Chapter 13

複雑な条件を表現する「仕様」

仕様はオブジェクトの評価を行うオブジェクトです。

オブジェクトの評価はときに複雑な手順が必要となります。こうした評価処理をオブジェクトのメソッドとして定義すると、オブジェクト本来の趣旨を見えづらくしてしまうことがあります。
評価処理はオブジェクトに定義する以外に、評価自体をオブジェクトとして切り出すことが可能です。そうして切り出された条件に合致しているかどうかを見極めるオブジェクトが、本章で解説する仕様です。

DDD 13.1 仕様とは

オブジェクトの評価は単純なものであればメソッドとして定義されますが、すべての評価が単純な処理であるとは限りません。評価処理にはオブジェクトのメソッドとして定義されるには似つかわしくないものも存在します。

そういった複雑な評価の手順は、アプリケーションサービスに記述されてしまうことが多いです。しかしながら、オブジェクトの評価はドメインの重要なルールです。サービスに記述されてしまうことは問題です。

この対策として挙げられるのが仕様です。仕様はあるオブジェクトがある評価基準に達しているかを判定するオブジェクトです。

まずは実際に複雑な評価を例にしながら、仕様がどういったものかを確認していきましょう。

13.1.1 複雑な評価処理を確認する

あるオブジェクトがある特定の条件にしたがっているかを評価する処理は、オブジェクトのメソッドとして定義されます。これまで取り扱ってきたサークルを表すオブジェクトにも、まさに評価を行うメソッドがありました（**リスト13.1**）。

リスト13.1：条件にしたがっているかを評価するふるまい

```
public class Circle
{
  (…略…)

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

これほど単純な条件であれば問題はありません。しかし、これよりも、もう少し複雑であった場合はどうでしょうか。

たとえばサークルの人数上限が、所属しているユーザのタイプにより変動するルールを考えてみましょう。

- ユーザにはプレミアムユーザと呼ばれるタイプが存在する
- 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
- プレミアムユーザが10名以上所属しているサークルはメンバーの最大数が50名に引き上げられる

Circleはサークルに所属するメンバーを保持していますが、UserIdのコレクションを保持しているにすぎず、プレミアムユーザが何名存在するかはユーザのリポジトリに問い合わせる必要があります。しかし、Circleはユーザのリポジトリを保持していません。そこで、リポジトリを保持しているアプリケーションサービス上で判定してみましょう（**リスト13.2**）。

リスト13.2：サークルのメンバー上限数は条件によって変更される

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

  (…略…)

  public void Join(CircleJoinCommand command)
  {
    var circleId = new CircleId(command.CircleId);
    var circle = circleRepository.Find(circleId);

    var users = userRepository.Find(circle.Members);
    // サークルに所属しているプレミアムユーザの人数により上限が変わる
    var premiumUserNumber = users.Count(user => user.IsPremium);
    var circleUpperLimit = premiumUserNumber < 10 ? 30 : 50;
    if (circle.CountMembers() >= circleUpperLimit)
```

```
    {
      throw new CircleFullException(circleId);
    }

    (…略…)
  }
}
```

本来サークルが満員かどうかの確認はドメインのルールです。これまで解説してきたとおり、サービスにドメインのルールに基づくロジックを記述することは避けなくてはいけません。これを放置するとドメインオブジェクトは何も語らず、ドメインの重要なルールはサービスのあちらこちらに記述されるようになってしまいます。

ドメインのルールはドメインオブジェクトに定義するべきです。Circleクラスの IsFullメソッドとして定義する道を考えてみましょう。すると今度はCircleクラスがユーザ情報として識別子しか保持していないことが問題となります。ユーザの識別子からユーザ情報を取得するためにはIsFullメソッドがリポジトリを受け取る必要があります（**リスト13.3**）。

