Chapter 14 アーキテクチャ

ドメイン駆動設計と同時に語られるアーキテクチャを解説します。

ドメイン駆動設計はドメインと向き合いながらモデルをコードに落とし込むことで、ドメインとコードを結びつけるプラクティスです。したがって、ドメイン駆動設計は特定のアーキテクチャを前提とすることはありません。それにもかかわらず、ドメイン駆動設計と同時にアーキテクチャが語られることは多いです。なぜアーキテクチャの話が出てくるのでしょうか。

これまでドメインモデル貧血症などの例を通じて、重大なルールが漏れ出した結果が引き起こす弊害を訴えてきました。ソフトウェアが利用者の役に立ち続けるものであり続けるには、継続的な改良に耐えうる構造でなくてはなりません。ひとつのモデルの変更がいくつものオブジェクトの変更に繋がるようでは、改良に二の足を踏むのも致し方ないことでしょう。

アーキテクチャはこれを解決するものです。アーキテクチャは知識を記述すべき箇所を示す方針です。ドメインのルールが流出することを予防するのと同時に一箇所にまとめるよう促します。ドメイン駆動設計とアーキテクチャが同時に語られる理由はまさにそこにあります。

DDD 14.1 アーキテクチャの役目

アーキテクチャは開発者の心を躍らせるものです。開発者はいつだって整然とした理論や仕組みに心惹かれるものです。

だからこそ、そこに水を差すのは憚られるのですが、アーキテクチャに対する態度について論じる必要があります。つまり、ドメイン駆動設計にとって「アーキテクチャは決して主役ではない」ことについてです。

この言葉の真意を理解するために、アーキテクチャ自体を学ぶ前に、まずはドメイン駆動設計にとってアーキテクチャがどのような立ち位置であるかを確認していきましょう。

14.1.1 アンチパターン：利口なUI

システムを致命的に硬直したものに仕立てるアンチパターンのひとつが「利口なUI（Smart UI）」です。利口なUIは本来であればドメインオブジェクトに記載されるべき重要なルールやふるまいが、ユーザーインターフェースに記述されてしまっている状態を揶揄しています。

利口なUIはドメインを分離することが適わなかったアプリケーションに多く見られます。そうしたシステムは改良に対するコストが異常に高くなってしまい、結果としてひどく硬直したソフトウェアになってしまっています。

たとえばECサイトのシステムを例に考えてみましょう。

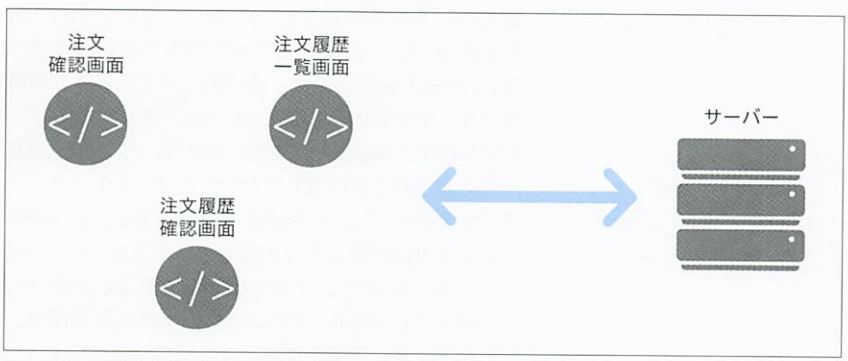

図14.1：典型的なECシステム

ECサイトでは利用者は商品をウェブサイト上から注文します。図14.1は主に利用者が商品を注文する際に利用するシステムの一部を表しています。

これらの画面には共通するビジネスロジックが存在します。ひとつずつ、システムの利用者の行動を追いながら確認していきましょう。

まずシステムの利用者は、注文したい商品を選択したのちに、商品を注文するために"注文確認画面"へ遷移します。"注文確認画面"では、これから行う注文の「合計金額」が表示されます。"注文確認画面"に表示されている内容が問題なければ、利用者は注文を確定します。

自分が注文した内容が間違いないかを確認するために、利用者は"注文履歴一覧画面"を開きます。この画面はこれまで行った注文の概要が一覧となって表示されています。このとき、注文それぞれに「合計金額」が表示されていると便利です。"注文履歴一覧画面"には「合計金額」が表示されています。

"注文履歴一覧画面"では注文ひとつひとつの具体的な内容がわかりません。システムの利用者は注文内容の詳細を確認するために"注文履歴確認画面"を開きます。もちろん、そこにも「合計金額」が表示されています。

ここで紹介したいずれの画面においても「合計金額」が表示されています。したがって、「合計金額の計算」をそれらすべての画面で行う必要があります。「合計金額の計算」処理はどこに記述されるべきか考えてみましょう。

