

# Chapter

# 8

## ソフトウェアシステムを 組み立てる

---

ユーザーインターフェースに組み込んで、システムを  
成り立たせます。

利用者がアプリケーションを利用するためにはユー  
ザーインターフェースが必要です。

ユーザーインターフェースと一口にいても、文字を  
ベースにしたものやグラフィックをベースとしたもの  
など、実にさまざまな種類がありますが、これまで取  
り扱ってきたアプリケーションはユーザーインター  
フェースを選びません。文字ベースであってもグラ  
フィックベースであっても、任意のユーザーインター  
フェースに組み込むことが可能です。

そこで、本章ではまず文字ベースのユーザーインター  
フェースに組み込む手順とWeb GUIに組み込む手順  
を紹介します。

これまでに登場した要素をユーザーインターフェース  
に組み込み、ソフトウェアとして成り立たせる方法を  
確認していきましょう。

ソフトウェアの利用者はユーザーインターフェースを通してアプリケーションを利用します。ソフトウェアとして成り立たせるためにはユーザーインターフェースが必要です。

ユーザーインターフェースには沢山の種類があります。たとえば利用者が文字列によって指示を出すCLI（コマンドラインインターフェース）や操作対象がグラフィックによって表現されるGUI（グラフィカルユーザーインターフェース）はその代表です。

本書ではWeb GUIをユーザーインターフェースとしたWebアプリケーションをメインのサンプルにしますが、これはもちろんドメイン駆動設計がWebアプリケーションに限ったものということを意味しません。その処理の指示方法が文字列によるものであっても、グラフィカルなアイコンを駆使して指示されたものであっても、「ユーザを登録する」というビジネスロジックに変わりはないからです。ユーザーインターフェースとして採用するのがCLIであったりGUIであったとしてもドメイン駆動設計の強力な力の恩恵を受けることは可能です。

それを証明するかのように、本章ではまず最初にCLIで問題なくアプリケーションの処理が実行できることを確認し、その後にWeb GUIをベースとした実践的なシステムを構築していきます。また、ソフトウェアはただ動作するだけでは完成ではありません。間違いなく動作することを確認めてこそ本当の完成です。したがって、本章の最後にはアプリケーションが正しく動作する確認のためにユニットテストを実施します。

アプリケーションのインターフェースがCLIやGUI、果てはユニットテストとまったく異なるものであったとしても、ソフトウェアとして成り立たせることが可能であることを確認していきましょう。

## COLUMN

### ソフトウェアとアプリケーションの使い分け

ソフトウェアとアプリケーションは一般的に同じものを指しますが、本書ではドメインの問題を解決するなど利用者の必要を満たす主要なモジュール群をアプリケーションと呼び、それらにユーザーインターフェースなどを付加してシステムとして成り立たせたものをソフトウェアと区別して記述しています。

開発者はしばしばCLIを好んで利用しますが、その理由はさまざまです。たとえばグラフィックに関わる処理の実装が不要で単純であることや、コマンドを正確に入力するよう要求されるため誤操作を起こしづらい、といったことが代表的な理由でしょう。ここでCLIを題材とする理由は、主に前者を理由としています。

さっそくCLIで動作させるコードを確認していきましょう。まずは依存関係の登録を行うコードです。依存関係のコントロールにはIoC Containerを利用します。ServiceCollectionはC#におけるIoC Containerです（[リスト8.1](#)）。

リスト8.1：依存関係の登録を行う

```
class Program
{
    private static ServiceProvider serviceProvider;

    static void Main(string[] args)
    {
        Startup();

        (...略...)
    }

    private static void Startup()
    {
        // IoC Container
        var serviceCollection = new ServiceCollection();
        // 依存関係の登録を行う（以下コメントにて補足）
        // IUserRepositoryが要求されたらInMemoryUserRepositoryを生成して➡
        // 引き渡す（生成したインスタンスはその後使いまわされる）
        serviceCollection.AddSingleton<IUserRepository, ➡
        InMemoryUserRepository>();

        // UserServiceが要求されたら都度UserServiceを生成して引き渡す
```

```

1      serviceCollection.AddTransient<UserService>();
2      // UserApplicationServiceが要求されたら都度➡
3      UserApplicationServiceを生成して引き渡す
4      serviceCollection.AddTransient<UserApplicationService>();
5      // 依存解決を行うプロバイダの生成
6      // プログラムはserviceProviderに依存の解決を依頼する
7      serviceProvider = serviceCollection.BuildServiceProvider();
8  }
9  }

