BS450 Yazılım Test Mühendisliği

Test Dublörleri (Doubles)

Bir Test Problemi

To: hoca@erciyes.edu.tr
From:babannemiyediler@erciyes.edu.tr
Subject: Test ile ilgili problem - Lütfen yardım edin!!!

Merhaba Hocam

Bitirme Ödevimdeki projem için bir davranışı test eden birim test yazmak istiyorum ancak bir veritabanı bağlantısı içeriyor.

Veritabanını testime dahil etmek biraz zor olacak ayrıca entegrasyon değil birim test yazmak istiyorum

Ne yapmalıyım?

Saygılarımla,
Ersoy

Bir Test Çözümü

To: babannemiyediler@erciyes.edu.tr
From:hoca@erciyes.edu.tr
Subject: Re: Test ile ilgili problem - Lütfen yardım edin!!!

Merhaba Ersoy,

Korkmaya gerek yok.

Test dublörü kullanmayı düşünebilirsin.

Bir sonraki derste bu konuyu ele alacağız.

Saygılarımla,
Hoca

Senaryo

`BankAccount` isimli bir sınıfımız var.

Banka hesap bilgilerini depolamak ve almak için bir veritabanı kullanılır.

Gerçek veritabanıyla etkileşime girmeden bu sınıfın mantığını nasıl birim testine tabi tutarız?

```
public class BankAccount {

    private final int bankAccountNumber;
    private final BankAccountDatabaseConnection bankAccountDatabaseConnection;

    public BankAccount(BankAccountDatabaseConnection bankAccountDatabaseConnection,
                       int openingBalance, int overdraft) {
        this.bankAccountDatabaseConnection = bankAccountDatabaseConnection;
        this.bankAccountNumber = bankAccountDatabaseConnection.createBankAccount();
        setOverdraft(overdraft);
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, openingBalance);
    }

    public void withdraw(int amount) {
        if (amount <= 0) {
            throw new BankAccountException("Cannot withdraw a zero or negative amount");
        }
        int balance = getBalance();
        int maxWithdrawalAmount = balance + getOverdraft();
        if (amount > maxWithdrawalAmount) {
            throw new BankAccountException(
              "Cannot withdraw more than the current balance plus the overdraft");
        }
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
    }

    public void deposit(int amount) {
        if (amount <= 0) {
            throw new BankAccountException("Cannot deposit a zero or negative amount");
        }
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);
    }

    public void setOverdraft(int amount) {
        if (amount < 0) {
```

Senaryo

`BankAccount` isimli bir sınıfımız var.

Banka hesap bilgilerini depolamak ve almak için bir veritabanı kullanılır.

Gerçek veritabanıyla etkileşime girmeden bu sınıfın mantığını nasıl birim testine tabi tutarız?

```
        int balance = getBalance();
        int maxWithdrawalAmount = balance + getOverdraft();
        if (amount > maxWithdrawalAmount) {
            throw new BankAccountException(
              "Cannot withdraw more than the current balance plus the overdraft");
        }
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
    }

    public void deposit(int amount) {
        if (amount <= 0) {
            throw new BankAccountException("Cannot deposit a zero or negative amount");
        }
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);
    }

    public void setOverdraft(int amount) {
        if (amount < 0) {
            throw new BankAccountException("Overdraft cannot be negative");
        }
        bankAccountDatabaseConnection.setOverdraft(bankAccountNumber, amount);
    }

    public int getBankAccountNumber() {
        return bankAccountNumber;
    }

    public int getBalance() {
        return bankAccountDatabaseConnection.getBalance(bankAccountNumber);
    }

    public int getOverdraft() {
        return bankAccountDatabaseConnection.getOverdraft(bankAccountNumber);
    }
}
```

Sorunlu Nesne

`BankAccount'taki` mantığı, veritabanına SQL sorguları gönderen ve içindeki değerleri döndüren bu sınıf tarafından uygulanan, kullandığı temel veritabanı hakkında endişelenmenize gerek kalmadan test etmek istiyoruz.

```
public class BankAccountDatabaseConnection {

    public int createBankAccount() {
        // make database calls
    }

    public int getBalance(int bankAccountNo) {
        // make database calls
    }

    public void setBalance(int bankAccountNo, int amount) {
        // make database calls
    }

    public int getOverdraft(int bankAccountNo) {
        // make database calls
    }

    public void setOverdraft(int bankAccountNo, int amount) {
        // make database calls
    }
}
```