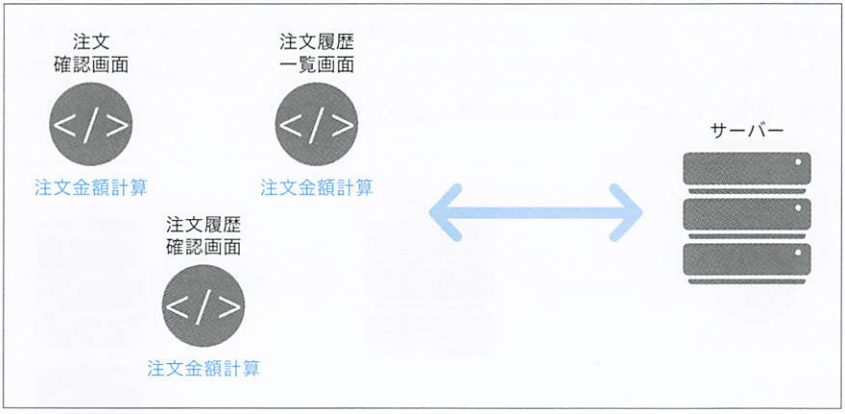

図14.2：UIに記述されるビジネスロジック

図14.2はもっとも短絡的な構成を示しています。3つの画面それぞれで直接「合計金額の計算」をすることで、各画面に「合計金額」を表示しています。

これは多くの問題を抱えています。たとえば、もしも「合計金額の計算」処理が変更されることになったときを考えてみてください。

まず弊害として挙げられるのは、「合計金額の計算」はひとつでありながら、その変更が3箇所に渡ることです。本来、正しくロジックがまとめられていれば、たった1箇所の修正で済んでいたはずなのに、作業量が3倍となってしまいます。

また、ひとつひとつの画面は似たような画面であってもそれぞれの事情は異なります。システムは成長していくものです。年月が経つにつれ、当初はまったく同じであったはずの「合計金額の計算」は、それぞれの画面特有の事情によって独自に成長していくでしょう。同じように見えるコードはわずかに異なり、慎重に修正を加える必要ができてしまいます。

もちろんひとりの開発者がこれら3つの画面を同時期に手掛けるのであれば、わざわざ同じ処理をそれぞれの画面に記述するようなことはしないでしょう。たいていの場合、開発者はロジックをまとめることに熱心です。3つの画面で行われる「合計金額の計算」がまったく同じ処理であることに気づき、共通化を図ることができます。

真に問題となるのは最初時点で画面が1つしかなかったときです。先の例でいえば、最初は"注文確認画面"しか存在せず、後から注文履歴を確認する機能が追加され、"注文履歴一覧画面"と"注文履歴確認画面"の2つの画面を開発することになったときです。

ロジックをまとめることだけに関心がある開発者にとって"注文確認画面"しか存在しないということは、「合計金額の計算」が"注文確認画面"に記述されることは肯定されます（**図14.3**）。このとき同じ計算を他の箇所でも利用することが発生したときのことが頭の片隅に過ぎることはあるでしょうが、それはそのときが来たときに実施すればよい、と楽観的に考えるのです。

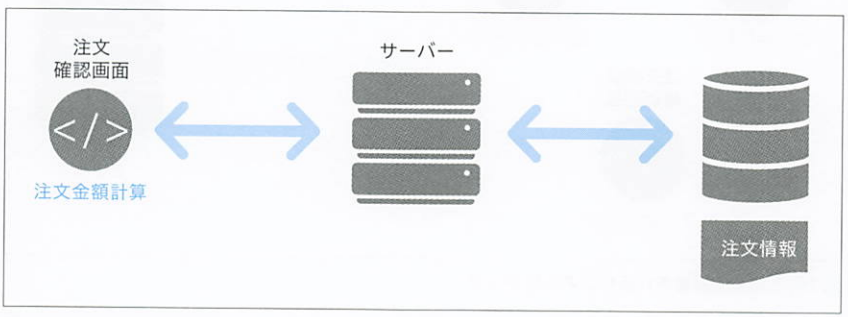

図14.3：重複が起きていない状態

開発者は「一事が万事」といった言葉に目を向けず、いつかリファクタリングをすべきタイミングが訪れる、という夢を信じがちです。同時に、正しい形は見えていて、いかなるときであっても容易に書き換えられるという無謀な自信に満ち溢れ

ています。もちろんそんなときは訪れません。短絡的な解決を図ったコードは複雑怪奇な進化を遂げて、いつの日かあなたの前に立ちはだかるのです。

UIは入力と表示がその責務です。ビジネスに関わるようなロジックは可能な限り記述されるべきではありません。UIはできるだけ愚かであるべきです。UIが利口になればなるほど改修の多くはコストがかさみ、その修正は痛みを伴うものになります。その痛みは開発者を怯えさせ、システムは段々と硬直していきます。

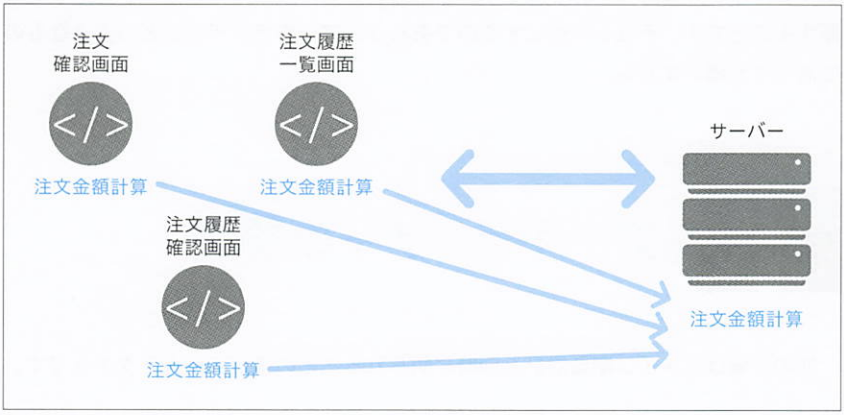

図14.4：ビジネスロジックの集約

利口なUIを捨て、ビジネスロジックを1箇所に集約すると図14.4になります。この構成であれば、画面固有の事情が「注文金額の計算」に入り込む余地はありませんし、計算に変更があったとしても1箇所の改修で済みます。もちろん修正の難易度もそう難しいものではないでしょう。

