8장. 통합 테스트를 하는 이유

8.1 통합 테스트는 무엇인가?
- 8.1.1 통합 테스트의 역할
  - 컨트롤러 사분면을 다루는 테스트가 단위 테스트일 수 있다.
- 8.1.2 다시 보는 테스트 피라미드
- 8.1.3 통합 테스트와 빠른 실패
  - 예외 상황을 다루는 지침
  - 빠른 실패 원칙

8.2 어떤 프로세스 외부 의존성을 직접 테스트해야 하는가?
- 8.2.1 프로세스 외부 의존성의 두 가지 유형
- 8.2.2 관리 의존성이면서 비관리 의존성인 프로세스 외부 의존성 다루기
- 8.2.3 통합 테스트에서 실제 데이터베이스를 사용할 수 없으면 어떻게 할까?
  - 레거시 데이터베이스 예시

8.3 통합 테스트 : 예제
- 8.3.1 어떤 시나리오를 테스트할까?
- 8.3.2 데이터베이스와 메시지 버스 분류하기
- 8.3.3 엔드 투 엔드 테스트는 어떤가?
- 8.3.4 통합 테스트 : 첫 번째 시도

8.4 의존성 추상화를 위한 인터페이스 사용
- 8.4.1 인터페이스와 느슨한 결합
- 8.4.2 프로세스 외부 의존성에 인터페이스를 사용하는 이유는 무엇인가?
- 8.4.3 프로세스 내부 의존성을 위한 인터페이스 사용

8.5 통합 테스트 모범 사례
- 통합 테스트 최대 활용을 위한 몇가지 지침
- 8.5.1 도메인 모델 경계 명시하기
- 8.5.2 계층 수 줄이기
- 8.5.3 순환 의존성 제거하기
  - 순환 의존성의 문제점
  - 순환 의존성 제거하기
  - 염두할 점
- 8.5.4 테스트에서 다중 실행 구절 사용
  - 원하는 상태로 만들기 어려운 프로세스 외부 의존성으로 작동하는 테스트가 있다면..?

8.6 로깅 기능을 테스트하는 방법
- 8.6.1 로깅을 테스트해야 하는가?
  - 식별할 수 있는 동작(지원 로깅) VS 구현 세부 사항(진단 로깅)
- 8.6.2 로깅을 어떻게 테스트해야 하는가?
  - ILogger 위에 래퍼 도입하기
  - 구조화된 로깅 이해하기
  - 지원 로깅과 진단 로깅을 위한 테스트 작성

8.6.3 로깅이 얼마나 많으면 충분한가?
8.6.5 로거 인스턴스를 어떻게 전달하는가?
해결 방안

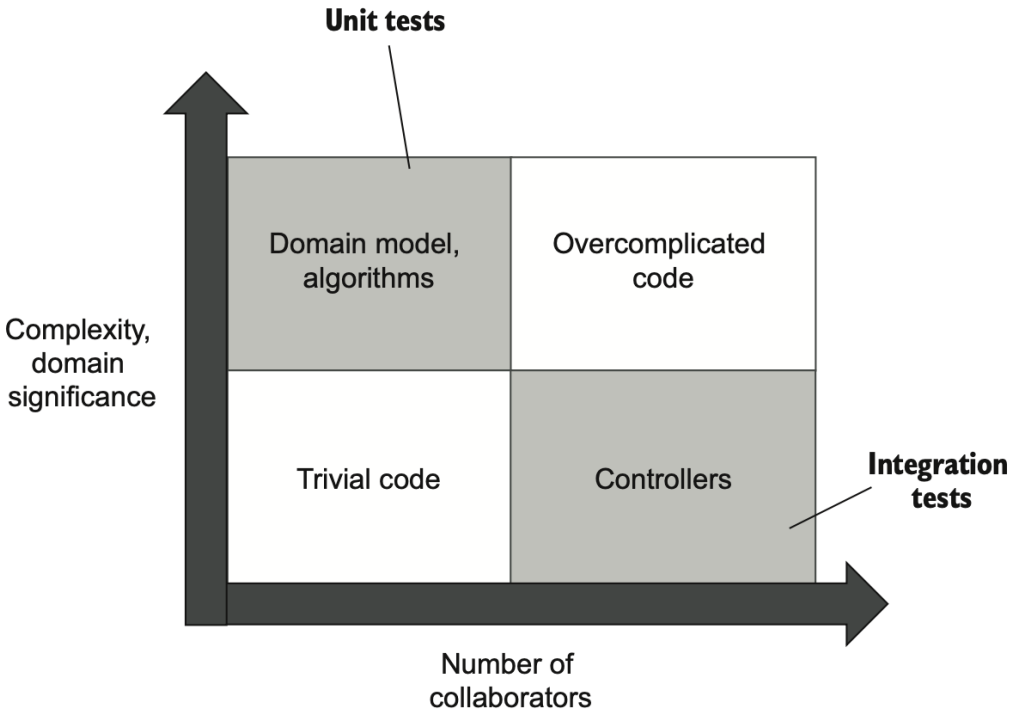
8.1 통합 테스트는 무엇인가?

8.1.1 통합 테스트의 역할

세 가지 요구 사항을 충족하는 것이 단위 테스트!

1. 단일 동작 단위 검증
2. 빠르게 수행
3. 다른 테스트와 별도로 처리

- 단위 테스트가 아닌 모든 테스트는 통합 테스트로 칭한다.
- 다시 말해 컨트롤러 사분면에 속하는 코드



- 단위 테스트는 도메인 모델을 다루고, 통합 테스트는 프로세스 외부 의존성과 도메인 모델을 연결하는 코드를 확인한다.

컨트롤러 사분면을 다루는 테스트가 단위 테스트일 수 있다.

