9장. 목 처리에 대한 모범 사례

9.1 목의 가치를 극대화하기

주요 모범 사례

- [ ] `비관리  의존성에만  목  적용하기`
- [ ] 시스템 끝에 있는 의존성에 대해 상호 작용 검증하기
- [ ] 통합 테스트에서만 목을 사용하고 단위 테스트에서는 하지 않기
- [ ] 항상 목 호출 수 확인하기
- [ ] 보유 타입만 목으로 처리하기

- 지난 챕터 사용자 컨트롤러 마지막 버전

```
public class UserController {

  private final Database database;
  private final EventDispatcher eventDispatcher;
```

```
  public UserController(Database database, IMessageBus messageBus, IDomainLogger domainLogger) {
      this.database = batabase;
      this.eventDispatcher = new EventDispatcher(messageBus, domainLogger);
  }

  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.getDomainEvents());
    return "OK";
  }

}
```

- EventDispatcher 클래스가 도입됨.
  - EventDispatcher는 도메인 모델에서 생성된 도메인 이벤트를 비관리 의존성에 대한 호출로 변환한다.
  - 이는 전에 컨트롤러가 수행하던 것이다.

```
public class EventDispatcher {

  private final IMessageBus messageBus;
  private final IDomainLogger domainLogger;

  public EventDispatcher(IMessageBus messageBus, IDomainLogger domainLogger) {
    this.messageBus = messageBus;
    this.domainLogger = domainLogger;
  }

  public void dispatch(List<IDomainEvent> events) {
    for (IDomainEvent event : events) {
        dispatch(event);
    }
  }

  private void dispatch(IDomainEvent event) {
      if (event instanceof EmailChangedEvent) {
          EmailChangedEvent emailChangedEvent = (EmailChangedEvent) event;
          messageBus.sendEmailChangedMessage(
            emailChangedEvent.getUserId(),
            emailChangedEvent.getNewEmail());
      } else if (event instanceof UserTypeChangedEvent) {
          UserTypeChangedEvent userTypeChangedEvent = (UserTypeChangedEvent) event;
```

```
          domainLogger.userTypeHashChanged(
            userTypeChangedEvent.getUserId(),
            userTypeChangedEvent.getOldType(),
            userTypeChangedEvent.getNewType());
      }
  }
}
```

- 모든 프로세스 외부 의존성(관리 의존성 + 비관리 의존성)을 거친 테스트
  - IMessageBus 와 IDomainLogger를 목으로 처리

```
@Test
public void changing_email_from_corporate_to_non_corporate() {

  // Arrange
  Database database = new Database("connection String");
  User user = new User("user@mycorp.com", UserType.Employee, database);
  createCompany("mycorp.com", 1, database);

  IMessageBus messageBusMock = Mockito.mock(IMessageBus.class);
  IDomainLogger loggerMock = Mockito.mock(IDomainLogger.class);
  UserController sut = new UserController(db, messageBusMock, loggerMock);

  // Act
  String result = sut.changeEmail(user.getUserId(), "new@egmail.com");

  // Assertion
  assertEquals("OK", result);

  Object[] userData = database.getUserById(user.getUserId());
  User userFromDb = UserFactory.create(userData);
  assertEquals("new@gmail.com", userFromDb.getEmail());
  assertEquals(UserType.Customer, userFromDb.getType());

  Object[] companyData = database.getCompany();
  Company companyFromDb = CompanyFactory.create(companyData);
  assertEquals(0, companyFromDb.getNumberOfEmployees());

  messageBusMock.verify(
    x -> x.sendEmailChangedMessage(
      user.getUserId(), "new@gmail.com"), Times.Once
    );
  loggerMock.verify(
    x -> x.userTypeHasChanged(
      user.getUserId(), UserType.Employee, UserType.Customer), Times.Once
    );
  );
}
```

9.1.1 시스템 끝에서 상호 작용 검증하기

주요 모범 사례

- [x] ~~비관리 의존성에만 목 적용하기~~
- [ ] `시스템 끝에 있는 의존성에 대해 상호 작용 검증하기`
- [ ] 통합 테스트에서만 목을 사용하고 단위 테스트에서는 하지 않기
- [ ] 항상 목 호출 수 확인하기
- [ ] 보유 타입만 목으로 처리하기

