7장. 가치 있는 단위 테스트를 위한 리팩터링

7.1 리팩터링할 코드 식별하기
    7.1.1 코드의 네가지 유형
        1) 코드 복잡도
        코드 복잡도 - 순환 복잡도 계산법
        도메인 유의성
        2) 협력자 수
        네 가지 코드 유형
    7.1.2 험블 객체 패턴을 사용해 지나치게 복잡한 코드 분할하기
7.2 가치 있는 단위 테스트를 위한 리팩터링하기
    7.2.1 고객 관리 시스템 소개
    7.2.2 1단계 : 암시적 의존성을 명시적으로 만들기
    7.2.3 2단계 : 애플리케이션 서비스 계층 도입
    7.2.4 3단계 : 애플리케이션 서비스 복잡도 낮추기
    7.2.5 4단계 : 새 Company 클래스 소개
7.3 최적의 단위 테스트 커버리지 분석
    샘플 프로젝트 험블 객체 패턴으로 리팩토링한 후의 코드 유형
    7.3.1 도메인 계층과 유틸리티 코드 테스트하기
    7.3.2 나머지 세 사분면에 대한 코드 테스트하기
    7.3.3 전제 조건을 테스트해야 하는가?
7.4 컨트롤러에서 조건부 로직 처리
    해결 방식
    세 가지의 특성의 균형을 맞추는 것
    단점
    7.4.1 CanExecute/Excute 패턴 사용
    7.4.2 도메인 이벤트를 사용해 도메인 모델 변경 사항 추적
        도메인 이벤트란?
        도메인 이벤트로 해법을 일반화할 수 있다.

7.1 리팩터링할 코드 식별하기

7.1.1 코드의 네가지 유형

- 모든 제품 코드는 2차원으로 분류할 수 있다.
    1. 복잡도 또는 도메인 유의성

2. 협력자 수

1) 코드 복잡도

- 코드 내 의사 결정(분기) 지점 수로 정의

코드 복잡도 - 순환 복잡도 계산법

- 순환 복잡도는 주어진 프로그램 또는 메서드의 분기 수를 나타낸다.
- 지표 계산은 다음과 같다.

```
1 + <분기점 수> = 순환 복잡도
```

- 예시
  - 제어 조건부가 없는 메서드의 경우 순환 복잡도가 `1 + 0 = 1` 이다.

도메인 유의성

- 문제 도메인에 대해 얼마나 의미있는지
- 유틸리티 코드는 그런 연관성이 없다.
- 메서드의 조건문이 없어서 순환 복잡도가 1인 경우라도 비즈니스에 중요한 기능이라면 테스트하는 것이 중요하다.
  - e.g. 주문 가격 계산하는 메서드

