BIM539, Ders 11: Test Edilebilir Kod Yazma

Test Edilebilir Kodun Anahtar Fikirleri

- Kodu parçalara böl - modüler hale getir
- DRY (Don’t repeat yourself, kendini tekrar etme)
- Kendinize test etmek için bir şeyler verin
- Teste uygun olmayan fonksiyonları teste uygun olmayan yapıların dışına koy
- Ön koşulların sağlanmasını kolaylaştır
- Tekrar üretilebilirliği kolaylaştır
- Yerelleştirmeyi kolaylaştır

Kodu Parçalara Böl

- Metotların yalnızca bir adet iyi tanımlanmış fonksiyonu olmalıdır

```
// Kötü
public int getNumMonkeysAndSetDatabase(Database d) {
  database = (d != null) ? d : DEFAULT_DATABASE;
  setDefaultDatabase(database);
  int numMonkeys = monkeyList.size();
  return numMonkeys;
}
```

- **Neden?**

Yeniden düzenleme

```
public void setDatabase(Database d) {
  database = (d != null) ? d : DEFAULT_DATABASE;
  setDefaultDatabase(database);
}
public int getNumMonkeys() {
  int numMonkeys = monkeyList.size();
  return numMonkeys;
}
```

- Bu neden daha iyidir?

1. Daha modüler ve yüksek kalitede kod
2. Ayrıca test etmesi çok daha kolay --- `Database d` bağımlılığı yok!

DRY (Don’t repeat yourself, kendini tekrar etme)

- Copy/Paste yapma
- Benzer fonksiyona sahip birden fazla fonksiyon yazma
- “generic” sınıf ve metotları kullan
  - Parametreli türe sahip sınıflar ve metotlar
  - Ör. Java `ArrayList<Type>`, `Type` ile parametreli hale getirilmiştir
  - Dil implementasyonları: Java generics, C++ templates, …

Neden DRY?

- Daha fazla hata ihtimali
- Şişmiş kod
- Bir bug veya iyileştirme kodun bütün kopyalarında düzeltilmeli
  – Hatanın bir başka kaynağı

Kötü: Farklı veri tipleri için kodun çoğullanması

```
private ArrayList<Animal> animalList;

public int addMonkey(Monkey m) {
  animalList.add(m);
  return animalList.count();
}

public int addGiraffe(Giraffe g) {
  animalList.add(g);
  return animalList.count();
}

public int addRabbit(Rabbit r) {
  animalList.add(r);
  return animalList.count();
}
```

Yeniden Düzenleme

```
// Animal is superclass of Giraffe, Monkey, Rabbit

private ArrayList<Animal> animalList;
public int addAnimal (Animal a) {
  animalList.add(a);
  return animalList.count();
}
```

Kötü: Bir superclass bulunmuyorsa?

```
// No superclass for List<Monkey>,List<Giraffe>,List<Rabbit>
public void addOne(List<Monkey> l, Monkey m) {
  l.add(m);
}
public void addOne(List<Giraffe> l, Giraffe g) {
  l.add(g);
}
public void addOne(List<Rabbit> l, Rabbit b) {
  l.add(b);
}
```

Yeniden Düzenleme

```
// generic kullan.
// List<T> gönder, T herhangi bir tipte
olabilir.

public <T> void addOne(List<T> l, T e) {
  l.add(e);
}
```

Çoğaltılmış kod metodların içerisinde de olabilir!

```
// Bir metot…
String name =
     db.where("user_id = " + id).get_names()[0];

// Bir başka metot…
String name =
     db.find(id).get_names().first();

// Her ikisi de temelde aynı şeyi yapmaktadır
```

Şöyle düzenlenebilir:

```
// Bir metot…
String name = getName(db, id);


// Bir başka metot…
String name = getName(db, id);


String getName(Database db,  int id) {
  // Buradaki bir iyileştirme bütün çağrıları
  iyileştirecektir
  return db.find(id).get_names().first();
}
```

Kendinize test etmek için bir şeyler verin

- Buradaki ”bir şeylerden” kasıt bir durumdur (genellikle return değeri)

- Çünkü durum onaylaması genellikle davranış onayından daha iyidir
  - Fonksiyon çağırıldıktan sonra programın durumunun kontrol edilmesi sonucu onaylamak için doğrudan bir yoldur (direct).
  - Davranış onaylama uygulama ayrıntılarını incelediği için kırılgandır ve indirect’dir.

