

# 크린-테스트

.NET, NUnit

AUSG 5기 이진성

# 발표자를 소개합니다.

---



## 이진성 AUSG 5기

- 테스트는 어떻게 코드를 더럽히는가?
- 테스트는 어떻게 더러워 지는가?

# 테스트는 어떻게 코드를 더럽히는가

---



## 테스트 코드를 작성하고 싶은 신입 개발자 이야기

# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

어떤 신입 개발자가 간단한 계산기 코드를 개발해야 했어요.  
이렇게 쉬운 코드라니!

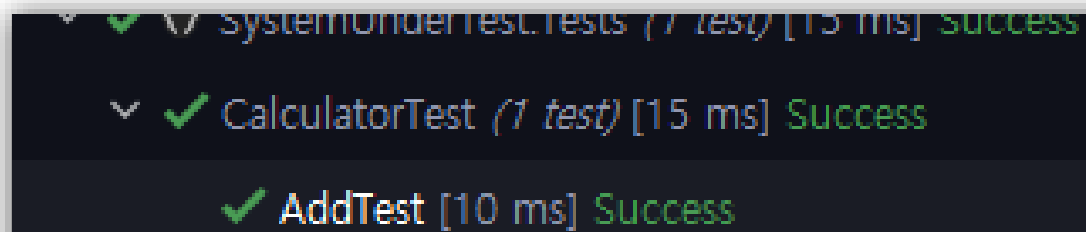
# 테스트 코드는 쉽다.

```
public class CalculatorTest
{
    [Test]
    public void AddTest()
    {
        var result = Calculator.Add(1, 2);

        Assert.Equals(result, 3);
    }
}
```

그래서 기왕에 작성하는 겸 테스트 코드도 쓰기로 했죠.

# 테스트 코드는 쉽다.



```
✓✓✓ SystemUnderTest.Tests (1 test) [15 ms] Success
  ✓ CalculatorTest (1 test) [15 ms] Success
    ✓ AddTest [10 ms] Success
```

첫 테스트 코드를 성공적으로 통과 시켰습니다.

## 코드 리뷰 코멘트 중...

☺ : 저희는 모든 메서드에 로거를 붙여야 해요.

헉!!



## 코드 리뷰 코멘트 중...

☺ : 저희는 모든 메서드에 로거를 붙여야 해요.

조금 이해할 수 없는 내용이지만

## 코드 리뷰 코멘트 중...

☺ : 저희는 모든 메서드에 로거를 붙여야 해요.

입사한지 얼마 되지 않았고 컨벤션이 그렇다는데 그냥 따라 주기로 합니다.

# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static int Add(int a, int b)
    {
        Logger.Info($"유저({ServerContext.User().Name})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

코드리뷰 코멘트를 반영하여 로깅 코드를 작성하고

# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static int Add(int a, int b)
    {
        Logger.Info($"유저({ServerContext.User().Name})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

테스트 코드를 실행해보려고 합니다.

# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static int Add(int a, int b)
    {
        Logger.Info($"유저({ServerContext.User().Name})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

문제될 만한 코드는 없는 것 같아요.

# 테스트 코드는 쉽다.

```
✓ - SystemUnderTest.Tests (2 tests) [150 ms] Failed: 1 test failed
  > ✓ CalculatorLoggerTest (1 test) [115 ms] Success
  ✓ - CalculatorTest (1 test) [34 ms] Failed: One or more child tests had errors: 1 test failed
    - AddTest [29 ms] Failed: System.OperationCanceledException : Couldn't fetch user principal for this call context.
```

??????

테스트 코드는 쉽다.



어떻게...?

# 테스트 코드는 쉽다.



```
System.OperationCanceledException : Couldn't fetch user principal for this call context.  
at SystemUnderTest.ServerContext.User() in D:\git\WhyTestCodWillBeDirty\SystemUnderTest\Ch1.cs:line 24  
at SystemUnderTest.Calculator.<>c.<.cctor>b__4_0() in D:\git\WhyTestCodWillBeDirty\SystemUnderTest\Ch1.cs:line 30  
at SystemUnderTest.Calculator.Add(Int32 a, Int32 b) in D:\git\WhyTestCodWillBeDirty\SystemUnderTest\Ch1.cs:line 36  
at SystemUnderTest.Tests.CalculatorTest.AddTest() in D:\git\WhyTestCodWillBeDirty\SystemUnderTest.Tests\UnitTest1.cs:line 12
```

조금 당황했지만,  
영리한 신입 개발자는 데이터를 받아오는 부분을 분리하도록 했어요.



# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static Func<string> UserName = () => ServerContext.User().Name;

    public static int Add(int a, int b)
    {
        Logger.Info($"유저({UserName})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

이렇게 유저 이름을 가져오는 방법을 별도로 분리하고

# 테스트 코드는 쉽다.

```
public class CalculatorTest
{
    [Test]
    public void AddTest()
    {
        Calculator.UserName = ( ) => "페페's Test";

        var result = Calculator.Add(1, 2);

        Assert.AreEqual(3, result);
    }
}
```

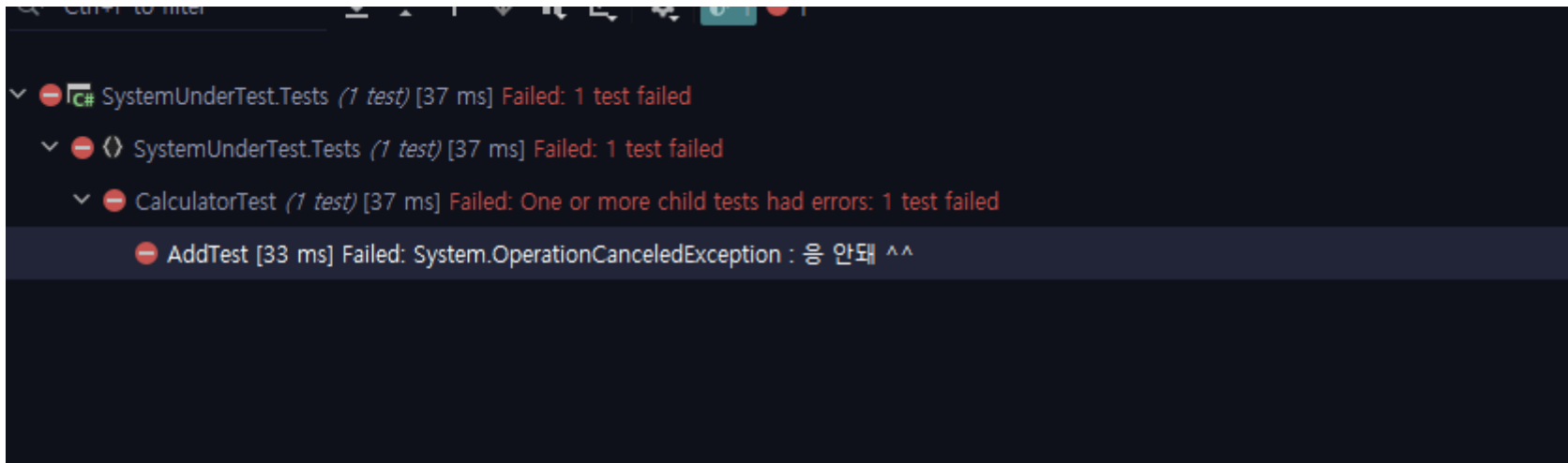
테스트 코드에서 이렇게 바꿔주면 되겠죠?

