D Appendix

ソリューション構成

ソリューション構成を紹介します。

ソリューション構成は最初に決定しなくてはいけないことながら、悩みどころでもあります。本書で学んだことをいますぐ実践するためには、このハードルを乗り越える必要があります。
本付録ではレイヤードアーキテクチャを例に取り、どのようなソリューション構成にしてそれぞれのレイヤーを配置していくかについて解説していきます。

DDD A.1 ソフトウェア開発の最初の一歩

ソフトウェアを開発するにあたって、まず最初にしなくてはいけない作業はソリューションの構成を決めることです。しかしながら、ソリューション構成は悩みどころです。なぜなら、ここでの決定はそのプロダクトを手放さない限り長い付き合いになるからです。

もちろん、開発者はリファクタリングに対して前向きです。しかし、プロジェクトをまたがるリファクタや、プロジェクト構成自体を変更するリファクタには抵抗を感じることも否めません。それゆえ、ソリューション構成を決めることは重大に感じられてしまうのです。

こうした事情から、開発者はソリューション構成を決定することに慎重になっています。そこでこの付録では、皆さんの背中を押す意味を込めて、ソリューション構成を決める際の考慮事項と具体的なソリューション構成について提示していきます。

COLUMN

C# 特有のプロジェクト管理用語

Visual Studio で C# を使ったプログラム開発を始めると、最初にプロジェクトとソリューションが作られます。

プロジェクトはプログラムを作るために必要なファイルを管理するもので、実際のソースコードや画像などのリソースが収められます。

ソリューションはそれらのプロジェクトを束ねて管理するものです。

Java に置き換えて説明するなら IntelliJ IDEA のプロジェクトがソリューションに相当するもので、モジュールがプロジェクトです（Eclipse ではそれぞれワークスペースとプロジェクト）。

D 4つのレイヤーのパッケージ構成

本付録では第14章『アーキテクチャ』で紹介したレイヤードアーキテクチャを例にパッケージ分けを考えていきます。

ここで例にするレイヤードアーキテクチャは次の4つのレイヤーで構成されています。

- プレゼンテーション
- アプリケーション
- ドメイン
- インフラストラクチャ

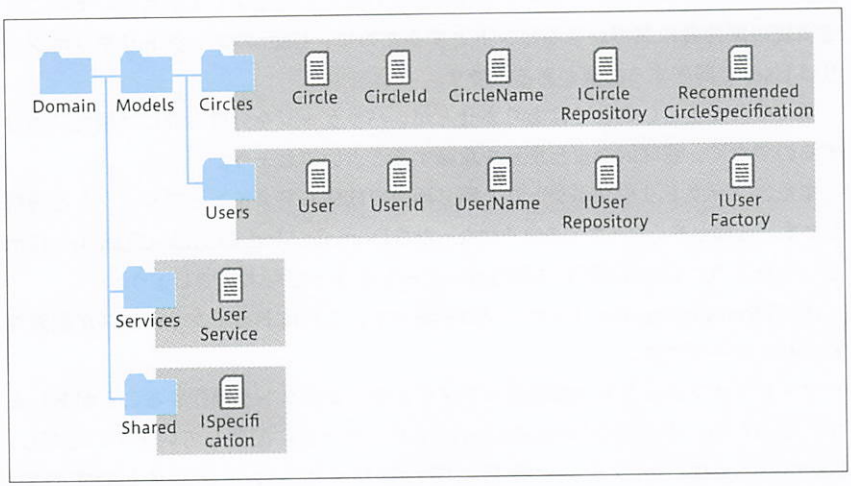
ソリューション構成について考える前に、まずはそれぞれのパッケージ構成を確認していきましょう。

なお、プレゼンテーションレイヤーはASP.NET Core MVCのプロジェクトになるのでパッケージ構成の解説は割愛します。

A.1.1 ドメインレイヤーのパッケージ構成

最初に確認するのはもっとも気になるであろうドメインレイヤーです。このレイヤーでは技術的なライブラリに対する依存はしません。

図A.1はドメインレイヤーのパッケージ構成図です。



図A.1：ドメインレイヤーの構成

図A.1ではルートパッケージはDomainとなっていますが、実際には境界付けられたコンテキストの名称になるでしょう。ドメインレイヤーのパッケージ構成は大きく3種類に分けられます。

1つ目のDomain.Modelパッケージはドメインオブジェクトが配置されます。集約を構成するエンティティや値オブジェクトはもちろんのこと、集約の生成を担うファクトリやリポジトリ、仕様もここに所属します。

Model直下のパッケージ名がCirclesやUsersといったように複数形になっているのは、C#ではクラス名と名前空間（パッケージ名）が衝突することが許されていないため[*1]です。C#以外のクラス名と名前空間が同じ名前でも問題が起きないプログラミング言語では単数形にすることが多いです。

