

# 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も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ず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。

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なら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）。

## 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

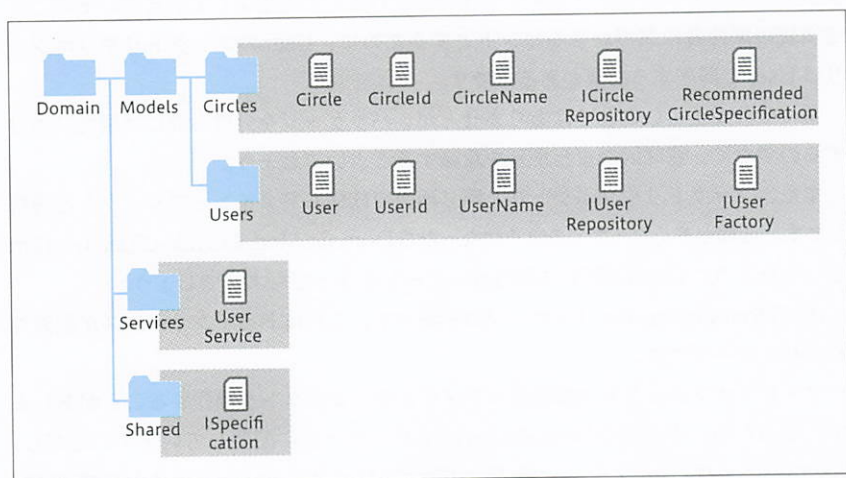
ソリューション構成について考える前に、まずはそれぞれのパッケージ構成を確認していきましょう。

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<sup>[\*1]</sup>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はなく、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。

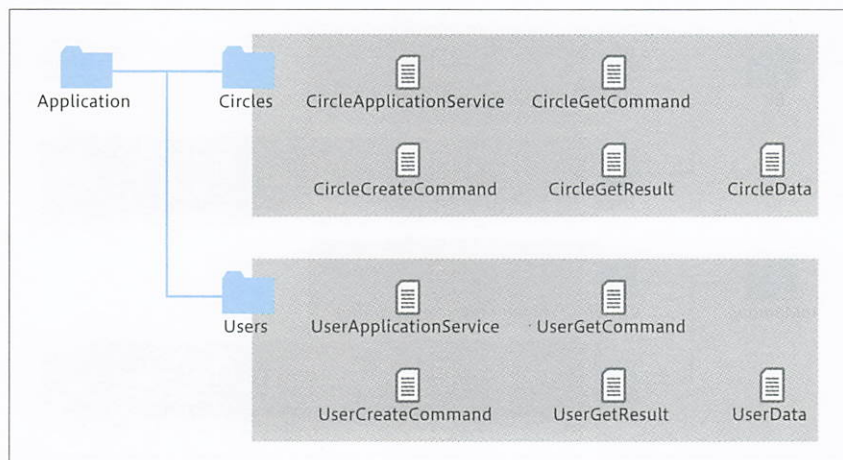
サービスオブジェクトは複数種のドメインオブジェクトを操作することがあります。そのため、中立的なDomain.Serviceパッケージに配置しています。ただし、UserServiceはUserクラスと密接に関わるサービスオブジェクトですので、Domain.Model.Usersパッケージに含める選択肢は考慮の余地があります。