```

IUserRepositoryの依存解決に利用するInMemoryUserRepositoryはAdd Singletonでシングルトンとして登録します。シングルトンでの登録は一度インスタンスを作成したら、そのインスタンスを使いまわす設定です。もしもInMemoryUserRepositoryがシングルトンとして登録されていないと、IoC ContainerはIUserRepositoryの依存解決が要求されるたびに、InMemoryUserRepositoryのインスタンスを新たに生成します。InMemoryUserRepositoryはインメモリで動作するオブジェクトであるので、インスタンスごとにデータを保持します。インスタンスの使いまわしをしないと余所で保存したデータが消えてしまいます。

## ✍ COLUMN

### シングルトンパターンと誤解

シングルトンパターンほど誤解を招きやすいデザインパターンはないでしょう。その誤解というのはシングルトンをstaticの代わりとして扱われてしまうことです。もしシングルトンパターンがstaticの代わりとして使うためのパターンであるなら、素直にstaticを利用すればいいはずです。

シングルトンを利用する理由はインスタンスをひとつに限定しながら、通常のオブジェクトと同様に取り扱えることです。言い換えるなら、staticと異なり、ポリモーフィズムなどのオブジェクト指向プログラミングが実現する機能の恩恵を受けることができます。

シングルトンをstaticの代わりとして扱うのは誤りです。通常のオブジェクト指向プログラミングの流れを組ませるためにシングルトンを利用するのが正しい利用方法です。

UserApplicationServiceはAddTransientで登録します。AddTransientでの登録はオブジェクトが要求されるたびに新しいインスタンスを生成する設定です。

今回のスクリプトではAddSingletonで登録したとしても問題ありませんが、インスタンスの生存期間はなるべくなら短くしておくことが管理をしやすくするコツです。パフォーマンスに問題がないようであれば、都度インスタンスを生成したとしても問題ないでしょう。

なお、AddSingletonやAddTransientといったメソッドはC#のIoC ContainerライブラリであるServiceCollectionに定義されたメソッド名にすぎません。皆さんがご利用のプログラミング言語やIoC Containerライブラリによって命名は異なりますので、対応するメソッドを確認してください。

### 8.2.1 メインの処理を実装する

スタートアップスクリプトにて依存の設定をしたのちは、いよいよメインの処理です（リスト8.2）。

リスト8.2: メインとなる処理を実装する

```
class Program
{
    private static ServiceProvider serviceProvider;
    static void Main(string[] args)
    {
        Startup();

        while (true)
        {
            Console.WriteLine("Input user name");
            Console.Write(">");
            var input = Console.ReadLine();
            var userApplicationService = serviceProvider.GetService<
<UserApplicationService>();
            var command = new UserRegisterCommand(input);
            userApplicationService.Register(command);

            Console.WriteLine("-----");
            Console.WriteLine("user created:");
```

```

1      Console.WriteLine("-----");
2      Console.WriteLine("user name:");
3      Console.WriteLine("- " + input);
4      Console.WriteLine("-----");

5
6      Console.WriteLine("continue? (y/n)");
7      Console.Write(">");
8      var yesOrNo = Console.ReadLine();
9      if (yesOrNo == "n")
10     {
11         break;
12     }
13 }
14 }
15
APP
(…略…)
}