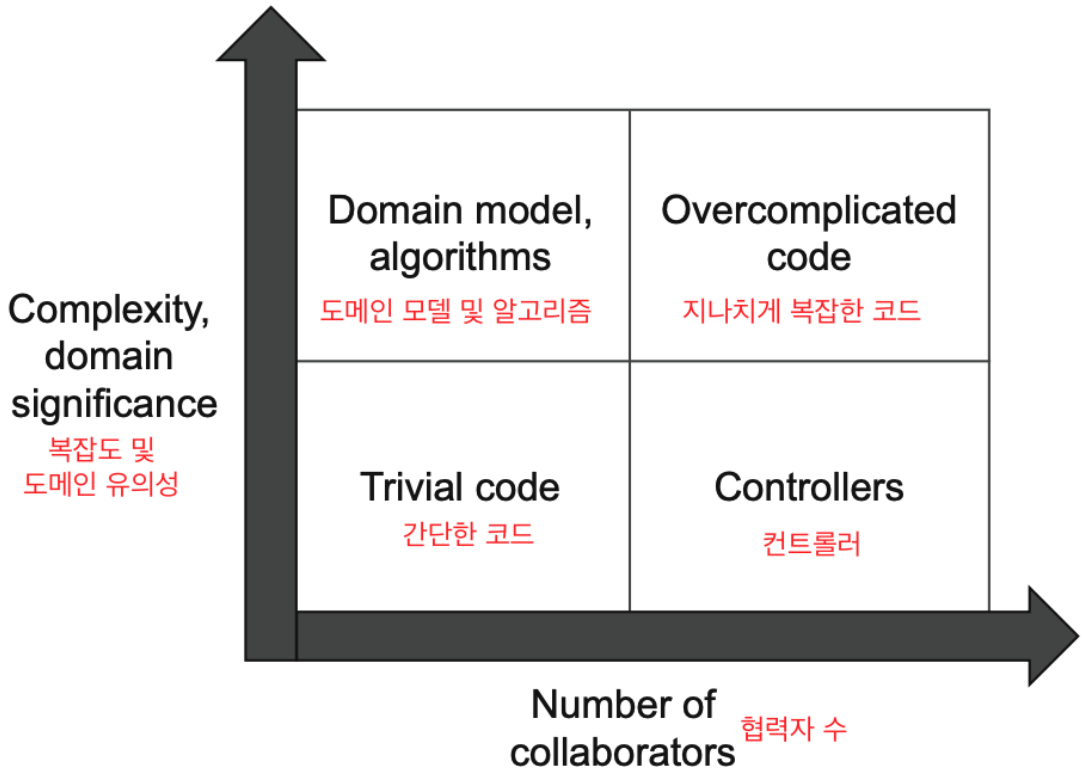
2) 협력자 수

- 협력자란?
  - 가변 의존성 또는 프로세스 외부 의존성(또는 둘 다)이다.
  - 협력자가 많을 수록 테스트 비용이 많이 든다.
- 협력자의 유형
  - 도메인 모델이라면?
    - 프로세스 외부 협력자를 사용하면 안된다. → 유지 비용이 많이 든다.

        - 도메인 클래스는 프로세스 내부 의존성에서만 동작해야 한다.
    - 암시적 협력자 VS 명시적 협력자
        - 테스트 대상 시스템이 협력자를 인수로 받거나 정적 메서드를 통해 암시적으로 참조해도 상관없다.
        - 테스트에서 이 협력자를 설정해야만 한다.
    - 불변 의존성
        - 불변 의존성(값 또는 값 객체)은 해당하지 않는다.
        - 설정과 검증이 훨씬 쉽다.

