Chapter 7

柔軟性をもたらす依存関係のコントロール

ソフトウェアに柔軟性をもたらすために必要なことは依存関係を制御することです。

プログラムには依存という概念があります。依存はオブジェクトがオブジェクトを参照するだけで発生します。したがって、オブジェクト同士に依存関係が発生するのは自然なことです。しかし、この依存関係を軽視するとソフトウェアはその柔軟性を失います。

ソフトウェアを柔軟に保つために必要なことは、特定の技術的要素への依存を避け、変更の主導権を主たる抽象に移すことです。本章で解説する依存のコントロールはその方法です。

オブジェクト同士の依存がどういったものかを確認し、ソフトウェアを柔軟に保つために技術的要素への依存から脱却する術を確認しましょう。

DDD 7.1 技術要素への依存がもたらすもの

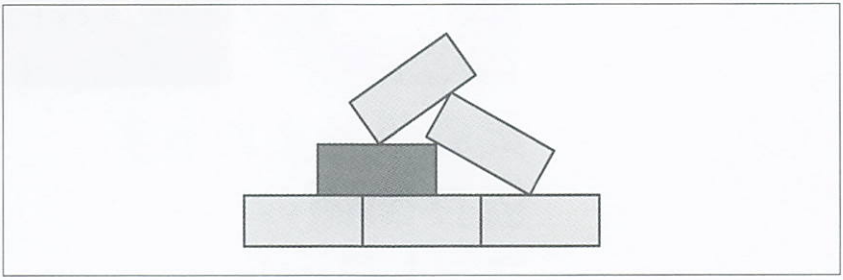

図7.1：複雑な依存関係

積み上がった積み木の中ほどに位置するブロックを抜き出すことを想像してみてください（図7.1）。抜き出そうとしたブロックは直上のブロックを支えています。手荒くブロックを抜き出せば、たちまちに積み木は崩れ去るでしょう。誰しもがブロックを抜き出すことを難しいと感じるはずです。プログラムにおける依存も同じことがいえます。

ソフトウェアの中核に位置するオブジェクトを変更することを想像してみてください。そのオブジェクトは多くのオブジェクトに依存されていますし、多くのオブジェクトに依存しています。たったひとつの変更が多くのオブジェクトに影響します。緻密に組み上げられたコードを変更することの厄介さは、ほとんど恐怖と似たような感覚となって開発者に襲い掛かるでしょう。

プログラムを組み上げていく過程でオブジェクト同士の依存は避けられません。依存関係はオブジェクトを利用するだけで発生するのです。重要なことは依存を避けることではなく、コントロールすることです。

本章で解説する依存のコントロールはドメインのロジックを技術的要素から解き放ち、ソフトウェアに柔軟性を与えるものです。コードが技術的要素に支配されることの問題を確認し、その解決法を確認していきましょう。

DDD 7.2 依存とは

まずは簡単なサンプルで依存がどういったものかを確認します。依存はあるオブジェクトからあるオブジェクトを参照するだけで発生します。**リスト7.1**の単純なコードに存在している依存関係を確認してみましょう。

リスト7.1：ObjectAはObjectBに依存する

```
public class ObjectA
{
  private ObjectB objectB;
}
```

ObjectAはObjectBを参照しています。そのため、ObjectBの定義が存在しない限りObjectAは成り立ちません。ObjectAはObjectBに依存しているといえます。

こういった依存関係は図で表すことができます。**図7.2**はObjectAとObjectBの依存関係を表した図です。

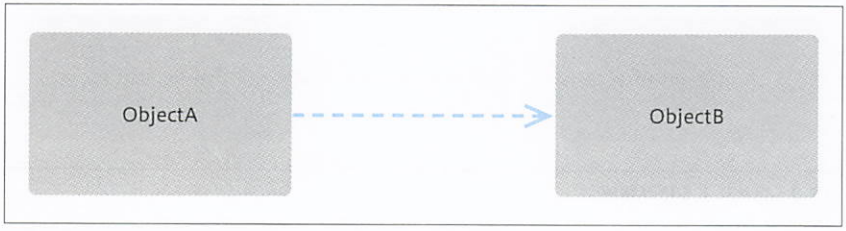

図7.2：参照による依存関係

依存は依存する側のモジュールから依存される側のモジュールへと矢印を伸ばして表現します。**図7.2**の矢印は参照による依存を表す矢印です。

依存関係が生じるのは参照に限ったことではありません。たとえばインターフェースとその実体になる実装クラスの関係にも依存が生まれます（**リスト7.2**）。

リスト7.2：UserRepositoryはIUserRepositoryに依存する

```
public interface IUserRepository
{
  User Find(UserId id);
}

public class UserRepository : IUserRepository
{
  public User Find(UserId id)
  {
    (…略…)
  }
}
```