```

IoC Container (serviceProvider) からUserApplicationServiceを取得し、ユーザ登録処理を呼び出しています。インスタンスを直接生成せずにIoC Container経由でインスタンスを取得するようにすることで、スタートアップスクリプトなどに依存関係の設定に関する記述を集中させられます。

プロダクション用のリレーショナルデータベースに接続するリポジトリを使用したいときはスタートアップスクリプトを変更します ([リスト8.3](#))。

**リスト8.3** : リポジトリを差し替える

```

class Program
{
    (…略…)

    private static void Startup()
    {
        var serviceCollection = new ServiceCollection();
    }
}

```

```
// UserRepositoryに差し替え
// serviceCollection.AddSingleton<IUserRepository, ➔
InMemoryUserRepository>();
serviceCollection.AddTransient<IUserRepository, ➔
UserRepository>();
serviceCollection.AddTransient<UserService>();
serviceCollection.AddTransient<UserApplicationService>();

serviceProvider = serviceCollection.BuildServiceProvider();
}
}
```

このようにIoC Containerを活用することでメインの処理（リスト8.2）にまったく手を加えることなく、データストアの変更を実現できるのです。

## DDD

### 8.3

## MVCフレームワークに組み込んでみよう

より本格的なソフトウェアとして成り立たせるべくWebアプリケーションを組み立ててみましょう。

Webアプリケーションを開発する際には、Webフレームワークを利用するのが一般的です。C#におけるWebフレームワークはASP.NETというフレームワークがメジャーです。ASP.NETにはいくつかのバージョンが存在していますが、本書ではASP.NET Core MVCを題材にします。

ここではCLIのときと同じように、まずスタートアップスクリプトにて依存関係を設定し、システムの利用者のアクションに応じて適宜必要なインスタンスをIoC Containerに要求するように設定と実装を確認していきます。その後に実際に利用者の操作から処理を実行する箇所を確認します。処理の流れは図8.1です。予め確認しておくといでしょう。

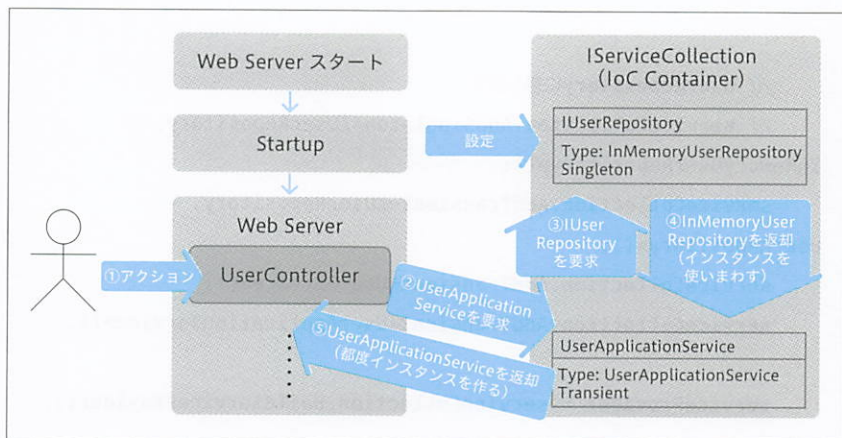


図 8.1 : MVC フレームワークと IoC Container の連携イメージ

ASP.NET Core MVC は MVC フレームワークとして一般的な機能を網羅しています。皆さんが普段お使いのプログラミング言語やフレームワークがまったく異なるものであったとしても、代替となる機能は存在するでしょう。適宜読み換えながら読み進めていってください。

### 8.3.1 依存関係を設定する

依存関係の設定は CLI のときと同じようにスタートアップスクリプト (リスト 8.4) で設定します。ASP.NET Core MVC では予めスタートアップスクリプトとして Startup クラスが用意されており、サーバー起動時にこの処理が実行されます。

リスト 8.4 : ASP.NET Core MVC が提供している Startup クラス

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }
```

```
// This method gets called by the runtime. Use this method →  
to add services to the container.  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddControllersWithViews();  
  
    services.AddSpaStaticFiles(configuration =>  
    {  
        configuration.RootPath = "ClientApp/build";  
    });  
}  
  