まずは、いま開発しているソフトウェアがひどく単純なものであるという先入観を捨てることから始めるべきです。ユーザインターフェースにビジネスロジックを記述することが肯定された段階で、豊かなドメインモデルを育てていく輝かしい道のりは閉ざされます。

14.1.2 ドメイン駆動設計がアーキテクチャに求めること

利口なUIを避けることに決めたとしても、それを実際に実践するのはそう簡単なことではありません。ビジネスロジックを正しい場所に配置し続けることは、いかにその大切さを熟知している開発者であっても難しいものなのです。それゆえ、開発者に強い自制心を促す以外の方法を考える必要があります。アーキテクチャはその解決策です。

アーキテクチャは方針です。何がどこに記述されるべきかといった疑問に対する解答を明確にし、ロジックが無秩序に点在することを防ぎます。開発者はアーキテクチャが示す方針にしたがうことで「何をどこに書くのか」に振り回されないようになります。これは開発者がドメイン駆動設計の本質である「ドメインを捉え、うまく表現する」ことに集中するために必要なことです。

ドメイン駆動設計がアーキテクチャに求めることは、ドメインオブジェクトが渦巻くレイヤーを隔離して、ソフトウェア特有の事情からドメインオブジェクトを防衛することです。それを可能にするのであれば、アーキテクチャがどのようなものであっても構いません。

DDD 14.2 アーキテクチャの解説

次の一覧はドメイン駆動設計と同時に語られることの多いアーキテクチャです。

- レイヤードアーキテクチャ
- ヘキサゴナルアーキテクチャ
- クリーンアーキテクチャ

本書ではこれらのアーキテクチャを解説しますが、ドメイン駆動設計にとってはドメインが隔離されることのみが重要であり、必ずしもこのいずれかのアーキテクチャにしたがわなければいけないというわけではありません。またアーキテクチャにしたがったからといって、それすなわちドメイン駆動設計を実践したことにはならないということも知っておくべきことです。

重要なのは、ドメインの本質に集中する環境を用意することです。

14.2.1 レイヤードアーキテクチャとは

レイヤードアーキテクチャはドメイン駆動設計の文脈で登場するアーキテクチャの中で、もっとも伝統的でもっとも有名なアーキテクチャです。

レイヤードアーキテクチャはその名のとおり、いくつかの層が積み重なる形で表現されます。

たとえば書籍『エリック・エヴァンスのドメイン駆動設計』では図14.5が提示されています。

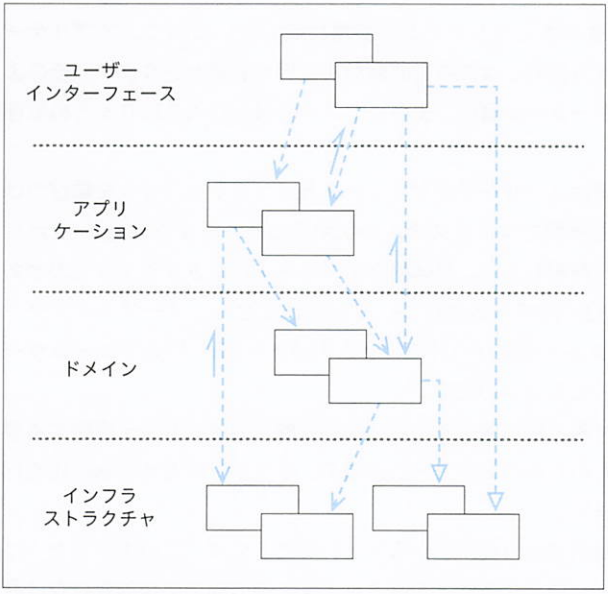

図14.5：エリック・エヴァンスが示したレイヤードアーキテクチャ

出典：『エリック・エヴァンスのドメイン駆動設計』（翔泳社）、P.66より引用

ドメイン駆動設計の文脈で紹介されるレイヤードアーキテクチャは図14.5のとおり、4つの層で構成されたものが代表的です。この図に倣って確認していきましょう。

レイヤードアーキテクチャを構成する4つの層の内訳は次のとおりです。

- プレゼンテーション層（ユーザーインターフェース層）
- アプリケーション層
- ドメイン層
- インフラストラクチャ層

ドメイン層はこの中でもっとも重要な層です。ソフトウェアを適用しようとしている領域で問題解決に必要な知識を表現します。この層を明示的にして、ドメイン層に本来所属すべきドメインオブジェクトの隔離を促し、他の層へ流出しないようにします。

アプリケーション層はドメイン層の住人を取りまとめる層です。この層の住人はアプリケーションサービスが挙げられます。アプリケーションサービスはドメインオブジェクトの直接のクライアントとなり、ユースケースを実現するための進行役になります。ドメイン層の住人はドメインの表現に徹しているので、アプリケーションとして成り立たせるためには彼らを問題解決に導く必要があります。そのような働きをするアプリケーション層は、まさにドメイン層の住人を取りまとめる存在です。

プレゼンテーション層はユーザーインタフェースとアプリケーションを結びつけます。主な責務は表示と解釈です。システムの利用者にわかるように表示を行い、システム利用者の入力を解釈します。具体的なアプローチにはさまざまなものがありますが、それが具体的に何であるかについては問いません。ユーザーインターフェースとアプリケーションを結びつけることさえできれば、Webフレームワークであってもcliであってもよいのです。

インフラストラクチャ層は他の層を支える技術的基盤へのアクセスを提供する層です。アプリケーションのためのメッセージ送信や、ドメインのための永続化を行うモジュールが含まれます。

