を真綿で締めるような行為です。その場限りの対応はいつかだれかの苦しみに変わり、その誰かが自分である可能性もあるのです。

デメテルの法則にしたがうとコードはリスト12.7のように変化します。

リスト12.7：デメテルの法則にしたがいオブジェクトにふるまいを追加する

```
public class Circle
{
```

```
    private readonly CircleId id;
    // メンバー一覧は非公開にできる
    private List<User> members;

    (…略…)

    public bool IsFull()
    {
      return members.Count >= 29;
    }

    public void Join(User user)
    {
      if (user == null) throw new ArgumentNullException➡
(nameof(user));

      if (IsFull())
      {
        throw new CircleFullException(id);
      }

      members.Add(user);
    }
}
```

メンバー数が上限に達しているかはIsFullメソッドを通じて確認されます。上限チェックのコードはすべてこれに置き換わります（**リスト12.8**）。

リスト12.8：リスト12.7のIsFullメソッドを利用して上限チェックを行う

```
if (circle.IsFull())
{
  throw new CircleFullException(circleId);
}
```

サークルに関わる上限メンバー数の知識はすべてIsFullメソッドに集約されてい

ます。もし上限数が変更されるようであれば**リスト12.9**のようにIsFullメソッドの修正だけで完結します。

リスト12.9：上限数の変更

```
public class Circle
{
  (…略…)

  public bool IsFull()
  {
    // return members.Count >= 29;
    return members.Count >= 49;
  }
}
```

このような形であればサークルのメンバー上限数はいくら変更しても構いません。ゲッターを避ける理由はまさにここにあります。フィールドがゲッターを通じて公開されていると、本来オブジェクトに記述されるべきルールがいつ何時にどこかで漏れ出すことを防げないのです。

デメテルの法則はソフトウェアのメンテナンス性を向上させ、コードをより柔軟なものへ導きます。それは集約が成し遂げようとしていることと同じことでしょう。

12.1.3 内部データを隠蔽するために

オブジェクトの内部データは無暗やたらに公開すべきものではありません。しかし、完全に非公開にしてしまうとリポジトリがインスタンスを永続化をしようとしたときに困ったことが発生します（**リスト12.10**）。

リスト12.10：リポジトリの永続化処理

```
public class EFUserRepository : IUserRepository
{
  public void Save(User user)
  {
    // ゲッターを利用しデータの詰め替えをしている
    var userDataModel = new UserDataModel
```

```
    {
      Id = user.Id.Value,
      Name = user.Name.Value
    };
    context.Users.Add(userDataModel);
    context.SaveChanges();
  }

  (…略…)
}
```

EFUserRepositoryはUserのインスタンスを永続化する際、フレームワーク用のデータモデルであるUserDataModelにデータを移し替えています。UserDataModelを生成する際にはUserクラスのIdやNameを利用しているので、もしもUserクラスのIdやNameが非公開になってしまうと、このコードはコンパイルエラーになってしまいます。この問題に対するアプローチにはどのようなものがあるでしょうか。

最初にもっとも単純な一般的なアプローチとして挙げられるのがルールによる防衛です。つまり、リポジトリ以外で無暗に集約の内部データを取得するようなコードを書かない（要するにゲッターを使わない）ようにするというものです。これはチームで十分に認識が共有されていれば、もっともコストをかけることなく機能します。その反面、こうした紳士協定は制限力がもっとも低いです。開発者にそのつもりがなくても、誤って紳士協定を破ってしまうことは起こりえます。

もうひとつのアプローチは通知オブジェクトを使う方法です。通知オブジェクトを利用する場合はまず専用のインターフェースを用意します（**リスト12.11**）。

リスト12.11：通知のためのインターフェース

```
public interface IUserNotification
{
  void Id(UserId id);
  void Name(UserName name);
}
```