- 모든 프로세스 외부 의존성을 목으로 처리하면 테스트가 공유하는 의존이 없어지므로 테스트간 별도 처리가 가능하다.
- 그러나 목으로 처리 불가능한 프로세스 외부 의존성이 있다.(데이터베이스)

`→ 모든 테스트는 도메인 모델과 컨트롤러 사분면에 초점 맞춰 작성한다.`

8.1.2 다시 보는 테스트 피라미드

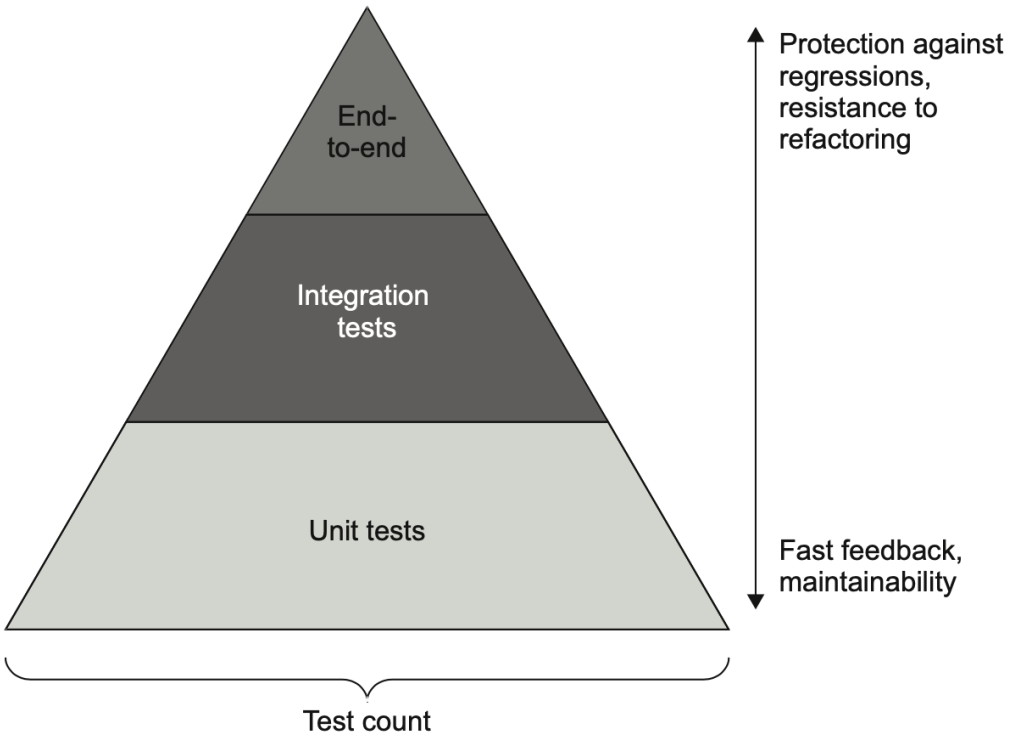
- 단위 테스트와 통합 테스트 간의 균형이 중요하다.
- 통합 테스트는 프로세스 외부 의존성에 직접 작동하기 때문에 느리고 유지 비용이 많이 들기 때문
- 유지 비용이 증가하는 이유
  1. 프로세스 외부 의존성 운영이 필요
  2. 관련 협력자가 많아 테스트가 비대해짐
- 통합 테스트는 회귀 방지가 단위 테스트보다 우수하다.
  - 애플리케이션코드와 애플리케이션에서 사용하는 라이브러리의 코드를 모두 포함하기 때문
- 제품 코드와의 결합도가 낮아서 리팩터링 내성도 우수하다.

- 단위 테스트로 가능한 많은 비즈니스 시나리오의 예외 상황을 확인
- 통합 테스트로 주요 흐름과 단위 테스트가 다루지 못하는 기타 예외 상황을 다룬다.

주요 흐름과 예외 상황의 정의

- 주요 흐름은 시나리오의 성공적인 실행
- 예외 상황은 비즈니스 시나리오 수행 중 오류가 발생하는 경우

- 대부분을 단위 테스트로 전환하면 유지비를 절감할 수 있다.
- 중요한 통합 테스트가 비즈니스 시나리오당 하나 또는 두 개 있으면 시스템 전체의 정확도를 보장할 수 있다. (1 비즈니스 시나리오 대 1~2 통합 테스트)



단순 애플리케이션

- 테스트 구성이 직사각형 모양을 띈다.

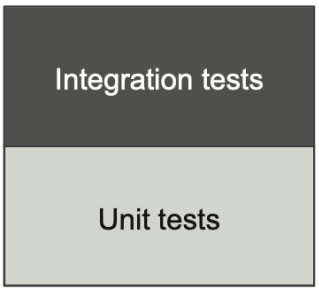


Figure 8.3 The Test Pyramid of a simple project. Little complexity requires a smaller number of unit tests compared to a normal pyramid.

- 단위 테스트와 통합 테스트 수가 같다는 의미이다. (아주 단순한 경우라면 단위 테스트가 없을 것이다.)

- 통합 테스트는 단순한 애플리케이션에서도 가치가 있다.
    - 코드 간단성 보다는 다른 서브 시스템과 통합해 어떻게 작동하는지 확인하는 것이 더 중요하다.

8.1.3 통합 테스트와 빠른 실패

예외 상황을 다루는 지침

- 가장 긴 주요 흐름을 선택하라
    - 모든 상호 작용을 거치는 흐름이 없으면, 외부 시스템과의 통신을 모두 확인하는 데 필요한 만큼 통합 테스트를 추가로 작성하라.
- 어떤 예외 상황에 잘못 실행돼 전체 애플리케이션이 즉시 실패하면 해당 예외 상황은 테스트할 필요가 없다.

- CanChangeEmail 메서드 구현

```
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);
    /* the rest of the method */
}
```

- 컨트롤러가 CanChangeEmail() 을 호출하고 해당 메서드가 오류를 반환하면 연산을 중단한다.

```
// UserController
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);
    string error = user.CanChangeEmail();
    if (error != null)
        return error;
    /* the rest of the method */
}
```

- 위의 예제는 충분한 가치를 가져다주지 못한다.

    - 좋지 않은 테스트라면 작성하지 않는 것이 좋다.
    - User 에서 사전 조건이 있는지 테스트를 해야한다.
- 버그를 빨리 나타나게 하는 것은 `빠른 실패 원칙(Fast Fail Principle)` 이라고 하며, 통합 테스트에서 사용할 수 있는 대안이다.

빠른 실패 원칙

- 예기치 않은 오류가 발생하자마자 현재 연산을 중단하는 것을 의미
- 다음과 같은 이유로 애플리케이션의 안정성을 높일 수 있다.
    1. 피드백 루프 단축

       : 버그를 빨리 발견하여 문제를 더 쉽게 해결한다. 운영 환경으로 넘어간 버그는 수정 비용이 훨씬 많이 든다.
    2. 지속성 상태 보호

       : 버그는 애플리케이션 상태를 손상시킨다. 빨리 실패를 발견하여 손상이 확산되는 것을 막는다.

8.2 어떤 프로세스 외부 의존성을 직접 테스트해야 하는가?