- 목을 사용할 때 항상 다음 지침을 따른다.
  - `시스템 끝에서 비관리 의존성과의 상호 작용을 검증`하라.
- 바로 이전 예제에서 messageBusMock 의 문제점은 IMessageBus 인터페이스가 시스템 끝에 있지 않다는 것이다.
  - IMessageBus의 구현은 다음과 같다.

```
public interface IMessageBus
{
    void SendEmailChangedMessage(int userId, string newEmail);
}

public class MessageBus : IMessageBus
{
    private readonly IBus _bus;
    public void SendEmailChangedMessage(
        int userId, string newEmail)
    {
        _bus.Send("Type: USER EMAIL CHANGED; " +
            $"Id: {userId}; " +
            $"NewEmail: {newEmail}");
    }
}

public interface IBus
{
    void Send(string message);
}
```

  - MessageBus 와 IBus 인터페이스 모두 프로젝트 코드베이스에 속한다.
  - IBus 는 `메시지 버스 SDK 라이브러리 위`에 있는 Wrapper 이다.
    - IBus(=Wrapper)는 기술 세부 사항을 캡슐화한다.
    - IBus 는 임의의 텍스트 메시지를 메시지 버스로 보낼 수 있다.

- IMessageBus 는 `IBus 위`에 있는 Wrapper 로 도메인과 관련된 메시지를 정의한다.
  - IMessageBus 는 모든 메시지를 한 곳에 보관하고 애플리케이션에서 재사용할 수 있다.
- IBus 와 IMessageBus를 합칠 수 있다. 그러나 차선책이다.
  - **두 가지 책임을 분리**하는 것이 좋다.
  - ILogger 와 IDomainLogger 예시와 같은 상황이다.
  - IDomainLogger 는 비즈니스에 필요한 특정 로깅 기능만 구현

헥사고날 아키텍처에서 IBus와 IMessageBus 의 위치

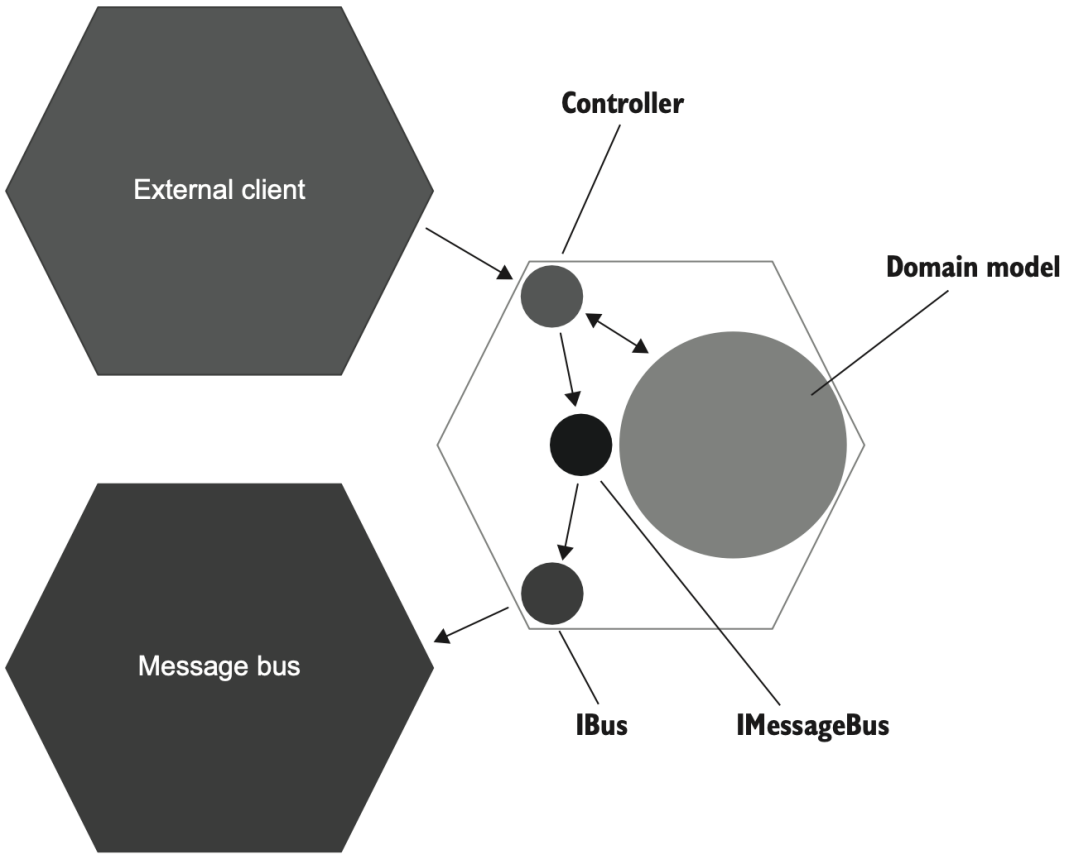

Figure 9.1 `IBus` **resides at the system's edge;** `IMessageBus` **is only an intermediate link in the chain of types between the controller and the message bus. Mocking** `IBus` **instead of** `IMessageBus` **achieves the best protection against regressions.**