次にこのインターフェースを実装した通知オブジェクトを実装します（**リスト12.12**）。

リスト12.12：リスト12.11を実装した通知オブジェクト

```
public class UserDataModelBuilder : IUserNotification
{
  // 通知されたデータはインスタンス変数で保持される
  private UserId id;
  private UserName name;

  public void Id(UserId id)
  {
    this.id = id;
  }

  public void Name(UserName name)
  {
    this.name = name;
  }

  // 通知されたデータからデータモデルを生成するメソッド
  public UserDataModel Build()
  {
    return new UserDataModel
    {
      Id = id.Value,
      Name = name.Value
    };
  }
}
```

Userクラスは通知オブジェクトのインターフェースを受け取り、内部の情報を通知するようにします（リスト12.13）。

リスト12.13：通知オブジェクトを受け取るメソッドを追加する

```
public class User
{
  // インスタンス変数はいずれも非公開
```

```
  private readonly UserId id;
  private UserName name;

  (…略…)

  public void Notify(IUserNotification note)
  {
    // 内部データを通知
    note.Id(id);
    note.Name(name);
  }
}
```

このようにすることでオブジェクトの内部データを非公開にしたまま、外部に対して引き渡せます（**リスト12.14**）。

リスト12.14：通知オブジェクトを利用してデータモデルを取得する

```
public class EFUserRepository : IUserRepository
{
  public void Save(User user)
  {
    // 通知オブジェクトを引き渡して内部データを取得
    var userDataModelBuilder = new UserDataModelBuilder();
    user.Notify(userDataModelBuilder);

    // 通知された内部データからデータモデルを生成
    var userDataModel = userDataModelBuilder.Build();

    // データモデルをO/R Mapperに引き渡す
    context.Users.Add(userDataModel);
    context.SaveChanges();
  }

  (…略…)
}
```

もちろんこの場合はコードの記述量が大幅に増えてしまうことが懸念事項です。その懸念を払しょくするには、リスト12.11やリスト12.12のような通知オブジェクトに関連するコードをひとまとめに生成する開発者用の補助ツールを用意するとよいでしょう。

COLUMN

よりきめ細やかなアクセス修飾子（Scala）

内部データを操作できる対象の条件をコードで表現できれば、開発者が不用意に内部データに対してアクセスすることがなくなります。たとえばScalaではリスト12.15のように記述することで操作するための条件を指定できます。

リスト12.15：よりきめ細やかなアクセス制御（Scala）

```
public class User (
  private [IUserRepository] val id: UserId,
  private [IUserRepository] val name: UserName
) {
}
```

この記述方法はアクセス限定子と呼ばれます。private修飾子が示すようにidやnameは基本的には非公開ですが、[] で指定したオブジェクトに対してアクセスを許可します。リスト12.15のように指定すればIUserRepositoryの実装クラスだけに公開する内部データを実現できるのです。

アクセス限定子はその機能もさることながら宣言的なところが素晴らしいです。コードを見れば、暗にリポジトリ以外から内部データを操作すべきでないという主張が読み取れます。これは開発者に対して無暗に内部データを公開したり操作しないように踏みとどまらせるヒントになるでしょう。

DDD 12.2 集約をどう区切るか

集約をどのように区切るか、というのはとても難しいテーマです。その方針としてもっともメジャーなものは「変更の単位」でしょう。変更の単位が集約の境界を引く理由となることを理解するためには、あえて違反してみるとわかりやすいです。