UserRepositoryクラスはIUserRepositoryインターフェースを実装しています。もしもIUserRepositoryの定義が存在しなかったとしたらクラスの宣言部にてコンパイルエラーが検出され、UserRepositoryは成り立ちません。UserRepositoryはIUserRepositoryに依存していることになります。

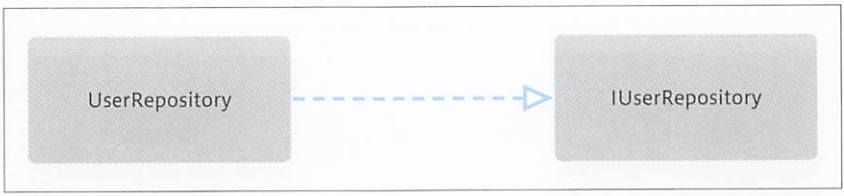

図7.3：汎化による依存関係

インターフェースと実装クラスの依存関係を表したものが図7.3です。依存の方向性を表す白抜きの矢印は汎化を示します。

これらの例を見てわかるとおり、プログラムを組み上げていく上で、依存は自然と発生するものです。もちろん、これまでの解説で取り扱ってきたオブジェクトにも依存は発生しています。たとえば次のUserApplicationServiceに登場するモジュールの依存関係を確認してみましょう（リスト7.3）。

リスト7.3：UserApplicationServiceの依存関係に着目

```
public class UserApplicationService
{
  private readonly UserRepository userRepository;

  public UserApplicationService(UserRepository userRepository)
  {
    this.userRepository = userRepository;
  }

  (…略…)
}
```

UserApplicationServiceにはUserRepositoryがフィールドとして定義されています。したがってUserApplicationServiceはUserRepositoryに依存している状態です（図7.4）。

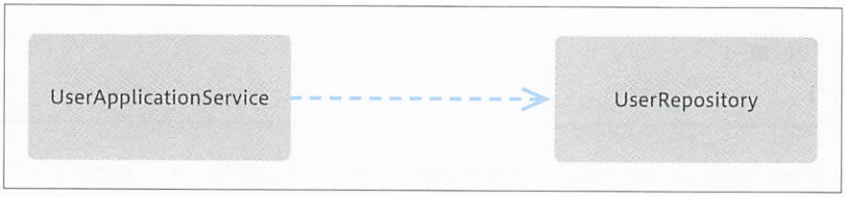

図7.4：リスト7.3の依存関係

実をいうと、このUserApplicationServiceには問題があります。具体的にいえば具象クラスであるUserRepositoryクラス、ひいてはそこで利用されているデータ永続化に関する特定の技術基盤に依存していることが問題です。

UserRepositoryが取り扱っているデータストアがリレーショナルデータベースなのか、それともNoSQLデータベースであるかということはリスト7.3を眺めても計り知れませんが、UserApplicationServiceがいずれかのデータストアに対して結びついていることは確かです。ソフトウェアが健全に成長するためには開発やテストで気軽にコードを実行できるように仕向けることが重要です。特定のデータストアに結びついてしまうとそれは不可能になります。コードを実行するためにはデータベースを準備して、必要なテーブルを作成する必要があります。取り扱うロジックによっては事前にデータを投入する必要が生じることもあるでしょう。ただ

動作させるというだけで、途方もない労力がのしかかってきます。

このような問題を解決するためには第5章『データにまつわる処理を分離する「リポジトリ」』で解説をしたリポジトリが役に立ちます。UserApplicationServiceが具象クラスであるUserRepositoryを受け取るのではなく、リポジトリのインターフェースを参照するようにしてみましょう（**リスト7.4**）。

リスト7.4：リポジトリのインターフェースを参照する

```
public class UserApplicationService
{
  // インスタンス変数として保持しているのはインターフェース
  private readonly IUserRepository userRepository;

  // コンストラクタが受け取る引数もインターフェースになる
  public UserApplicationService(IUserRepository userRepository)
  {
    this.userRepository = userRepository;
  }

  (…略…)
}
```