8.2.1 프로세스 외부 의존성의 두 가지 유형

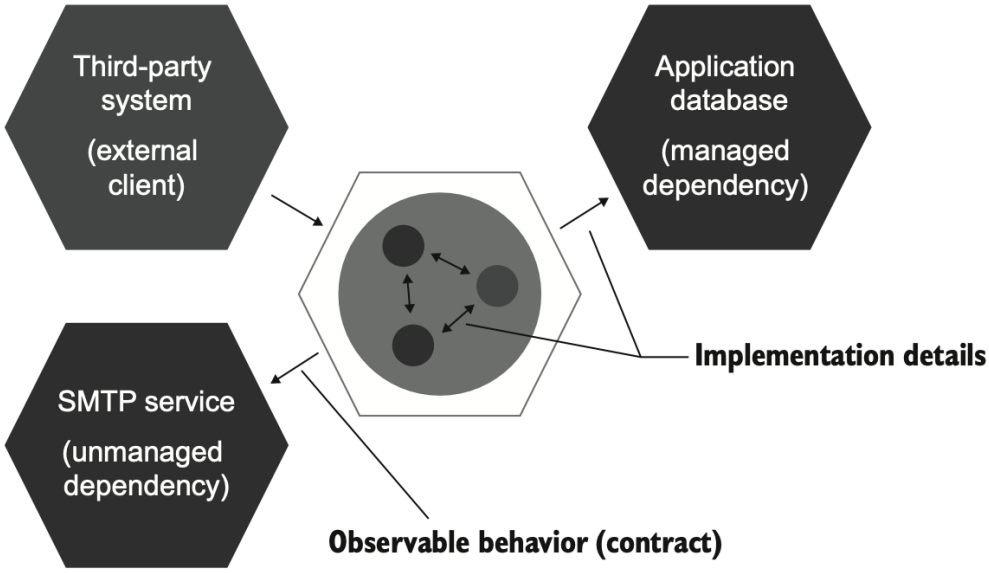
1. 관리 의존성 : 전체를 제어할 수 `있는` 프로세스 외부 의존성
    - 예로는 데이터베이스가 있다.
    - 해당 의존성과의 상호 작용은 외부 환경에서 볼 수 없다.
    - 외부 시스템이 데이터베이스에 접근할 때 보통 애플리케이션에 제공하는 API 를 통해 접근한다.
2. 비관리 의존성 : 전체를 제어할 수 `없는` 프로세스 외부 의존성
    - 예로는 SMTP 서버와 메시지 버스 등이 있다.
    - 해당 의존성과의 상호 작용을 외부 환경에서 볼 수 있다.

    - 둘 다 다른 애플리케이션에서 볼 수 있는 사이드 이펙트를 발생시킨다.

- 관리 의존성과의 통신은 구현 세부 사항이다.
- 비관리 의존성과의 통신은 시스템의 식별할 수 있는 동작이다.

  → **이러한 차이로 통합 테스트에서 프로세스 외부 의존성의 처리가 달라진다.**

  → **관리 의존성은 실제 인스턴스를 사용하고, 비관리 의존성은 목으로 대체한다.**
- 비관리 의존성에 대한 통신 패턴을 유지해야 하는 것은 하위 호환성을 지켜야 하기 때문이다. (?)
    - 이 때 목이 제격이다.
    - 목을 사용함으로써 모든 가능한 리팩터링을 고려해서 통신 패턴 영속성을 보장할 수 있다.



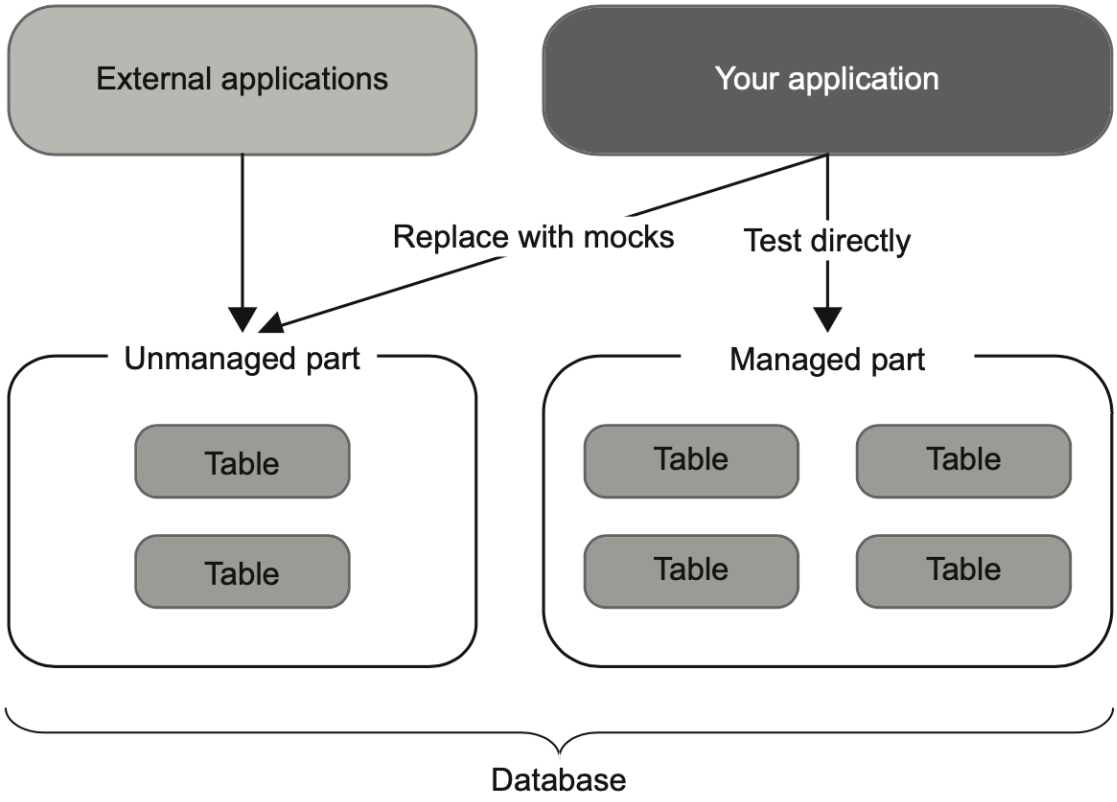
- 관리 의존성과의 통신하는 대상은 애플리케이션뿐이기 때문에 하위 호환성을 유지할 필요가 없다.
    - 시스템의 최종 상태가 중요하다.
    - 통합 테스트에서 관리 의존성에 실제 인스턴스를 사용하면 외부 클라이언트 관점에서 최종 상태를 확인할 수 있다.
    - 데이터베이스 리팩터링에도 도움(컬럼 변경 사항, 데이터베이스 이관 등)

8.2.2 관리 의존성이면서 비관리 의존성인 프로세스 외부 의존성 다루기

- 관리 의존성과 비관리 의존성 모두의 속성을 가진 프로세스 외부 의존성이 있을 수 있다.
- 예로는 다른 애플리케이션이 접근할 수 있는 데이터베이스가 있다.
    - 일부 테이블만 접근 권한을 공유하는 방식으로 해결
    - 이는 시스템이 서로 결합되고 추가 개발을 복잡하게 만들 수 있어 좋지 않다.
    - 다른 방법이 없는 경우에만 이 방식을 사용하라.
    - API(동기적 통신) 또는 메시지 버스(비동기적 통신)을 사용하는 것이 낫다.