さっそく違反をしてみましょう。現在、サークル集約は図12.4のように区切られています)。

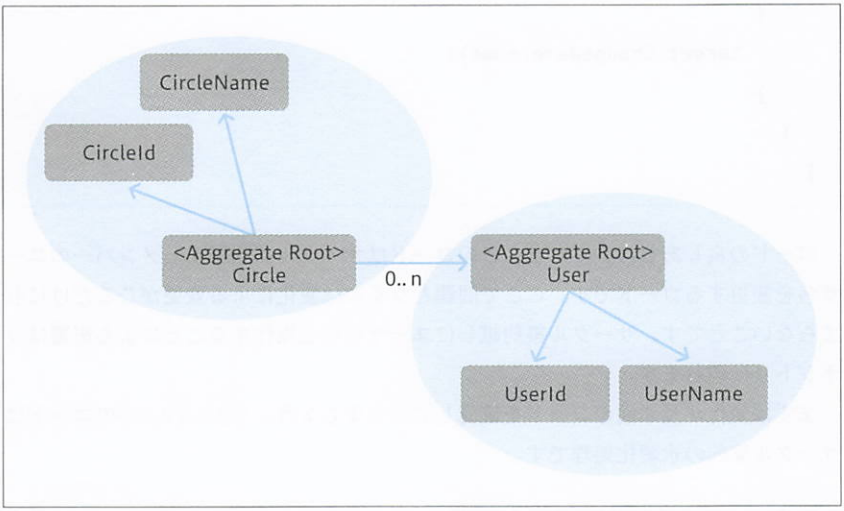

図12.4：サークル集約（図12.2を再掲）

サークルとユーザは別の集約です。集約は変更の単位ですので、サークルを変更するときはサークルの集約内部で納めるべきですし、ユーザを変更するときはユーザの集約内部の事柄だけを変更するべきです。もしも集約のルールに違反をして、サークル集約越しにユーザ集約へ変更を加えるとプログラムはどのようになってしまうでしょうか。

リスト12.16のコードはサークル集約からあえてユーザ集約の変更をしています。

リスト12.16：サークル集約を通じてユーザ集約のふるまいを呼び出す

```
public class Circle
{
  private List<User> members;

  (…略…)

  public void ChangeMemberName(UserId id, UserName name)
  {
```

```
    var target = members.FirstOrDefault(x => x.Id.Equals(id));
    if (target != null)
    {
      target.ChangeName(name);
    }
  }
}
```

コードの良しあしはさておき、このコードはサークルに所属するメンバーのユーザ名を変更するコードです。ここで問題となるのは変化による変更がここだけに収まらないことです。サークル集約越しにユーザ集約を操作することによる影響はリポジトリに現れます。

まずは変化が及ぶ前のコードを確認しておきましょう。**リスト12.17**のコードはサークル集約の永続化処理です。

リスト12.17：サークル集約を永続化する処理

```
public class CircleRepository : ICircleRepository
{
  (…略…)

  public void Save(Circle circle)
  {
    using (var command = connection.CreateCommand())
    {
      command.CommandText = @"
 MERGE INTO circles
   USING (
   SELECT @id AS id, @name AS name, @ownerId AS ownerId
   ) AS data
   ON circles.id = data.id
   WHEN MATCHED THEN
   UPDATE SET name = data.name, ownerId = data.ownerId
   WHEN NOT MATCHED THEN
```

```
    INSERT (id, name, ownerId)
    VALUES (data.id, data.name, data.ownerId);
";
        command.Parameters.Add(new SqlParameter("@id", ➡
circle.Id.Value));
        command.Parameters.Add(new SqlParameter("@name", ➡
circle.Name.Value));
        command.Parameters.Add(new SqlParameter("@ownerId", ➡
(object)circle.Owner?.Id.Value ?? DBNull.Value));

      command.ExecuteNonQuery();
    }

    using (var command = connection.CreateCommand())
    {
      command.CommandText = @"
 MERGE INTO userCircles
    USING (
    SELECT @userId AS userId, @circleId AS circleId
    ) AS data
    ON userCircles.userId = data.userId AND ➡
userCircles.circleId = data.circleId
    WHEN NOT MATCHED THEN
    INSERT (userId, circleId)
    VALUES (data.userId, data.circleId);
";
      command.Parameters.Add(new SqlParameter("@circleId", ➡
circle.Id.Value));
      command.Parameters.Add(new SqlParameter("@userId", ➡
null));

      foreach (var member in circle.Members)
      {
```

```
            command.Parameters["@userId"].Value = member.Id.Value;
            command.ExecuteNonQuery();
        }
    }
  }
}
```