(…略…)  
}
```

StartupクラスのConfigureServicesメソッドはIoC Containerを利用して依存関係を登録する箇所です。このメソッドにリポジトリなどの依存解決の設定を行っていきます（[リスト8.5](#)）。

**リスト8.5**： MVCフレームワークのスタートアップスクリプトで依存解決の設定をする

```
public class Startup  
{  
    (…略…)  
  
    public void ConfigureServices(IServiceCollection services)  
    {  
        services.AddControllersWithViews();  
  
        services.AddSpaStaticFiles(configuration =>  
        {  
            configuration.RootPath = "ClientApp/build";  
        });  
    }  
}
```

```
// リポジトリやアプリケーションサービスの依存解決を設定する
services.AddSingleton<IUserRepository, ➡
InMemoryUserRepository>();
services.AddTransient<UserService>();
services.AddTransient<UserApplicationService>();
}
}
```

**リスト 8.5** に追加した依存関係の登録は **リスト 8.1** で行っていた登録と同じ内容です。設定自体に問題はありませんが、このままではプロダクション用としてデータベースに接続して動作させたいとき、この設定スクリプトに変更を加える必要があります。現段階では登録されているリポジトリはたったひとつなのであまり問題にはなりませんが、システムが大きくなるにつれて比例するようにリポジトリの数は増えてきます。それらすべてを起動のたびにいちいち設定を書き直すのは大きな手間です。

こうした手間を省くためにデバッグ用とプロダクション用でコンフィグスクリプトを分けるのはよいアイデアです (**リスト 8.6**、**リスト 8.7**)。

#### リスト 8.6 : テスト用の設定スクリプト

```
public class InMemoryModuleDependencySetup : IDependencySetup
{
    public void Run(IServiceCollection services)
    {
        SetupRepositories(services);
        SetupApplicationServices(services);
        SetupDomainServices(services);
    }

    private void SetupRepositories(IServiceCollection services)
    {
        services.AddSingleton<IUserRepository, ➡
InMemoryUserRepository>();
```

```
}

private void SetupApplicationServices(IServiceCollection
services)
{
    services.AddTransient<UserApplicationService>();
}

private void SetupDomainServices(IServiceCollection services)
{
    services.AddTransient<UserService>();
}
}
```

リスト 8.7 : プロダクション用の設定スクリプト

```
public class SqlConnectionDependencySetup : IDependencySetup
{
    private readonly IConfiguration configuration;

    public SqlConnectionDependencySetup(IConfiguration ➡
configuration)
    {
        this.configuration = configuration;
    }

    public void Run(IServiceCollection services)
    {
        SetupRepositories(services);
        SetupApplicationServices(services);
        SetupDomainServices(services);
    }
}
```

```

1 private void SetupRepositories(IServiceCollection services)
2 {
3     services.AddTransient<IUserRepository, SqlUserRepository>();
4 }
5
6 private void SetupApplicationServices(IServiceCollection →
7 services)
8 {
9     services.AddTransient<UserApplicationService>();
10 }
11
12 private void SetupDomainServices(IServiceCollection services)
13 {
14     services.AddTransient<UserService>();
15 }
16 }

```

これらのスクリプトはプロジェクトの構成ファイルによって切り替えます。ASP.NET Coreではappsettings.jsonというjson形式のファイルがその構成ファイルにあたります（[リスト8.8](#)）。

**リスト8.8**：利用する設定スクリプトを構成ファイルに記述する

```

{
  "Dependency": {
    "SetupName": "InMemoryModuleDependencySetup"
  }
}

```

スタートアップスクリプトでは[リスト8.8](#)を読み込み、実施する依存関係の設定処理を切り替えます（[リスト8.9](#)、[リスト8.10](#)）。

リスト8.9: リスト8.8の設定により設定スクリプトを選定するモジュール