ここに存在する原則は依存の方向が上から下ということです。上位のレイヤーは自身より下位のレイヤーに依存することが許されます。逆方向の直接的な依存は許されません。

依存の観点で考えると、ドメイン層からインフラストラクチャ層に依存の矢印が伸ばされていることが奇妙に見えることでしょう。これはドメイン層のオブジェクトがインフラストラクチャ層のオブジェクトを取り扱うことを意味していません。**図14.5**の右下に位置する白抜きの矢印を確認すれば、汎化が含まれていることがわかります。ちょうどリポジトリのインターフェースと実装クラスの関係がこの矢印にあたるでしょう。

また、『エリック・エヴァンスのドメイン駆動設計』にはレイヤー間のオブジェクト同士の結びつきを表した図が登場します（**図14.6**）。

「a234：口座」がaddToUnitOfWork(a234)というメッセージを「作業ユニットマネージャ」に送っていることから、古典的なユニットオブワークの実装（**10.4.4**項）を想定していることがわかります。昨今ではあまり利用されることのないパターンですが、ドメインオブジェクトがインフラストラクチャ層のオブジェクトに依存している例として挙げられるでしょう。

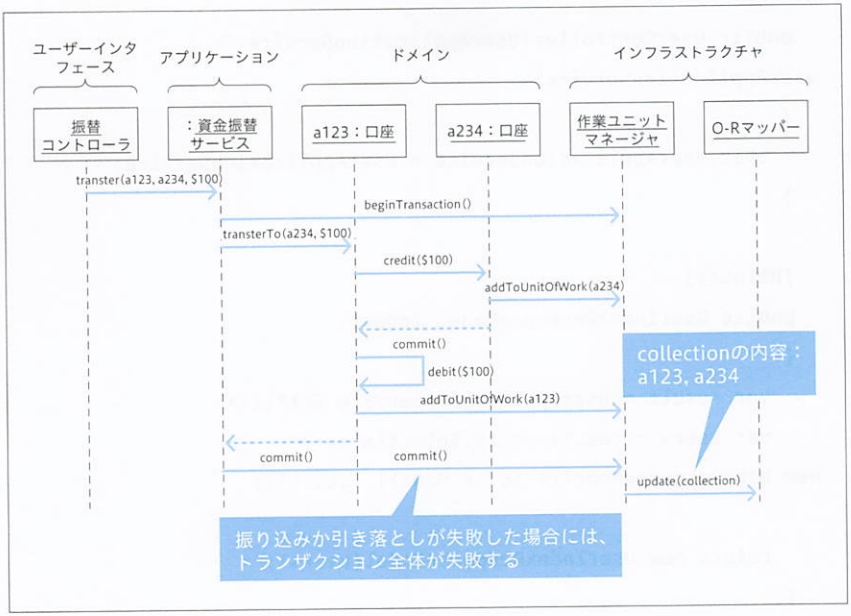

図14.6：オブジェクト同士の結びつきを示した図

出典：『エリック・エヴァンスのドメイン駆動設計』（翔泳社）、P.70の図4.1より引用

レイヤードアーキテクチャの実装サンプル

本書で解説をしながら実装してきたアプリケーションは実はレイヤードアーキテクチャを意識して実装されたものです。改めてレイヤードアーキテクチャの観点でコードを観察してみましょう。

最初に確認をするのはプレゼンテーション層に所属するUserControllerです（**リスト14.1**）。

リスト14.1：プレゼンテーション層の住人であるコントローラ

```
[Route("api/[controller]")]
public class UserController : Controller
{
  private readonly UserApplicationService ➡
userApplicationService;
```

```
    public UserController(UserApplicationService ➡
userApplicationService)
    {
        this.userApplicationService = userApplicationService;
    }

    [HttpGet]
    public UserIndexResponseModel Index()
    {
        var result = userApplicationService.GetAll();
        var users = result.Users.Select(x => ➡
new UserResponseModel(x.Id, x.Name)).ToList();

        return new UserIndexResponseModel(users);
    }

    [HttpGet("{id}")]
    public UserGetResponseModel Get(string id)
    {
        var command = new UserGetCommand(id);
        var result = userApplicationService.Get(command);

        var userModel = new UserResponseModel(result.User);

        return new UserGetResponseModel(userModel);
    }

    [HttpPost]
    public UserPostResponseModel Post([FromBody] ➡
UserPostRequestModel request)
    {
        var command = new UserRegisterCommand(request.UserName);
        var result = userApplicationService.Register(command);
```

```
    return new UserPostResponseModel(result.CreatedUserId);
  }

  [HttpPut("{id}")]
  public void Put(string id, [FromBody] UserPutRequestModel ➡
request)
  {
    var command = new UserUpdateCommand(id, request.Name);
    userApplicationService.Update(command);
  }

  [HttpDelete("{id}")]
  public void Delete(string id)
  {
    var command = new UserDeleteCommand(id);
    userApplicationService.Delete(command);
  }
}
```

HTTPリクエストという利用者からの入力データをアプリケーションに伝えるための変換を行っているMVCフレームワークのコントローラは、まさに入力を解釈してアプリケーションに結びつけるプレゼンテーション層の住人です。アプリケーション層の住人であるアプリケーションサービスのクライアントにもなっているので、**図14.5**で示した依存の方向性が守られています。