サークル集約は自身の内部データのみを変更するというルールであればこれは問題がないコードです。しかし、今回のコードはユーザ集約のデータを変更しています。このままではサークル集約越しに操作をしたユーザ集約に対する変更が保存されません。ユーザ集約に対する変更を許容する場合、リポジトリのコードを変更する必要があります（**リスト12.18**）。

リスト12.18：サークル集約越しに操作されたユーザ集約に対する変更をサポートする

```
public class CircleRepository : ICircleRepository
{
  (…略…)

  public void Save(Circle circle)
  {
    // ユーザ集約に対する更新処理を行う
    using (var command = connection.CreateCommand())
    {
      command.CommandText = "UPDATE users SET username = ➡
@username WHERE id = @id";
      command.Parameters.Add(new SqlParameter("@id", null));
      command.Parameters.Add(new SqlParameter("@username", ➡
null));

      foreach (var user in circle.Members)
      {
        command.Parameters["@id"].Value = user.Id.Value;
        command.Parameters["@username"].Value = user.Name.Value;
        command.ExecuteNonQuery();
```

```
    }
}

// その後サークルの更新処理を行う
(…略…)
```

サークル集約越しにユーザの操作をサポートした結果、サークルリポジトリのロジックの多くがユーザの更新処理に汚染されてしまいました。同時に、サークルリポジトリに追加されたコードとほとんど同じコードがユーザのリポジトリにも存在しています。やむを得ない場合を除けばコードの重複は可能であれば避けたいところです。

これらの問題は本来の変更の単位を超えて変更を行っているために発生しています。集約に対する変更はあくまでその集約自身に実施させ、永続化の依頼も集約ごとに行われる必要があります。ここまでリポジトリはどの単位で作るのかということに言及していませんでしたが、こういった理由からリポジトリは変更の単位である集約ごとに用意します。

12.2.1 IDによるコンポジション

これまで何度か説いてきたように、そもそもできてしまうことを問題視する考え方もあります。つまりCircleオブジェクトはUserのインスタンスをコレクションで保持していて、プロパティを経由してそのメソッドを呼び出すことが可能であることこそが問題であると見做す考えです。

変更しないことを不文律として課すよりももっと有効な手段はないでしょうか。

もちろんあります。それはとても単純なもので、つまりインスタンスをもたない選択肢です。インスタンスをもたなければメソッドを呼び出しようがありません。インスタンスをもたないけれど、それを保持しているように見せかける、そんな便利なものがエンティティにありました。そう、識別子です。

サークル集約をユーザ集約を直接保持するのではなく、識別子をインスタンスの代わりとして保持するように修正してみましょう（**リスト12.19**）。

リスト12.19：識別子をインスタンスの代わりとして保持する

```
public class Circle
{
  public CircleId Id { get; }
  public CircleName Name { get; private set; }
  // public List<User> Members { get; private set; }
  public List<UserId> Members { get; private set; }

  (…略…)
}
```

このようにしておくことで、たとえMembersプロパティを公開していたとしてもUserオブジェクトのメソッドを呼び出すことができなくなります。あえて呼び出したい場合はUserRepositoryにUserIdを引き渡してUserオブジェクトのインスタンスを再構築し、その上でメソッドを呼び出すことになります。そのような手順が必要になれば、少なくとも不意にメソッドを呼び出して変更してしまうことはないでしょう。

また、これは同時にメモリの節約にも繋がります。たとえばサークルの名前を変更する処理を例にします（**リスト12.20**）。

リスト12.20：サークルの名前を変更する処理

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

  (…略…)

  public void Update(CircleUpdateCommand command)
  {
    using (var transaction = new TransactionScope())
    {
      var id = new CircleId(command.Id);
      // この時点でUserのインスタンスが再構築されるが
```