リスト13.3：エンティティがリポジトリを受け取る

```
public class Circle
{
  // プレミアムユーザの人数を探したいが保持しているのはUserIdのコレクションだけ
  public List<UserId> Members { get; private set; }

  (…略…)

  // ユーザのリポジトリを受け取る？
  public bool IsFull(IUserRepository userRepository)
  {
    var users = userRepository.Find(Members);
    var premiumUserNumber = users.Count(user => user.IsPremium);
    var circleUpperLimit = premiumUserNumber < 10 ? 30 : 50;
    return CountMembers() >= circleUpperLimit;
```

```
  }
}
```

これはあまりよくない解決法です。リポジトリはドメイン設計を完成させるといった意味ではドメインのオブジェクトですが、ドメイン由来のものではありません。Circleはドメインモデルの表現に徹していません。

エンティティや値オブジェクトがドメインモデルの表現に専念するためには、リポジトリを操作することを可能な限り避ける必要があります。

13.1.2 「仕様」による解決

エンティティや値オブジェクトにリポジトリを操作させないために取られる手段は仕様と呼ばれるオブジェクトを利用した解決です。サークルが満員かどうかを評価する処理を仕様として切り出してみましょう（**リスト13.4**）。

リスト13.4：サークルが満員かどうかを評価する仕様

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

  public CircleFullSpecification(IUserRepository userRepository)
  {
    this.userRepository = userRepository;
  }

  public bool IsSatisfiedBy(Circle circle)
  {
    var users = userRepository.Find(circle.Members);
    var premiumUserNumber = users.Count(user => user.IsPremium);
    var circleUpperLimit = premiumUserNumber < 10 ? 30 : 50;
    return circle.CountMembers() >= circleUpperLimit;
  }
}
```

仕様はオブジェクトの評価のみを行います。複雑な評価手順をオブジェクトに埋もれさせず切り出すことで、その趣旨は明確になります。

仕様を利用したときのサークルメンバー追加処理は**リスト13.5**です。

リスト13.5：仕様を利用する

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

  (…略…)

  public void Join(CircleJoinCommand command)
  {
    var circleId = new CircleId(command.CircleId);
    var circle = circleRepository.Find(circleId);

    var circleFullSpecification = new CircleFullSpecification(➡
userRepository);
    if (circleFullSpecification.IsSatisfiedBy(circle))
    {
      throw new CircleFullException(circleId);
    }

    (…略…)
  }
}
```

複雑な評価手順はカプセル化され、コードの意図は明確になっています。

趣旨が見えづらいオブジェクト

オブジェクトの評価処理を安直にオブジェクト自身に実装すると、オブジェクトの趣旨はぼやけます。オブジェクトが何のために存在し、何を為すのかが見えづらくなるのです（**リスト13.6**）。

リスト13.6：評価メソッドにまみれた定義

```
public class Circle
{
  public bool IsFull();
  public bool IsPopular();
  public bool IsAnniversary(DateTime today);
  public bool IsRecruiting();
  public bool IsLocked();
  public bool IsPrivate();
  public void Join(User user);
}
```

こうした評価の処理を放置しておくと、オブジェクトに対する依存は手の施しようがないほど増加し、変化に対して痛みを伴うようになります。

あるオブジェクトを評価する方法はメソッドに限ったことではありません。仕様のように外部のオブジェクトとして切り出すことで扱いやすくなることもあると知っておきましょう。

13.1.3 リポジトリの使用を避ける

仕様はれっきとしたドメインオブジェクトであり、その内部で入出力を行う（リポジトリを使用する）ことを避ける考えもあります。その場合はファーストクラスコレクションを利用することが選択肢に挙げられるでしょう。ファーストクラスコレクションはListといった汎用的な集合オブジェクトを利用するのではなく、特化した集合オブジェクトを用意するパターンです。

たとえばサークルのメンバー群を表現するファーストクラスコレクションは**リスト13.7**です。

リスト13.7：サークルに所属するメンバーを表すファーストクラスコレクション

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

```
  public CircleMembers(CircleId id, User owner, List<User> ➡
members)
  {
    Id = id;
    this.owner = owner;
    this.members = members;
  }

  public CircleId Id { get; }

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

  public int CountPremiumMembers(bool containsOwner = true)
  {
    var premiumUserNumber = members.Count(member => member.➡
IsPremium);
    if (containsOwner)
    {
      return premiumUserNumber + (owner.IsPremium ? 1 : 0);
    }
    else
    {
      return premiumUserNumber;
    }
  }
}
```