- IBus 는 컨트롤러와 메시지 버스 사이의 타입 사슬에서 마지막 고리이다.
- IMessageBus 는 중간에 위치한다.
- IMessageBus 대신 IBus 를 목으로 처리하면 `회귀 방지를 극대화`할 수 있다.

- **비관리 의존성과 통신하는 마지막 타입을 목으로 처리하면** 통합 테스트가 거치는 클래스의 수가 증가하므로 `보호가 향상` 된다.

- IMessageBus 대신 IBus를 대상으로 한 통합 테스트

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busMock = new Mock<IBus>();
    var messageBus = new MessageBus(busMock.Object);
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busMock.Verify(
        x => x.Send("Type: USER EMAIL CHANGED; " +
                    $"Id: {user.UserId}; " +
                    "NewEmail: new@gmail.com"),
        Times.Once);
}
```

- IMessageBus 인터페이스가 아닌 MessageBus 구체 클래스를 사용하고 있다.
  - 인터페이스를 두는 이유는 목으로 처리하기 위한 용도일 뿐이다.
  - IMessageBus를 더이상 목으로 처리하지 않기 때문에 구체 클래스로 대체했다.
- IBus 목으로 외부 시스템에 전송한 실제 텍스트 메시지 검증하는 방식은 회귀 방지가 좋고 리팩터링 내성도 향상된다.
  - 외부 시스템은 애플리케이션으로부터 텍스트 메시지를 수신할 뿐 MessageBus 와 같은 클래스를 호출하지 않는다.
  - 실제 텍스트 메시지가 외부에서 식별 가능한 유일한 사이드 이펙트이다.
  - 메시지 생성에 참여하는 클래스는 단지 구현 세부 사항일 뿐이다.
  - 잠재적 거짓 양성에 노출될 가능성이 낮아지게 된다.
  - 코드 베이스와 결합이 낮기 때문에 낮은 수준의 리팩터링에도 영향을 많이 받지 않는다.
- 애플리케이션을 떠나기 전 비관리 의존성에 대한 호출 단계들 중 마지막 단계를 선택하라.
  - 외부 시스템과의 하위 호환성을 보장하는 가장 좋은 방법이다.
  - 목을 통해 하위 호환성을 달성할 수 있다.

9.1.2 목을 스파이로 대체하기

스파이란

- 스파이는 목과 같은 목적을 수행하는 테스트 대역이다.
- 스파이는 수동으로 작성하는 방면에 목은 목 프레임워크의 도움을 받아 생성한다.
- 직접 작성한 목이라고도 한다.

스파이 사용 케이스

- 시스템 끝에 있는 클래스의 경우 목 대신 스파이를 사용하는 것이 낫다.
- 검증 단계에서 코드를 재사용할 수 있어 테스트 크기를 줄이고 가독성을 향상시킨다.

- IBus 위에서 작동하는 스파이 예시

```
public interface IBus
{
    void Send(string message);
}

public class BusSpy : IBus
{
    private List<string> _sentMessages =
        new List<string>();

    public void Send(string message)
    {
        _sentMessages.Add(message);
    }

    public BusSpy ShouldSendNumberOfMessages(int number)
    {
        Assert.Equal(number, _sentMessages.Count);
        return this;
    }

    public BusSpy WithEmailChangedMessage(int userId, string newEmail)
    {
        string message = "Type: USER EMAIL CHANGED; " +
                         $"Id: {userId}; " +
                         $"NewEmail: {newEmail}";

        Assert.Contains(_sentMessages, x => x == message);
        return this;
    }
}
```

- 스파이 사용한 통합 테스트

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busSpy = new BusSpy();
```

```
    var messageBus = new MessageBus(busSpy);
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busSpy.ShouldSendNumberOfMessages(1)
        .WithEmailChangedMessage(user.UserId, "new@gmail.com");
}
```

→ BusSpy가 제공하는 `플루언트 인터페이스` 덕분에 메시지 버스와의 상호 작용을 검증하는 것이 간결해졌고 표현력이 생겼다.