```
class DependencySetupFactory
{
    public IDependencySetup CreateSetup(IConfiguration ➡
configuration)
    {
        var setupName = configuration["Dependency:SetupName"];
        switch (setupName)
        {
            case nameof(InMemoryModuleDependencySetup):
                return new InMemoryModuleDependencySetup();

            case nameof(SqlConnectionDependencySetup):
                return new SqlConnectionDependencySetup(configuration);

            default:
                throw new NotSupportedException(setupName + " is ➡
not registered.");
        }
    }
}
```

リスト8.10: スタートアップスクリプトはリスト8.9を利用する

```
public class Startup
{
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        // 依存関係の設定スクリプトを取得して実行
        var factory = new DependencySetupFactory();
        var setup = factory.CreateSetup(Configuration);
        setup.Run(services);
    }
}
```

```

1  services.AddControllersWithViews();
2
3
4  services.AddSpaStaticFiles(configuration =>
5  {
6      configuration.RootPath = "ClientApp/build";
7  });
8  }
9
10 (...略...)
11 }

```

### 8.3.2 コントローラを実装する

さあ、依存関係の設定をしている箇所の確認が済んだところで、いよいよコントローラの実装の確認です。まずはユーザ登録のデータを受け取り、ユーザを登録する処理（アクション）を確認しましょう。

多くのMVCフレームワークはIoC Containerと連携しており、IoC Containerに登録されたオブジェクトをコントローラのコンストラクタで受け取ることができます。UserApplicationServiceを利用したい場合、[リスト8.11](#)のようにコンストラクタで受け取り、アクションから呼び出すように記述します。

リスト8.11：ユーザを作成するアクション

```

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

    // IoC Containerと連携して依存の解決が行われる
    public UserController(UserApplicationService ➡
    userApplicationService)

```

```

{
    this.userApplicationService = userApplicationService;
}

(...略...)

[HttpPost]
public void Post([FromBody] UserPostRequestModel request)
{
    var command = new UserRegisterCommand(request.UserName);
    userApplicationService.Register(command);
}
}

```

Postアクションの引数であるUserPostRequestModelはビューから受け渡されるデータがバインドされるオブジェクトです。このオブジェクトはアプリケーションサービスが受け取るUserRegisterCommandオブジェクトとほとんど同じデータ構造ですので、それを使いまわすアイデアも思い浮かびますが、フロントから引き渡されるデータの入れ物と、アプリケーションサービスのふるまいを実行するためのコマンドオブジェクトは用途が違うものです。特に理由がないようであれば、オブジェクトの使いまわしをしない方がよいでしょう。

その他の処理も確認してみましょう（[リスト8.12](#)）。

**リスト8.12**：コントローラのその他の処理

```

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

```

```

1 public UserController(UserApplicationService ➡
2 userApplicationService)
3 {
4     this.userApplicationService = userApplicationService;
5 }
6
7 [HttpGet]
8 public UserIndexResponseModel Index()
9 {
10     var result = userApplicationService.GetAll();
11     var users = result.Users.Select(x => new ➡
12 UserResponseModel(x.Id, x.Name)).ToList();
13
14     return new UserIndexResponseModel(users);
15 }
16
17 [HttpGet("{id}")]
18 public UserGetResponseModel Get(string id)
19 {
20     var command = new UserGetCommand(id);
21     var result = userApplicationService.Get(command);
22
23     var userModel = new UserResponseModel(result.User);
24
25     return new UserGetResponseModel(userModel);
26 }
27
28 (...略...)
29
30 [HttpPut("{id}")]
31 public void Put(string id, [FromBody] UserPutRequestModel ➡
32 request)
33 {

```