CircleMembersは汎用的なListと異なり、サークルの識別子と所属するメンバーをすべて保持しています。また独自の計算処理をメソッドとして定義できます。

このCircleMembersを利用した仕様は**リスト13.8**です。

リスト13.8：CircleMembersを利用した仕様

```
public class CircleMembersFullSpecification
{
  public bool IsSatisfiedBy(CircleMembers members)
  {
    var premiumUserNumber = members.CountPremiumMembers➡
(false);
    var circleUpperLimit = premiumUserNumber < 10 ? 30 : 50;
    return members.CountMembers() >= circleUpperLimit;
  }
}
```

ファーストクラスコレクションを利用すると決めた場合には、アプリケーションサービスでファーストクラスコレクションへのデータ詰め替え処理が必要になります（**リスト13.9**）。

リスト13.9：ファーストクラスコレクションに詰め替える

```
var owner = userRepository.Find(circle.Owner);
var members = userRepository.Find(circle.Members);
var circleMembers = new CircleMembers(circle.Id, owner, ➡
members);
var circleFullSpec = new CircleMembersFullSpecification();
if (circleFullSpec.IsSatisfiedBy(circleMembers)) {
   (…略…)
}
```

入出力をドメインオブジェクトから可能な限り排除することは重要です。ファーストクラスコレクションを利用した解決法は、その方針を支える手立てになるでしょう。

DDD 13.2 仕様とリポジトリを組み合わせる

仕様は単独で取り扱う以外にもリポジトリと組み合わせて活用する手法が存在します。つまり、リポジトリに仕様を引き渡して、仕様に合致するオブジェクトを検索する手法です。

リポジトリには検索を行うメソッドが定義されますが、検索処理の中には重要なルールを含むものが存在します。こうした検索処理をリポジトリのメソッドとして定義してしまうと、重要なルールはリポジトリの実装クラスに記述されてしまいます。

そういったとき、重要なルールを仕様オブジェクトとして定義し、リポジトリに引き渡せば、重要なルールがリポジトリの実装クラスに漏れ出すことを防げます。

13.2.1 お勧めサークルに見る複雑な検索処理

ユーザがサークルに参加したいと考えたとき、自分に合ったお勧めのサークルを検索できると便利です。お勧めサークルの検索機能を開発することを考えてみましょう。

お勧めサークル検索機能を作るにあたって、まずはお勧めサークルの定義を決めなくてはいけません。たとえば「活気があるサークル」や「新しく作られたばかりのサークル」など、考えられる条件はいくつもありますが、ひとまずは次の2つの条件にしたがったサークルをお勧めサークルとしましょう。

- 直近1か月以内に結成されたサークルである
- 所属メンバー数が10名以上である

お勧めサークルの定義が決まったところで次に決めるべきことは、どこにそれを記述するか、です。お勧めサークルの検索を行う処理はどこに実装すべきでしょうか。

これまでユーザやサークルの検索を実質的に行ってきたのはリポジトリでした。お勧めサークルの検索もこれと同様に、リポジトリに定義してみましょう（**リスト13.10**）。

リスト13.10：リポジトリにお勧めサークルを探し出すメソッドを追加する

```
public interface ICircleRepository
{
  (…略…)
  public List<Circle> FindRecommended(DateTime now);
}
```

FindRecommendedメソッドは引き渡された日付にしたがって、最適なサークルをいくつか見繕ってくれるメソッドです。アプリケーションサービスはFindRecommendedメソッドを利用して、ユーザにお勧めサークルを提案します（**リスト13.11**）。