테스트 코드는 쉽다.



역시 천재 신입 개발자인 것 같아요.

# 테스트 코드는 쉽다.



???????

# 테스트 코드는 쉽다.



```
System.OperationCanceledException : 응 안돼 ^^  
at SystemUnderTest.Logger.Info(String message) in D:\git\WhyTestCodWillBeDirty\SystemUnderTest\Ch1.cs:line 9  
at SystemUnderTest.Calculator.Add(Int32 a, Int32 b) in D:\git\WhyTestCodWillBeDirty\SystemUnderTest\Ch1.cs:line 34  
at SystemUnderTest.Tests.CalculatorTest.AddTest() in D:\git\WhyTestCodWillBeDirty\SystemUnderTest.Tests\UnitTest1.cs:line 12
```

믿었던 Logger에게 배신당한 신입개발자.

# 테스트 코드는 쉽다.

```
public class Calculator
{
    public static Func<string> UserName = () => ServerContext.User().Name;

    public static Action<string> Info = Logger.Info;

    public static int Add(int a, int b)
    {
        Info($"유저({UserName})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

```
public class CalculatorTest
{
    [Test]
    public void AddTest()
    {
        Calculator.UserName = () => "페페's Test";
        Calculator.Info = _ => { };

        var result = Calculator.Add(1, 2);

        Assert.AreEqual(3, result);
    }
}
```

여기서 포기할 순 없었습니다!

# 테스트 코드는 쉽다.



역시 천재 개발자예요!

# 테스트 코드는 쉽다.



하지만 이미 알아차리신 분들이 있겠지만,



# 테스트 코드는 쉽다.

테스트 가능한 코드가 항상 좋은 코드인가? = X

```
public class Calculator
{
    public static Func<string> UserName = () => ServerContext.User().Name;
    public static Action<string> Info = Logger.Info;
    public static int Add(int a, int b)
    {
        Info($"유저({UserName})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

정말 테스트를 위해 전역 상태를 노출하는 것은 적절하지 못해 보여요.

# 테스트 코드는 쉽다.

테스트 가능한 코드가 항상 좋은 코드인가? = X

```
public class Calculator
{
    public static Func<string> UserName = () => ServerContext.User().Name;
    public static Action<string> Info = Logger.Info;
    public static int Add(int a, int b)
    {
        Info($"유저({UserName})가 Calculator.Add를 호출했습니다.");
        return a + b;
    }
}
```

전역 상태는 프로그램의 상태 예측을 어렵게 만들기 때문입니다.  
다른 프로덕션/테스트 코드에서 수정 가능해지기 때문이죠.

# 테스트 코드는 쉽다.

## 코드 리뷰 코멘트 중...

☺ : 로거도 테스트해주세요!

만약 이러한 코멘트가 있었다면

# 테스트 코드는 쉽다.

```
public class CalculatorLoggerTest
{
    [Test]
    public void AddTest()
    {
        var result = false;
        Calculator.UserName = () => "귀찮게 하네";
        Calculator.Info = message =>
        {
            result = "유저(귀찮게 하네)가 Calculator.Add를 호출했습니다." == message;
        };

        Calculator.Add(1, 2);

        Assert.True(result);
    }
}
```

이러한 코드가 생성할 수 있고, 이는 병렬 테스트를 실행시켰을 때 전역 상태로 인해 결과가 테스트 대상에 격리에 실패하게 됩니다.

# 테스트 코드는 쉽다.

```
public class CalculatorLoggerTest
{
    [Test]
    public void AddTest()
    {
        var result = false;
        Calculator.UserName = () => "귀찮게 하네";
        Calculator.Info = message =>
        {
            result = "유저(귀찮게 하네)가 Calculator.Add를 호출했습니다." == message;
        };

        Calculator.Add(1, 2);

        Assert.True(result);
    }
}
```

테스트 대상을 격리해야 하는 이유는  
항상 일관된 결과를 만들기 위해서이기도 하구요.

# 테스트 코드는 쉽다.

```
public class CalculatorLoggerTest
{
    [Test]
    public void AddTest()
    {
        var result = false;
        Calculator.UserName = () => "귀찮게 하네";
        Calculator.Info = message =>
        {
            result = "유저(귀찮게 하네)가 Calculator.Add를 호출했습니다." == message;
        };

        Calculator.Add(1, 2);

        Assert.True(result);
    }
}
```

이는 단순히 전역 상태에 대해서만 적용되는 이야기는 아닙니다.

# 테스트 코드는 쉽다.

```
public class UserRepository
{
    public UserRepository()
    {
        _database = Database.Target(EnvironmentTypes.Production);
    }
}
```

이러한 코드가 있을 때, UserRepository는 이미 Production Database에 직접 의존을 하고 있기 때문에 격리되었다고 할 수 없습니다.

# 테스트 코드는 쉽다.

```
public class UserRepository
{
    public UserRepository()
    {
        _database = Database.Target(EnvironmentTypes.Production);
    }
}
```

망분리 환경에선 데이터베이스에 접근할 수 없을 수 있고, 또한 데이터 베이스가 죽거나 변경됨에 의해 UserRepository는 영향을 받기 때문이죠.



# 테스트 코드는 쉽다.

```
public class UserRepository
{
    private IDatabase Database { get; set; }

    public UserRepository()
    {
        if (Environment.GetEnvironmentVariable("ENV") == "testing")
        {
            Database = Database.InMemoryDatabase();
        }
        else
        {
            Database = Database.ProductionDatabase();
        }
    }
}
```

테스트를 하기 위해 대상 클래스를 격리시키려고  
이러한 코드를 작성할 수 있습니다.

# 테스트 코드는 쉽다.

```
public async Task UpdateUserNameByUserId(UserId id, string userName)
{
    var user = await Database.Users.Query().FindByIdAsync(id.ToInt());

    user.Name = userName;
    await user.SaveAsync();

    if (Environment.GetEnvironmentVariable("ENV") != "testing")
        EventNotifier.Send(new UpdateUserNameCommand(id, userName));
}
```

(테스트를 위해) 오류를 피하기 위해서 이런 코드로 작성할 수 있죠.

# 테스트 코드는 쉽다.

```
public async Task UpdateUserNameById(UserId id, string userName)
{
    var user = await Database.Users.Query().FindByIdAsync(id.ToInt);

    user.Name = userName;
    await user.SaveAsync();

    if (Environment.GetEnvironmentVariable("ENV") != "testing")
        EventNotifier.Send(new UpdateUserNameCommand(id, userName));
}
```

ENV 환경 변수가 testing으로 정의된 환경에서는  
격리된 것이 맞긴 하니까요.

# 테스트 코드는 쉽다.

**✕: 테스트를 위한 코드를 프로덕션에 작성하지 마세요.**

- 프로덕션 코드에 테스트 환경(변수, 상태, 상수)에 대해 의존하지 마세요.
  - 테스트를 하기위해 불필요한 상태나 상태의 노출을 하지 마세요.

물론 현실(레거시)는 이미 테스트 가능한 코드가 아닐 수 있어요.

# 테스트 코드는 쉽다.

**✕: 테스트를 위한 코드를 프로덕션에 작성하지 마세요.**

- 프로덕션 코드에 테스트 환경(변수, 상태, 상수)에 대해 의존하지 마세요.
  - 테스트를 하기위해 불필요한 상태나 상태의 노출을 하지 마세요.