Test Dublörleri

Test Dublörü, bazı orijinal sınıfların "yerine geçen" sınıflardır ve testlerin, bunun yerine o orijinal sınıfın kullanılması durumunda ihtiyaç duyulan karmaşıklığın bir kısmından kaçınmasına olanak tanır.

Test dublörleri biraz aktör dublörlerine benzer; gerçek aktörü kullanmak yerine ona benzeyen ama tüm zorlu şeyleri kolaymış gibi gösteren başka bir aktör kullanıyoruz!

Ryan Gosling ve dublörü "Barbie" filminin setinde.

Test Dublörü Türleri

1. Dummies
2. Stubs
3. Fakes
4. Mocks
5. Spies

Dummies

Dummies (Yapay nesneler), gerçek nesnenin yerine geçen nesnelerdir.

Ancak test hiçbir zaman dummy'i kullanmaz, amacı yalnızca derleyiciyi memnun etmektir.

Bu ve sonraki slaytlarda şunu varsayıyoruz: `BankAccountDatabaseConnection`, test amacıyla farklı şekillerde uygulayabileceğimiz bir Java arayüzüdür. Ancak aynı etkiyi elde etmek için gerçek bir sınıfın yöntemleri kolayca geçersiz kılınabilir.

```
public class DummyBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 0;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {
    }
}
```

Dummy

Dummies

Method under test

```
public void withdraw(int amount) {
    if (amount <= 0) {
        throw new BankAccountException("Cannot withdraw a zero or negative amount");
    }
    int balance = getBalance();
    int maxWithdrawalAmount = balance + getOverdraft();
    if (amount > maxWithdrawalAmount) {
        throw new BankAccountException(
          "Cannot withdraw more than the current balance plus the overdraft");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
}
```

Veritabanının kendisi bu testin amaçları açısından önemli değildir. Bu yüzden testi derlemek için sadece bir kuklaya ihtiyacımız var.

Dummy kullanan test

```
@Test
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {
    DummyBankAccountDatabaseConnection dummy = new DummyBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(dummy, 0, 0);

    // When a negative amount is withdrawn, Then an exception is thrown
    assertThrows(BankAccountException.class, () -> {
        bankAccount.withdraw(-1000);
    });
}
```

Stub

Stub'lar, başka bir sınıfın/yöntemin test edilebilmesi için orijinalin belirli yöntemlerini geçersiz kılan nesnelerdir.

```
public BankAccount(BankAccountDatabaseConnection bankAccountDatabaseConnection,
                   int openingBalance, int overdraft) {
    this.bankAccountDatabaseConnection = bankAccountDatabaseConnection;
    this.bankAccountNumber = bankAccountDatabaseConnection.createBankAccount();
    setOverdraft(overdraft);
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, openingBalance);
}
```

`BankAccount` – **original constructor**

```
public class StubbedBankAccountDatabaseConnection
        implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 1000;
    }

    // ...
```

Stubbed method of `BankAccountDatabaseConnection`

Stub kullanan test

```
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    StubbedBankAccountDatabaseConnection stub = new StubbedBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(stub, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

1 2 3

Fakes

Fake'ler, gerçek nesnenin sahte uygulamalarını sağlar.
Burada – veritabanı işlevselliğinin "bellek içi" uygulaması.

Fake'lerin dezavantajlarına dikkat edin; esasen kendisinin test edilmesi gereken daha fazla işlevsellik uygulanabilir.