リスト13.11：お勧めサークルを探し出すアプリケーションサービスの処理

```
public class CircleApplicationService
{
  private readonly DateTime now;

  (…略…)

  public CircleGetRecommendResult GetRecommend(➡
CircleGetRecommendRequest request)
  {
    // リポジトリに依頼するだけ
    var recommendCircles = circleRepository.FindRecommended➡
(now);

    return new CircleGetRecommendResult(recommendCircles);
  }
}
```

この処理自体は正しく動作しますが、ひとつ問題があります。お勧めサークルを導き出す条件がリポジトリの実装クラスに依存している点です。

お勧めサークルの条件は本来であれば重要なルールです。インフラストラクチャのオブジェクトであるリポジトリの実装クラスに左右されることは推奨されません。

リポジトリは強力なパターンです。しかしその強力さゆえに、ドメインの重要なルールをインフラストラクチャの領域に染み出させてしまうことを助長します。

13.2.2 仕様による解決法

ドメインの重要な知識はできる限りドメインのオブジェクトとして表現すべきです。お勧めサークルかどうかを判断する処理はまさにオブジェクトの評価であり、仕様として定義できます（**リスト13.12**）。

リスト13.12：お勧めサークルかどうかを見極める仕様オブジェクト

```
public class CircleRecommendSpecification
{
  private readonly DateTime executeDateTime;

  public CircleRecommendSpecification(DateTime executeDateTime)
  {
    this.executeDateTime = executeDateTime;
  }

  public bool IsSatisfiedBy(Circle circle)
  {
    if (circle.CountMembers() < 10)
    {
      return false;
    }
    return circle.Created > executeDateTime.AddMonths(-1);
  }
}
```

CircleRecommendedSpecificationはお勧めサークルかどうかを判定するオブジェクトです。お勧めサークル検索処理は**リスト13.13**になります。

リスト13.13：仕様を利用しお勧めサークルを検索する

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

  (…略…)

  public CircleGetRecommendResult GetRecommend(➡
CircleGetRecommendRequest request)
  {
    var recommendCircleSpec = new CircleRecommendSpecification➡
(now);

    var circles = circleRepository.FindAll();
    var recommendCircles = circles
      .Where(recommendCircleSpec.IsSatisfiedBy)
      .Take(10)
      .ToList();

    return new CircleGetRecommendResult(recommendCircles);
  }
}
```

このように仕立てれば、お勧めサークルの条件をリポジトリに記述する必要はなくなります。

また直接的に仕様のメソッドをスクリプト上で呼び出す以外にも、リポジトリに仕様を引き渡してメソッドを呼び出させることにより、対象となるオブジェクトを抽出させる手法もあります。この手法を採用する場合は、仕様のインターフェースを用意します（**リスト13.14**）。

リスト13.14：仕様のインターフェースと実装クラス

```
public interface ISpecification<T>
{
  public bool IsSatisfiedBy(T value);
}

public class CircleRecommendSpecification : ISpecification➡
<Circle>
{
   (…略…)
}
```

リポジトリはこのインターフェースを受け取り、結果となるセットを返却するようになります（**リスト13.15**）。

リスト13.15：リポジトリは仕様のインターフェースを受け取り結果セットを返す

```
public interface ICircleRepository
{
   (…略…)

  public List<Circle> Find(ISpecification<Circle> ➡
specification);
}
```

仕様をインターフェースにすることでリポジトリには仕様ごとにメソッドを追加定義する必要がありません。ISpecification<Circle>を実装した新たな仕様を定義すれば、ICircleRepositoryに引き渡しての検索が可能です。

リスト13.14を利用したお勧めサークル検索処理は**リスト13.16**です。

リスト13.16：リスト13.14を利用してお勧めサークルを探す

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

  (…略…)

  public CircleGetRecommendResult GetRecommend(➡
CircleGetRecommendRequest request)
  {
    var circleRecommendSpecification = new ➡
CircleRecommendSpecification(now);
    // リポジトリに仕様を引き渡して抽出（フィルタリング）
    var recommendCircles = circleRepository.Find(➡
circleRecommendSpecification)
      .Take(10)
      .ToList();

    return new CircleGetRecommendResult(recommendCircles);
  }
}
```