이미 공유 데이터베이스가 있다면?

- 다른 애플리케이션에서 볼 수 있는 테이블을 비관리 의존성으로 취급해라.
    - 이런 테이블을 사실상 메시지 버스 역할을 하고 각 행이 메시지 역할을 한다.
    - 나머지 데이터베이스를 관리 의존성으로 처리하고 최종 상태를 확인해라.



데이터베이스에서 이 두 부분을 구분하는 것이 중요하다.

- 공유 테이블은 외부에서 볼 수 있기 때문에 애플리케이션과 테이블간 통신 방식에 주의를 해야한다.
- **꼭 필요한 경우가 아니라면 시스템과 해당 테이블의 상호 작용 방식을 변경하지 마라!**

8.2.3 통합 테스트에서 실제 데이터베이스를 사용할 수 없으면 어떻게 할까?

레거시 데이터베이스 예시

- 목으로 대체하는 방식?
  - 리팩터링 내성이 저하되고 회귀 방지도 떨어진다.
  - 이 데이터베이스가 유일한 프로세스 외부 의존성이라면 통합 테스트는 기존 단위 테스트 세트와 다를 바가 없기 때문이다.

- 컨트롤러가 어떤 리포지터리 메서드를 호출하는지 검증하는 것뿐이다.

→ 데이터베이스를 그대로 테스트할 수 없으면 통합 테스트를 아예 작성하지 말고 도메인 모델의 단위 테스트에만 집중해라.

8.3 통합 테스트 : 예제

- 시나리오 : 데이터베이스에서 사용자와 회사를 검색 → 의사 결정을 도메인 모델에 위임 → 결과를 데이터베이스에 반영 → 필요한 경우 메시지 버스에 메시지를 보냄

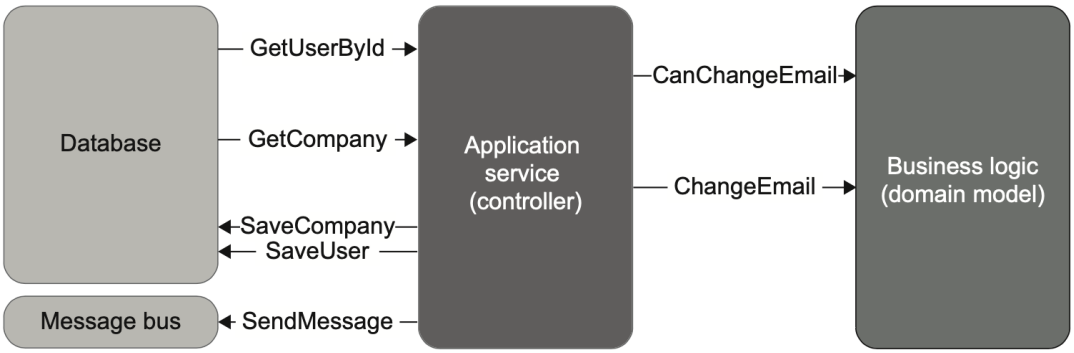


Figure 8.6 The use case of changing the user's email. The controller orchestrates the work between the database, the message bus, and the domain model.

- 현재 사용자 컨트롤러 코드 예시

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();
    public string ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);
        string error = user.CanChangeEmail();
        if (error != null)
            return error;
        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);
        user.ChangeEmail(newEmail, company);
        _database.SaveCompany(company);
        _database.SaveUser(user);
        foreach (EmailChangedEvent ev in user.EmailChangedEvents)
        {
            _messageBus.SendEmailChangedMessage(ev.UserId, ev.NewEmail);
        }
        return "OK";
```