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**

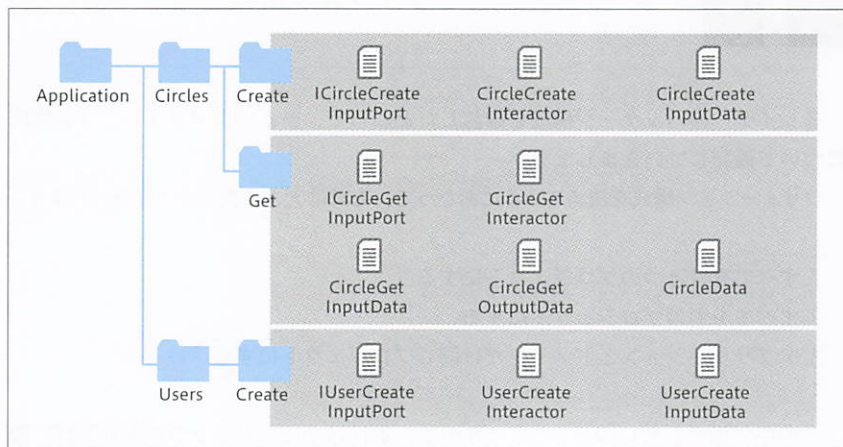
アプリケーションレイヤーのパッケージ構成は図A.2です。



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えます。

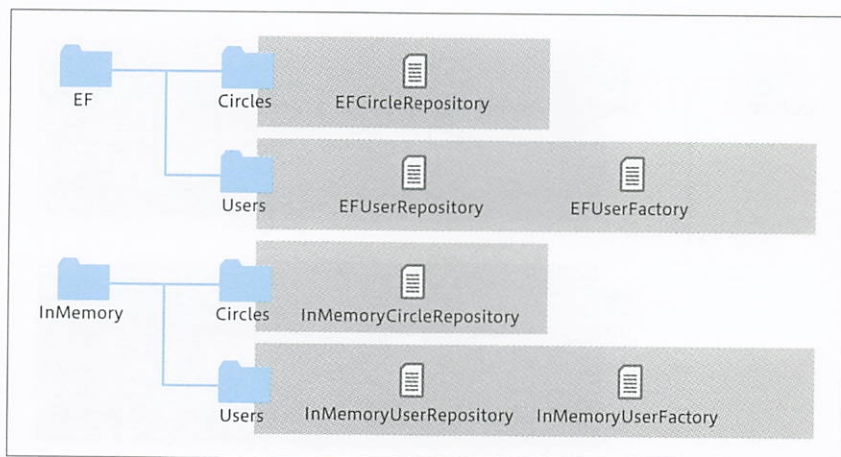
なお、第14章『アーキテクチャ』で紹介したクリーンアーキテクチャのようにユースケースごとにクラスを分けた場合は図A.3の構成になります。



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。

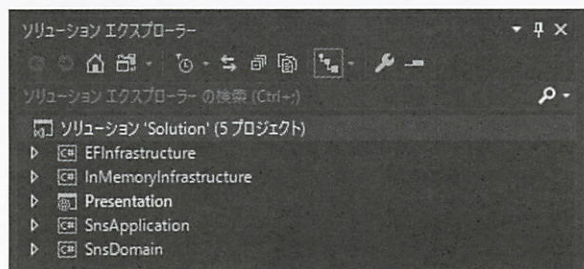
1. すべてをひとつのプロジェクトにする
2. すべてを別のプロジェクトにする
3. アプリケーションとドメインだけ同じプロジェクトにする

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。Sns Domain 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...)