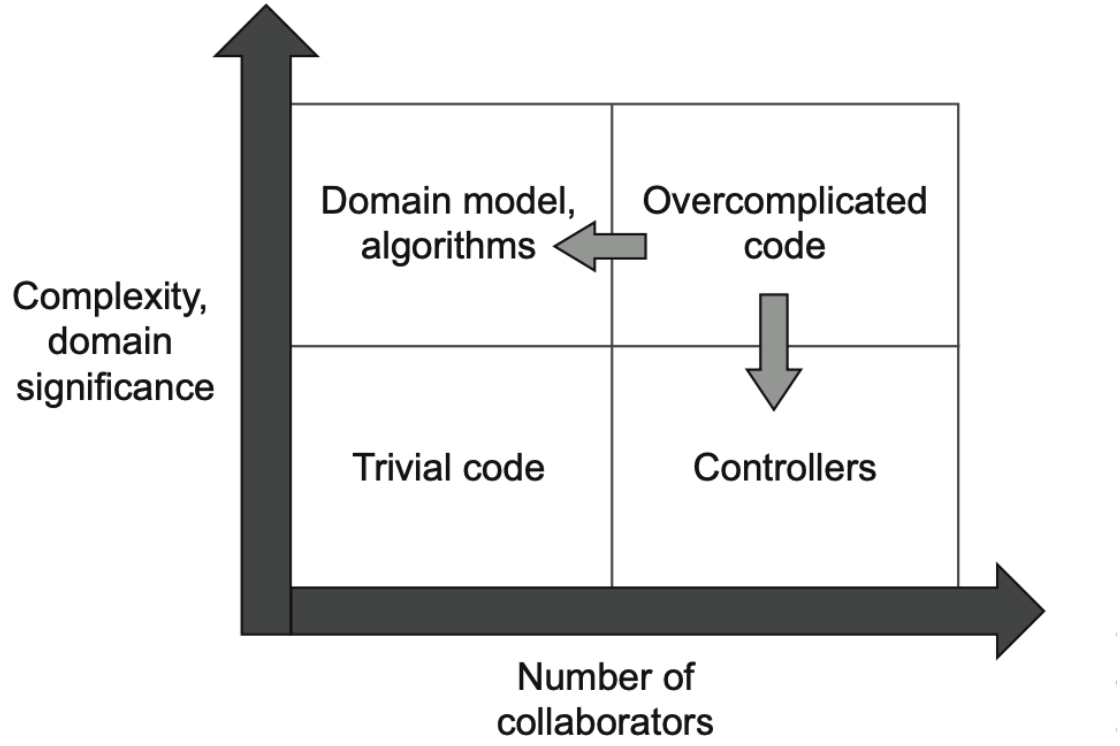
네 가지 코드 유형

1. 도메인 모델과 알고리즘
    - 보통 복잡한 코드는 도메인 모델이지만 도메인과 직접적으로 관련 없는 복잡한 알고리즘이 있을 수 있다.
2. 간단한 코드
    - 매개변수가 없는 생성자나 한 줄 속성이 있다.
    - 협력자가 없는 경우 복잡도나 도메인 유의성도 거의 없다.
3. 컨트롤러
    - 도메인 클래스와 외부 애플리케이션 같은 다른 구성 요소의 작업을 조정한다.
4. 지나치게 복잡한 코드
    - 두 가지 지표 모두 높은 경우이다. 협력자가 많고 복잡하거나 중요하다.
    - e.g. 덩치 큰 컨트롤러



- `도메인 및 알고리즘`을 단위 테스트하면 노력 대비 가장 가치있고 비용이 저렴하다.
  - 복잡하거나 중요한 로직을 수행해서 회귀 방지가 향상되기 때문이다.
- `간단한 코드`는 테스트할 필요가 전혀 없다.
  - 테스트 가치가 0에 가깝다.
- `복잡한 코드`가 가장 문제가 된다.
  - 테스트가 어렵겠지만 테스트 커버리지 없이 두는 것은 위험하다.
  - `알고리즘`과 `컨트롤러` 두 부분으로 나눈것이 일반적이다.

→ 지나치게 복잡한 코드를 피하고 도메인 모델과 알고리즘만 단위 테스트하는 것이 매우 가치있고 유지 보수가 쉬운 테스트 스위트로 가는 길이다.



7.1.2 험블 객체 패턴을 사용해 지나치게 복잡한 코드 분할하기

- 지나치게 복잡한 코드를 쪼개려면 `험블 객체 패턴`을 써야한다.
  - 테스트가 가능한 부분을 추출한다.
  - 결과적으로 테스트 사능한 부분을 둘러싼 얇은 `험블 래퍼`가 된다.
- 함수형 아키텍처는 더 나아가 프로세스 외부 의존성뿐만 아니라 **모든 협력자와의 커뮤니케이션에서 비즈니스 로직을 분리**한다.
  - 함수형 코어에는 `아무런 협력자가 없다.`
- 험블 객체 패턴을 보는 또 다른 방법?
  - `단일 책임 원칙(Single Responsibility principle)`을 지키는 것이다.
- 비즈니스 로직과 오케스트레이션을 분리하는 경우
  - 둘다 가능하지는 않다.
- MVP, MVC 패턴, 육각형 아키텍처 등

- DDD(Domain Driven Design) 도메인 주도 설계에 나오는 `집계 패턴(Aggregate pattern)` 이 있다.
    - 클래스를 클러스터(집계)로 물어서 클래스 간 연결을 줄이는 것이 향상

→ 코드 복잡도를 해결

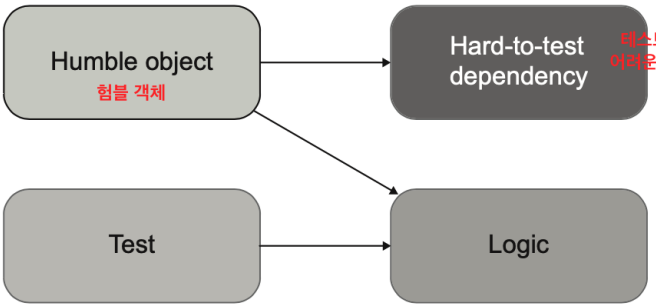


Figure 7.4 The Humble Object pattern extracts the logic out of the overcomplicated code, making that code so humble that it doesn't need to be tested. The extracted logic is moved into another class, decoupled from the hard-to-test dependency.