```
    }
}
```

8.3.1 어떤 시나리오를 테스트할까?

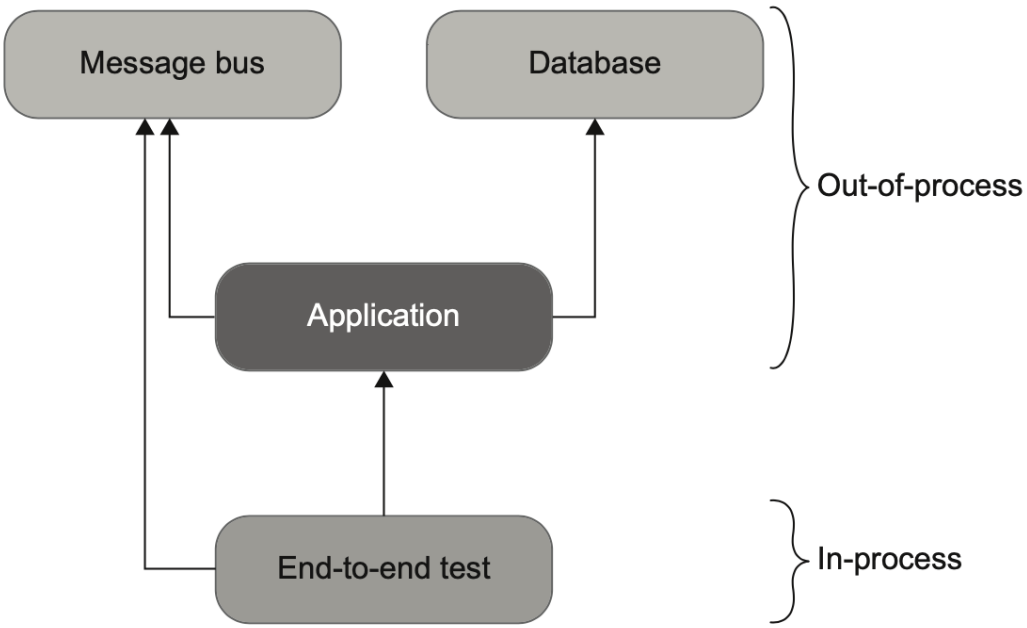
- 일반적 지침은 `가장 긴 주요 흐름`과 (단위 테스트로는 수행할 수 없는) `모든 예외 상황을 다루는 것`이다.
- 여기서 가장 긴 주요 흐름은 기업 이메일에서 일반 이메일로 변경하는 것이다.
  - 데이터베이스에서 사용자와 회사 모두 업데이트
    - 사용자는 유형과 이메일을 변경
    - 회사는 직원 수를 변경
    - 이메일을 변경할 수 없는 예외 시나리오가 있지만 빨리 실패하기 때문에 테스트할 필요가 없다.

8.3.2 데이터베이스와 메시지 버스 분류하기

- 직접 테스트할 대상과 목으로 대체할 대상을 결정해야한다.
- 데이트베이스는 어떤 시스템도 접근할 수 없기 때문에 관리 의존성이다.
  - 실제 인스턴스 사용
  - 데이터베이스에 사용자와 회사를 삽입한다.
  - 해당 데이터베이스에서 이메일 변경 시나리오를 실행한다.
  - 데이터베이스 상태를 검증한다.
- 메시지 버스는 비관리 의존성이다.
  - 메시지 버스의 목적은 다른 시스템과의 통신을 가능하게 하는 것뿐이다.
  - 메시지 버스를 목으로 대체
  - 컨트롤러와 목 간의 상호 작용을 검증한다.

8.3.3 엔드 투 엔드 테스트는 어떤가?

- 어떤 프로세스 외부 의존성도 목으로 대체하지 않는다.
- 사용 여부는 판단에 따라 결정한다.
- 대부분 통합 테스트 보호 수준(비관리 의존성만 목으로 대체)으로 엔드 투 엔드 테스트를 생략할 수 있다.
- 배포 후 상태 점검을 위해 한 개 또는 두 개 정도 엔드 투 엔드 테스트를 작성할 수 있다.
    - 외부 클라이언트의 동작을 모방하기 위해 메시지 버스는 직접 확인하고, 데이터베이스 상태는 애플리케이션을 통해 검증한다.



8.3.4 통합 테스트 : 첫 번째 시도

- 코드

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate(){

    // 준비
    var db = new Database(ConnectionString);
    User user = CreateUser(
        "user@mycorp.com", UserType.Employee, db);
```

```
    CreateCompany("mycorp.com", 1, db);

    var messageBusMock = new Mock<IMessageBus>();
    var sut = new UserController(db, messageBusMock.Object);

    // 실행
    string result = sut.ChangeEmail(user.UserId, "new@gmail.com");

    // 검증
    Assert.Equal("OK", result);

    object[] userData = db.GetUserById(user.UserId);
    User userFromDb = UserFactory.Create(userData);
    Assert.Equal("new@gmail.com", userFromDb.Email);
    Assert.Equal(UserType.Customer, userFromDb.Type);

    object[] companyData = db.GetCompany();
    Company companyFromDb = CompanyFactory
        .Create(companyData);
    Assert.Equal(0, companyFromDb.NumberOfEmployees);

    messageBusMock.Verify(
      x => x.SendEmailChangedMessage(
          user.UserId, "new@gmail.com"),
      Times.Once);

}
```

- 사용자와 회사를 데이터베이스에 삽입하지 않는다.
- 재사용 가능한 CreateUser 와 CreateCompany 헬퍼 메서드를 호출하고 있다.
- 검증 구절에서 사용자와 회사 데이터를 각각 조회한다.
- userFromDb와 companyFromDb 인스턴스를 생성해서 상태를 검증한다.
- 읽기와 쓰기 모두 수행하여 회귀 방지를 최대로 얻을 수 있다.
- 읽기는 내부적으로 구현된 동일한 코드를 사용해서 구현해야 한다.
- messageBusMock 은 회귀 방지가 그다지 좋지 않다.

8.4 의존성 추상화를 위한 인터페이스 사용

8.4.1 인터페이스와 느슨한 결합

- 일반적으로 인터페이스를 사용하는 이유

1. 프로세스 외부 의존성 추상화로 느슨한 결합을 달성한다.
2. 기존 코드 변경 없이 새로운 기능을 추가한다. : 공개 폐쇄 원칙(OCP)을 지키기 위함

- 단일 구현을 위한 인터페이스는 추상화가 아니다.
  - 구체 클래스보다 결합도가 낮지 않다.
  - 인터페이스가 제대로 추상화되려면 적어도 두 가지는 구현해야 한다.
- YAGNI 원칙을 위반하기 때문이다.
  - 현재 필요하지 않는 기능에 시간을 들이지 말것!
  - 기회 비용
    - 현재 필요하지 않는 기능에 시간을 보낸다면 지금 당장 피룡한 기능을 제치고 시간을 허비하는 것
    - 처음부터 실제 필요에 따라 기능을 구현하는 것이 더 유리하다.
  - 프로젝트 코드가 적을 수록 좋음
    - 코드베이스의 소유 비용이 불필요하게 증가한다.

`→ 코드 작성은 문제를 해결하는 값비싼 방법이다. 해결책에 필요한 코드가 적고 간단할 수록 더 좋다.`

8.4.2 프로세스 외부 의존성에 인터페이스를 사용하는 이유는 무엇인가?

- 간단히 목을 사용하기 위함이다.
  - 인터페이스 없이 테스트 대역을 만들 수 없다.
  - 목으로 처리할 필요가 없는 한, 프로세스 외부 의존성에 대한 인터페이스를 둘 필요가 없다.
  - 결국 비관리 의존성에만 인터페이스를 쓰라는 것이다.
- 진정한 추상화(구현이 둘 이상인 케이스) 는 목과 상관없이 인터페이스를 둘 수 있다.
- 적용된 컨트롤러 코드 예시

```
public class UserController
{
    private readonly Database _database; // 관리 의존성
```

```
      private readonly IMessageBus _messageBus;  // 비관리 의존성

    public UserController(Database database, IMessageBus messageBus)
    {
        _database = database;
        _messageBus = messageBus;
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        /* the method uses _database and _messageBus */
    }
}
```

8.4.3 프로세스 내부 의존성을 위한 인터페이스 사용

- 프로세스 내부 의존성도 인터페이스 기반인 코드를 볼 수 있다.
- User 인터페이스 코드 예시

```
public interface IUser
{
    int UserId { get; set; }
    string Email { get; }
    string CanChangeEmail();
    void ChangeEmail(string newEmail, Company company);
}


public class User : IUser
{
    /* ... */
}
```

- 프로세스 외부 의존성 인터페이스와 마찬가지로 구현이 하나만 있다면 좋지 않는 신호다.
- 도메인 클래스에 대해 단일 구현으로 인터페이스를 도입하는 이유는 목으로 처리하기 위한 것 뿐이다.
- 그러나, 다음과 같은 이유로 프로세스 외부 의존성과 달리 도메인 클래스 간의 상호 작용을 확인해서는 안된다.
    - 깨지기 쉬운 테스트로 이어진다.