UserApplicationServiceがIUserRepositoryという抽象型（インターフェースは抽象型とも呼ばれる）に対して依存するようになったことで、IUserRepositoryを実装した具象クラスであればその実体が何であっても引き渡すことができます。つまりUserApplicationServiceはUserRepositoryという具象クラス、ひいては特定のデータストアに結びつくことがなくなったのです。現在のUserApplicationServiceであれば、たとえばインメモリで動作するテスト用のリポジトリを引き渡してユニットテストを実施することが可能です。あるいは、別のデータストアを利用するリポジトリを用意することで、主たるロジックに変更を加えることなくデータストアを差し替えることも可能になります。

リスト7.4の依存関係を表した図が**図7.5**です。

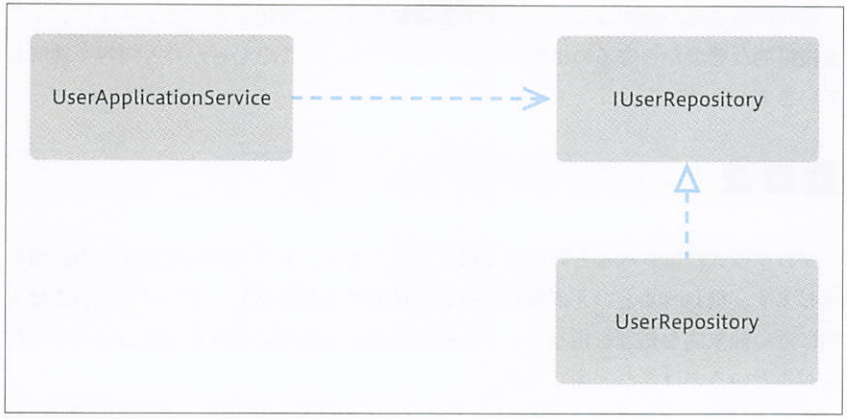

図7.5：抽象型に依存

抽象型を利用するようになると、具象型に向いていた依存の矢印が抽象型へ向くようになります。このように依存の方向性を制御し、すべてのモジュールが抽象へ依存するように制御することはビジネスロジックを具体的な実装から解き放ち、より純粋なものに昇華する効果があります。

この抽象型を用いた依存関係の制御は「依存関係逆転の原則」として知られています。

DDD 7.3 依存関係逆転の原則とは

依存関係逆転の原則（Dependency Inversion Principle）は次のように定義されています（以下引用[*1]）。

> A. 上位レベルのモジュールは下位レベルのモジュールに依存してはならない、どちらのモジュールも抽象に依存すべきである。

> B. 抽象は、実装の詳細に依存してはならない。実装の詳細が抽象に依存すべきである。

[*1] 『実践ドメイン駆動設計』（翔泳社）、P.119より引用。なお本書では、「上位」を「上位レベル」、「下位」を「下位レベル」と表記しています。

依存関係逆転の原則はソフトウェアを柔軟なものに変化させ、ビジネスロジックを技術的な要素から守るのに欠かせないものです。ここでしっかりと内容を理解していきましょう。

7.3.1 抽象に依存せよ

プログラムにはレベルと呼ばれる概念があります。レベルは入出力からの距離を示します。低レベルといえば機械に近い具体的な処理を指し、高レベルといえば人間に近い抽象的な処理を指します。依存関係逆転の原則に表れる上位レベルや下位レベルというのはこれと同じです。

たとえばデータストアを操作するUserRepositoryの処理は、UserRepositoryを操作するUserApplicationServiceよりも機械に近い処理です。レベルの概念に照らし合わせるとUserRepositoryは下位レベルでUserApplicationServiceは上位レベルになります。抽象型を利用しなかったとき（**リスト7.3**）、UserApplicationServiceは具体的な技術基盤と比べて上位レベルのモジュールでありながら、データストアの操作を行う下位レベルのモジュールであるUserRepositoryに依存していました。これはまさに「上位レベルのモジュールは下位レベルのモジュールに依存してはならない」という原則に反しています。

この依存の関係はUserApplicationServiceが抽象型であるIUserRepositoryを参照する（**リスト7.4**）ようになると**図7.6**のように変化します。