このように仕様を使って表現することで、お勧めサークルの条件はサービスに記述されることなく、ドメインの知識を語るオブジェクトとして存在できるのです。

13.2.3 仕様とリポジトリが織りなすパフォーマンス問題

仕様をリポジトリに引き渡す手法はルールをオブジェクトに表現しつつ、拡張性を高める有効な手段ですが、残念ながらデメリットが存在します。

リスト13.16で利用されているICircleRepositoryの実装クラスを確認してみましょう（リスト13.17）。

リスト13.17：仕様オブジェクトを受け取るリポジトリの実装

```
public class CircleRepository : ICircleRepository
{
  private readonly SqlConnection connection;

  (…略…)

  public List<Circle> Find(ISpecification<Circle> specification)
  {
    using(var command = connection.CreateCommand())
    {
      // 全件取得するクエリを発行
      command.CommandText = "SELECT * FROM circles";
      using (var reader = command.ExecuteReader())
      {
        var circles = new List<Circle>();
        while (reader.Read())
        {
          // インスタンスを生成して条件に合うか確認している（合わなければ➡
捨てられる）
          var circle = CreateInstance(reader);
          if (specification.IsSatisfiedBy(circle))
          {
            circles.Add(circle);
          }
        }
        return circles;
      }
    }
  }
}
```

仕様に合致するかはオブジェクトを生成して仕様に引き渡し、検査を行わない限りわかりません。結果としてこのコードは全件検索を行い、ひとつひとつのインス

タンスに対して条件に適合するか確認を行っています。データの総数が数件であれば大した問題にはなりませんが、数万件とデータが存在している場合には、ひどく遅い処理になってしまうことがあります。

仕様をリポジトリのフィルターとして扱うときは、パフォーマンスのことを常に考慮しておく必要があります。

13.2.4 複雑なクエリは「リードモデル」で

お勧めサークルの検索処理のように特殊な条件下にあるオブジェクトを検索したいという要求は、便利なソフトウェアを開発していく上で必ずといっていいほど必要になります。そうした操作はたいていの場合は利用者の利便性のための操作であることが多く、パフォーマンスに関する要求も高くなりがちです。

こういった事情のあるときは仕様やリポジトリといったパターンを扱わないことも視野に入ります。この章の主題である仕様から離れますが、ここで一度それについて確認しておきましょう。

リスト13.18はサークルの一覧を取得し、そのサークルのオーナーとなるユーザの情報を取得するスクリプトです。

リスト13.18：サークル一覧を取得する処理

```
public class CircleApplicationService
{
  public CircleGetSummariesResult GetSummaries➡
(CircleGetSummariesCommand command)
  {
    // 全件取得して
    var all = circleRepository.FindAll();
    // その後にページング
    var circles = all
      .Skip((command.Page - 1) * command.Size)
      .Take(command.Size);

    var summaries = new List<CircleSummaryData>();
    foreach(var circle in circles)
    {
      // サークルのオーナーを改めて検索
```

```
            var owner = userRepository.Find(circle.Owner);
            summaries.Add(new CircleSummaryData(circle.Id.Value, ➡
owner.Name.Value));
        }

        return new CircleGetSummariesResult(summaries);
    }

    (…略…)
}
```

この処理には2つの問題があります。

1つ目の問題は、冒頭でサークル集約を全件取得していることです。この処理ではページングが行われているので、すべてのサークルが必要なわけではありません。むしろ、ほとんどのサークルが不要です。せっかく再構築したインスタンスの多くは日の目を見ることなく捨てられます。

2つ目の問題はサークルに所属するユーザの検索処理が繰り返し文によって何度も行われていることです。リポジトリの具体的な実装が何で構成されているかはわかりませんが、もしもSQLであったのならばクエリが大量に発行されます。本来であればJOIN句などを活用することで発行されるクエリはたった1回で済むはずです。

リスト13.18は正しく動作するでしょうが、最適化された状態には程遠い状態です。ドメインのレイヤーにあるべき知識の流出を防ぐことを考えれば、このコードが正解であるはずなのですが、果たしてそれを理由にシステムの利用者の利便性からくる最適化の依頼を拒否してよいものでしょうか。