그렇다고 해서 테스트를 위해 프로덕션 코드를 더럽힌다면,  
우리가 목적했던 것에서 본말 전도하는 것이겠죠.

# 테스트 코드는 쉽다.

## ✓: 테스트 가능한 코드를 작성하세요.

- 대상 코드(클래스, 함수)는 전역 상태나 함수에 대해 **직접 의존**보단 되도록 추상 인터페이스에 **간접 의존(DI)**하세요.
  - 객체 생성에 어플리케이션 로직을 혼용하지 마세요!
  - 전역 상태/정적 함수에 의존을 피하세요.

하지만 우리가 작성하는 코드부터는 위와 같은 테스트 가능한 코드를 작성하면 좋을 것 같아요.

# 테스트 코드는 쉽다.

## ✓: 테스트 가능한 코드를 작성하세요.

- 대상 코드(클래스, 함수)는 전역 상태나 함수에 대해 **직접 의존**보단 되도록 추상 인터페이스에 **간접 의존(DI)**하세요.
  - 객체 생성에 어플리케이션 로직을 혼용하지 마세요!
    - 전역 상태/정적 함수에 의존을 피하세요.

객체지향 원칙을 이미 지키고 있다면 테스트 가능한 코드일 확률이 큼니다.  
OCP/ISP/DIP가 어느정도 이를 보장해주기 때문이죠.

# 테스트 코드는 쉽다.

## ✓: 테스트 가능한 코드를 작성하세요.

- 대상 코드(클래스, 함수)는 전역 상태나 함수에 대해 **직접 의존**보단 되도록 추상 인터페이스에 **간접 의존(DI)**하세요.
  - 객체 생성에 어플리케이션 로직을 혼용하지 마세요!
  - 전역 상태/정적 함수에 의존을 피하세요.

위와 같은 원칙은 사실 모두가 알고 있지만,  
구현에 초점을 맞춰 코딩하다 보면 잊게 되는 것 같아요.



# 테스트 코드는 쉽다.

## ✓: 테스트 가능한 코드를 작성하세요.

- 대상 코드(클래스, 함수)는 전역 상태나 함수에 대해 **직접 의존**보단 되도록 추상 인터페이스에 **간접 의존(DI)**하세요.
  - 객체 생성에 어플리케이션 로직을 혼용하지 마세요!
    - 전역 상태/정적 함수에 의존을 피하세요.

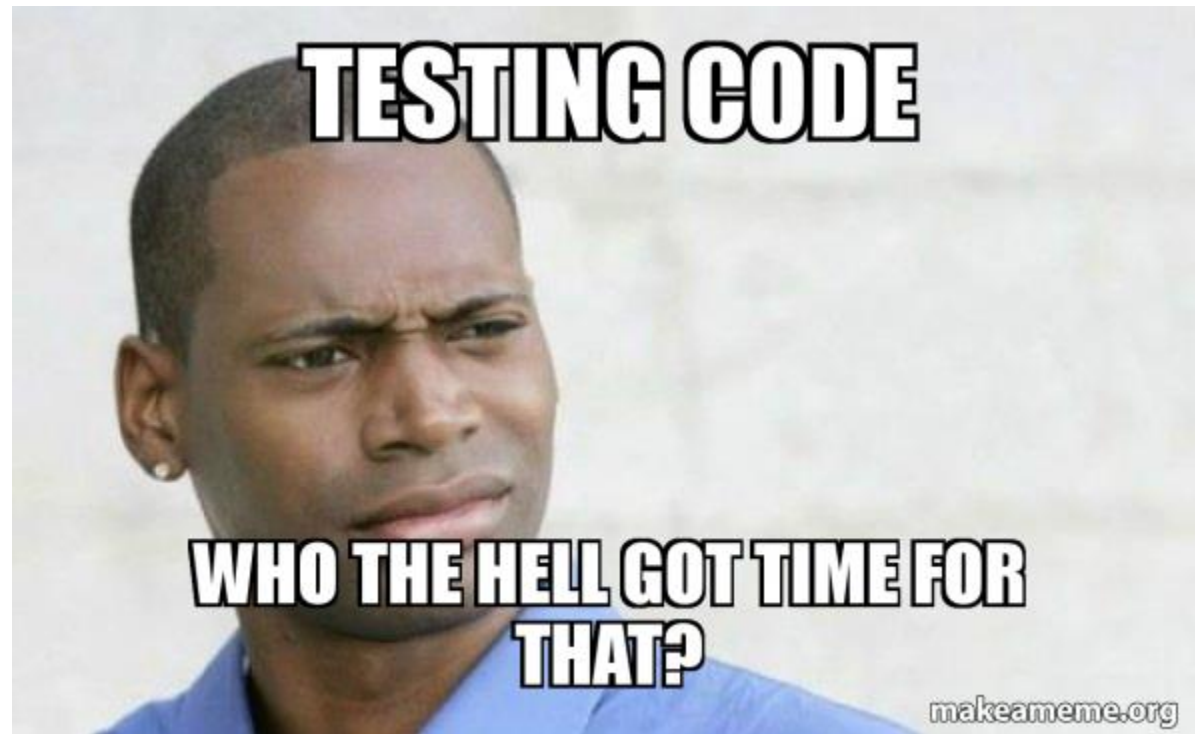
하지만 이러한 것은 지속적인 연습과 노력을 하다 보면

# 테스트 코드는 쉽다.

## ✓: 테스트 가능한 코드를 작성하세요.

- 대상 코드(클래스, 함수)는 전역 상태나 함수에 대해 **직접 의존**보단 되도록 추상 인터페이스에 **간접 의존(DI)**하세요.
  - 객체 생성에 어플리케이션 로직을 혼용하지 마세요!
  - 전역 상태/정적 함수에 의존을 피하세요.

어느 순간부터 테스트 가능하며 좋은 코드를 작성할 수 있지 않을까요?



일정에 여유가 있다면요!

# 테스트는 어떻게 더러워지는가?

---

큁큁,, 🦋



객체지향의 원칙 뿐만 아니라  
우리는 코드 냄새에 대해서도 이미 알고 있어요.

큁큁,, 🦋



테스트 코드도 역시 “코드”이기 때문에 냄새가 날 수 있습니다.

큁큁,, 🦋



테스트 냄새이나 다양한 관점에서의 도구들에 대해서는  
시간 관계상 다음 번 발표를 할 수 있다면 진행하도록 하겠습니다. ☺

# 테스트는 어떻게 더러워지는가?

---

**DRY, KISS Principles**



[principles]