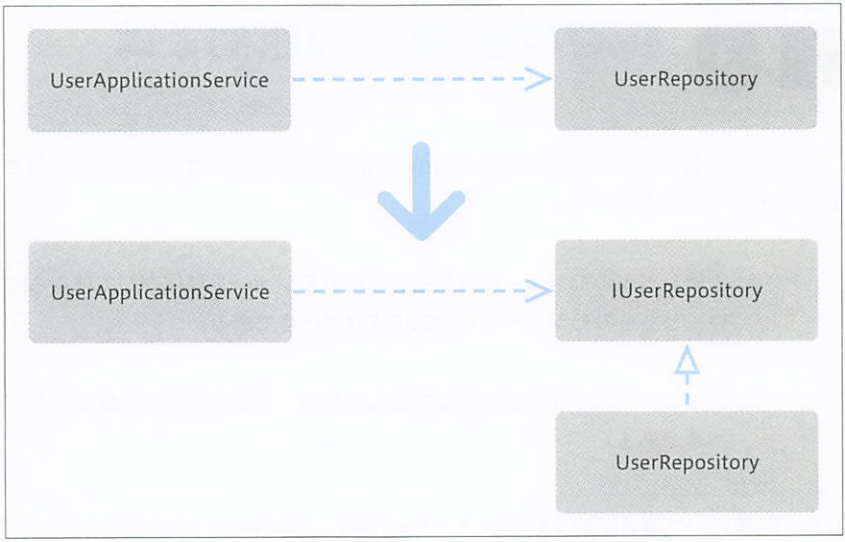

図7.6：依存関係の変化

抽象型を導入することでUserApplicationServiceとUserRepositoryは、双方ともに抽象型であるIUserRepositoryに依存の矢印を伸ばすことになります。もはや上位レベルのモジュール（UserApplicationService）が下位レベルのモジュール（UserRepository）に依存しなくなり、「どちらのモジュールも抽象に依存すべきである」という原則にも合致します。もともと具体的な実装に依存していたものが抽象に依存するように、依存関係は逆転したのです。

一般的に抽象型はそれを利用するクライアントが要求する定義です。IUserRepositoryはいわばUserApplicationServiceのために存在しているといっても過言ではありません。このIUserRepositoryという抽象に合わせてUserRepositoryの実装を行うことは、方針の主導権をUserApplicationServiceに握らせることと同義です。「抽象は実装の詳細に依存してはならない。実装の詳細が抽象に依存すべきである」という原則はこのようにして守られます。

7.3.2 主導権を抽象に

伝統的なソフトウェア開発手法では高レベルなモジュールが低レベルなモジュールに依存する形で作成される傾向がありました。言い換えるなら抽象が詳細に依存するような形で構築されていました。

抽象が詳細に依存するようになると、低レベルのモジュールにおける方針の変更が高レベルのモジュールに波及します。これはおかしな話です。重要なドメインのルールが含まれるのはいつだって高レベルなモジュールです。低レベルなモジュールの変更を理由にして、重要な高レベルのモジュールを変更する（たとえばデータストアの変更を理由にビジネスロジックを変更する）などということは起きてほしくない事態です。

主体となるべきは高レベルなモジュール、すなわち抽象です。低レベルなモジュールが主体となるべきではありません。

高レベルなモジュールは低レベルのモジュールを利用するクライアントです。クライアントがすべきはどのような処理を呼び出すかの宣言です。先述したとおり、インターフェースはそれを利用するクライアントが宣言するものであり、主導権はそのクライアントにあります。インターフェースを宣言し、低レベルのモジュールはそのインターフェースに合わせて実装を行うことで、より重要な高次元の概念に主導権を握らせることが可能になるのです。

DDD 7.4 依存関係をコントロールする

UserApplicationServiceがインメモリで動作するテスト用のリポジトリを利用してほしいのか、それともリレーショナルデータベースに接続するプロダクション用のリポジトリを利用してほしいのかどうかは、ときと場合によります。開発中であれば前者でしょうし、リリースビルドはもちろん後者です。重要なのはどれを扱うかではなく、それをどのようにして制御するかです。依存関係をコントロールする手段について確認していきます。

まずはあまりよくない例から見ていきましょう。たとえば開発中にインメモリのリポジトリを利用するという目的を達成することだけを考えた短絡的なコードは**リスト7.5**です。

リスト7.5：インメモリのリポジトリをコンストラクタで生成する

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

  public UserApplicationService()
  {
    this.userRepository = new InMemoryUserRepository();
  }