- 리팩터링 내성이 떨어진다.

8.5 통합 테스트 모범 사례

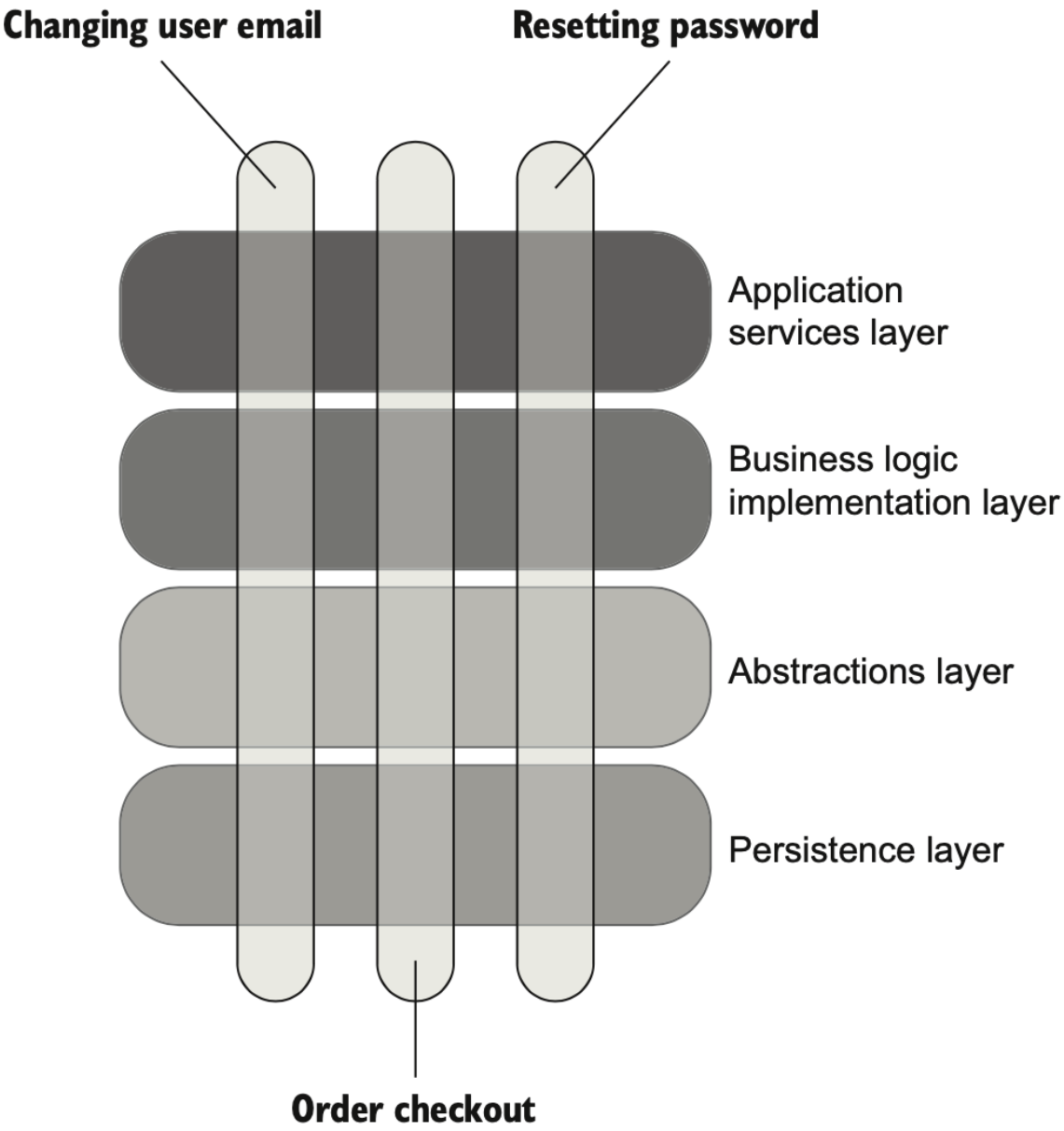
통합 테스트 최대 활용을 위한 몇가지 지침

1. 도메인 모델 경계 명시하기
2. 애플리케이션 내 계층 줄이기
3. 순환 의존성 제거하기

8.5.1 도메인 모델 경계 명시하기

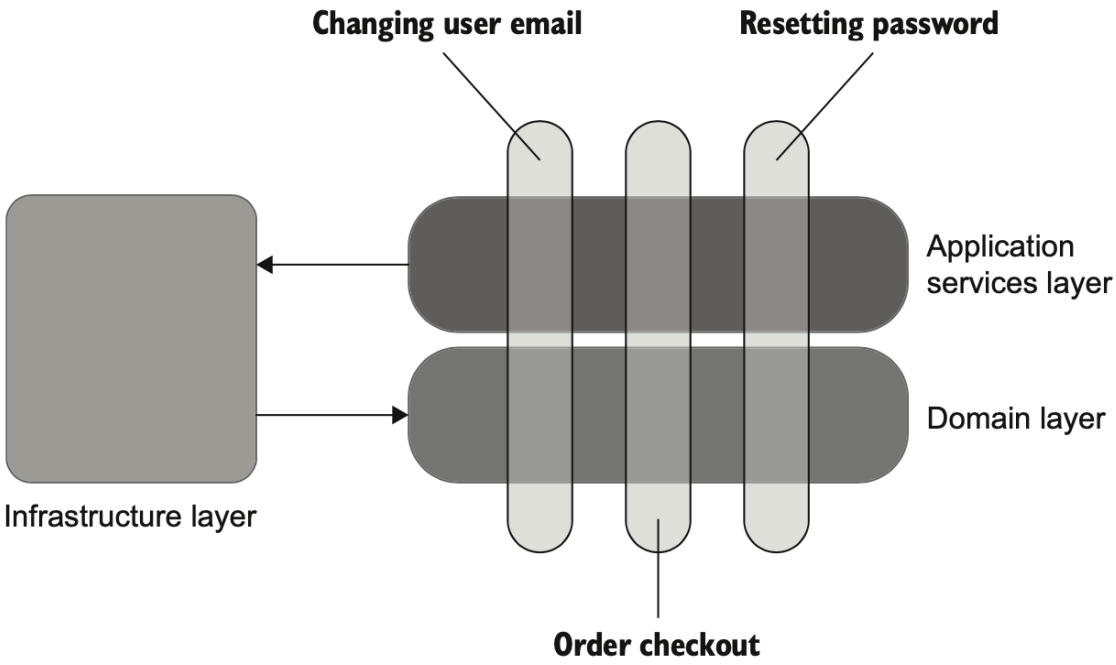
- 도메인 모델을 명시적이고 잘 알려진 위치에 두어라.
- 코드의 해당 부분을 더 잘 보여주고 잘 설명할 수 있다.
- 단위 테스트는 도메인 모델과 알고리즘이 대상이기 때문에 통합 테스트와의 차이점을 쉽게 구분할 수 있다.
- 경계는 별도 어셈블리 또는 네임스페이스 형태를 취할 수 있다.
- (어셈블리는 .NET 애플리케이션의 구성 요소로 참조 라이브러리에 해당, 네임스페이스는 다른 언어의 패키지에 상응하는 개념)