- (용어가 익숙치 않다는 이유로 BusSpy의 이름을 BusMock으로 바꿀 수 있지만 동료에게 불필요한 혼란을 줄 수 있으므로 그대로 사용하는 것을 권장한다.)

- 플루언트 인터페이스(fluent interface)

  ▼ 코드 예시

```
class Person {
  private String name;
  private int age;

  public Person setName(String name) {
    this.name = name;
    return this;
  }

  public Person setAge(int age) {
    this.age = age;
    return this;
  }

  public void introduce() {
    System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
  }

  public static void main(String[] args) {
    Person person = new Person();
    person.setName("Peter").setAge(21).introduce();
    // Hello, my name is Peter and I am 21 years old.
  }
}
```

MessageBus 와의 비교

- BusSpy 와 MessageBus 모두 IBus 의 wrapper 이기 때문에 검증은 비슷하다.
  - IMessageBus 목으로 처리한 코드

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

- 검증은 유사할 수 있으나 결정적 차이가 있다.
  - BusSpy는 `테스트 코드`에 속한다.
  - MessageBus는 `제품 코드`에 속한다.
- 테스트에서 검증문 작성 시 제품 코드에 의존하면 안 되므로 이 차이는 중요하다.
  - 테스트는 감시자다.
  - 모든 것을 재확인한다.
  - 메시지 구조가 변경될 때 알람이 생기게끔 별도의 검사점이 있는 셈이다.
  - IMessageBus를 목으로 처리시 제품 코드를 너무 많이 신뢰하게 된다.

9.1.3 IDomainLogger는 어떤가?

```
busSpy.ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(
        user.UserId, "new@gmail.com");

loggerMock.Verify(
    x => x.UserTypeHasChanged(
        user.UserId,
        UserType.Employee,
        UserType.Customer),
    Times.Once);
```

- MessageBus는 IBus위의 래퍼로 IBus 를 목으로 대체하여 테스트하고 있다.
- 위와 같이 DomainLogger 는 ILogger 위의 래퍼고 ILogger 인터페이스도 애플리케이션 경계에 있기 때문에 목 대상을 다시 지정해야하나?
  - NO. 로거와 메시지 버스는 비관리 의종성이므로 둘다 모두 하위 호환성을 유지해야 한다. 하지만, 호환성의 정확도가 같을 필요는 없다.
  - 메시지 버스를 사용하면 외부 시스템이 이런 변경에 어떻게 반응하는지 알 수 없으므로 메시지 구조를 변경하지 않는 것이 중요하다.
  - 그러나, **텍스트 로그의 정확한 구조는 대상 독자에게 중요하지 않다.**
  - 로그가 있다는 사실과 로그에 있는 정보만이 중요할 뿐이다.
  - 따라서 `IDomainLogger 만 목으로 대체해도 보호 수준은 충분`하다.

9.2 목 처리에 대한 모범 사례

주요 모범 사례

- [x] ~~비관리 의존성에만 목 적용하기~~
- [x] ~~시스템 끝에 있는 의존성에 대해 상호 작용 검증하기~~
- [ ] `통합 테스트에서만 목을 사용하고 단위 테스트에서는 하지 않기`
- [ ] `항상 목 호출 수 확인하기`
- [ ] 보유 타입만 목으로 처리하기

9.2.1 목은 통합 테스트만을 위한 것

- 도메인 모델에 대한 테스트는 단위 테스트 범주에 속하며, 컨트롤러를 다루는 테스트는 통합 테스트다.
    - 비즈니스 로직과 오케스트레이션의 분리 원칙을 따른다.
- 목은 비관리 의존성에만 해당하며 컨트롤러만 이러한 의존성을 처리하는 코드이다.
    - 따라서 통합 테스트에서 컨트롤러를 테스트할 때만 목을 적용해야 한다.

9.2.2 테스트당 목이 하나일 필요는 없음

- 목이 둘 이상인 경우 한 번에 여러가지를 테스트할 가능성이 있다.
- `단위` 라는 용어는 코드 단위가 아니라 **동작 단위를 의미**한다.
    - 동작 단위 구현에 필요한 **코드의 양은 관계가 없다.**
    - 목을 사용해도 같은 원칙이 적용된다.
        - 동작 단위를 검증하는 데 필요한 **목의 수는 관계가 없다.**
        - 목의 수는 운영에 참여하는 비관리 의존성 수에만 의존한다.(제품 코드상 비관리성 의존성 개수와는 같아야한다..)