  (…略…)
}
```

リスト7.5は、フィールドであるuserRepositoryは抽象型であるものの、具象クラスを内部でインスタンス化しているためにUserApplicationServiceはInMemoryUserRepositoryという詳細なオブジェクトに依存してしまっています。この依存が引き起こす問題は単純で、完成したはずのコードに修正を加えることが必要になることです。ある程度動作するようになってからか、それとも考えうる限りのテストをしきってからかはわかりませんが、**リスト7.6**のようにリリースの際にはプロダクション用リポジトリを利用するように完成したはずのコードを修正する必要があります。

リスト7.6：プロダクション用のリポジトリに差し替える

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

  public UserApplicationService()
  {
    // this.userRepository = new InMemoryUserRepository();
    this.userRepository = new UserRepository();
  }

  (…略…)
}
```

もちろんこの作業はここだけにとどまりません。この造りが許されるなら、きっと他のコードも似たような構造をしているでしょう。それらも間違いなくプロダクション用リポジトリを取り扱うようにコードを修正していく必要があります。修正自体は至極単純な作業と予測できますが、いかにも開発者向きでない愚直で面倒な作業です。

また修正作業を無事完遂してリリースしたとしても、場合によってはインメモリのリポジトリを利用したくなるときがあります。たとえばソフトウェアに不具合が発生したときの原因究明をしたいときなどがそれにあたります。ソフトウェアに不具合が生じたとき、その不具合の状況を再現するためのデータをデータベースに用意するのは手間がかかります。多くの場合「エラーを発生させるための整合性が取れたデータ」は用意しづらいものです。

こういったときはテスト用のリポジトリを用意して、それを利用するように差し替えて、プログラムの挙動を確かめたいところです。そのとき開発者に与えられるのは、プロダクション用のリポジトリをまたテスト用のリポジトリに差し替える単調な作業です。

こういった問題を解決するために取られるパターンとして、Service Locatorパターンと IoC Containerパターンがあります。それぞれどういったものか確認していきましょう。

7.4.1 Service Locatorパターン

Service LocatorパターンはServiceLocatorと呼ばれるオブジェクトに依存解決先となるオブジェクトを事前に登録しておき、インスタンスが必要となる各所でServiceLocatorを経由してインスタンスを取得するパターンです。

言葉だけではイメージがしづらいので具体例を見てみましょう。**リスト7.7**はService Locatorパターンを適用したUserApplicationServiceです。

リスト7.7：ServiceLocatorを適用する

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

  public UserApplicationService()
  {
    // ServiceLocator経由でインスタンスを取得する
    this.userRepository = ServiceLocator.Resolve➡
<IUserRepository>();
  }

   (…略…)
}
```

コンストラクタでServiceLocatorにIUserRepositoryの依存を解決するように依頼しています。この依頼に対して返却される実際のインスタンスはスタートアップスクリプトなどで事前に登録しておきます（**リスト7.8**）。

リスト7.8：事前にインスタンスを登録する

```
ServiceLocator.Register<IUserRepository, ➡
InMemoryUserRepository>();
```

リスト7.8のように登録するとIUserRepositoryの依存解決が依頼された際にInMemoryUserRepositoryをインスタンス化して引き渡します。もしもプロダクション用データベースに接続するリポジトリを利用したいときはServiceLocator

への登録を**リスト7.9**のように変更することで対応します。

リスト7.9：プロダクションに移行するためリポジトリを切り替える