次にアプリケーション層に所属するアプリケーションサービスを確認してみましょう(**リスト14.2**)。

リスト14.2：アプリケーション層の住人であるアプリケーションサービス

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

```
  public UserApplicationService(IUserFactory userFactory, ➡
IUserRepository userRepository, UserService userService)
  {
    this.userFactory = userFactory;
    this.userRepository = userRepository;
    this.userService = userService;
  }

  public UserGetResult Get(UserGetCommand command)
  {
    var id = new UserId(command.Id);
    var user = userRepository.Find(id);
    if (user == null)
    {
      throw new UserNotFoundException(id, ➡
"ユーザが見つかりませんでした。");
    }

    var data = new UserData(user);

    return new UserGetResult(data);
  }

  public UserGetAllResult GetAll()
  {
    var users = userRepository.FindAll();
    var userModels = users.Select(x => ➡
new UserData(x)).ToList();
    return new UserGetAllResult(userModels);
  }

  public UserRegisterResult Register(UserRegisterCommand ➡
command)
  {
```

```
    using (var transaction = new TransactionScope())
    {
      var name = new UserName(command.Name);
      var user = userFactory.Create(name);
      if (userService.Exists(user))
      {
        throw new CanNotRegisterUserException(user, ➡
"ユーザは既に存在しています。");
      }

      userRepository.Save(user);

      transaction.Complete();

      return new UserRegisterResult(user.Id.Value);
    }
  }

  public void Update(UserUpdateCommand command)
  {
    using (var transaction = new TransactionScope())
    {
      var id = new UserId(command.Id);
      var user = userRepository.Find(id);
      if (user == null)
      {
        throw new UserNotFoundException(id);
      }

      if (command.Name != null)
      {
        var name = new UserName(command.Name);
        user.ChangeName(name);
```

```
        if (userService.Exists(user))
        {
          throw new CanNotRegisterUserException(user, ➡
"ユーザは既に存在しています。");
        }
      }

      userRepository.Save(user);

      transaction.Complete();
    }
  }

  public void Delete(UserDeleteCommand command)
  {
    using (var transaction = new TransactionScope())
    {
      var id = new UserId(command.Id);
      var user = userRepository.Find(id);
      if (user == null)
      {
        return;
      }

      userRepository.Delete(user);

      transaction.Complete();
    }
  }
}
```

アプリケーションサービスはその名に冠しているとおりアプリケーション層に所属するオブジェクトです。自身のレイヤーより下位に位置するドメイン層とインフラストラクチャ層に対して依存をしています。

アプリケーション層の目的はアプリケーションサービスの目的と一致しています。すなわち問題を解決するためにドメインオブジェクトが実施するタスクの進行管理を行います。場合によっては他のサービスと協調することもあります。

このレイヤーで注意すべきはドメインのルールやふるまいを直接的に記述してはいけないことです。ビジネスの重要なルールやふるまいはドメイン層に収められるべき事柄です。

次に確認するドメイン層はもっとも重要な層です。ユーザのコード上の表現であるUserクラスやドメインサービスであるUserServiceクラスはまさにこの層に所属するオブジェクトです(**リスト14.3**)。

リスト14.3：ドメイン層の住人であるエンティティやドメインサービス

```
public class User
{
  public User(UserId id, UserName name, UserType type)
  {
    if (id == null) throw new ArgumentNullException(nameof(id));
    if (name == null) throw new ArgumentNullException(nameof➡
(name));

    Id = id;
    Name = name;
    Type = type;
  }

  public UserId Id { get; }
  public UserName Name { get; private set; }
  public UserType Type { get; private set; }

  public bool IsPremium => Type == UserType.Premium;

  public void ChangeName(UserName name)
  {
    if (name == null) throw new ArgumentNullException(nameof➡
(name));
```

```
    Name = name;
  }

  public void Upgrade()
  {
    Type = UserType.Premium;
  }

  public void Downgrade()
  {
    Type = UserType.Normal;
  }

  public override string ToString()
  {
    var sb = new ObjectValueStringBuilder(nameof(Id), Id)
      .Append(nameof(Name), Name);

    return sb.ToString();
  }
}

public class UserService
{
  private readonly IUserRepository userRepository;

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

  public bool Exists(User user)
  {
    var duplicatedUser = userRepository.Find(user.Name);
```

```
    return duplicatedUser != null;
  }
}
```

ドメインモデルに表現するコードはすべてこの層に集中します。また、ドメインオブジェクトをサポートする役割のあるファクトリやリポジトリのインターフェースもこの層に含まれます。

最後に確認するインフラストラクチャ層のオブジェクトは永続化を実施するリポジトリです（**リスト14.4**）。

リスト14.4：インフラストラクチャ層の住人であるリポジトリ

```
public class EFUserRepository : IUserRepository
{
  private readonly ItdddDbContext context;

  public EFUserRepository(ItdddDbContext context)
  {
    this.context = context;
  }

  public User Find(UserId id)
  {
    var target = context.Users.Find(id.Value);
    if (target == null)
    {
      return null;
    }

    return ToModel(target);
  }

  public List<User> Find(IEnumerable<UserId> ids)
  {
```

```
    var rawIds = ids.Select(x => x.Value);

    var targets = context.Users
      .Where(userData => rawIds.Contains(userData.Id));

    return targets.Select(ToModel).ToList();
  }

  public User Find(UserName name)
  {
    var target = context.Users
      .FirstOrDefault(userData => userData.Name == name.Value);
    if (target == null)
    {
      return null;
    }

    return ToModel(target);
  }

  (…略…)
}
```

インフラストラクチャ層には**リスト14.4**のようにドメインオブジェクトを直接的に支える技術的機能の他に、アプリケーション層やプレゼンテーション層のための技術的機能を担うオブジェクトも含まれます。