```
        var circle = circleRepository.Find(id);
        if (circle == null)
        {
          throw new CircleNotFoundException(id);
        }

        if (command.Name != null)
        {
          var name = new CircleName(command.Name);
          circle.ChangeName(name);

          if (circleService.Exists(circle))
          {
            throw new CanNotRegisterCircleException(circle, ➡
"サークルは既に存在しています。");
          }
        }

        circleRepository.Save(circle);

        transaction.Complete();

        // Userのインスタンスは使われることなく捨てられる
      }
    }
}
```

サークルの名前を変更する処理ではユーザを操作するようなことはありません。それゆえ、Circleオブジェクトがサークルのメンバーを Userオブジェクトとして保持している場合、リポジトリがインスタンスを再構築しますが、まったく利用されずに捨てられることとなります。これは明らかにリソースの無駄遣いです。Userオブジェクトを保持する代わりにUserIdを保持することで、Userオブジェクトを再構築するための処理能力も節約できますし、インスタンスを保持するメモリも節約できます。

COLUMN

IDのゲッターに対する是非

ここまでゲッターについては可能な限り排除すべきものとして説明してきたつもりです。しかし、その対象が識別子であった場合は少し事情が変わってきます（リスト12.21）。

リスト12.21：識別子をゲッターで公開する

```
public class Circle
{
  private CircleName name;
  private UserId owner;
  private List<UserId> members;

  public Circle(CircleId id, CircleName name, ➡
UserId owner, List<UserId> members)
  {
    if (id == null)
      throw new ArgumentNullException(nameof(id));
    if (name == null)
      throw new ArgumentNullException(nameof(name));
    if (owner == null)
      throw new ArgumentNullException(nameof(owner));
    if (members == null)
      throw new ArgumentNullException(nameof(members));

    Id = id;
    this.name = name;
    this.owner = owner;
    this.members = members;
  }

  public CircleId Id { get; }

  public void Notify(ICircleNotification note)
  {
```

```
        note.Id(Id);
        note.Name(name);
        note.Owner(owner);
        note.Members(members);
    }