```
// この修正のみで全体に変更が行き渡る
ServiceLocator.Register<IUserRepository, UserRepository>();
```

IUserRepositoryを要求するオブジェクトがすべてServiceLocator経由でインスタンスを取得していれば、修正箇所は依存関係を設定しているスタートアップスクリプトの修正だけでこと足ります。

このようにServiceLocatorに依存を解決させることによりInMemoryUserRepositoryないしUserRepositoryのインスタンス化を行うコードがプログラムの随所に点在しなくなり、アプリケーションの中核を担うロジックに修正を加えることなく、リポジトリの実体を差し替えられるようになります（**図7.7**）。

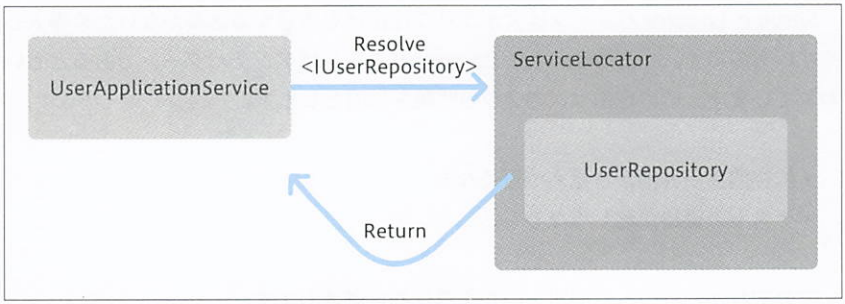

図7.7：Service Locatorによる依存の解決

ServiceLocatorに登録されるインスタンスの設定はプロダクション用とテスト用など用途に応じて一括で管理すると便利です。スタートアップスクリプトでプロジェクトの構成設定などをキーにして、用途ごとのインスタンス設定に切り替えを行えるようにします（**図7.8**）。

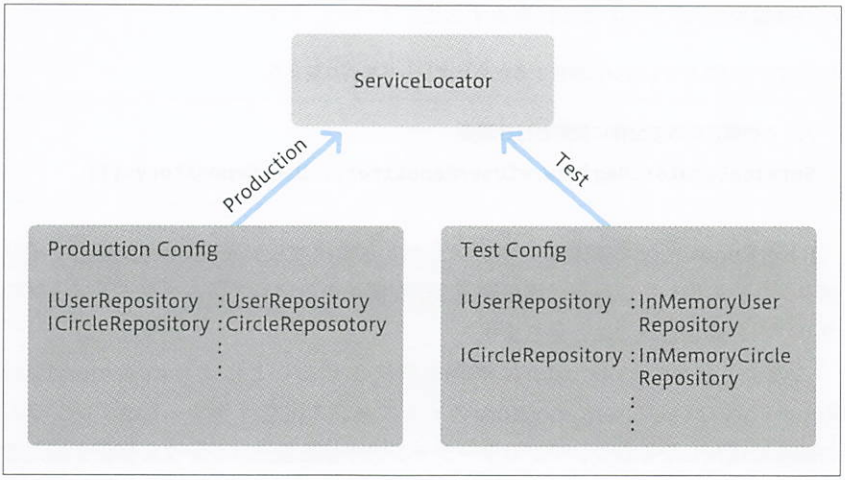

図7.8：スタートアップスクリプトによる切り替え

Service Locatorパターンは大がかりな仕掛けを用意する必要がないため導入しやすいものです。一方でService Locatorパターンはアンチパターンであるともいわれています。その理由は次の2つの問題を起因とします。

- 依存関係が外部から見えづらくなる
- テストの維持が難しくなる

それぞれがどういった問題なのかを具体的に確認します。

依存関係が外部から見えづらくなる

Service Locatorパターンを採用した場合、コンストラクタはたいていひとつになります。これはService Locatorから必要なインスタンスを取り出すようになるためです。このとき外部から見えるクラス定義は**リスト7.10**です。

リスト7.10：外部から確認したときのクラスの定義

```
public class UserApplicationService
{
  public UserApplicationService();
  public void Register(UserRegisterCommand command);
}
```

この定義を見たとき、開発者はUserApplicationServiceをインスタンス化してRegisterメソッドを呼び出すでしょう。それ以外にこのオブジェクトに対してできることがないからです。しかし、それは実行時エラーによってプログラムを強制終了させる結果に終わります。なぜならUserApplicationServiceのコンストラクタはServiceLocatorにIUserRepositoryの依存解決を依頼するからです（**リスト7.11**）。

リスト7.11：リスト7.10のコンストラクタ

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

  public UserApplicationService()
  {
    // IUserRepositoryの依存解決先が設定されていないのでエラーを起こす
    this.userRepository = ServiceLocator.Resolve➡
<IUserRepository>();
  }

   (…略…)
}
```

事前にServiceLocatorに対して依存解決の設定を行っていないため、UserApplicationServiceが要求する依存解決は失敗します。

「UserApplicationServiceを正しく動作させるためには、事前にIUserRepositoryが要求された際に引き渡すオブジェクトの登録が必要である」ことが、クラスの定義を確認しただけではわからないことはあまり良い傾向とはいえません。UserApplicationServiceを動作させるためにUserRepositoryをServiceLocatorに登録できるのは、UserApplicationServiceの実装を確認したか、さもなければ超能力に目覚めたかのどちらかでしょう。もちろんコメントにより補足を行う手立ても考えられますが、コメントは実際のコードと乖離することがある以上、解決策として上策ではありません。