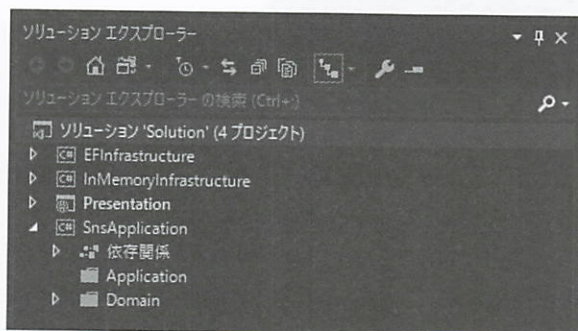
    public void ChangeName(Username name)
    {
        (...略...)
    }
}
```

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ム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



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・コミュニケーションズ)

## INDEX

### アルファベット

AOP	237
ASP.NET	185
Aspect Oriented Programming	237
C#	027
CLI	180, 181
Command Query Responsibility Segregation	313
Command-query separation	313
Commitメソッド	239
CQRS	313
CQS	313
Data Transfer Object	122
Dependency Injectionパターン	175
Dependency Inversion Principle)	165
DTO	122
EntityFramework	315
Globally Unique Identifier	207
GUI	180
GUID	207, 208
IoC Container	184
IoC Containerパターン	169, 175
MVCフレームワーク	185
NoSQLデータベース	077, 091
null	094
Service Locatorパターン	169, 170
SQL	094
Web GUI	180
Webアプリケーション	194

### あ

アーキテクチャ	317
アクセス修飾子	122, 280
値	016, 018
値オブジェクト	010, 015, 016, 018, 028, 049, 086
アプリケーションサービス	010, 113, 114, 144, 153
誤った代入	040
アンチパターン	318
依存	160, 161
依存関係	159, 168, 172, 186
依存関係逆転の原則	165
イテレーティブ	033
折り	100
折り信者のテスト理論	099
インスタンス	034, 086
インスタンス生成の処理	211
インスタンスの永続化処理	227
インスタンス変数	273
インターフェース	092, 152
インフラストラクチャ	078
インフラストラクチャレイヤー	370
永続化	108, 227
エラー	043, 133
エリック・エヴァンスのドメイン駆動設計	001
エンティティ	010, 047, 049, 058, 086
オーバーライド	102
オブジェクト自身	273
オブジェクト同士の依存	160
オブジェクトの生成	206
オブジェクトリレーショナルマップ	104

### か

会計システム	003
下位レベル	165, 166
カプセル化	220
可変	049
凝集度	144, 147
クエリ	309
クラス	017, 021, 152
クリーンアーキテクチャ	322, 338
軽量DDD	344
結果整合性	290
ゲッター	107, 219
交換	022
更新	127
コマンドオブジェクト	132
コマンドラインインターフェース	180, 181
コミット処理	233
コレクション	272
コンストラクタ	175
コンテキスト	356
コンテキストマップ	360
コンテナ	034
コントローラ	191, 195
コントロール	159, 168
コンパイラ	039
コンパイルエラー	176
コンポジション	285

### さ

サークル機能	253
サークル集約	281
サービス	066, 155
再構築	109
採番処理	207
識別子	053, 207

自動採番機能	214
重複確認	260
集約	010, 268
出庫	080
仕様	010, 293, 297, 302, 304
上位レベル	165, 166
上限チェック	271
状態	156
シングルトン	182
シングルトンパターン	182
スーパークラス	248
スタートアップスクリプト	183
整合性	224, 235
セーフティネット	052
セッター	107, 215
属性	027
ソフトウェア開発	009, 366
ソフトウェアシステム	179
ソリューション構成	370

## た

退会処理	134
代入	040
遅延実行	314
致命的な不具合	225
抽象	166
直接インスタンス化したオブジェクト	273
ディープコピー	102
データストア	086, 091
データストレージ	106
データ転送用オブジェクト	122
データの整合性	223
データベースコネクション	233
テスト	098
テストの維持	173
テスト用のリポジトリ	100
デバッグ用	188
メテルの法則	274
同一性	055
等価性	023
ドキュメント性	060
ドメイン	003
ドメインエキスパート	345
ドメインオブジェクト	007, 048, 087, 115
ドメイン駆動設計	001

ドメインサービス	010, 065, 066, 091
ドメインサービスの濫用	070
ドメインの概念	082
ドメインの概念 + DomainService	082
ドメインの概念 + Service	082
ドメインのルール	267
ドメインモデル	004
ドメインレイヤー	367
トランザクション	231
トランザクション処理	235
トランザクションスコープ	235
トランザクションの開始	233

## な

入庫	080
----	-----

## は

バグ	035
パターン	010
パッケージ	152
パフォーマンス問題	307
パラメータ	131
引数として渡されたオブジェクト	273
表現力	037
ファクトリ	010, 205, 206
ファクトリの存在	213
不自然なふるまい	067
不正な値	039
物流拠点のふるまい	079
物流システム	002, 079
不変	019
不変条件	271
不変のメソッド	021
ふるまい	033, 056, 156
プログラミング	016, 019
プロダクション用	188
分散トランザクション	236
ヘキサゴナルアーキテクチャ	322, 334
ボトムアップドメイン駆動設計	363
ポリモーフィズム	220

## ま

メソッド	218
メソッドのシグネチャ	130
モジュール	161, 165
文字列型	028
モチベーション	036
モデリング	345
モデル	005
漏れ出したルール	263

## や

ユーザID	041
ユーザーインターフェース	180
ユーザ登録処理	226, 234
ユーザエンティティ	073
ユーザ作成処理	075, 090
ユーザ情報更新処理	136, 142
ユーザ情報取得処理	120, 127
ユーザ退会処理	148, 150
ユーザ登録処理	138, 149
ユーザの重複	138
ユーザ名	039
ユーザ登録処理	196
ユースケース	073, 115, 258
輸送ドメインサービス	081
ユニークキー制約	228, 230
ユニットオブワーク	239, 241
ユニットテスト	180, 195
ユビキタス言語	349, 355

## ら

ライフサイクル	047, 058
リードモデル	309
リポジトリ	010, 085, 086, 087, 091, 092, 096
リレーショナルデータベース	076, 089, 091
ルール	030
ルールの流出	135
例外	133
レイヤードアーキテクチャ	322, 325
レベル	166
ロールバック	246
ロック	289

## 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 ( <a href="https://www.shoeisha.co.jp">https://www.shoeisha.co.jp</a> )
印刷・製本	株式会社シナノ

©2020 MASANOBU NARUSE

※本書は著作権法上の保護を受けています。本書の一部または全部について(ソフトウェアおよびプログラムを含む)、株式会社 翔泳社から文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

※本書へのお問い合わせについては、iiページに記載の内容をお読みください。

※落丁・乱丁の場合はお取替えいたします。03-5362-3705までご連絡ください。

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