そもそもシステムは何のために存在するのでしょう。それは間違いなく利用者の問題を解決するためです。徹頭徹尾それは変わりません。システムはその利用者に対して友好的である必要があります。もしもシステムの利用者に対する態度が友好的でなくなれば、そのシステムはやがて使われなくなっていくでしょう。利用者を無視し続けた未来にはシステムの緩やかな死が待ち受けています。

ドメインの防衛を理由に、利用者に対して不便を強いるのは正しい道ではありません。ドメインの表現を守り、領域を保護することは大切なことですが、アプリケーションの領域はプレゼンテーション（ひいてはシステムの利用者）を強く意識する必要があります。

こういった問題は特に読み取りの際に発生します。通常のリポジトリから読み取りを行うよりもずっと複雑な読み取り動作をプレゼンテーションは要求することが

多いです。概要取得やページングはまさにその最たるものです。

複雑な読み取り動作においてパフォーマンスを起因とする懸念があった場合には、局所的にドメインオブジェクトから離れることがあります。より具体的には**リスト13.19**のようにページングしたクエリを直接実行して結果を組み立てます。

リスト13.19：最適化のために直接クエリを実行する

```
public class CircleQueryService
{
  (…略…)

  public CircleGetSummariesResult GetSummaries➡
(CircleGetSummariesCommand command)
  {
    var connection = provider.Connection;

    using (var sqlCommand = connection.CreateCommand())
    {
      sqlCommand.CommandText = @"
 SELECT
 circles.id as circleId,
 users.name as ownerName
 FROM circles
 LEFT OUTER JOIN users
 ON circles.ownerId = users.id
 ORDER BY circles.id
 OFFSET @skip ROWS
 FETCH NEXT @size ROWS ONLY
";
      var page = command.Page;
      var size = command.Size;
      sqlCommand.Parameters.Add(new SqlParameter("@skip", ➡
(page - 1) * size));
      sqlCommand.Parameters.Add(new SqlParameter("@size", ➡
size));
```

```
      using (var reader = sqlCommand.ExecuteReader())
      {
        var summaries = new List<CircleSummaryData>();
        while (reader.Read())
        {
          var circleId = (string) reader["circleId"];
          var ownerName = (string) reader["ownerName"];
          var summary = new CircleSummaryData(circleId, ➡
ownerName);
          summaries.Add(summary);
        }

        return new CircleGetSummariesResult(summaries);
      }
    }
  }
}
```

リスト13.19はSQLを用いたモジュールですが、もちろんORMなどを利用したモジュールでも構いません（**リスト13.20**）。

リスト13.20：ORM（EntityFramework）を利用する

```
public class EFCircleQueryService
{
  private readonly MyDbContext context;

  public EFCircleQueryService(MyDbContext context)
  {
    this.context = context;
  }

  public CircleGetSummariesResult GetSummaries➡
(CircleGetSummariesCommand command)
  {
```

```
    var all =
        from circle in context.Circles
        join owner in context.Users
        on circle.OwnerId equals owner.Id
        select new { circle, owner };

    var page = command.Page;
    var size = command.Size;

    var chunk = all
      .Skip((page - 1) * size)
      .Take(size);

    var summaries = chunk
      .Select(x => new CircleSummaryData(x.circle.Id, ➡
x.owner.Name))
      .ToList();

    return new CircleGetSummariesResult(summaries);
  }
}
```

プレゼンテーション側の要求に特化したユースケースの提供をすることは便利なシステムを提供する上で必ずといってよいほど必要になります。

読み取り（クエリ）で要求されるデータは複雑ですが、その動作自体は単純でドメインのロジックと呼べるものがほとんどありません。反対に書き込み（コマンド）はドメインとしての制約が多く存在します。このことから、コマンドにおいてはドメインを隔離するためにドメインオブジェクトやそれに関わるものを積極的に利用し、クエリにおいてはある程度の緩和をすることがあります。本書で詳しくは取り扱いませんが、この考えはCQS（Command-query separation）やCQRS（Command Query Responsibility Segregation）といった考えに基づくものです。オブジェクトのメソッドをコマンドとクエリに大別し、それらを別個に扱うというこれらの考えはプレゼンテーションが要求するパフォーマンスを維持しながら、システムを統制することに寄与する考えです。