テストの維持が難しくなる

優れた開発者は「人間が間違いを犯す生き物である」という事実を知っています

し、その最たる例が自分であることも熟知しているでしょう。テストはそのような間違いを未然に発見できるツールです。すべての間違いを防ぐことは叶いませんが、思い違いや意図しない動作をある程度は見つけ出してくれます。**リスト7.12**はUserApplicationServiceのテストを行うスクリプトの一部です。

リスト7.12：テストを行うための準備

```
ServiceLocator.Register<IUserRepository, ➡
InMemoryUserRepository>();
var userApplicationService = new UserApplicationService();
```

UserApplicationServiceが実装された当初、このコードは問題なく動作していました。テストは開発者の思い違いを正し、大いに役立ったものです。そうしてうまくやってのけたコードも月日が経つにつれて変化が求められます。UserApplicationServiceもまたそのうちのひとつでした（**リスト7.13**）。

リスト7.13：UserApplicationServiceに変化が起きた

```
public class UserApplicationService
{
  private readonly IUserRepository userRepository;
  // 新たなフィールドが追加された
  private readonly IFooRepository fooRepository;

  public UserApplicationService()
  {
    this.userRepository = ServiceLocator.Resolve➡
<IUserRepository>();
    // ServiceLocator経由で取得している
    this.fooRepository = ServiceLocator.Resolve➡
<IFooRepository>();
  }

  (…略…)
}
```

新しいコードには新たな依存関係が追加されています。しかし、**リスト7.12**のテ

ストコードでは当然のことながらIFooRepositoryに対する依存解決が登録されていません。この変更によってテストは破壊されるのです。

とはいえ、テストが破壊されること自体はそれほど問題ではありません。UserApplicationServiceを変更したことで、それを取り扱うテストコードにも変更が必要になるというのはよくあることです。ここで問題とすべきは、テストが破壊されたことが、テストを実行するそのときまでわからないことです。

開発者にとってテストは自身を助けるものですが、同時に途方もなく面倒なものです。テストを維持するにはある程度の強制力が必要です。今回のような依存関係の変更に自然と気づき、テストコードの変更を余儀なくさせる強制力をもたせることができなければ、近い将来テストは維持されなくなってしまうでしょう。

7.4.2 IoC Containerパターン

IoC[*2] Container (DI Container) について知るにはまずDependency Injectionパターンについて知る必要があります。

Dependency Injectionパターンは依存の注入という言葉で訳されます。ほとんど直訳ですので、わかったようなわからないような言葉です。具体例を確認して理解しましょう。**リスト7.14**はUserApplicationServiceにInMemoryUserRepositoryに対する依存を注入、つまりDependency Injectionをしています。

リスト7.14：依存を注入する

```
var userRepository = new InMemoryUserRepository();
var userApplicationService = new UserApplicationService(➡
userRepository);
```

この形式はコンストラクタで依存するオブジェクトを注入しているのでコンストラクタインジェクションとも呼ばれます。これまでの解説で何度も登場してきたパターンです。Dependency Injectionパターンはこれ以外にメソッドで注入するメソッドインジェクションなど多くのパターンが存在します。注入する方法はいくつもありますが、いずれも依存するモジュールを外部から注入することに変わりありません。

Dependency Injectionパターンであれば依存関係の変更に強制力をもたせら

[*2] IoCはInversion of Controlの略で、「制御の反転」を意味します。

れます。たとえば**リスト7.15**のようにUserApplicationServiceに新たな依存関係を追加してみましょう。

リスト7.15：新たな依存関係を追加する

```
public class UserApplicationService
{
  private readonly IUserRepository userRepository;
  // 新たにIFooRepositoryへの依存関係を追加する
  private readonly IFooRepository fooRepository;

  // コンストラクタで依存を注入できるようにする
  public UserApplicationService(IUserRepository ➡
userRepository, IFooRepository fooRepository)
  {
    this.userRepository = userRepository;
    this.fooRepository = fooRepository;
  }

  (…略…)
}
```

UserApplicationServiceでは新たな依存関係を追加するためコンストラクタに引数が追加されています。これによりUserApplicationServiceをインスタンス化して実施しているテストはコンパイルエラーにより実行できなくなります（**リスト7.16**）。

リスト7.16：テストがコンパイルエラーになる

```
var userRepository = new InMemoryUserRepository();
// 第2引数にIFooRepositoryの実体が渡されていないためコンパイルエラーとなる
var userApplicationService = new UserApplicationService(➡
userRepository);
```