```
public class FakeBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    private final int bankAccountNumber;
    private int balance;
    private int overdraft;

    public FakeBankAccountDatabaseConnection(int bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }

    @Override
    public int createBankAccount() {
        return bankAccountNumber;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return balance;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        this.balance = amount;
    }

    // ...
```

Fake

Fakes

Method under test

```
public void withdraw(int amount) {
    if (amount <= 0) {
        throw new BankAccountException("Cannot withdraw a zero or negative amount");
    }
    int balance = getBalance();
    int maxWithdrawalAmount = balance + getOverdraft();
    if (amount > maxWithdrawalAmount) {
        throw new BankAccountException(
          "Cannot withdraw more than the current balance plus the overdraft");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
}
```

Fake kullanan test

```
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    FakeBankAccountDatabaseConnection fake = new FakeBankAccountDatabaseConnection(0);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(fake, 500, 0);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    assertThat(bankAccount.getBalance(), equalTo(400));
}
```

Mocks

Mock’lar (taklit) saplama fikrini genişletir; bir yöntemin döndürdüğü değerleri kontrol etmenize olanak tanır, aynı zamanda **yöntemlerin argüman olarak doğru değerlerle çağrıldığını** da doğrulayabilir.

Method under test

```
public void deposit(int amount) {
    if (amount <= 0) {
        throw new BankAccountException("Cannot deposit a zero or negative amount");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);
}
```

Fake kullanmadıkça (ve fake için daha fazla test yazmadıkça), banka hesabının bakiyesini ayarlamak için veritabanına giren değerin doğru olduğunu tespit etmenin hiçbir yolu yoktur.

Mocks

Mock

```
public class MockedBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    public boolean verify = false;

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 100;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        verify = (bankAccountNo == 1000 && amount == 200);
    }

    // ...
```

Mock kullanan test

```
@Test
public void shouldDepositAmount() {
    MockedBankAccountDatabaseConnection mock
        = new MockedBankAccountDatabaseConnection();

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    // (as signalled by a flag - mock.verify - being set to true)
    assertThat(mock.verify, equalTo(true));
}
```

Veritabanına, hesap numarası 1000 için bakiye tutarını 200 £ olarak ayarlama talimatı verildiğini açıkça doğrulayın.

Spies

Casuslar, mock’lara benzerler ancak stub yöntemler (önceden belirlenmiş değerleri döndüren yöntemler) yoktur.

Yani sadece **yöntem çağrısı kayıt etme ve kontrol etme** kısmını yaparlar.

Bir birim ile harici bir bileşen arasındaki arayüzü kontrol etmek için kullanışlıdırlar. (Bazen entegrasyon testlerinin bir parçası olarak bile kullanılırlar.)

Örneğin, yöntemleri gözetlemek ve doğru SQL'in oluşturulduğunu kontrol etmek için kullanılabilirler.

Veya bir e-postanın içeriğinin, onu göndermek için bir hizmet çağrılmadan önce beklendiği gibi olması.

Dublörlere dikkat edin

Örneklerden kaçının uygulama detayını iki katına çıkardığına dikkat edin. Özellikle:

- **Fake'lerin** kendi testlerine(!) ihtiyacı vardır, çünkü daha fazla uygulama gerektirirler
- **Mock'lar**, bireysel yöntem çağrılarıyla ilgili ayrıntıları kaydeder ve bu da onları kırılganlığa yatkın hale getirir.

Bu nedenle dublörleri dikkatli ve yalnızca gerektiğinde kullanın.

Her şeyi olabildiğince gerçek tutmak çoğu zaman en iyi yoldur ve dublörlerden tamamen kaçınmaktır.

Mockito

Yeni bir şey test etmek istediğinizde her seferinde bir test dublörü yazmak oldukça acı verici olabilir.

Mockito, JUnit ile kullanılmak üzere modeller oluşturmak için kullanışlı bir frameworktür.

Mock'lar stub ve spy olduğundan ve stub'lar da dummy'lerin daha özel versiyonları olduğundan, Mockito fake'ler dışında her türlü dublörü üretebilir.

Mockito ile Mock örnek

```java
@Test
public void shouldDepositAmount() {
    MockedBankAccountDatabaseConnection mock
        = new MockedBankAccountDatabaseConnection();

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    // (as signalled by a flag - mock.verify - being set to true)
    assertThat(mock.verify, equalTo(true));
}
```

Elle yazılmış mock kullanan bir test