Davranış onaylama neden indirect'tir?

```
class Number {
  private int val;
  public void setVal(int v) { val = v; }
}
class Example {
  public void setSquared(Number n, int v) {
    n.setVal(v*v);
  }
}
@Test void testSetSquared() {
  Example ex = new Example();
  Number n = Mockito.mock(Number.class)
  ex.setSquared(n, 3);
  Mockito.verify(n).setVal();
}
```

Davranış onaylama neden indirect'tir?

```
class Number {
  private int val;
  public void setVal(int v) { val = v; }
}
class Example {
  public void setSquared(Number n, int v) {
    n.setVal(v+v); // KUSUR!
  }
}
@Test void testAddTen() {
  Example ex = new Example();
  Number n = Mockito.mock(Number.class)
  ex.setSquared(n, 3);
  Mockito.verify(n).setVal(); // YİNE DE
  BAŞARILIDIR!
}
```

Davranış onaylama neden kırılgandır?

```
class Number {
  private int val;
  public void setVal(int v) { val = v; }
  public void setSquared(int v) { val = v*v; }
}
class Example {
  public void setSquared(Number n, int v) {
    n.setVal(v*v);
  }
}
@Test void testAddTen() {
  Example ex = new Example();
  Number n = Mockito.mock(Number.class)
  ex.setSquared(n, 3);
  Mockito.verify(n).setVal();
}
```

Davranış onaylama neden kırılgandır?

```
class Number {
  private int val;
  public void setVal(int v) { val = v; }
  public void setSquared(int v) { val = v*v; }
}

class Example {
  public void setSquared(Number n, int v) {
    n.setSquared(v); // n.setVal(v*v) ile AYNI ŞEY
  }
}

@Test void testAddTen() {
  Example ex = new Example();
  Number n = Mockito.mock(Number.class)
  ex.setSquared(n, 3);
  Mockito.verify(n).setVal(); // BİR KUSUR YOK ANCAK
  HATAYA DÜŞER!
}
```

Yeniden düzenleme

```
class Number {
  private int val;
  public void setVal(int v) { val = v; }
}
class Example {
  public int setSquared(Number n, int v) {
    int ret = v*v;
    n.setVal(ret);
    return ret;
  }
}
@Test void testAddTen() {
  Example ex = new Example();
  Number n = Mockito.mock(Number.class);
  assertEquals(9, ex.setSquared(n, 3));
}
```

Teste uygun olmayan fonksiyonları (TUF) teste uygun olmayan yapıların (TUC) dışına koy

Teste uygun olmayan özellik örnekleri

- Konsola yazdırma
- Veritabanına yazma/okuma
- Dosya sistemine yazma/okuma
- Farklı bir program veya sisteme erişim
- Ağa erişim

Stub kullanarak sahtelerini oluşturacağınız kodlardır

Teste uygun olmayan yapılar

- Private metotlar
- Final metotlar
- Final sınıflar
- Sınıf constructor / destructor
- Static metotlar

Stub veya override ile sahtelerini oluşturmanın zor olduğu metotlardır

TUC içinde TUF bulunmamalı

- Yani…
- Sahtesini oluşturmak istediğin kodu (TUF) sahtesini oluşturmanın zor olduğu yapıların (TUC) içine koyma

Ön koşulların sağlanmasını kolaylaştır

- Harici veriye bağımlılık == Kötü

- Harici veri nedir?
  - Global değişken değeri
  - Global veri yapısından çıkarılmış bir değer
  - Dosya veya veritabanından bir değer
  - Temelde parametre olarak göndermediğin herhangi bir değer
  - Yan etki (side-effect) olarak da bilinirler

Ön koşulların sağlanmasını kolaylaştır

```
// Kötü
public float getCatWeight(Cat cat) {
   int fishWeight = Fish.weight;
   // Çünkü kedi balığı yedi
   return cat.weight + fishWeight;
}
```