9.2.3 호출 횟수 검증하기

- 비관리 의존성과의 통신에 대해서 다음 두 가지 모두 확인해야 한다.
  1. 예상하는 호출이 있는가?
  2. 예상치 못한 호출은 없는가?

- 두 가지 요구사항은 비관리 의존성과 하위호환성을 지켜야 하는 데서 비롯된다.
  - 호환성은 양방향이어야 한다.
  - 외부 시스템이 예상하는 메시지를 생략해서는 안 된다.
  - 예상치 못한 메시지도 생성해서는 안된다.

- 아래와 같이 메시지 전송을 확인하는 것만으로는 충분하지 않다.

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

- 메시지 호출 횟수도 함께 확인해야 한다.

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

- 대부분 목 라이브러리는 목에 다른 호출이 없는지 명시적으로 확인할 수 있게 도와준다.
- Moq 라이브러리 사용한 검증문 예시

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

- BusSpy에서도 구현하고 있다.

```
public BusSpy ShouldSendNumberOfMessages(int number)
{
    Assert.Equal(number, _sentMessages.Count);
    return this;
}

public BusSpy WithEmailChangedMessage(int userId, string newEmail)
{
```

```
    string message = "Type: USER EMAIL CHANGED; " +
                     $"Id: {userId}; " +
                     $"NewEmail: {newEmail}";

    Assert.Contains(_sentMessages, x => x == message);
    return this;
}
```

```
busSpy
    .ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(user.UserId, "new@gmail.com");
```

9.2.4 보유 타입만 목으로 처리하기

주요 모범 사례

- [x] ~~비관리 의존성에만 목 적용하기~~
- [x] ~~시스템 끝에 있는 의존성에 대해 상호 작용 검증하기~~
- [x] ~~통합 테스트에서만 목을 사용하고 단위 테스트에서는 하지 않기~~
- [x] ~~항상 목 호출 수 확인하기~~
- [ ] `보유 타입만 목으로 처리하기`

- 스티브 프리먼, 냇 프라이스의 지침
  - 서드파티 라이브러리 위에 항상 어댑터를 작성한다.
  - 기본 타입 대신 해당 어댑터를 목으로 처리해야 한다.

- 해당 지침 관련 몇몇 주장
  1. 서드파티 코드의 작동 방식에 대해 깊이 이해하지 못하는 경우가 많다.
  2. 해당 코드가 내장 인터페이스를 제공하더라도 목으로 처리한 동작이 실제 외부 라이브러리와 일치하는지 확인해야 하므로 해당 인터페이스를 목 처리하는 것은 위험하다.
  3. 서드파티 코드의 기술 세부 사항까지 꼭 필요하지 않다. 어댑처는 이를 추상화하고, 애플리케이션 관점에서 라이브러리와의 관계를 정의한다.

- 실제 어댑터는 코드와 외부 환경 사이의 손상 방지 계층(anti-corruption layer)으로 작동한다.
  1. 기본 라리브러리의 복잡성을 추상화
  2. 라이브러리에서 필요한 기능만 도출

3. 프로젝트 도메인 언어를 사용해 수행할 수 있다.

IBus 인터페이스

- IBus 인터페이스가 이 목적에 부합한다.
  - 기본 메시지 버스 라이브러리가 IBus 인터페이스만큼 깔끔한 인터페이스를 제공해도 고유 래퍼를 그위 두는 것이 좋다.
  - 라이브러리 업그레이드시 서드파티 코드가 어떻게 변경될지 알 수 없다.
  - 이런 추상 계층을 두면 파급 효과를 하나의 클래스(어댑터 등)로 제한할 수 있다.

지침의 적용 범위

- 프로세스 내부 의존성에는 해당하지 않는다.
- `비관리 의존성에만 해당` 한다.
- 인메모리 의존성이나 관리 의존성을 추상화할 필요가 없다.
  - 라이브러리 날짜와 시간 API 제공하는 경우 비관리 의존성에 도달하지 않으므로 해당 API를 있는 그대로 사용한다.
  - ORM이 외부 애플리케이션에서 볼 수 없는 데이터베이스 접근에 사용되는 한 ORM 을 추상화할 필요가 없다.
- 모든 라이브러리에 고유 래퍼를 둘 수 있지만 비관리 의존성 이외에는 노력을 들일 만한 가치가 없다.