8.5.2 계층 수 줄이기



- 추상 계층(Abstractions layer)이 너무 많으면 코드 베이스 탐색이 어렵고 숨은 로직을 이해하기 어려워진다.
- 간접 계층은 코드를 추론하는 데 부정적인 영향을 준다.
- 간접 계층이 많은 코드베이스는 컨트롤러와 도메인 모델 사이에 경계가 없는 편이다.
- 각 계층을 따로 검증하는 경향이 강해서 통합 테스트 가치가 떨어지고, 각 테스트는 특정 계층 코드만 실행하고 하위 계층을 목으로 처리한다.
- 최종 결과가 항상 똑같이 낮은 리팩터링 내성과 불충분한 회귀 방지를 나타낸다.

- 가능한 한 간접 계층을 적게 사용하라.
- 대부분 백엔드에서는 `도메인 모델`, `애플리케이션 서비스 계층`, `인프라 계층` 이 세가지만 활용하면 된다.



8.5.3 순환 의존성 제거하기

- 순환 의존성 예시 : 콜백

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        service.GenerateReport(orderId, this);
        /* other code */
    }
}


public class ReportGenerationService
{
    public void GenerateReport(
        int orderId,
        CheckOutService checkOutService)
    {
        /* calls checkOutService when generation is completed */
```

```
    }
}
```

순환 의존성의 문제점

- 코드를 읽을 때 알아야할 것이 많아진다.
- 해결책을 위한 출발점이 명확하지 않다.
- 하나의 클래스 주변 클래스 그래프 전체를 봐야한다.
- 소규모 독립 클래스조차도 파악하기 어려울 수 있다.
- 테스트를 방해한다.
    - 클래스 그래프를 나눠서 동작 단위 하나를 분리하려면 인터페이스에 의존해 목으로 처리해야 하는 경우가 늘어난다.
    - 이는 도메인 모델 테스트의 경우 해서는 안된다.
    - ReportGenerationService이 ICheckOutService를 의존하게 해도 런타임에는 순환이 여전히 존재한다.
        - 코드를 이해하는 데 알아야하는 부담이 줄지 않고 인터페이스 추가로 인해 오히려 부담이 늘어난다.

순환 의존성 제거하기

- ReportGenerationService를 리팩터링하여 ICheckOutService 를 의존하는 것이 아니라 CheckOutService를 호출하지 않고 작업 결과를 따로 반환하게 수정한다.

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        Report report = service.GenerateReport(orderId);
        /* other work */
    }
}

public class ReportGenerationService
{
    public Report GenerateReport(int orderId)  // 작업 결과를 리턴
    {
        /* ... */
```

```
    }
}
```

염두할 점

- 모든 순환 의존성을 제거하는 것은 거의 불가능하기 때문에 가능한 한 의존적인 클래스의 그래프를 작게 만들면 손상을 최소화할 수 있다.

8.5.4 테스트에서 다중 실행 구절 사용

- 테스트에서 두 개 이상의 준비나 실행 또는 검증 구절을 두는 것은 코드 악취(code smell)에 해당한다.
- 유지 보수성을 떨어드리는 신호다.
- 흐름이 자연스럽기 때문에 도메인의 상태를 확인할 수 있어 설득력이 있지만, 이런 테스트는 초점을 읽고 순식간에 너무 커질 수 있다.

원하는 상태로 만들기 어려운 프로세스 외부 의존성으로 작동하는 테스트가 있다면..?

- 여러 동작을 하나의 테스트로 묶어서 문제가 있는 프로세스 외부 의존성에 대한 상호 작용 횟수를 줄이는 것이 유리하다.
- 이 경우가 아니라면 절대로 실행 구절이 여러 개 있어서는 안된다.
- 다단계 테스트는 거의 엔드 투 엔드 테스트 범주에 속한다.

8.6 로깅 기능을 테스트하는 방법

8.6.1 로깅을 테스트해야 하는가?

- 로깅은 횡단 기능이다.
- 로깅이 애플리케이션의 식별할 수 있는 동작인가, 아니면 구현 세부 사항인가? 질문해 보자.

식별할 수 있는 동작(지원 로깅) VS 구현 세부 사항(진단 로깅)

- 개발자 이외 사람이 보는 경우라면 식별할 수 있는 동작이므로 반드시 테스트한다.
- 개발자가 보는 경우라면 테스트해서는 안된다.

8.6.2 로깅을 어떻게 테스트해야 하는가?