- Neden? `Fish.weight` parametre olarak gönderilmemiş harici bir veridir.
  - Uygulamada derinlerdeki bu bağımlılık kodlama için iyi bir yaklaşım değildir (Geliştirici `Fish.weight`'i değiştirebilir ve yanlışlıkla `getCatWeight'i etkileyebilir`)
  - Test için de iyi bir kod değildir: **preconditions**'da belirtilmesi kolaylıkla unutulabilir

Tekrar düzenleme

```
// Daha iyi
public float getCatWeight(Cat cat, int fishWeight) {
   return cat.weight + fishWeight;
}
```

- Neden? `fishWeight` bir parametre ile alınır oldu.
  - Bu yüzden de `fishWeight` değiştiğinde `getCatWeight` değişmesi kimseyi şaşırtmayacaktır
  - Test etmesi kolay: artık ön koşul yok!
- Bütün değerler parametre olarak gönderiliyor
  - Bu tür metotlara saf (pure) metot denir.
  - Neden fonksiyonel dillerin test edilmesi kolay ve daha az hata meydana geliyor?

Ön koşulların sağlanmasını kolaylaştır

- Harici veriye erişim sağlamak zorunda kalırsak erişimi nerede yapmalıyız?

1. Parametreleri kullanırsanız, daha az global değişkene ihtiyaç duyarsınız
   - Orijinalde: `getCatWeight`:
     ```
     Fish.weight = 5;
     int weight = getCatWeight(cat);
     ```
     Yeniden düzenlenince: `getCatWeight`:
     ```
     int weight = getCatWeight(cat, 5);
     ```
2. Geri kalan harici veri için:
   - Yan etkileri olan test edilmesi zor metotları olabildiği kadar azaltın ve bunları bir köşeye ayırarak belirtin
   - Olabildiği kadar çok metodu saf (pure) metot yapın

Tekrar üretilebilirliği kolaylaştır

- Rastgele veriye bağımlılık == Kötü
  - Rastgele veri aynı sonucun tekrar üretilmesini imkansız hale getirir

```
// Kötü
public Result playOverUnder() {
   // rastgele zar at
   int dieRoll = (new Die()).roll();
   if (dieRoll > 3) {
      return RESULT_OVER;
   }
   else {
    return RESULT_UNDER;
   }
}
```

Tekrar düzenleme

```
public Result playOverUnder(Die d) {
   int dieRoll = d.roll();
   if (dieRoll > 3) {
      return RESULT_OVER;
   }
   else {
    return RESULT_UNDER;
   }
}
```

- Neden daha iyi? Şimdi Die sınıfı kopyalanıp d.roll() stub fonksiyon oluşturulabilir

  d.roll():

```
Die d = Mockito.mock(Die.class);
Mockito.when(d.roll()).thenReturn(6);
playOverUnder(d);
```

Daha da iyisi

```
public Result playOverUnder(int dieRoll) {
   if (dieRoll > 3) {
      return RESULT_OVER;
   }
   else {
    return RESULT_UNDER;
   }
}
```

- Neden? Herhangi bir mock veya stub'a ihtiyaç yok!

```
playOverUnder(6);
```

Yerelleştirmeyi kolaylaştır

```
// Kötü
public class House {
   private Room bedRoom;
   private Room bathRoom;
   public House() {
      bedRoom = new Room("bedRoom");
      bathRoom = new Room("bathRoom");
   }
   public String toString() {
      return bedRoom.toString() + " " + bathRoom.toString();
   }
}
```

- Neden? `bedRoom` ve `bathRoom`'u mock ve `toString()`'i stub yapma şansı yok

Yeniden Düzenleme

```
public class House {
  private Room bedRoom;
  private Room bathRoom;
  public House(Room r1, Room r2) {
    bedRoom = r1;
    bathRoom = r2;
  }
  public String toString() {
    return bedRoom.toString() + " " + bathRoom.toString();
  }
}
```

- Şimdi kolaylıkla mock ve stub işlemleri yapılabilir:

```
Room bedRoom = Mockito.mock(Room.class);
Room bathRoom = Mockito.mock(Room.class);
House house = new House(bedRoom, bathRoom);
```