14.2.2 ヘキサゴナルアーキテクチャとは

ヘキサゴナルアーキテクチャは六角形をモチーフとした**図14.7**に代表されるアーキテクチャです。コンセプトはアプリケーションとそれ以外のインターフェースや保存媒体は付け外しできるようにするというものです。ヘキサゴナルアーキテクチャが成し遂げようとしていることを説明するにあたって、ゲーム機はよい例えです。

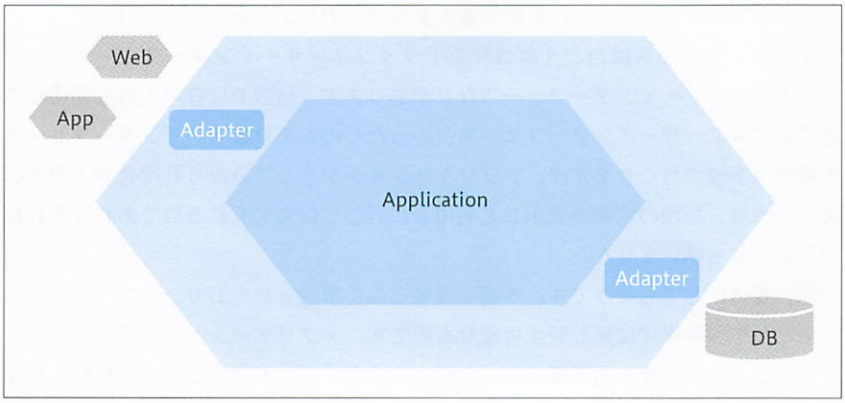

図14.7：ヘキサゴナルアーキテクチャ

ゲーム機にはコントローラやモニターといった利用者が直接触れることのできるインターフェースが存在します（図14.8）。ゲームコントローラにはいわゆる純正品がありますが、利用者の好みによってサードパーティ製のものを差し込んでもうまく動作します。モニターもコントローラと同じく、メーカーや液晶かプラズマかといった描画手法など細かい点が異なりますが、いずれにせよゲーム機にとっては些細な違いです。表示さえできればそれで十分です。

記憶媒体はどうでしょうか。昨今のゲーム機は内蔵されたハードディスクにゲームデータを保存する選択肢以外に、クラウド上に保存する選択肢も用意されています。ゲーム機からするとゲームデータの保存さえできるのであれば、実際に保存される媒体が何であっても構わないのです。

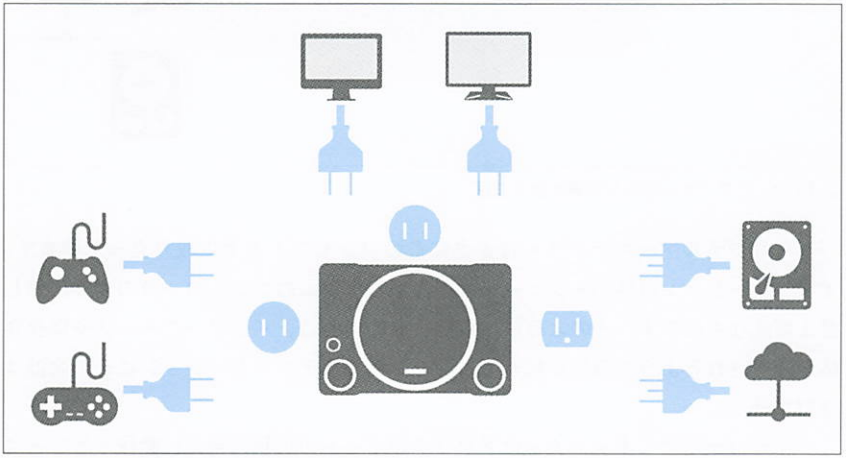

図14.8：ゲーム機と付け外し可能なアダプタ

これをアプリケーションに転用するとどうなるでしょうか（図14.9）。

インターフェースはたとえばコマンドラインユーザーインターフェースやグラフィカルユーザーインターフェースなどがあります。最近では音声入力も発達してきてボイスユーザーインターフェースといったものもあります。インターフェースの種類は多岐にわたりますが、アプリケーションからしてみたら利用者の入力を伝えてくれて、処理の結果を表示して利用者に対して伝えてくれるのであればそれが何であっても構いません。

保存媒体にしてもそうです。本書で幾度となく解説したとおり、アプリケーションにとって保存媒体は差し替え可能なものです。アプリケーションが求めるのはインスタンスの永続化と再構築です。それさえこなすことができるのであれば、具体的な保存媒体がデータベースなのか、それとも磁気テープであるのかといったことは些細な問題です。