```
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);
}
}
```

いずれのアクションも、コントローラはフロントからのデータをビジネスロジックが必要とする入力データへ変換する作業に集中しています。ビジネスロジックをアプリケーションサービスに寄せるようになると、結果としてコントローラのコードはリスト8.11やリスト8.12とほとんど同じようなコードばかりのシンプルなものになるでしょう。

以上でCRUD機能をもった最小限のWebアプリケーションは完成です。

## COLUMN

### コントローラの責務

コントローラの責務は入力の変換です。たとえばゲーム機を思い浮かべてみてください (図8.2)。

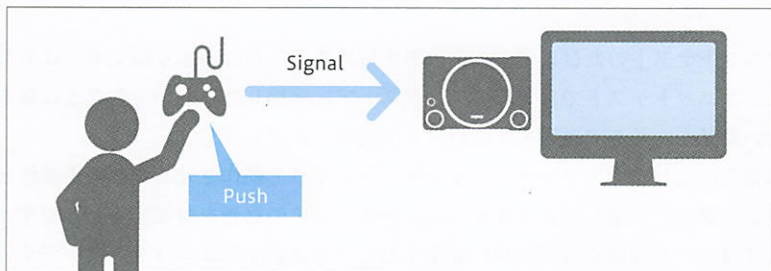


図8.2: コントローラ役目

ゲームのプレイヤーはコントローラのボタンを押すことでキャラクターを動かします。このときコントローラは「ボタンが押された」という事実をそのままゲーム機に送っているわけではありません。コントローラは「ボタンが押された事実」をゲーム機が解釈できる電気信号に変換して送信しているのです。ゲーム機のコントローラの責務はプレイヤーの入力をゲーム機が理解できる形に変換する作業です。

MVCパターンのコントローラもこれと同じです。コントローラはユーザからの入力をモデルが要求するメッセージに変換し、モデルに伝えることが責務です。もしもコントローラがそれ以上のことをこなしているように見受けられたのなら、ドメインの重要な知識やロジックがコントローラに漏れ出している可能性を疑うべきです。

## DDD 8.4

# ユニットテストを書こう

ソフトウェアを完成とするには意図したとおりに動くことを証明する必要があります。プログラムが正しく動作することを証明するのにユニットテストは最高のツールです。百聞は一見に如かずという言葉もあるとおり、動作することを口頭で説明するも、正しく動作することを実証してしまう方がずっと簡単です。

この最小限のアプリケーションにもユニットテストを用意して、完成を証明しましょう。

## 8.4.1 ユーザ登録処理のユニットテスト

ユニットテストのたびにテストデータを用意することはあまり現実的ではありません。ユニットテストでは実際のデータストアに接続したりといったことは基本的に行いません。そこで登場するのがテスト用のリポジトリです。

本書ではこれまでアプリケーションをインメモリで動作させることの重要性和そのための努力を何度も訴えてきました。いよいよその真価を発揮するときです。

まずはユーザ登録処理が正常に完了することを確認するユニットテストです（[リスト8.13](#)）。なお、C#ではユニットテストクラスに [TestClass] や [TestMethod] といったアトリビュートを付与します。

```
[TestClass]
public class UserRegisterTest
{
    [TestMethod]
    public void TestSuccessMinUserName()
    {
        var userRepository = new InMemoryUserRepository();
        var userService = new UserService(userRepository);
        var userApplicationService = new UserApplicationService➡
(userRepository, userService);

        // 最短のユーザ名（3文字）のユーザが正常に生成できるか
        var userName = "123";
        var minUserNameInputData = new UserRegisterCommand➡
(userName);
        userApplicationService.Register(minUserNameInputData);

        // ユーザが正しく保存されているか
        var createdUserName = new UserName(userName);
        var createdUser = userRepository.Find(createdUserName);
        Assert.IsNotNull(createdUser);
    }

    [TestMethod]
    public void TestSuccessMaxUserName()
    {
        var userRepository = new InMemoryUserRepository();
        var userService = new UserService(userRepository);
        var userApplicationService = new UserApplicationService➡
(userRepository, userService);
```