ILogger 위에 래퍼 도입하기

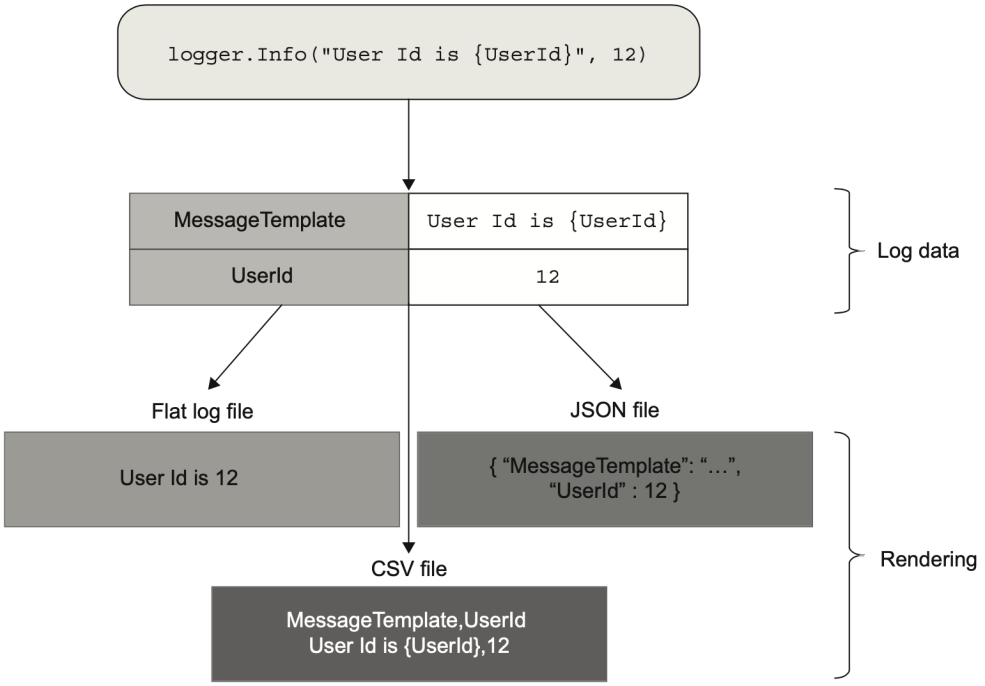
- ILogger 인터페이스를 목으로 처리하지 말라.
- 지원 로깅은 비즈니스 요구 사항이므로 코드베이스에 명시적으로 반영해야 한다.
- 대신 DomainLogger 클래스를 생성해서 해당 클래스와의 상호 작용을 확인 한다.

```
public class DomainLogger : IDomainLogger
{
    private readonly ILogger _logger;
    public DomainLogger(ILogger logger)
    {
        _logger = logger;
    }
    public void UserTypeHasChanged(
        int userId, UserType oldType, UserType newType)
    {
        _logger.Info(
            $"User {userId} changed type " +
            $"from {oldType} to {newType}");
    }
}
```

구조화된 로깅 이해하기

- **구조화된 로깅이란?**

- 로그 데이터 캡처와 렌더링을 분리하는 로깅 기술이다.
- 구조화된 로깅은 로그 저장소에 구조가 있다.
- 메시지 템플릿의 해시를 계산하고 해당 해시를 입력 매개변수와 결합해서 캡처한 데이터 세트를 형성한다.
- 데이터 랜더링을 할 때 기존 로깅과 마찬가지로 평범한 로그 파일을 사용할 수도 있고 JSON 또는 CSV 파일로 변경할 수도 있다.



- UserTypeHasChanged() 메서드를 메시지 템플릿의 해시로 볼 수 있다. 이 해시에 userId, oldType, newType 매개변수를 두어 로그 데이터를 만든다.

```
public void UserTypeHasChanged(
    int userId, UserType oldType, UserType newType)
{
    _logger.Info(
        $"User {userId} changed type " +
        $"from {oldType} to {newType}");
}
```

지원 로깅과 진단 로깅을 위한 테스트 작성

- DomainLogger는 프로세스 외부 의존성(로그 저장소)이 있다.

- 별도의 도메인 이벤트를 도입할 수 있다.

```
public void ChangeEmail(string newEmail, Company company)
{
    _logger.Info($"Changing email for user {UserId} to {newEmail}");

    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;

    UserType newType = company.IsEmailCorporate(newEmail)
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee ? 1 : -1;
        company.ChangeNumberOfEmployees(delta);
        // DomainLogger 대신 도메인 이벤트를 사용한다.
        AddDomainEvent(
            new UserTypeChangedEvent(
                UserId, Type, newType));
    }
    Email = newEmail;
    Type = newType;
    // DomainLogger 대신 도메인 이벤트를 사용한다.
    AddDomainEvent(new EmailChangedEvent(UserId, newEmail));

    _logger.Info($"Email is changed for user {UserId}");
}
```

- 이벤트 발생이 반영된 UserController

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null)
        return error;

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _eventDispatcher.Dispatch(user.DomainEvents);
```

```
    return "OK";
}
```

- EventDispatcher는 도메인 이벤트를 받아서 프로세스 외부 의존성에 대한 호출로 변환해주는 새로운 클래스다.
  - EmailChangedEvent는 _messageBus.SendEmailChangedMessge()로 변환
  - UserTypeChangedEvent는 _domainLogger.UserTypeHasChanged()로 변환

- UserTypeChangedEvent 를 사용하면 두 가지 책임(프로세스 외부 의존성 통신과 도메인 로직)을 분리할 수 있다.
  - 단위 테스트는 테스트 대상 User 에서 UserTypeChangedEvent 인스턴스를 확인해야 한다.
  - 단일 통합 테스트는 목을 사용해 DomainLogger 와의 상호 작용이 올바른지 확인해야 한다.

- 도메인 클래스가 아니라 컨트롤러에서 지원 로깅이 필요한 경우 도메인 이벤트를 사용할 필요가 없다.
  - UserController 에서는 해당 로거를 직접 사용한다.
  - 가능한 한 User 나 다른 도메인에서는 진단 로깅을 사용하지 않아야 한다.

8.6.3 로깅이 얼마나 많으면 충분한가?

- 도메인에서 진단 로깅을 사용하지 않아야 하는 이유는 다음과 같다.
  1. 과도한 로깅은 코드를 혼란스럽게 한다.
  2. 로그의 신호 대비 잡음 비율이 높아진다.
- 무언가를 디버깅 해야한다면?
  - 일시적으로 사용 후 디버깅이 끝나면 제거한다.

8.6.5 로거 인스턴스를 어떻게 전달하는가?

- 정적 메서드를 사용
  - 정적 메서드를 통해 ILogger 를 처리하고 비공개 정적 필드에 저장하는 방식은 앰비언트 컨텍스트에 속한다.
  - 이 방식은 **안티 패턴**으로 다음과 같은 단점이 있다.
    1. 의존성이 숨어있어 변경하기 어렵다.
    2. 테스트가 더 어려워진다.
    3. 코드의 잠재적인 문제를 가린다.
    4. 로그가 너무 많이 남겨질 수 있다.

해결 방안

- 명시적으로 주입하는 방식
  - 메서드 인수로 주입
  - 클래스 생성자를 통한 주입
- 메서드 인수 주입 방식 예시 코드

```
public void ChangeEmail(
    string newEmail,
    Company company,
    ILogger logger)
{
      logger.Info($"Changing email for user {UserId} to {newEmail}");
      /* ... */
      logger.Info($"Email is changed for user {UserId}");
}
```