ところで、ファクトリやリポジトリがエンティティや値オブジェクトといったドメインオブジェクトのパッケージに同居することに驚いたでしょうか。パッケージ分けをする際に、属性に着目するとDomain.FactoriesやDomain.Repositoriesといったパッケージを準備することを思いつくこともあります。しかし、それはあまりよい考えではありません。

たとえば同じ切るものであるからといって、カッターと包丁を同じ戸棚にしまっておくでしょうか。同じすくうものであるからといって、レードルとスコップを同じ引き出しにしまっておくでしょうか。属性に着目したパッケージ分けには、これと同種のおかしさがあります。

Userにはファクトリやリポジトリがあります。ファクトリでオブジェクトは作成され、リポジトリでオブジェクトは再構築されます。Userのコンストラクタは必ずファクトリやリポジトリによって呼び出されることを想定しています。そのことを後続の開発者に気づかせるヒントとするために、Userのファクトリやリポジトリは Userと同居する必要があるのです。

常にそれが正しいことではありませんが、パッケージを分ける際には属性に着目するのでなく、意味的なまとまりを意識するとよいでしょう。

また、ファクトリやリポジトリと同じ理由で仕様もドメインオブジェクトと同居します。仕様が多くなってくるようになったらDomain.Model.Circles.Specificationといったように Circles直下に専用パッケージを作ってもよいでしょう。

次にDomain.Serviceパッケージの解説です。これはドメインサービスが配置されるパッケージです。

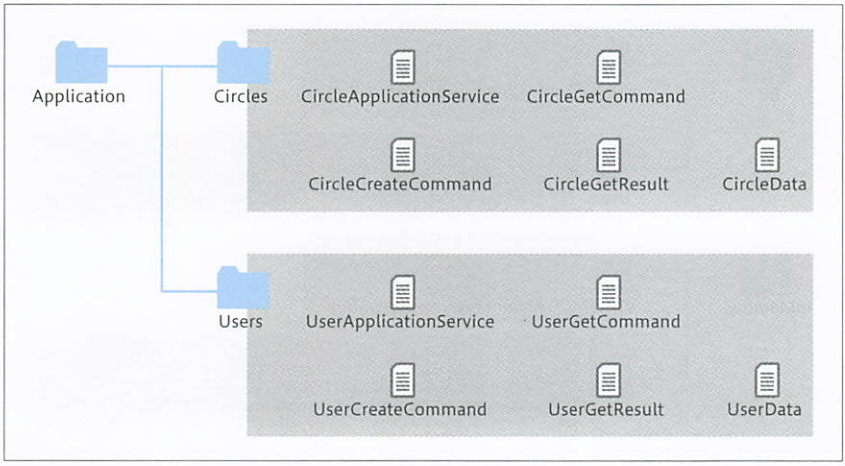
サービスオブジェクトは複数種のドメインオブジェクトを操作することがあります。そのため、中立的なDomain.Serviceパッケージに配置しています。ただし、UserServiceはUserクラスと密接に関わるサービスオブジェクトですので、Domain.Model.Usersパッケージに含める選択肢は考慮の余地があります。

残りのDomain.Sharedパッケージは必ずしも必要になるわけではありません。**図A.1**でこのパッケージに配置されているISpecificationは他のプロジェクトでも利用できるものです。共通プロジェクトとして括り出してしまい、Domainパッケージがそれに依存する形に仕立てることも可能です。

[*1] 厳密には衝突しても問題ありませんが、修飾名が必要になってしまいます。

A.1.2 アプリケーションレイヤーのパッケージ構成

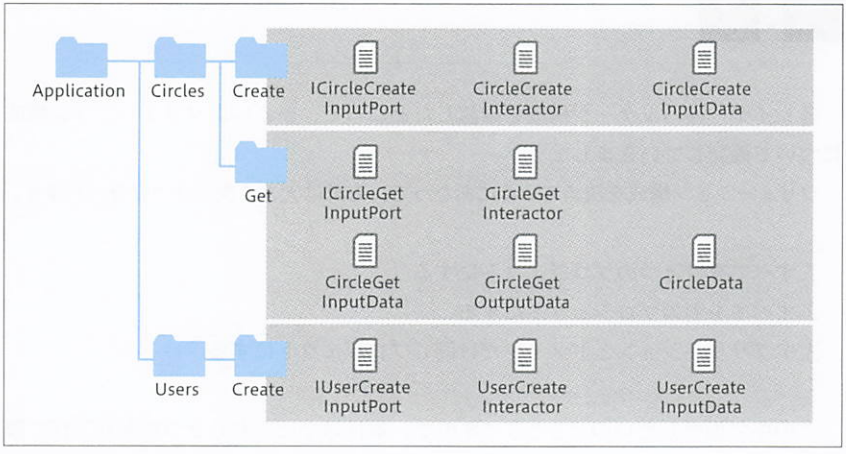
アプリケーションレイヤーのパッケージ構成は図A.2です。



図A.2：アプリケーションレイヤーのパッケージ構成

コマンドオブジェクトなどを利用するために、アプリケーションサービスごとにパッケージを分けています。パッケージ内部のファイルが多くなりすぎる場合には直下にパッケージを作って整理することを考慮します。