**KISS  
DRY**

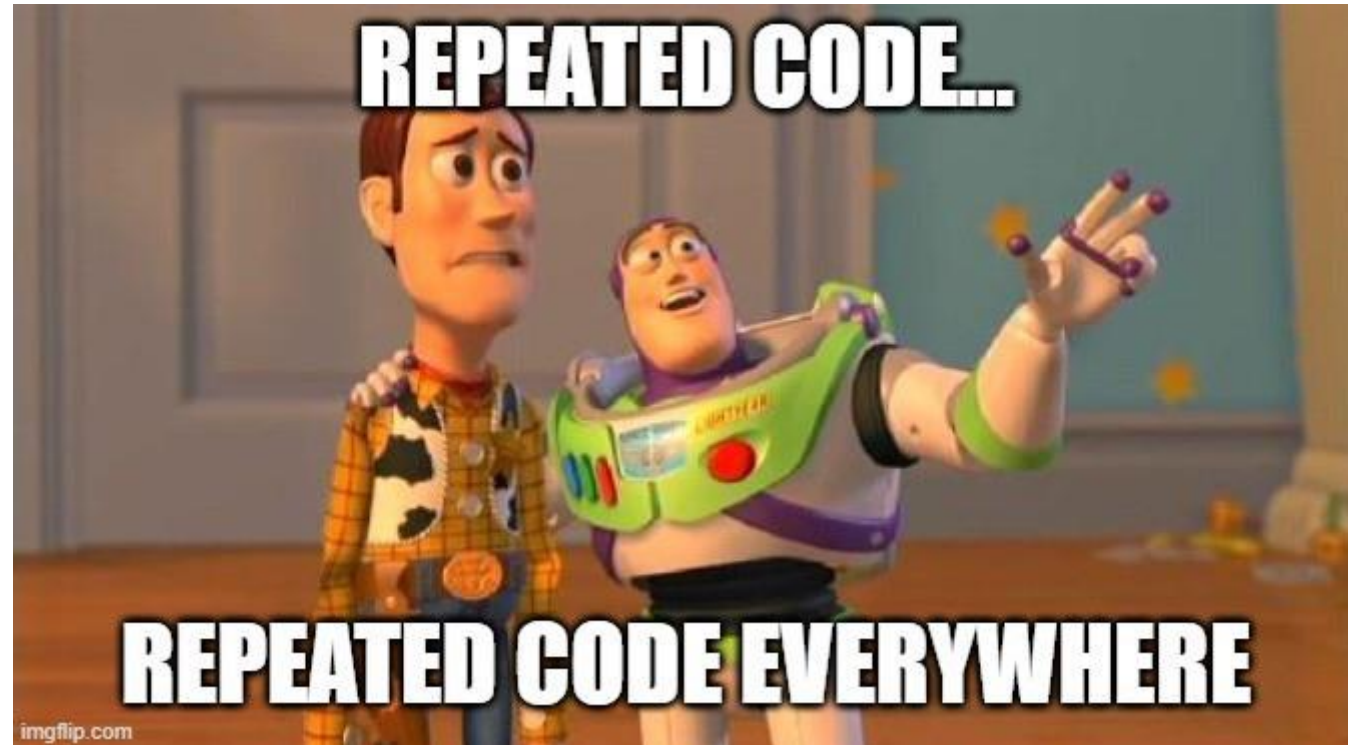
KISS와 DRY원칙에 대해서 간단히 설명하자면,

THE KISS PRINCIPLE | **KEEP  
IT  
SIMPLE,  
STUPID**

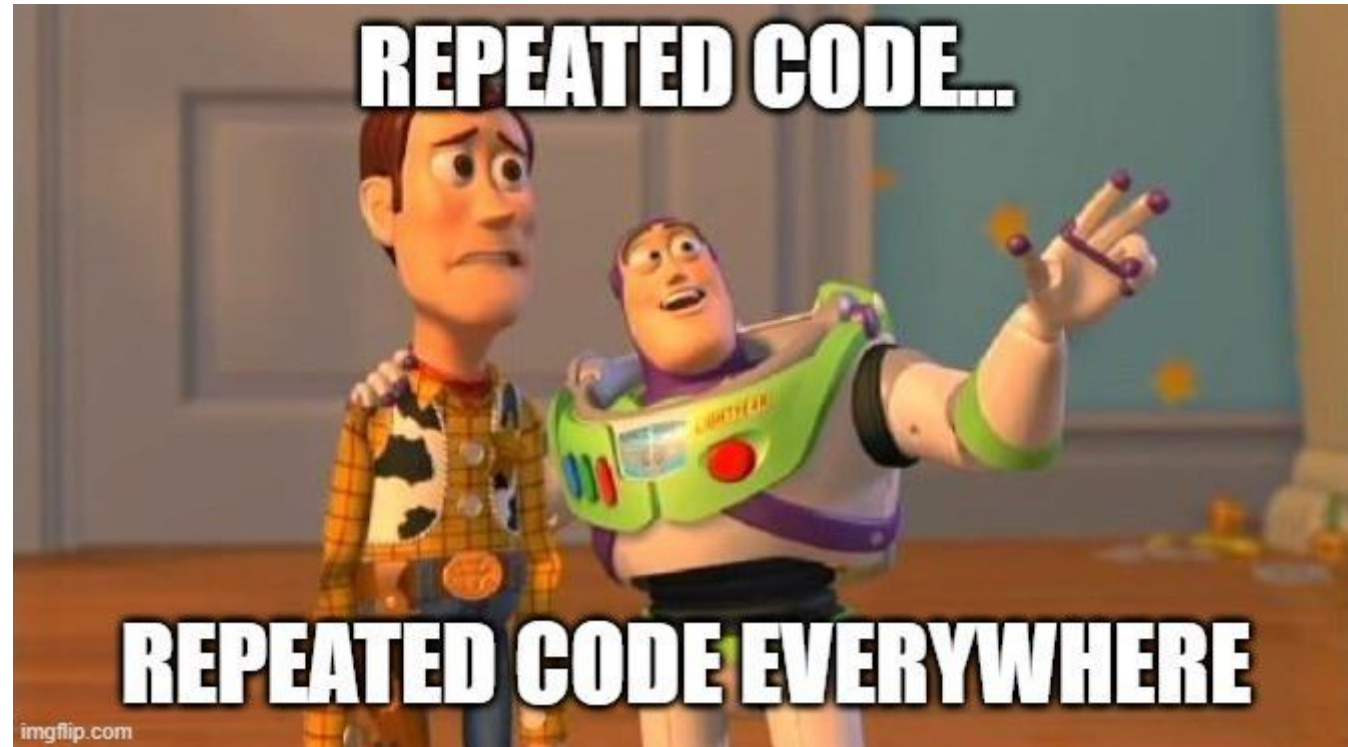
“멍청아! 단순하게 유지해!”

DO NOT REPEAT YOURSELF DO  
NOT REPEAT YOURSELF DO NO  
T REPEAT YOURSELF DO NOT R  
EPEAT YOURSELF DO NOT REPE  
AT YOURSELF DO NOT REPEAT  
YOURSELF DO NOT REPEAT YOU  
RSELF DO NOT REPEAT YOURS  
ELF DO NOT REPEAT YOURSELF  
DO NOT REPEAT YOURSELF DO  
NOT REPEAT YOURSELF DO NO  
T REPEAT YOURSELF DO NOT R  
EPEAT YOURSELF DO NOT REPE

스스로 반복하지 마라!!!!!!



하지만 현실은 😬 ...