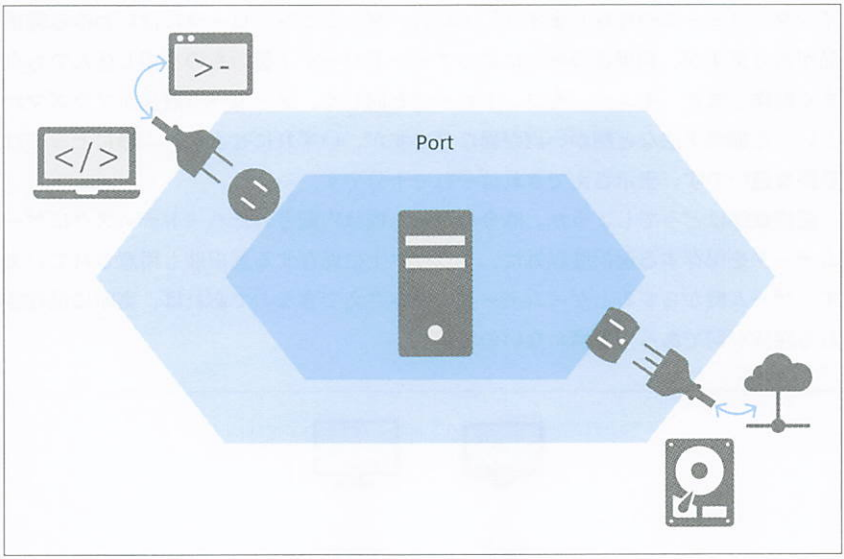

図14.9：アプリケーションに置き換えた図

ヘキサゴナルアーキテクチャはまさにこのコンセプトの上に成り立っています。アプリケーション以外のモジュールはさながらゲームのコントローラのように差し替え可能なものです。そのように仕立てられていれば、インターフェースや保存媒体が変更されるような事態が起きても、コアとなるアプリケーションにその余波は及びません。

ヘキサゴナルアーキテクチャはアダプタがポートの形状に合えば動作することに見立てて、ポートアンドアダプタと呼ばれることもあります。このとき、アプリ

ケーションに対する入力を受けもつポートとアダプタをそれぞれプライマリポートとプライマリアダプタといいます。反対にアプリケーションが外部に対してインタラクトするポートをセカンダリポートと表現し、実装するオブジェクトをセカンダリアダプタと呼びます。

実をいうとこれまで見てきたコードはヘキサゴナルアーキテクチャのコンセプトを達成しています。レイヤードアーキテクチャの紹介をしているときに例示したアプリケーションサービスであるUserApplicationServiceを確認してみましょう(**リスト14.5**)。

リスト14.5：ユーザアプリケーションサービスのコード

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

   (…略…)

  public void Update(UserUpdateCommand command)
  {
    using (var transaction = new TransactionScope())
    {
      var id = new UserId(command.Id);
      var user = userRepository.Find(id);
      if (user == null)
      {
        throw new UserNotFoundException(id);
      }

      if (command.Name != null)
      {
        var name = new UserName(command.Name);
        user.ChangeName(name);

        if (userService.Exists(user))
        {
```

```
            throw new CanNotRegisterUserException(user, ➡
"ユーザは既に存在しています。");
          }
        }

        // セカンダリポートであるIUserRepositoryの処理を呼び出す
        // 処理は実体であるセカンダリアダプタに移る
        userRepository.Save(user);

        transaction.Complete();
      }
    }
}
```

ユーザ情報の更新であるUpdateメソッドを呼び出すクライアントはプライマリアダプタで、Updateメソッドはプライマリポートにあたります。プライマリアダプタはアプリケーションを操作するための値をプライマリポートが求めるUserUpdateCommandに変換し、アプリケーションを呼び出します。

アプリケーションはIUserRepositoryというセカンダリポートを呼び出すことで、具体的な実装（セカンダリアダプタ）からインスタンスを再構築したり、永続化を依頼します。

先に紹介したレイヤードアーキテクチャとの違いとして挙げられるのは、インターフェースを利用した依存関係の整理に言及している点です。レイヤードアーキテクチャは層分けを言及しているに過ぎないので、インターフェースを取り扱うかはまったく任意です。とはいえ、昨今のシステム開発の現場においてはレイヤードアーキテクチャでインターフェースを利用して、依存関係の逆転を達成することは当たり前のように行われているため、両者の垣根はほとんどないように感じられます。

14.2.3 クリーンアーキテクチャとは

クリーンアーキテクチャは4つの同心円が特徴的な図によって説明されるアーキテクチャです（図14.10）。

図14.10が示すのはビジネスルールをカプセル化したモジュールを中心に据えるというコンセプトです。図中のEntitiesはドメイン駆動設計のエンティティを示

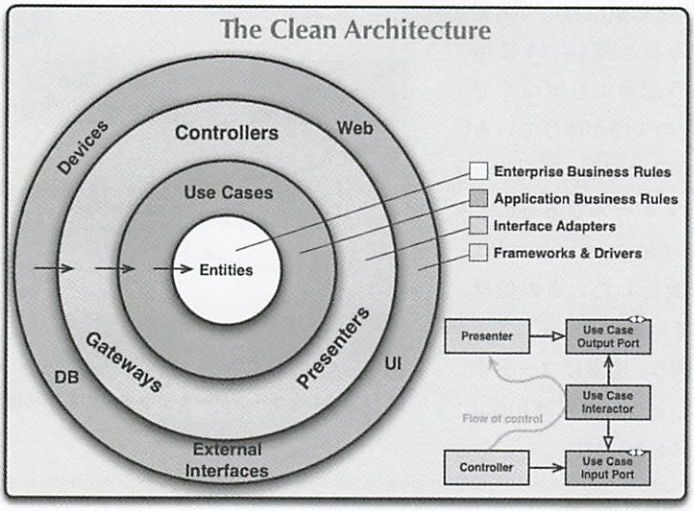

図14.10：クリーンアーキテクチャ

出典：「The Clean Code Blog」より引用
URL：https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

しません。クリーンアーキテクチャの文脈で語られるエンティティはビジネスルールをカプセル化したオブジェクトないし、データ構造と関数のセットを指すので、どちらかといえばドメインオブジェクトに近い概念です。

クリーンアーキテクチャはユーザインターフェースやデータストアなどの詳細を端に追いやり、依存の方向を内側に向けることで、詳細が抽象に依存するという依存関係逆転の原則を達成します。

この説明を聞くと勘のよい方であれば次のことに気づくでしょう。すなわち、ヘキサゴナルアーキテクチャと目的としているところが同じであることです。コンセプトが同じであれば、具体的に違う箇所はどこなのかというのは気になるところです。

ヘキサゴナルアーキテクチャとクリーンアーキテクチャの大きな違いはその実装の仕方が詳しく言及されているか否かです。ヘキサゴナルアーキテクチャはポートとアダプタによりつけ外しを可能にするという方針だけがありました。それに比べてクリーンアーキテクチャには、コンセプトを実現する具体的な実装方針が明示されています。

図14.10の右下の図に着目してください。この図こそが具体的な実装方法を表しています。

まずは矢印を見てみましょう。この矢印はよく見ると2種類存在しています。片方は普通の矢印、もう片方は白抜きの矢印。これはそれぞれ依存と汎化を表してい

ます。その観点を念頭において図を改めて眺めてみると**図14.11**に示す＜I＞という記号にも気づくでしょうか。汎化の矢印が伸びていることからもわかるとおり、モジュールがインターフェースであることを示す印です。Flow of controlはプログラムを実行したときの処理の流れを示しています。

右下の図に則り、実際にコードを実装してみましょう。まずはInput Portです（**リスト14.6**）。

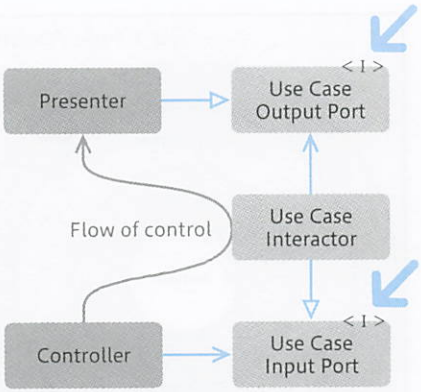

図14.11：クリーンアーキテクチャの右下の図

リスト14.6：InputPortの実装