COLUMN 遅延実行による最適化

リポジトリを利用しつつもパフォーマンス問題を解決する手法として遅延実行という手段が挙げられます。遅延実行を実現する場合、リポジトリに定義する検索メソッドはList型ではなく、IEnumerable型を戻り値とします（**リスト13.21**）。

リスト13.21：遅延実行を考慮したリポジトリ

```
public interface ICircleRepository
{
  IEnumerable<Circle> FindAll();
}
```

IEnumerableはコレクション操作を行える型ですが、実際にコレクションに対する操作を行うまでコレクションを確定しません（厳密には実装クラス次第です）。**リスト13.18**の前半部分にこのリポジトリを適用したときを例にして確認してみましょう（**リスト13.22**）。

リスト13.22：**リスト13.18**の前半部分に適用

```
public class CircleApplicationService
{
  // リスト 13.21のリポジトリ
  private readonly ICircleRepository circleRepository;

  public CircleGetSummariesResult GetSummaries➡
(CircleGetSummariesCommand command)
  {
    // この段階ではデータを取得しない
    var all = circleRepository.FindAll();
    // ページング処理は条件を付与しているに過ぎないためデータを取得しない
    var chunk = all
      .Skip((command.Page - 1) * command.Size)
      .Take(command.Size);
```

```
    // ここではじめてコレクションが処理されるため、条件に応じて➡
データ取得がされる
    var summaries = chunk
        .Select(x =>
        {
          var owner = userRepository.Find(x.Owner);
          return new CircleSummaryData(x.Id.Value, ➡
owner.Name.Value);
        })
        .ToList();

    return new CircleGetSummariesResult(summaries);
  }
}
```

リスト13.22では最初期にFindAllメソッドを利用して全サークルのコレクションを取得していますが、この段階ではコレクションの内容を参照する必要がないため、実際のデータ取得は実施されません。また、その後に続くページング処理もコレクションを取得する際の条件を付与しているに過ぎず、実際にデータを参照する必要がありません。リスト13.22で実際にデータ取得が行われるタイミングはToListメソッドによりコレクションが確定されるときです。このとき、操作対象にはページングの条件処理が付与されているため、データ取得前にページングが行われ、無駄なデータ取得が発生しなくなります。

このように、実際に必要となるまで処理を実行しないことを遅延実行といいます。遅延実行を利用するとクエリの発行を本当に必要になるまで遅らせることができます。それゆえ、ロジックの最初期段階で全件取得を行い、条件を付与して取得範囲を狭めていく処理を組み立てやすくなります。

実際に、C#のデータ操作ライブラリであるEntityFrameworkはこの動作をサポートしています。ただし、それをあてにすることは、コードが特定の技術基盤に依存することを意味します。リポジトリの実装クラスがEntityFrameworkをベースとしたものであり続ければよいですが、もし、そうでなくなったときを考えると、悩ましい問題であることは間違いないでしょう。

DDD 13.3 まとめ

オブジェクトの評価はそれ単体で知識になりえます。仕様は評価の条件や手順をモデルにしたオブジェクトです。

オブジェクトの評価をオブジェクト自身にさせることが常に正しいとは限りません。仕様のような外部のオブジェクトに評価させる手法の方が素直なコードになることも多いでしょう。

本章では、仕様をリポジトリに引き渡してフィルタリングを行う手法も紹介しました。残念ながらこれは銀の弾丸ではなく、パフォーマンス上の問題が付いて回ります。

読み取り操作は単純ながら最適化が求められることも多いです。読み取り操作においてはドメインという考え方を棚上げにして、クライアントが利用しやすい形で提供することもあると覚悟しておく必要があります。