테스트 코드를 작성하는데 이런 유혹이 많이 발생하는데요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    await repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

만약 우리가 API 테스트 코드를 작성하고 싶을 때,  
처음에 파일럿으로 짰다면 이런 코드를 작성할 수 있을 것 같아요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    await repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var resposne = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, resposne.StatusCode);
}
```

대체 무슨 코드인지 한눈에 알아볼 수 없지만,  
함수명을 보니 무슨 테스트인지는 알 수 있을 것 같네요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    await repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsNotUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

만약 이런 코드들을 복붙해서 계속해서 작성해 나간다면



```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    await repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

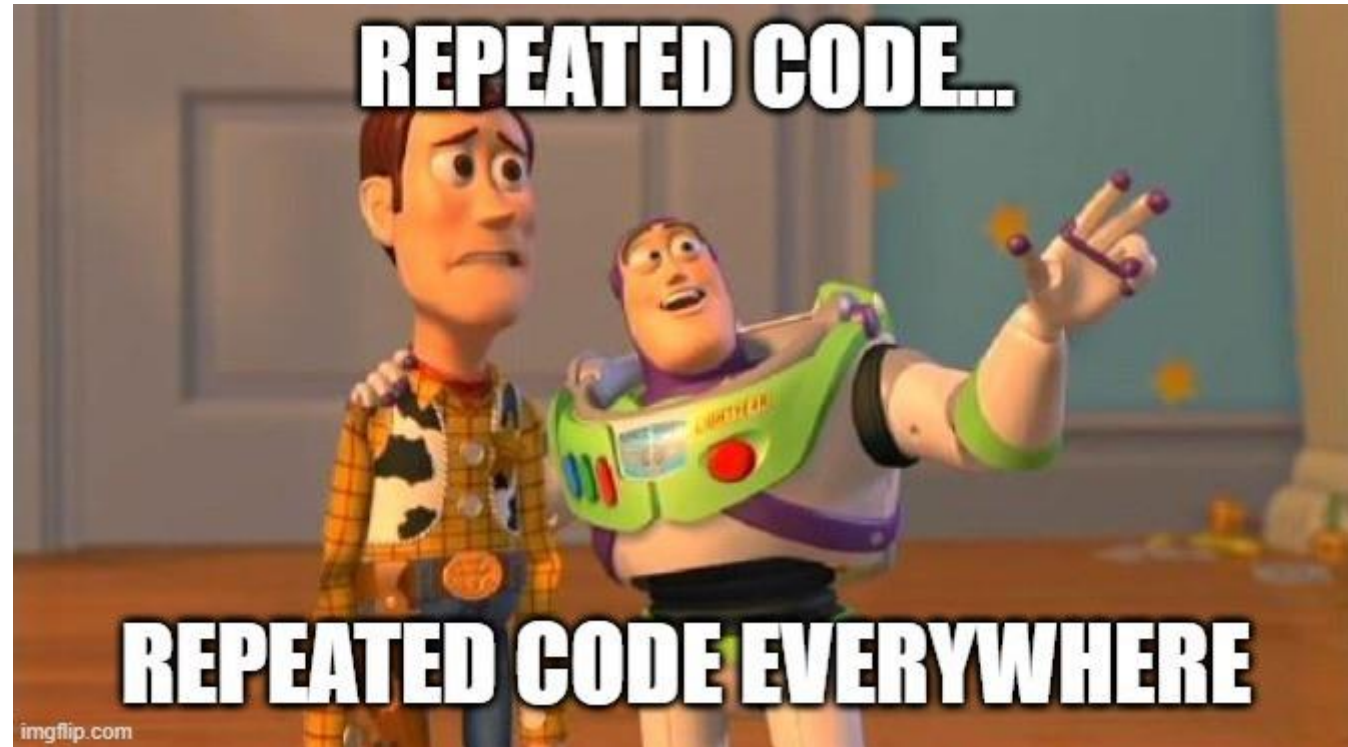
```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsNotUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

boilerplate 코드들로 테스트 코드들이 더러워질 것입니다.



하지만 이건 중복 코드를 설명하기에는 너무 뻔한 예제입니다.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    var user = new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    };
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(database);
    await repository.SaveAsync(user);

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await httpClient.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(JsonConvert.SerializeObject(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }), Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, response.StatusCode);
}
```

한번 정리를 해볼까요?

```
[SetUp]
public void Setup()
{
    _database = new Database(new InMemoryDriver());
    _userRepository = new UserRepository(_database);
    _testServer = new TestServer(new List<object>
    {
        new Replacement<IDatabase>(database)
    });
    _client = _testServer.CreateHttpClient();
}
```

우선 테스트 마다 공통적으로 생성하는 부분은 셋업 함수로 골라내고

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed( )
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var resposne = await _client.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }, Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, resposne.StatusCode);
}
```

Setup 함수에서 생성한 객체들을 사용하면 이 정도까지 줄일 수 있어요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed( )
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var resposne = await _client.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }, Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, resposne.StatusCode);
}
```

조금 더 낫죠? 이제는 좀 읽을 만한 것 같아요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed( )
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var resposne = await _client.SendAsync(new HttpRequestMessage(HttpMethod.Post, "/api/users")
    {
        Content = new StringContent(new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }, Encoding.UTF8, "application/json")
    });

    Assert.AreEqual(400, resposne.StatusCode);
}
```

더 개선할 점은 없을까요?

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

일부 반복되는 코드를 유틸 함수로 추출해서 가독성을 개선시킬 수 있어요.  
\*Extract Method(-refactoring)



```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

이를 다른 테스트 코드에도 적용하도록 합니다.

[pattern]

# Given/When/Then Arrange/Act/Assert

여기서 잠깐 GWT, A3 패턴에 대해서 설명하자면

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

Given

When

Then

테스트 코드를 작성하는데 있어  
준비/실행/검증(단언)를 분리해서 작성하는 형태를 말합니다.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

Given

When

Then

테스트 코드는 하나의 구현 코드에 여러 테스트가 생성되므로

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

Given

When

Then

이러한 테스트 코드 중 단언(Then, Assert)구문을 제외한  
상황/실행 코드에서 많은 중복이 나타날 수 있어요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe2",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

비슷하지 않나요?

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

하나의 구현코드에 보통 최소 2개 ~ 10+개까지  
테스트 코드를 작성하게 되는데

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe2",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