```

1 // 最長のユーザ名（20文字）のユーザが正常に生成できるか
2 var userName = "12345678901234567890";
3 var maxUserNameInputData = new UserRegisterCommand➡
4 (userName);
5 userApplicationService.Register(maxUserNameInputData);
6
7 // ユーザが正しく保存されているか
8 var createdUserName = new UserName(userName);
9 var maxUserNameUser = userRepository.Find(createdUserName);
10 Assert.IsNotNull(maxUserNameUser);
11
12 }
13
14 }

```

ユーザ登録処理で確認すべきは「生成されたユーザが保存されているか」ということです。ユーザ名には文字数に関して条件があるので境界値の検査も同時に行っています。生成されたユーザが保存されているかを確認するためにインメモリのリポジトリを用いて処理を実行し、処理が完了したのちにリポジトリに対して問い合わせをしています。

テストしたい内容によってはリポジトリに収められた必要な情報を取得するためのメソッドが提供されていないことがあります。そういったときにはリポジトリに対する問い合わせを行わず、リポジトリがデータを保管しているフィールドを公開することで対応できます（[リスト8.14](#)）。

**リスト8.14**：テストで確認するためにテスト用リポジトリの内部データを公開する

```

public class InMemoryUserRepository : IUserRepository
{
    // 直接のデータ保管先となる連想配列を公開している
    public Dictionary<UserId, User> Store { get; } = new ➡
    Dictionary<UserId, User>();

    (...略...)
}

```

このオブジェクトを利用したテストコードはリポジトリに問い合わせを行わず、リポジトリのプロパティを直接操作するように変化します（リスト8.15）。

リスト8.15：リスト8.14を利用してユーザが保存されたかを確認する

```
[TestMethod]
public void TestSuccessMinUserName()
{
    var userRepository = new InMemoryUserRepository();
    var userService = new UserService(userRepository);
    var userApplicationService = new UserApplicationService➡
(userRepository, userService);

    // 最短のユーザ名（3文字）のユーザが正常に生成できるか
    var userName = "123";
    var minUserNameInputData = new UserRegisterCommand(userName);
    userApplicationService.Register(minUserNameInputData);

    // ユーザが正しく保存されているか
    var createdUser = userRepository.Store.Values
        .FirstOrDefault(user => user.Name.Value == userName);
    Assert.IsNotNull(createdUser);
}
```

テスト用のリポジトリがデータ保管先としているフィールドを外部から操作できるようにすることは、きめ細かい検索を可能にし、テスト用モジュールの利便性を向上させます。フィールドを無暗に公開することは避けるべきですが、通常利用されるのはIUserRepositoryであるためリスト8.14のStoreプロパティを操作はできません。InMemoryUserRepositoryを直接利用するのはテストコードだけです。Storeプロパティのように直接のデータ保管オブジェクトを公開しても問題は起きないのです。

さて、正常系のテストを確認したのちは異常系のテストです。ユーザ生成処理はパラメータによってはエラーを発生させることがあります。エラーの条件をまとめると次のリストのとおりになります。

- 登録しようとしたユーザ名の長さが3文字以上20文字以下でない
- 既に登録されているユーザ名である

異常系は正常系に比べて多く検査する項目があるので若干複雑になります。まずはユーザ名の長さが異常なときの動作の確認テストです（リスト8.16）。

リスト8.16：ユーザ名の長さに関するエラーをテストする

```
[TestClass]
public class UserRegisterTest
{
    (...略...)

    [TestMethod]
    public void TestInvalidUserNameLengthMin()
    {
        var userRepository = new InMemoryUserRepository();
        var userService = new UserService(userRepository);
        var userApplicationService = new UserApplicationService➡
(userRepository, userService);

        bool exceptionOccurred = false;
        try
        {
            var command = new UserRegisterCommand("12");
            userApplicationService.Register(command);
        }
        catch
        {
            exceptionOccurred = true;
        }

        Assert.IsTrue(exceptionOccurred);
    }
}
```

```
[TestMethod]
public void TestInvalidUserNameLengthMax()
{
    var userRepository = new InMemoryUserRepository();
    var userService = new UserService(userRepository);
    var userApplicationService = new UserApplicationService➡
(userRepository, userService);

    bool exceptionOccured = false;
    try
    {
        var command = new UserRegisterCommand➡
("123456789012345678901");
        userApplicationService.Register(command);
    }
    catch
    {
        exceptionOccured = true;
    }