```
public interface IUserGetInputPort
{
  public void Handle(UserGetInputData inputData);
}
```

InputPortはクライアントのためのインターフェースです。コントローラから呼び出されます。

Interactorはこの InputPortを実装してユースケースを実現します（**リスト14.7**）。

リスト14.7：Interactorの実装

```
public class UserGetInteractor : IUserGetInputPort
{
  private readonly IUserRepository userRepository;
  private readonly IUserGetPresenter presenter;

  public UserGetInteractor(IUserRepository userRepository, ➡
IUserGetPresenter presenter)
  {
    this.userRepository = userRepository;
```

```
    this.presenter = presenter;
  }

  public void Handle(UserGetInputData inputData)
  {
    var targetId = new UserId(inputData.userId);
    var user = userRepository.Find(targetId);

    var userData = new UserData(user.Id.Value, user.Name.Value);
    var outputData = new UserUpdateOutputData(userData);
    presenter.Output(outputData);
  }
}
```

UserGetInteractorはちょうどアプリケーションサービスのメソッドをそのままクラスにしたものです。これまでのアプリケーションサービスと異なる点は、結果を出力する先がpresenterと呼ばれるオブジェクトになっている点です。

UserGetInteractorはIUserGetInputPortを実装しているので、**リスト14.8**のようなスタブを作ることが可能です。

リスト14.8：テスト用のスタブ

```
public class StubUserGetInteractor : IUserGetInputPort
{
  private readonly IUserGetPresenter presenter;

  public StubUserGetInteractor(IUserGetPresenter presenter)
  {
    this.presenter = presenter;
  }

  public void Handle(UserGetInputData inputData)
  {
    var userData = new UserData("test-id", "test-user-name");
    var outputData = new UserUpdateOutputData(userData);
```

```
        presenter.Output(outputData);
    }
}
```

クライアントはIUserGetInputPort越しにInteractorを呼び出すので、スタブに差し替えることでテストの実施が可能です。このようにして、テスタビリティを随所で確保することもクリーンアーキテクチャの重要なテーマです。

クリーンアーキテクチャのコンセプトでもっとも重要なことはビジネスルールをカプセル化したモジュールを中心に据え、依存の方向を絶対的に制御することです。これはヘキサゴナルアーキテクチャのコンセプトとほとんど同じものです。

いずれにせよ、ドメイン駆動設計の文脈上でもっとも重要なことはドメインの隔離を促すことです。すべての詳細がドメインに対して依存するようにすることは、ソフトウェアの方針をもっとも重要なドメインに握らせることを可能にします。

DDD 14.3 まとめ

アーキテクチャには共通点があります。それは、一度に多くのことを考えすぎないこと。

人は多くのことを同時に考えることが苦手な生き物です。複数の作業をこなすのであれば、並行的に作業をこなすマルチタスクよりも単一の作業を複数こなすシングルタスクを連続して行う方が作業効率はよかったりします。アーキテクチャは方針を示し、各所で考える範囲を狭めることで集中を促します。

何をすべきかが明確になると、同時に考える余地も出てくるでしょう。アーキテクチャを採用することはより深いモデルの考察の時間を増やすことに寄与します。

本章で紹介したアーキテクチャに固執する必要はありません。やり方はひとつではないはずです。ドメイン駆動設計においてアーキテクチャは主役ではありません。ドメインの隔離を促すことができるのであれば、どのようなものを採用しても構いません。

ソフトウェアにとってもっとも重要なことはシステム利用者の必要を満たすことや問題の解決を実現することです。その本質に集中するために、ご自身のプロジェクトにとって最適なアーキテクチャを選択してください。