테스트 코드는 보통 작성된 코드를 복사하여 추가 작성하기 때문에



```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

이런 코드들은 중복이 될 가능성이 높아요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

그렇다고 해서 Given 코드를 Setup에는 넣을 수 없어요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

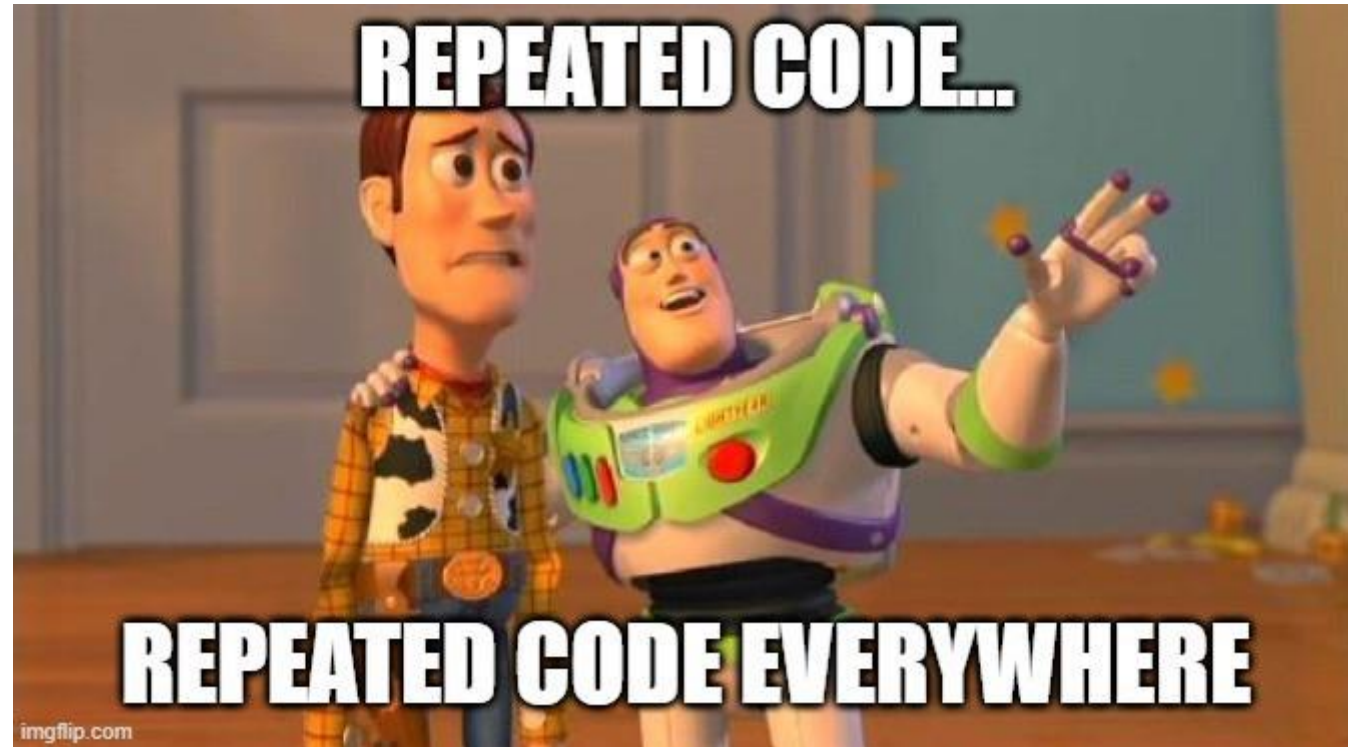
    var response = await _client.PostAsync("/api/users",
        new CreateUserRequest
        {
            Name = "pepe2",
            Age = 1,
            Gender = GenderType.Male,
            Address = Address.None()
        }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

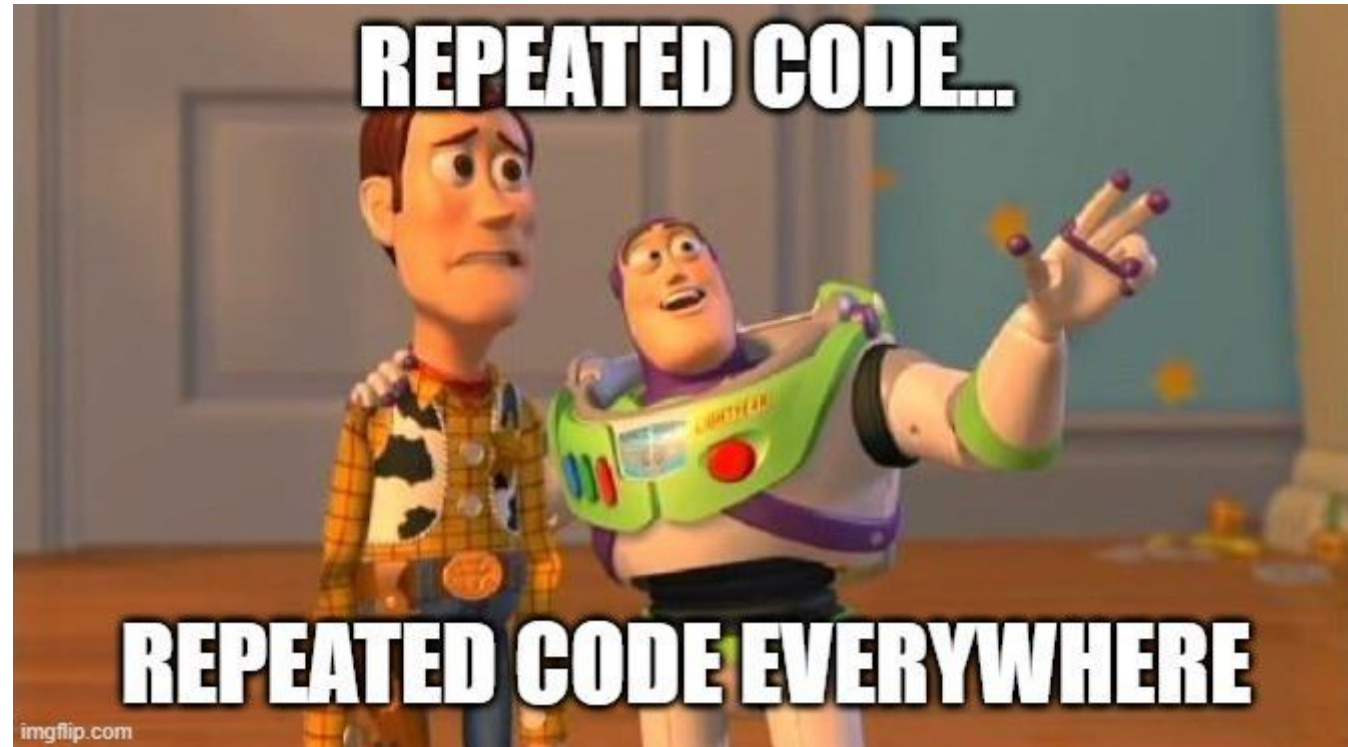
Given

When

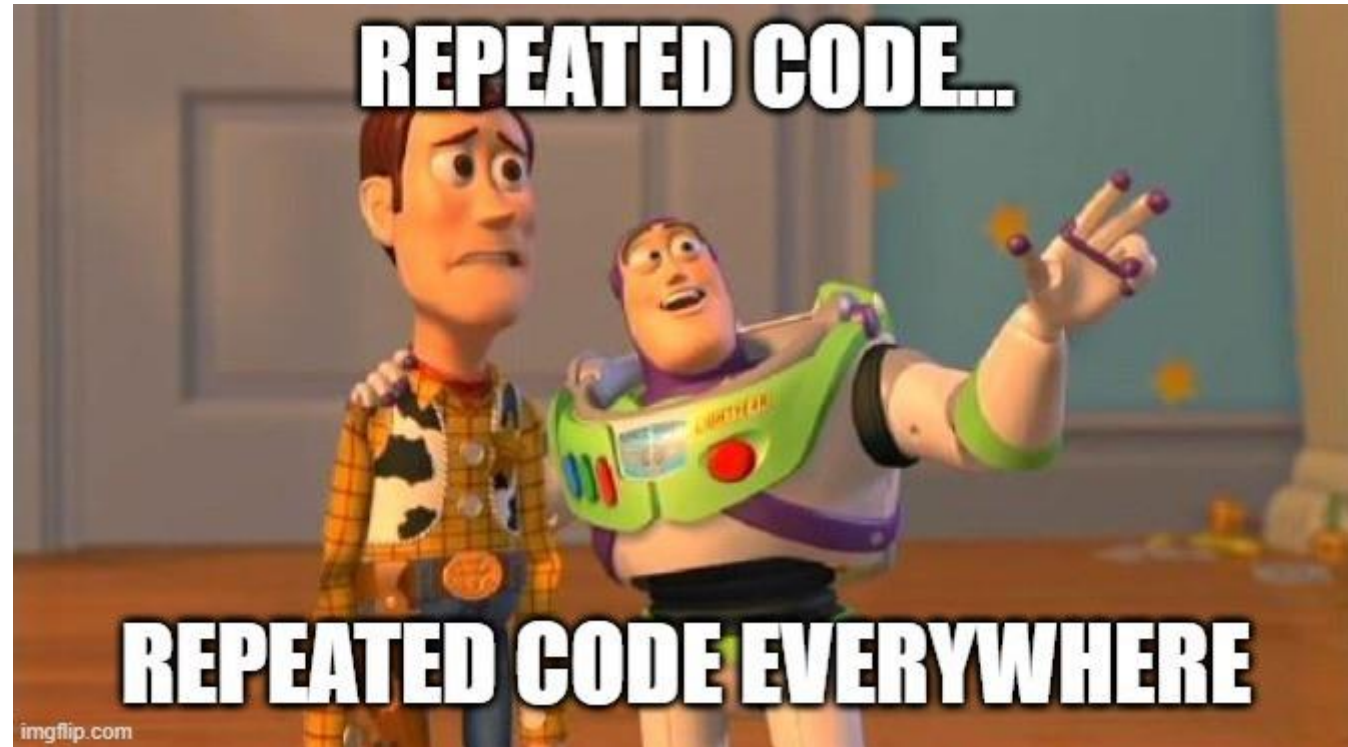
테스트 코드마다 상황/실행의 코드가 다를 수 있기 때문입니다.



약간 이러한 상황이 DRY를 위반한 상태인데요.



“테스트 코드니까, 뭐 좀 더러워도 괜찮지 않을까?”



그러기엔 이미 문제가 심각해지는 예제가 있습니다.