7.2 가치 있는 단위 테스트를 위한 리팩터링하기

7.2.1 고객 관리 시스템 소개

- 코드 예제

```
public class User {
    private int userId;
    private String email;
    private UserType type;

    public int getUserId() {
        return userId;
    }

    private void setUserId(int userId) {
        this.userId = userId;
    }

    public String getEmail() {
        return email;
    }

    private void setEmail(String email) {
```

```
        this.email = email;
    }

    public UserType getType() {
        return type;
    }

    private void setType(UserType type) {
        this.type = type;
    }

    // 파라미터로 받는 userId, newEmail 은 명시적 의존성을 보여준다.
    public void changeEmail(int userId, String newEmail) {
        Object[] data = Database.getUserById(userId);
        setUserId(userId);
        setEmail((String) data[1]);
        setType((UserType) data[2]);
        if (getEmail().equals(newEmail)) {
            return;
        }
        Object[] companyData = Database.getCompany();
        String companyDomainName = (String) companyData[0];
        int numberOfEmployees = (int) companyData[1];

        String emailDomain = newEmail.split("@")[1];
        boolean isEmailCorporate = emailDomain.equals(companyDomainName);
        UserType newType = isEmailCorporate ? UserType.Employee : UserType.Customer;
        if (getType() != newType) {
            int delta = newType == UserType.Employee ? 1 : -1;
            int newNumber = numberOfEmployees + delta;
            Database.saveCompany(newNumber);
        }
        setEmail(newEmail);
        setType(newType);
        // 암시적 의존성을 보여준다
        Database.saveUser(this);
        // 암시적 의존성을 보여준다.
        MessageBus.sendEmailChangedMessage(getUserId(), newEmail);
    }
}

public enum UserType {
    Customer(1),
    Employee(2);

    private final int value;

    UserType(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

- 명시적 의존성은 userId 와 newEmail 이다.(생성자)
  - 테스트 협력자 수에 포함되지 않는다.
- 암시적 의존성은 Database, MessageBus 이다.
  - 이 둘은 프로세스 `외부 협력자`이다.

활성 레코드 패턴

- 도메인 클래스가 스스로 데이터베이스를 조회
- 다시 저장
- 비즈니스 로직과 프로세스 외부 의존성과의 통신 사이 분리가 없어서 코드베이스가 커지면 확장하기 어렵다.

7.2.2 1단계 : 암시적 의존성을 명시적으로 만들기

- 암시적 의존성을 명시적 의존성으로 만든다.
- 해당 의존성은 여전히 프로세스 `외부`에 있다.
  - 데이터가 없는 **프록시 형태**다.
  - 복잡한 목 체계가 필요할 수 있다.

7.2.3 2단계 : 애플리케이션 서비스 계층 도입

- 다른 클래스인 험블 컨트롤러로 책임을 옮겨야 한다.
- 애플리케이션 서비스 도입의 문제점
  1. 외부 의존성이 주입되지 않고 직접 인스턴스화된다.
  2. 컨트롤러가 데이터베이스에서 받은 원시 데이터를 User 인스턴스로 재구성된다. 애플리케이션 서비스의 역할은 오케스트레이션만 해당한다.
  3. User의 업데이트된 직원 수를 반환하는 부분을 보면 특정 사용자와 관련이 없기 때문에 이 책임을 다른 곳으로 옮겨야한다.
  4. 이메인 결과 여부와 관계 없이 무조건 데이터를 수정해서 저장하고, 메시지 버스에 알림을 보낸다.

7.2.4 3단계 : 애플리케이션 서비스 복잡도 낮추기

- UserController 가 컨트롤러 사분면에 확실히 위치하려면 재구성 로직을 추출해야 한다.
  - ORM 사용하는 경우
    - 도메인 모델 매핑시 재구성 로직을 옮기기에 적절한 위치가 될 수 있다.
  - ORM 사용하지 않는 경우
    - 도메인 클래스를 인스턴스화하는 팩토리 클래스를 생성해라.
- 3단계 재구성 로직은 복잡하지만 도메인 유의성이 `없다.`

7.2.5 4단계 : 새 Company 클래스 소개

- 컨트롤러 코드 중 User에서 업데이트된 직원 수를 반환하는 부분이 어색하다.

```
object[] companyData = _database.GetCompany();
string companyDomainName = (string)companyData[0];
int numberOfEmployees = (int)companyData[1];

int newNumberOfEmployees = user.ChangeEmail(
    newEmail, companyDomainName, numberOfEmployees);
```

- 회사 관련 로직과 데이터를 함께 묶는 또 다른 도메인 클래스 Company 를 생성한다.