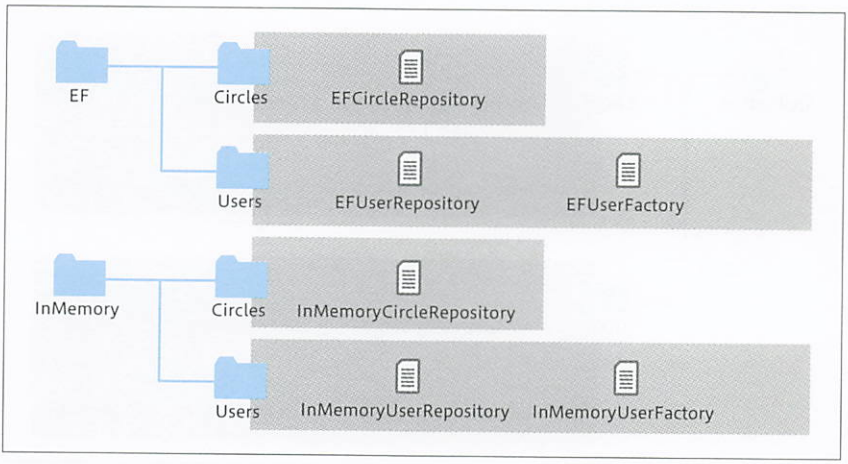
なお、第14章『アーキテクチャ』で紹介したクリーンアーキテクチャのようにユースケースごとにクラスを分けた場合は図A.3の構成になります。



図A.3：クリーンアーキテクチャに寄せた構成

A.1.3 インフラストラクチャレイヤーのパッケージ構成

インフラストラクチャレイヤーのパッケージ構成図は図A.4です。



図A.4：インフラストラクチャレイヤーパッケージ構成

インフラストラクチャはベースとなる技術基盤ごとにパッケージを分けていますが、同一のパッケージにする選択肢もあります。

DDD A.2 ソリューション構成

各レイヤーのパッケージ構成を確認したところで、いよいよソリューション構成について確認していきましょう。

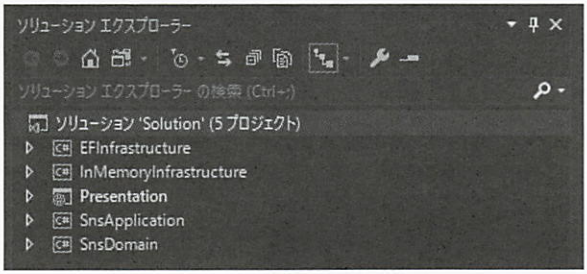
ソリューション構成を決めていくにあたって、方針は大きく次の3つに分かれます。

1. すべてをひとつのプロジェクトにする
2. すべてを別のプロジェクトにする
3. アプリケーションとドメインだけ同じプロジェクトにする

この中で推奨したいのは2と3ですので、本付録ではこれら2つを掘り下げて確認していきます。

A.2.1 すべてを別のプロジェクトにする

すべてを別のプロジェクトにしたときのソリューション構成は図A.5です。



図A.5：インフラストラクチャパッケージの構成

このパッケージ構成はドメインレイヤーの再利用を考慮しています。SnsDomainパッケージは他のプロジェクトからも参照できるので、そこに含まれるオブジェクトを再利用して新たなアプリケーションを作ることが可能です。

その反面、アプリケーションサービスとドメインオブジェクトが別のプロジェクトになるため、リストA.1のようにドメインオブジェクトのメソッドをpublicにして公開する必要があります。

リストA.1：ドメインオブジェクトのメソッドがpublicになる

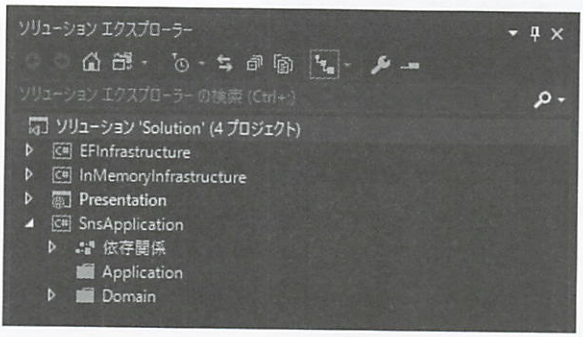
```
public class User
{
  (…略…)

  public void ChangeName(UserName name)
  {
     (…略…)
  }
}
```