```
[Test]
public async Task Should_ResponseSortByName_When_RequestUsersOrderedByName()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe1",
        Age = 2022,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe2",
        Age = 2023,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe3",
        Age = 2024,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.GetAsync("/api/users",
        new GetUsersQuery
        {
            OrderBy = "name",
        });

    Assert.AreEqual(200, response.StatusCode);
    Assert.AreSame(new List<string> { "pepe", "pepe1", "pepe2", "pepe3" },
        response
            .Content
            .FromJsonAsync<GetUsersResponse>()
            .Select(u => u.Name)
            .ToList());
}
```

```
[Test]
public async Task Should_ResponseSortByAge_When_RequestUsersOrderedByAge()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe1",
        Age = 2022,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe2",
        Age = 2023,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe3",
        Age = 2024,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.GetAsync("/api/users",
        new GetUsersQuery
        {
            OrderBy = "age",
        });

    Assert.AreEqual(200, response.StatusCode);
    Assert.AreSame(new List<string> { 2021, 2022, 2023, 2024 },
        response
            .Content
            .FromJsonAsync<GetUsersResponse>()
            .Select(u => u.Age)
            .ToList());
}
```

상황이 구조(계층)적이거나 반복(iterative)한 경우  
Given 코드가 복잡해지고 어려워질 수 있습니다.

```
[Test]
public async Task Should_ResponseSortByName_When_RequestUsersOrderedByName()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe1",
        Age = 2022,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe2",
        Age = 2023,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe3",
        Age = 2024,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.GetAsync("/api/users",
        new GetUsersQuery
        {
            OrderBy = "name",
        });

    Assert.AreEqual(200, response.StatusCode);
    Assert.AreSame(new List<string> { "pepe", "pepe1", "pepe2", "pepe3" },
        response
            .Content
            .FromJsonAsync<GetUsersResponse>()
            .Select(u => u.Name)
            .ToList());
}
```

```
[Test]
public async Task Should_ResponseSortByAge_When_RequestUsersOrderedByAge()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe1",
        Age = 2022,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe2",
        Age = 2023,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe3",
        Age = 2024,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.GetAsync("/api/users",
        new GetUsersQuery
        {
            OrderBy = "age",
        });

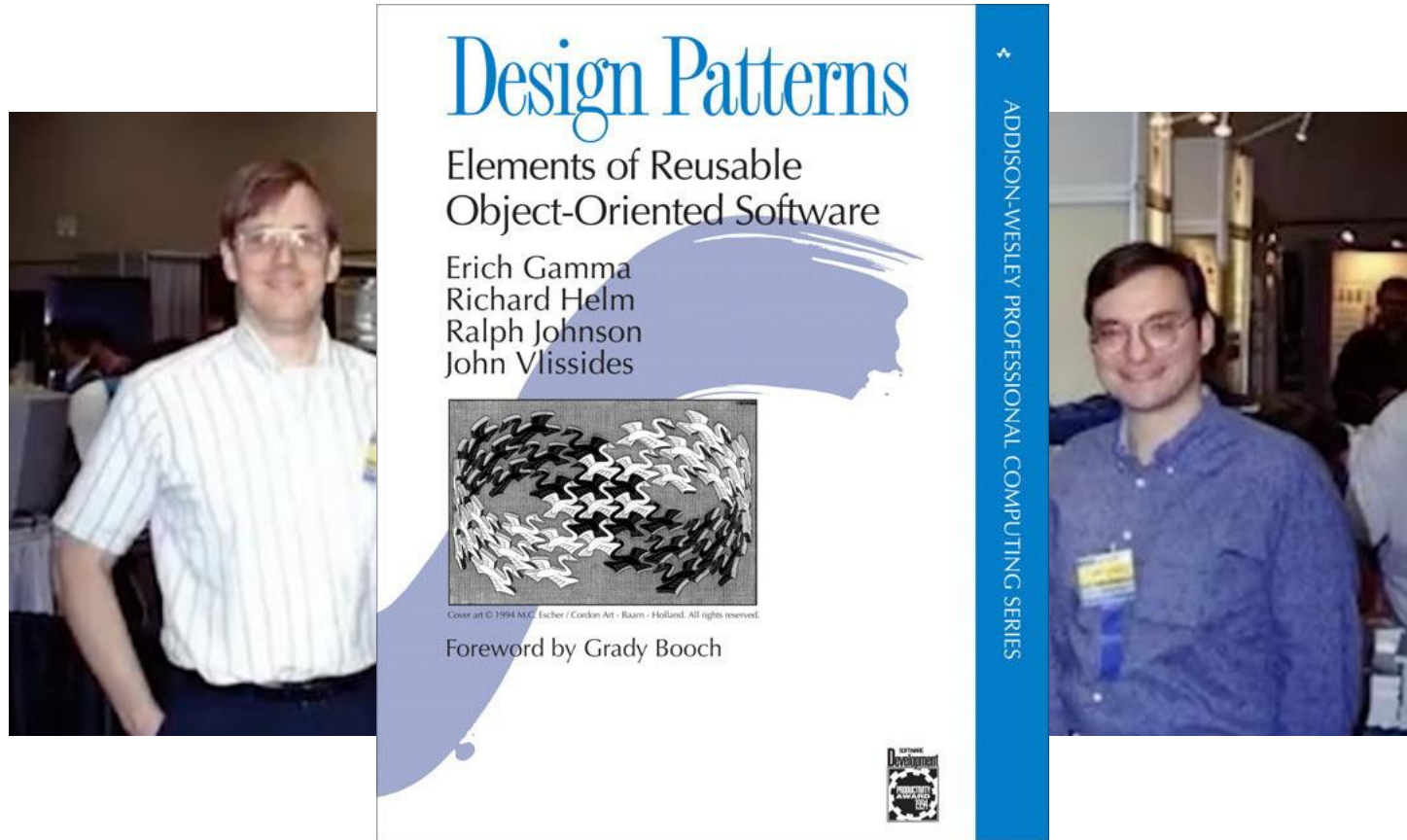
    Assert.AreEqual(200, response.StatusCode);
    Assert.AreSame(new List<string> { 2021, 2022, 2023, 2024 },
        response
            .Content
            .FromJsonAsync<GetUsersResponse>()
            .Select(u => u.Age)
            .ToList());
}
```

When 또한 마찬가지예요.





사실 이러한 문제들을 해결할 수 있는 방법은  
우리는 이미 다 알고 있습니다.



언어에 따라서 쓰지 않아도 되는 경우가 있긴 하지만,  
GoF의 생성패턴 (팩토리/프로토타입/빌더)가 이런 경우를 도와줍니다.