    (…略…)
}
```

このCircleクラスは識別子をゲッターで公開しています。理想論でいえばこのゲッターも排除すべきですが、識別子はエンティティを表現するためのシステマチックな属性で、それ自体が集約の代わりとして扱える便利なものです。一意な識別子自体に関心が寄せられることはありますが（宅急便の追跡番号など）、識別子に対してビジネスルールが記述されることは多くありません。そのようなときは識別子を公開することで発生するデメリットよりも公開することのメリットの方が大きいこともあるでしょう。

DDD 12.3 集約の大きさと操作の単位

トランザクションはデータをロックします。集約が大きくなればなるほどロックの範囲もそれに比例して大きくなります。

集約を不用意に大きくしてしまうと、それだけ処理が失敗する可能性を高めます。

集約の大きさはなるべく小さく保つべきです。もしも巨大な集約ができあがってしまったのであれば、それは今一度集約の境界線を見つめ直すチャンスです。

また複数の集約を同一トランザクションで操作することも可能な限り避けます。複数の集約にまたがるトランザクションは、巨大な集約と同様に広範囲なデータロックを引き起こす可能性を高めます。

COLUMN

結果整合性

それでもなお、複数の集約をまたがるような処理を取り扱いたいときもあります。そういったときに利用できるのが結果整合性です。

トランザクション整合性は即時的な整合性ですが、結果整合性はあるタイミングにおいて矛盾が発生することを許容します。もちろんそのままではシステムが破綻してしまいますので、最終的には整合性を保つような仕組みにより解決をします。

たとえば、これは極端な例ですが、一日１回cron（ジョブを自動実行するデーモンプロセス）にてユーザをすべて検査し、もしも同じユーザ名のユーザがいたらそのユーザの名前をランダムで重複しない文字列に変更してしまうといったような仕組みです。図12.5はひどく乱暴な処理ですが、システム全体としての整合性は保たれます。

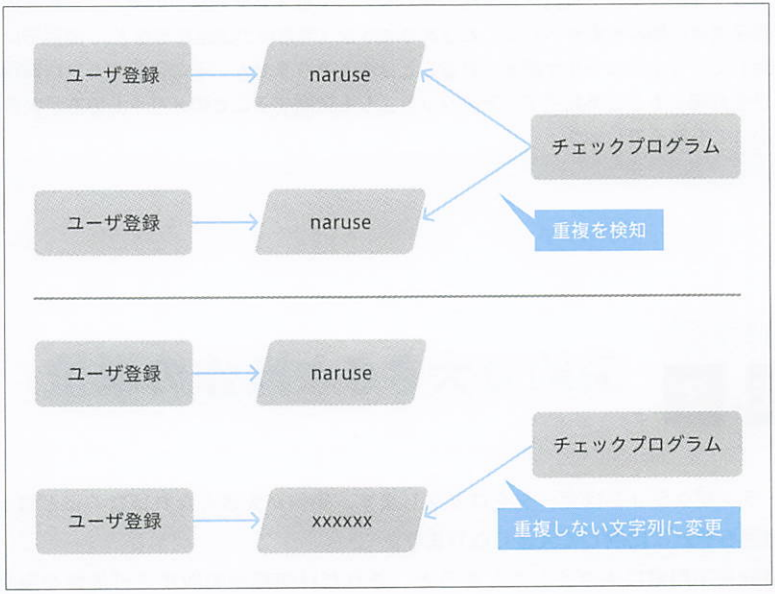

図12.5：結果整合性の一例

システムに必要な整合性を選り分けてみると、即時的な整合性が求められるものは思ったよりも少ないものです。もしも、トランザクションによって問題が発生した際には、結果整合性について一考してみてもよいでしょう。

DDD 12.4 言葉との齟齬を消す

サークルには「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」という不変条件があります。30といった具体的な数字が出ていますが、コードに出てくる数字は29です（リスト12.22）。

リスト12.22：30ではなく29が現れている

```
public class Circle
{
  private User owner;
  private List<User> members;

   (…略…)

  public bool IsFull()
  {
    return members.Count >= 29;
  }
}
```

これはコードが間違っているわけではありません。Circleにはサークルのメンバーを表すmembersとは別にサークルのオーナーのユーザが保持されており、membersとは別のフィールドで管理されているため、リスト12.22のIsFullメソッドでは30から1引いて29という数値で比較をしています。

しかしながら、コードに問題はないものの、言葉との齟齬は誤解を招きます。後続の開発者がリスト12.22のコードを見て、間違っているのではないかと考えることも否めません。

コードは可能な限り言葉との齟齬がないようにすべきです。そのためにもCircleにはメンバーを数えるメソッドを追加するとよいでしょう（リスト12.23）。

リスト12.23：サークルのオーナーとメンバーの定義

```
public class Circle
{
  private User owner;
  private List<User> members;

   (…略…)

  public bool IsFull()
  {
    return CountMembers() >= 30;
  }

  public int CountMembers()
  {
    return members.Count + 1;
  }
}
```


DDD 12.5 まとめ

この章ではオブジェクトがもつ不変条件を守る境界として集約を学びました。

集約はシステマチックに定義できるものではありません。そもそもドメインに渦巻く概念はそのほとんどが連なっているものです。そこに境界線を引くことは簡単なことではありません。

集約の境界線を引くことは、ドメインの概念を捉え、そこにある不変条件を導き出し、ドメインとシステムを両天秤にかけながら最適解を目指すような作業です。どちらか一方によりすぎることのないバランスが取れた解を目指しましょう。