    Assert.IsTrue(exceptionOccured);
}
}
```

ユーザ名が下限値よりも短いときと上限値よりも長いときのふたとおりを確認しています。正常系と合わせれば境界値検査が網羅されていることがわかります。

最後のテストはユーザ名が重複しているときの動作を確認するテストです（リスト8.17）。

リスト8.17: ユーザ名の重複に関するエラーをテストする

```
[TestClass]
public class UserRegisterTest
{
    (...略...)

    [TestMethod]
    public void TestAlreadyExists()
    {
        var userRepository = new InMemoryUserRepository();
        var userService = new UserService(userRepository);
        var userApplicationService = new UserApplicationService→
(userRepository, userService);

        var userName = "test-user";
        userRepository.Save(new User(
            new UserId("test-id"),
            new UserName(userName)
        ));

        bool exceptionOccured = false;
        try
        {
            var command = new UserRegisterCommand(userName);
            userApplicationService.Register(command);
        }
        catch
        {
            exceptionOccured = true;
        }

        Assert.IsTrue(exceptionOccured);
    }
}
```

ユニットテストを作成するとそこにはどういった入力をすればよいのか、その入力によって得られる結果はどうあるべきか、といったことが記述されます。あまり考えたくないことですがドキュメントの類が一切存在しないプロダクトにおいては、ユニットテストがそのロジックのあるべき姿を語る最後の手掛かりになるでしょう。

DDD

8.5

## まとめ

この章ではこれまで解説してきたパターンをまとめあげ、アプリケーションを実際にユーザーインターフェースへ組み込み、ソフトウェアとして成り立たせる方法を確認しました。

アプリケーションにとってユーザーインターフェースは交換可能なものです。ユーザーインターフェースをソフトウェアの核心から分離し、オブジェクトの責務を明瞭にすることはソフトウェアの未来を守るに等しい行為です。

実際にユーザーインターフェースを交換するような事態は稀ですが、ユーザーインターフェースを交換可能であるということはアプリケーションが単独で実行できるということで、つまりユニットテストを実施できるということに他なりません。

ユニットテストがそのままソフトウェアの品質向上になるわけでは必ずしもありませんが、ユニットテストができるような形に仕立てることは品質向上の第一歩です。

ソフトウェアに対する変化の要求を満たすために、開発者はリファクタリングを余儀なくされることがあります。そのとき、ユニットテストが準備されていれば、リファクタリングによってアプリケーションを破壊していないかを確認しながら、作り変えることができるのです。

ドメインの変化をドメインオブジェクトまできっちりと伝え、システムがドメインと同期するように仕向けるためにユニットテストを用意しておくことは大事な戦略です。

## 本当に稀な怪談話

「そうはいうものの、ユーザーインターフェースを交換するような事態なんて起こらない」と高を括ってはいないでしょうか。残念ながら実際にユーザーインターフェースを交換する羽目になった経験が筆者にはあります。

そのプロダクトは主力のサービスで、ASP.NET Web Formsで構築されたソフトウェアでした。Web Formsは技術者の獲得が難しく、より一般的なMVCに乗り換える必要がありました。運命だったとは信じたくないのですが、そのお鉢はなぜか筆者に回ってきました。

15年程の手垢にまみれたコードは筆者を恐怖に陥れました。すべてのロジックはユーザーインターフェースに記述されており、似たようなロジックが至るところに記述されています。もともと同一だったロジックはそれぞれの画面の事情にしたがって出鱈目な進化を遂げていたのです。

当然のようにユニットテストはありません。それどころかドキュメントすらも存在しなかったのです。結局のところ筆者にできたのは、愚直にすべてのコードを読み解いて、ゼロベースでコードを組み立てることでした。