テストを実施するために開発者はコンパイルエラーを解消することを余儀なくされます。この強制は大いなる力です。

しかし、これは便利な一方で、依存するオブジェクトのインスタンス化をあちこ

ちに記述する必要が生み出します。たとえば開発時にインメモリのリポジトリでプログラムを動作させていた場合、プロダクション環境へ移行するときにはデータベースに接続するリポジトリを利用するように変更する必要があります。**リスト7.16**のようなリポジトリのインスタンス化を行っている箇所すべてを見つけ出し、依存させたいリポジトリに差し替える必要があるのです。

この問題を解決するために活躍するのがIoC Containerパターンです。**リスト7.17**のコードはC#のIoC Containerを利用してUserApplicationServiceをインスタンス化するコードです。

リスト7.17：IoC Containerを利用して依存関係を解決させる

```
// IoC Container
var serviceCollection = new ServiceCollection();
// 依存解決の設定を登録する
serviceCollection.AddTransient<IUserRepository,
InMemoryUserRepository>();
serviceCollection.AddTransient<UserApplicationService>();

// インスタンスはIoC Container経由で取得する
var provider = serviceCollection.BuildServiceProvider();
var userApplicationService = provider.GetService➡
<UserApplicationService>();
```

IoC Containerは設定にしたがって依存の解決を行い、インスタンスを生成します。

リスト7.17を例に処理の流れを追ってみましょう。オブジェクトのインスタンス化が始まるのは**リスト7.17**の最終行からです。GetService<UserApplicationService>が呼び出され、IoC ContainerはUserApplicationServiceを生成しようとします。UserApplicationServiceはコンストラクタでIUserRepositoryを必要とするので、内部的に依存関係を解決し、IUserRepositoryを取得しようとします。IUserRepositoryはInMemoryUserRepositoryを利用するように登録されているので、UserApplicationServiceはInMemoryUserRepositoryのインスタンスを受け取り、インスタンス化されます（**図7.9**）。

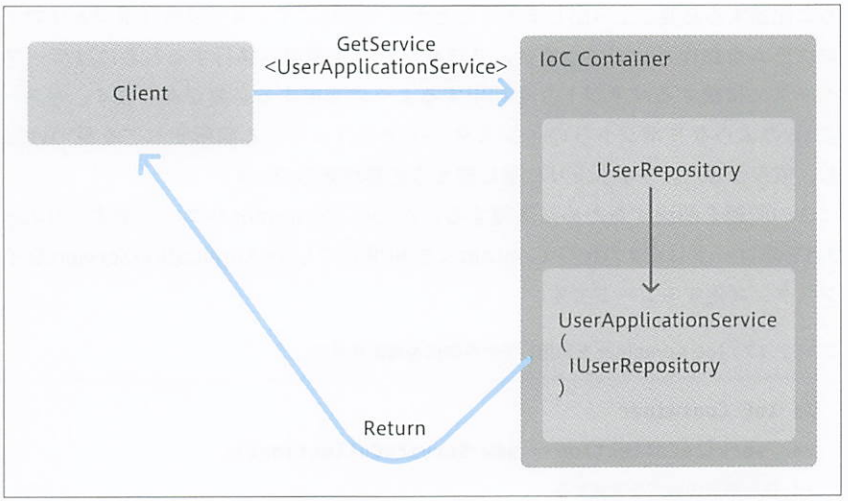

図7.9：IoC Containerによる依存の解決

IoC Containerに対する設定方法はService Locatorと同じくスタートアップスクリプトなどで行います。

DDD 7.5 まとめ

この章ではプログラムとは切っても切れない関係にある重要な概念の依存と、そのコントロールの仕方について学びました。

依存関係はソフトウェアを構築する上で自然と発生するものです。しかしながらその取り扱い方を間違えると手の施しようがないほど硬直したソフトウェアを生み出すことに繋がります。

ソフトウェアは本来柔軟なものです。利用者を取り巻く環境の変化に対応し、利用者を助けるために柔軟に変化できるからこそ「ソフト」ウェアと呼ばれるのです。

依存を恐れる必要はありません。依存すること自体は避けられなくとも、依存の方向性は開発者が絶対的にコントロールできるものです。この章で学んだようにドメインを中心にして、主要なロジックを技術的な要素に依存させないように仕立て上げ、ソフトウェアの柔軟性を保つことを目指してください。