```
_userRepository.SaveAsync(new User
{
    Name = "pepe",
    Age = 2021,
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
});
_userRepository.SaveAsync(new User
{
    Name = "pepe1",
    Age = 2022,
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
});
_userRepository.SaveAsync(new User
{
    Name = "pepe2",
    Age = 2023,
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
});
_userRepository.SaveAsync(new User
{
    Name = "pepe3",
    Age = 2024,
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
});
```



```
var builder = new UserBuilder()
    .WithGender(GenderType.Other)
    .WithAddress(new Address("서울", "동작구"));

_userRepository.SaveAsync(builder.WithName("pepe").WithAge(2021).Build());
_userRepository.SaveAsync(builder.WithName("pepe2").WithAge(2022).Build());
_userRepository.SaveAsync(builder.WithName("pepe3").WithAge(2023).Build());
_userRepository.SaveAsync(builder.WithName("pepe4").WithAge(2024).Build());
```

전자보단 후자가 나을 것이고

```
var builder = new UserBuilder()
    .WithGender(GenderType.Other)
    .WithAddress(new Address("서울", "동작구"));

_userRepository.SaveAsync(builder.WithName("pepe").WithAge(2021).Build());
_userRepository.SaveAsync(builder.WithName("pepe2").WithAge(2022).Build());
_userRepository.SaveAsync(builder.WithName("pepe3").WithAge(2023).Build());
_userRepository.SaveAsync(builder.WithName("pepe4").WithAge(2024).Build());
```



```
var baseUser = new User
{
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
};

_userRepository.SaveAsync(baseUser with { Name = "pepe1", Age = 2021 });
_userRepository.SaveAsync(baseUser with { Name = "pepe2", Age = 2022 });
_userRepository.SaveAsync(baseUser with { Name = "pepe3", Age = 2023 });
_userRepository.SaveAsync(baseUser with { Name = "pepe4", Age = 2024 });
```

애플리케이션에 프로토타입 패턴 비슷한 것을  
지원하는 언어를 쓰는게 좋을 것 같습니다.

## Test Fixture ?

그리고 테스트하는 데 지속적으로 사용되는 환경을  
“Test Fixture”라고 칭합니다.

## Test Infrastructure ?

그리고 그런 Test Fixture나 테스트 작성에 도움이 되는 여러 구조적인 코드를 “Test Infrastructure”라고 부릅니다.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe2",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

실제 테스트 코드에서는 계속해서 중복되는 User 정보와 같은 테스트 데이터가 Fixture로 만들어서 사용할 수 있을 것 같네요.

```
[Test]
public async Task Should_ResponseBadRequest_When_ProvidedUserNameIsAlreadyUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(400, response.StatusCode);
}
```

```
[Test]
public async Task Should_ResponseCreated_When_ProvidedUserNameIsNotUsed()
{
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var response = await _client.PostAsync("/api/users",
    new CreateUserRequest
    {
        Name = "pepe2",
        Age = 1,
        Gender = GenderType.Male,
        Address = Address.None()
    }.ToJsonMessage());

    Assert.AreEqual(201, response.StatusCode);
}
```

Given

When

또한 Fixture Generator같은 클래스를 만들어 사용하면  
더더욱 직관적으로 코드가 변할 것입니다.



```
var baseUser = new User
{
    Gender = GenderType.Other,
    Address = new Address("서울", "동작구")
};
_userRepository.SaveAsync(baseUser with { Name = "pepe1", Age = 2021});
_userRepository.SaveAsync(baseUser with { Name = "pepe2", Age = 2022});
_userRepository.SaveAsync(baseUser with { Name = "pepe3", Age = 2023});
_userRepository.SaveAsync(baseUser with { Name = "pepe4", Age = 2024});
```



```
await Fixture.Of(_database)
    .AddUser(Fixture.User)
    .AddUser(Fixture.User with {Name = "pepe2", Age = 2022})
    .AddUser(Fixture.User with {Name = "pepe3", Age = 2023})
    .AddUser(Fixture.User with {Name = "pepe4", Age = 2024})
    .SetupAsync();
```

이러한 모양세이지 않을까요?

```
[Test]
public async Task Should_ResponseSortedByAge_When_RequestUsersOrderedByAge()
{
    var database = new Database(new InMemoryDriver());
    var repository = new UserRepository(new Database());
    _userRepository.SaveAsync(new User
    {
        Name = "pepe",
        Age = 2021,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe1",
        Age = 2022,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe2",
        Age = 2023,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });
    _userRepository.SaveAsync(new User
    {
        Name = "pepe3",
        Age = 2024,
        Gender = GenderType.Other,
        Address = new Address("서울", "동작구")
    });

    var testServer = new TestServer(new List<ServiceReplacement>
    {
        new Replacement<IDatabase>(database)
    });

    var httpClient = testServer.CreateHttpClient();
    var response = await _client.GetAsync("/api/users",
        new GetUsersQuery
        {
            OrderBy = "age",
        });

    Assert.AreEqual(200, response.StatusCode);
    Assert.AreSame(new List<string> {2021, 2022, 2023, 2024},
        response
            .Content
            .FromJsonAsync<GetUsersResponse>()
            .Select(u => u.Age)
            .ToList());
}
```



```
[Test]
public async Task Should_ResponseSortedByAge_When_RequestUsersOrderedByAge()
{
    await Fixture.Of(_database)
        .AddUser(Fixture.User)
        .AddUser(Fixture.User with {Name = "pepe2", Age = 2022})
        .AddUser(Fixture.User with {Name = "pepe3", Age = 2023})
        .AddUser(Fixture.User with {Name = "pepe4", Age = 2024})
        .SetupAsync();

    var response = await _client.GetAsync("/api/users", new GetUsersQuery {OrderBy = "age",});

    response.Should().BeOk();
    response.Body<GetUsersResponse>().Should().SortBy(u => u.Age);
}
```

처음과 이후를 비교한다면 이렇게 될 것입니다.

1 production method => N tests.

말씀 드렸드시피 하나의 프로덕션 메서드/함수에 대해서  
N개 만큼 테스트가 생성되는데요.

M production method => M\*N tests.

너무나도 당연하게도

10000 production lines => 50000+ test lines

저희는 정말 많은 테스트 코드를 작성해야 합니다.

If test is important,  
Test code will also be important.

테스트를 중요하게 생각한다면,  
테스트 코드 또한 중요합니다.

[principles]

**KISS  
DRY**

다양한 도구를 통해서 테스트코드를 깔끔하게 유지하세요.  
그리고 반복되는 코드를 작성하지 마세요.



긴 발표(?) 시간 내어 들어 주셔서 감사합니다.