```
public class Company
{
    public string DomainName { get; private set; }
    public int NumberOfEmployees { get; private set; }
    public void ChangeNumberOfEmployees(int delta)
    {
        Precondition.Requires(NumberOfEmployees + delta >= 0);
        NumberOfEmployees += delta;
    }
    public bool IsEmailCorporate(string email)
    {
        string emailDomain = email.Split('@')[1];
        return emailDomain == DomainName;
```

```
    }
}
```

- 리팩토링 후 컨트롤러 코드

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

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

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

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);

        _messageBus.SendEmailChangedMessage(userId, newEmail);
    }
}
```

- 리팩토링 후 User 클래스
    - 이메일이 회사 이메일인지, 회사의 직원 수 변경하는 책임을 해당 인스턴스에 위임하고 있다.

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(string newEmail, Company company)
    {
        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);
        }

        Email = newEmail;
        Type = newType;
    }
}
```

7.3 최적의 단위 테스트 커버리지 분석

샘플 프로젝트 험블 객체 패턴으로 리팩토링한 후의 코드 유형

	협력자 거의 없음	협력자 많음
복잡도와 도메인 유의성이 높음	User의 ChangeEmail(newEmail, company), Company의 ChangeNumberOfEmployees(delta), isEmailCorporate(email), CompanyFactory의 Create(data)	
복잡도와 도메인 유의성이 낮음	User 와 Company 의 생성자	UserController 의 ChangeEmail(userId, newMail)

7.3.1 도메인 계층과 유틸리티 코드 테스트하기

- 비용 편의 측면에서 최상의 결과를 가져다 준다.
- 복잡도나 도메인 유의성이 높으면?
  - 회귀 방지 → `GOOD`
  - 협력자 X → `GOOD`
  - 유지 비용 낮음 → `GOOD`

7.3.2 나머지 세 사분면에 대한 코드 테스트하기

- 복잡도가 높고 협력자가 많은 코드를 리팩터링으로 제거했다.
- 간단한 코드는 테스트할 필요가 없다.

7.3.3 전제 조건을 테스트해야 하는가?

- 기본 규칙은 `도메인 유의성이 있는 모든 전제 조건`을 테스트해라.
    - **`전제 조건에 대한 테스트가 테스트 스위트에 있을 만큼 충분히 가치가 있는가?`**

7.4 컨트롤러에서 조건부 로직 처리

- 비즈니스 로직과 오케스트레이션의 분리
- 아래 3단계의 경우 가장 효과적이다.
    1. 저장소에서 데이터 검색
    2. 비즈니스 로직 실행
    3. 데이터를 다시 저장소에 저장

해결 방식

1. 외부에 대한 모든 읽기와 쓰기를 가장자리로 밀어낸다.
2. 도메인 모델에 프로세스 외부 의존성을 주입, 비즈니스 로직이 해당 의존성을 호출할 시점을 직접 결정한다.
3. 의사 결정 프로세스 단계를 더 세분화하고, 각 단계별 컨트롤러를 실행한다.

세 가지의 특성의 균형을 맞추는 것

1. 도메인 모델 테스트 유의점
    - 도메인 클래스의 협력자 수와 유형에 따른 함수

2. 컨트롤러 단순성
    - 의사 결정 지점이 있는지에 따라 다름
3. 성능
    - 프로세스 외부 의존성에 대한 호출 수로 정의

단점

- 해결 방식 1은 `성능이 저하`될 수 있다.
- 해결 방식 2는 도메인 모델의 테스트 유의성이 떨어진다.
- 해결 방식 3은 컨트롤러가 단순하지 않다. 컨트롤러에 의사 결정 지점이 있어야 한다.

7.4.1 CanExecute/Excute 패턴 사용

- 비즈니스 로직이 도메인 모델에서 컨트롤러로 유출되는 것을 방지한다.

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }
    public bool IsEmailConfirmed
        { get; private set; }
    /* ChangeEmail(newEmail, company) method */
}

public string ChangeEmail(string newEmail, Company company)
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";
    /* the rest of the method */
}
```

- 모든 의사 결정을 제거한 **컨트롤러**
    - 성능저하를 감수해야한다.

```
public string ChangeEmail(int userId, string newEmail)
{
    // 데이터 준비
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);
    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    // 의사 결정
    string error = user.ChangeEmail(newEmail, company);

    // 결정에 따라 실행
    if (error != null)
        return error;

    _database.SaveCompany(company);
    _database.SaveUser(user);

    _messageBus.SendEmailChangedMessage(userId, newEmail);
    return "OK";
}
```

- 사용자의 이메일을 변경할지 여부를 결정하는 컨트롤러
  - 성능은 유지된다.
  - 그러나 의사 결정 프로세스가 두 부분으로 나뉜다.
    1. 이메일 변경 진행 여부(컨트롤러에서 수행)
    2. 변경 시 해야할 일(User 에서 수행)
  - 도메인 모델의 캡슐화가 떨어진다.