```java
public class MockedBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    public boolean verify = false;

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 100;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        verify = (bankAccountNo == 1000 && amount == 200);
    }

    // ...
```

Elle yazılmış mock sınıf

Mockito ile Mock Örneği

```
@Test
public void shouldDepositAmount() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // Set the database mock to return a balance of £100 for this account number
    // as would have been instructed to have been set in the database via
    // the constructor
    when(mock.getBalance(1000)).thenReturn(100);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    verify(mock).setBalance(1000, 200);
}
```

Mock nesne oluşturun. Gerçek kod (buna ihtiyacımız zaten bulunmaz) asla görünmez

Mock için “stubbed” metot oluşturun

Mock'a belirli çağrıların yapıldığını doğrulayın. `setBalance` bir komutla çağrıldı mı? `1000` numaralı banka hesabı, bakiyesi `200` mü?

Virtual mock kullanan test

Fake'in, Mock'a dönüşümü

```java
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    FakeBankAccountDatabaseConnection fake = new FakeBankAccountDatabaseConnection(0);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(fake, 500, 0);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    assertThat(bankAccount.getBalance(), equalTo(400));
}
```

Manuel yazılmış Fake ile test

Manuel yazılmış fake sınıf.
Bu sadece test amaçlı ama bizim de bunu test etmemiz gerekecek!

```java
public class FakeBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    private final int bankAccountNumber;
    private int balance;
    private int overdraft;

    public FakeBankAccountDatabaseConnection(int bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }

    @Override
    public int createBankAccount() {
        return bankAccountNumber;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return balance;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        this.balance = amount;
    }

    // ...
```

Fake'in Mock'a dönüşümü

```
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(mock, 500, 0);

    // Set the database mock to return a balance of £500 for this account number
    // as would have been instructed to have been set in the database via
    // the constructor
    when(mock.getBalance(1000)).thenReturn(500);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    verify(mock).setBalance(1000, 400);
}
```

Bunun yerine yalnızca mock kullanılabilir

Bu kod, son mock örneğine benzemektedir.

Mockito ile Dummy Örneği

```java
@Test
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {
    DummyBankAccountDatabaseConnection dummy = new DummyBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(dummy, 0, 0);

    // When a negative amount is withdrawn, Then an exception is thrown
    assertThrows(BankAccountException.class, () -> {
        bankAccount.withdraw(-1000);
    });
}
```

Elle yazılmış dummy ile test

Elle yazılan dummy sınıf

```java
public class DummyBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 0;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {
    }
}
```

Mockito ile Dummy Örneği

```
@Test
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {
    // Setup the mock
    BankAccountDatabaseConnection mock = mock();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(mock, 0, 0);

    // When a negative amount is withdrawn, Then an exception is thrown
    assertThrows(BankAccountException.class, () -> {
        bankAccount.withdraw(-1000);
    });
}
```

Mock nesneyi oluşturun. Herhangi bir yönteme stub yazmadığımız veya doğrulamadığımız için bu metot aslında bir dummy'dir.

Virtual mock kullanarak test (bu senaryoda bir dummy)

Mockito ile Stub örneği

```java
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    StubbedBankAccountDatabaseConnection stub = new StubbedBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(stub, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

Elle yazılmış stub kullanan test

Elle yazılmış stub sınıf

```java
public class StubbedBankAccountDatabaseConnection
             implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {

    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {

    }
}
```

Mockito ile Stub örneği

```
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a bank account
    BankAccount bankAccount = new BankAccount(mock, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

Mock nesne ve stub metot oluştur.
Gidip herhangi bir yöntemi doğrulamadığımız için bu aslında bir stub'tır.

Mockito – Özet

Mockito, doblörleri manuel olarak yazarken birçok işten tasarruf sağlayabilir.

Mockito burada anlattıklarımızdan daha fazlasını yapabilir, bkz. https://site.mockito.org/

…ancak cazip olan, alternatifleri düşünmeden doğrudan testi yapabilmektir.
Dublörler kırılgan testlere yol açabilir.

Her zaman bir entegrasyon testinin daha uygun olup olmadığını değerlendirin.