メソッドの公開範囲が広がると、本来アプリケーションサービスで呼び出されることを想定したメソッドがどこか他のところで呼び出せるようになります。そのため、本来アプリケーションサービスに記述されるべきコードがプレゼンテーションレイヤーに分散してしまうこともあるでしょう。

もちろんレイヤーをまたぐデータの受け渡しをする際に、確実にDTOに詰め替えを行うことでそれは防げます。しかし、可能であればシステマチックに避けたいところです。

A.2.2 アプリケーションとドメインだけ同じプロジェクトにする

ドメインオブジェクトのメソッドを呼び出せるクライアントはアプリケーションサービスに限定する。そんな願いを叶えるのがアプリケーションとドメインだけ同じプロジェクトにする選択肢です（図A.6）。



図A.6：アプリケーションとドメインだけ同じプロジェクトにする

この構成であればリストA.2のようにinternal修飾子を使うことで、公開範囲を狭められます。

リストA.2：internal修飾子を使う

```
public class User
{
  (…略…)

  internal void ChangeName(UserName name)
  {
    (…略…)
  }
}
```

internalの公開範囲は同一プロジェクトです。別のプロジェクト（図A.6でいうEFInfrastructureやInMemoryInfrastructure、及びPresentation）からはアクセ

スできません。

この構成であれば、意図せぬメソッド呼び出しを防止できるでしょう。ただし、プロジェクト内に定義されているドメインオブジェクトをそのまま再利用して別のアプリケーションを構築することはできなくなります。

A.2.3 言語機能が与える影響

プログラミング言語にはそれぞれ特色があり、その特色がパッケージ構成に影響を与えることがあります。

たとえばJavaではアクセス修飾子を付けないとパッケージプライベートと呼ばれる公開範囲になります。C#のinternalは同一プロジェクトであればどこからでもアクセスできましたが、パッケージプライベートはそれより狭い同一パッケージに限定します。そのため、本付録で紹介したinternalを使ったアクセス制限のアプローチとはまた異なったやり方になります。

またScalaには限定子と呼ばれる機能があります。これはprivate[A]とすることで、非公開でありながらAとその派生型からのアクセスを許可できる機能です。これを活用するとC#のinternalよりもきめ細かいアクセスの制御ができます。

プログラミング言語の特性はパッケージ構成を左右します。パッケージ構成に決定版はありません。本付録の例はあくまで一例です。参考にしてもよいですし、まったく別の構成を検討しても構いません。いずれにせよ、なぜその構成を選択するのかの理由付けだけは行うようにしてください。

DDD A.3 まとめ

開発者はコードに美しさを感じると同時に、構造にも美しさを見出すことのできる生き物です。考え抜かれたソリューション構成には美しさが宿ります。

後続の開発者にヒントを与えたり、意図せぬ呼び出しの危険性を減らすために、プログラミング言語と相談しながらソリューション構成を決めることは、開発者の楽しみのひとつです。

コードを配置する場所とその理由を自問自答しながら、最適なソリューションの構成を導き出すよう心がけていくことをお勧めします。

参考文献

- 『エリック・エヴァンスのドメイン駆動設計』(翔泳社)
- 『実践ドメイン駆動設計』(翔泳社)
- 『Clean Architecture　達人に学ぶソフトウェアの構造と設計』(アスキードワンゴ)

INDEX

Profile 著者プロフィール

成瀬 允宣（なるせ・まさのぶ）

岐阜県出身。プログラマ。プログラミングにはじめて触れたのは25歳のとき。業務システム開発からキャリアをはじめ、ゲーム、Webと業種を変えながらもアプリケーション開発全般に従事。好きな原則はDRY原則。趣味は車輪の再開発。

装丁・本文デザイン	大下 賢一郎
装丁写真	iStock.com:Imam Fathoni
DTP	株式会社シンクス
編集協力	佐藤 弘文
検証協力	村上 俊一

ドメイン駆動設計入門

ボトムアップでわかる! ドメイン駆動設計の基本

2020年 2月13日 初版第1刷発行
2024年12月10日 初版第8刷発行

著 者	成瀬 允宣（なるせ・まさのぶ）
発行人	佐々木 幹夫
発行所	株式会社翔泳社（https://www.shoeisha.co.jp）
印刷・製本	株式会社シナノ

ISBN978-4-7981-5072-7 Printed in Japan