Yerelleştirmeyi kolaylaştır

- Buna *dependency injection* adı verilmektedir

- *Dependency injection*: Bağımlı nesneyi parametre olarak gönderme (Dahili olarak oluşturmak yerine)
  - Nesneyi kopyalayarak (mock) daha rahat test etmenizi sağlar
  - Ayrıca tekrarlanabilirlik için de mock etmemizi sağlar
  - Yazılım mühendisliği açısından bir başka faydası da iki sınıf arasındaki decoupling'tir. (*Decoupled*: bir sınıfı bir başka sınıf ile değiştirmenin kolay olduğu anlamına gelir)

Miras, Eski (Legacy) kodlar

Image from https://goiabada.blog

Eski Kodlar ile Başa Çıkma

- Sınıfların büyük bir çoğunluğunda oldukça kolaydır
  - Ya sıfırdan kod yazılır
  - Ya da yazılmış kod değiştirilir

- Ancak gerçek dünya nadiren bu kadar düzenlidir
  - Kodlar genellikle baskı altında aceleyle yazılır ve test etme düşüncesi yoktur
  - Genellikle hiçbir dokümantasyon yoktur ve kodun nasıl çalıştığından bile emin olamazsınız.
- Peki nereden başlamalı?

Pinning testleri yazarak başla

- *Pinning Test*: **Mevcut davranışı** test etmek için yazılan test
  - Not: Mevcut davranış **beklenen davranıştan farklı** olabilir
  - Değiştirmeden önce tüm davranışlar, hatalar gibi ayrıntıların tespit edilmesi için yapılır
  - Metot özelliklerini ihlal eden kusurlu davranışlar bile kullanılır!
    - Bunların bile yanlışlıkla değiştirilmediğinden emin olmalısınız
    - Use case değişmeden bunları düzenlerseniz, uygulamanız bozulacaktır!

- Pinning test genellikle birim test kullanılarak gerçekleştirilir
  - Birim test yapabileceğim noktalar nereler?
  - *Dikişleri* arayın!

Eski Kodda Dikiş Bulma

- Yazılım QA'da dikişler:
  - İki modülün kesiştiği ve birinin bir başkası yerin değiştirilebildiği noktalardır
  - Kaynak kodda herhangi bir değişiklik yapmadan bu değişim yapılabilir
- Pinning test için dikişler neden önemlidir?
  - Eski kodda her birimin davranışını tespit etmek için
  - Dikiş, birim testlerini yerelleştirmek için sahte nesnelerin kullanılabileceği yerdir.
- Kaynak kodu değiştirmemek neden önemlidir?
  - Pinning testin tüm amacı eski kodu olduğu gibi test etmektir
  - Eğer kodu değiştirirsek, davranışı değiştirme tehlikesi doğar

Eski Kodda Dikiş Bulma

- Dikişsiz eski kod:

```
String read(String sql) {
  DatabaseConnection db = new DatabaseConnection();
  return db.executeSql(sql);
}
```

  - Gerçek bir veritabanına ihtiyaç duyduğu için test etmesi zordur

- Dikişli eski kod:

```
String read(String sql, DatabaseConnection db) {
  return db.executeSql(sql);
}
```

  - `db` mock nesnesi ve `db.executeSql` stub'ını kullanarak test etmek kolaydır.

Eski Kodda Dikiş Bulma

- Peki burada gerçekten dikiş yok mudur?

```
String read(String sql) {
  DatabaseConnection db = new DatabaseConnection();
  return db.executeSql(sql);
}
```

- Daha yakından inceleyelim!

Eski Kodda Dikiş Bulma

- Bu eski sınıfın var olduğunu var sayalım

```
class Database
  String read(String sql) {
    DatabaseConnection db = new DatabaseConnection();
    return db.executeSql(sql);
  }
}
```

- Ve yeni bir sınıf oluşturalım: `DatabaseForTesting`

```
class DatabaseForTesting : public Database
  String read(String sql) {
    // Bir veritabanı bağlantısı olduğunu varsayıp bir
    girdi döndürün
    return "test veritabanı girdisi";
  }
}
```

- `DatabaseForTesting` sınıfını test amacıyla kullanın

Eski Kod ile Başa Çıkma

- Davranışı belirledikten sonra kodu tekrar düzenlemeye başlayabilirsiniz

- Kodu, bulduğunuz halinden daha iyi bir hale getirin.

Textbook Chapter16'yı okuyun