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

    if (user.IsEmailConfirmed) Decision-making
      return "Can't change a confirmed email";

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

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
```

```
    _messageBus.SendEmailChangedMessage(userId, newEmail);

    return "OK";
}
```

- canExcute/Excute 패턴을 사용한 이메일 변경
  - 컨트롤러는 이메일 변경 프로세스를 알 필요가 없다.
  - ChangeEmail()의 전제 조건이 추가되어도 먼저 확인하지 않으면 이메일을 변경할 수 없음이 보장된다.
- 이 패턴은 도메인 계층의 모든 결정을 통합할 수 있다.
  - User 클래스의 전제 조건을 단위 테스트하는 것으로 충분해진다.

```
public string CanChangeEmail()
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";
    return null;
}

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

7.4.2 도메인 이벤트를 사용해 도메인 모델 변경 사항 추적

- 무슨 일이 일어나는지 외부 시스템에 알리는 것이 목적이다.
- `도메인 이벤트`로 이러한 추적을 구현한다.

도메인 이벤트란?

- 애플리케이션 내에서 도메인 전문가에게 중요한 이벤트를 말한다.
- 일반 이벤트(버튼 클릭 등)와 도메인 이벤트를 구분하는게 중요하다.
- 중요한 변경 사항을 외부 애플리케이션에 알리는 데 사용된다.
- 외부 시스템에 통보하는 데 필요한 데이터가 포함된 클래스이다.
    - e.g. 사용자 ID, 이메일 등
- 이벤트 클래스는 항상 과거 시제로 명명한다.

```
public class EmailChangedEvent
{
    public int UserId { get; }
    public string NewEmail { get; }
}
```

- 리팩터링한 ChangeEmail() 메서드

```
public void ChangeEmail(string newEmail, Company company)
{
    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);
    }

    Email = newEmail;
    Type = newType;
    EmailChangedEvents.Add(
        new EmailChangedEvent(UserId, newEmail));
}
```

- 리팩터링한 ChangeEmail() 을 호출하는 컨트롤러

```
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 (var ev in user.EmailChangedEvents)
    {
        _messageBus.SendEmailChangedMessage(
            ev.UserId, ev.NewEmail);
    }
    return "OK";
}
```

- 데이터베이스와의 통신은 CRM 의 식별할 수 잇는 동작이 아니라 `구현 세부사항`에 속한다.
- 반면, 메시지 버스와의 통신은 애플리케이션의 `식별할 수 있는 동작`이다.

  → 외부 시스템과의 계약을 지키려면 이메일이 변경될 때만 메시지를 메시지 버스에 넣어야 한다!!

도메인 이벤트로 해법을 일반화할 수 있다.

- DomainEvent 기초 클래스를 추출
- 도메인 클래스가 이 기초 클래스를 참조
- (`List<XXXEvent>` 같이 컬렉션을 포함할 수 있음)
- 별도의 이벤트 디스패처를 작성할 수 있다.
- 도메인 이벤트 발송 전 병합하는 매커니즘이 필요할 수 있다.
- 컨트롤러에서 의사 결정 책임을 제거

- 해당 책임을 도메인 모델에 위임함으로써 외부 시스템과의 통신에 대한 단위 테스트를 간결하게 한다.
- 아래와 같이 컨트롤러 검증, 프로세스 외부 의존성을 목으로 대체하기 보단, 단위 테스트에서 직접 도메인 이벤트 생성을 테스트할 수 있다.

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var company = new Company("mycorp.com", 1);
    var sut = new User(1, "user@mycorp.com", UserType.Employee, false);

    sut.ChangeEmail("new@gmail.com", company);

    company.NumberOfEmployees.Should().Be(0);
    sut.Email.Should().Be("new@gmail.com");
    sut.Type.Should().Be(UserType.Customer);
    // 컬렉션 크기와 요소를 동시에 검증
    sut.EmailChangedEvents.Should().Equal(
        new EmailChangedEvent(1, "new@gmail.com"));
}
```