

Kendinize profesyonel demeniz için temiz kod yazmanız gerekiyor. En iyisinden azını yapmak için hiçbir mantıklı bahaneniz olamaz.

Clean Code. Robert C Martin, (namı diğer bob amca)

## **Clean Code'dan Notlar**

## İçindekiler

Bölüm 1 — Temiz Kod Derken? .....	6
Peki Koda Ne Oldu? .....	7
Temiz Kod Nedir? .....	8
 Bölüm 2 - Anlamlı İsimlendirmelerle <i>Takım</i> Arkadaşlarınıza Küçük Sürprizler Yapın.....	10
Telaffuz Edilebilir İsimler Kullanın.....	12
Aranabilir İsimler Kullanın.....	12
Arayüzler ve Gerçekleştirmeler (Interfaces - Implementations).....	12
Kelime Oyunu Yapmaktan Kaçının .....	13
Gereksiz Bağlamlardan Kaçının.....	13
 Bölüm 3 - Fonksiyonlar .....	16
 Bölüm 4 - <i>Yorum Yok</i> .....	28
Yorumlarla Kötü Kodu Süslemeyin .....	29
İyi Yorumlar .....	29
Yasal Yorumlar .....	29
Bilgilendirici Yorumlar.....	30
TODO Yorumları.....	31
Javadoc'lar .....	31
Kötü Yorumlar.....	31
Gereksiz Yorumlar.....	32
Zorunlu Yorumlar.....	33
Günlük Gibi Yazılan Yorumlar .....	33
Bir Fonksiyon ya da Değişken Kullanabilecekken Yorum Yazmayın.....	35
Kapama Parantezlerine Koyduğumuz Yorumlar .....	35
Kapatılmış Kodlar .....	36
Yersiz Bilgi .....	37
 Bölüm 5 - Formatlama .....	42

Formatlamanın Amacı .....	42
Dikey Formatlama.....	42
Gazete Metaforu .....	43
Kavramlar Arası Dikey Açıklık.....	44
Dikey Yoğunluk .....	45
Dikey Uzaklık.....	46
Örnek Değişkenler (Instance Variables).....	46
Bağımlı (Dependent) Fonksiyonlar.....	47
Bir satır ne kadar geniş olmalı? .....	48
Yatay Açıklık ve Yoğunluk.....	49
Girintileme .....	49
Takım Kuralları .....	50
Uncle Bob'un Formatlama Kuralları.....	50
 Bölüm 6 - Nesneler ve Veri Yapıları .....	 53
Veri Soyutlama.....	53
Veri/Nesne Anti-Simetrisi .....	54
Demeter Kuralı (Law of Demeter).....	56
Tren Enkazları .....	57
Veri Aktarım Nesneleri (Data Transfer Objects) .....	59
Melez Yapılar (Hybrids).....	59
Aktif Kayıt (Active Record) .....	60
 Bölüm 7 - Hata İşleme.....	 61
Dönüş Kodları Yerine İstisnaları Kullanın .....	61
İlk Önce Try-Catch-Finally Bloklarını Yazın .....	62
Kontrolsüz (Unchecked) İstisnalar Kullanın .....	64
İstisnalarla Bağlam Sağlayın.....	65
Çağırının İhtiyaçlarına Uygun İstisna Sınıfları Tanımlayın .....	65
Normal Bir Akış Tanımlayın.....	66
Null Değerler Dönmeyin .....	67
Null Argümanlar Geçmeyin.....	68

Bölüm 8 - Sınırlar .....	70
Üçüncü Taraf Yazılım (Kod) Kullanmak .....	70
Sınırları Keşfetmek ve Öğrenmek .....	72
Log4j Öğrenmek.....	72
Temiz Sınırlar .....	75
 Bölüm 9 - Birim Testleri .....	76
TDD'nin 3 Kuralı .....	76
Testlerimizi Temiz Tutmak.....	77
Temiz Testler .....	78
Çifte Standart.....	80
Test Başına Bir Doğrulama (Assert) .....	82
Test Başına Tek Bir Konsept.....	82
F.I.R.S.T Kuralı .....	83
Sonuç .....	84
 Bölüm 10 — Sınıflar .....	85
Sınıf Düzeni .....	85
Kapsülleme (Encapsulation).....	85
Sınıflar Küçük Olmalıdır.....	85
Birbirine Bağlılık (Cohesion).....	88
Değişim İçin Düzenleme .....	90
Değişimden İzole Etme .....	92
 Bölüm 11 - Sistemler.....	94
Oluşturma Aşamalarının Kullanımdan Ayrılması .....	94
“main”in Ayrılması .....	95
Fabrikalar .....	95
Bağımlılık Enjeksiyonu .....	95
Big Design Up Front .....	96
 Bölüm 12 (Final) - Temiz Tasarımın 4 Kuralı.....	97

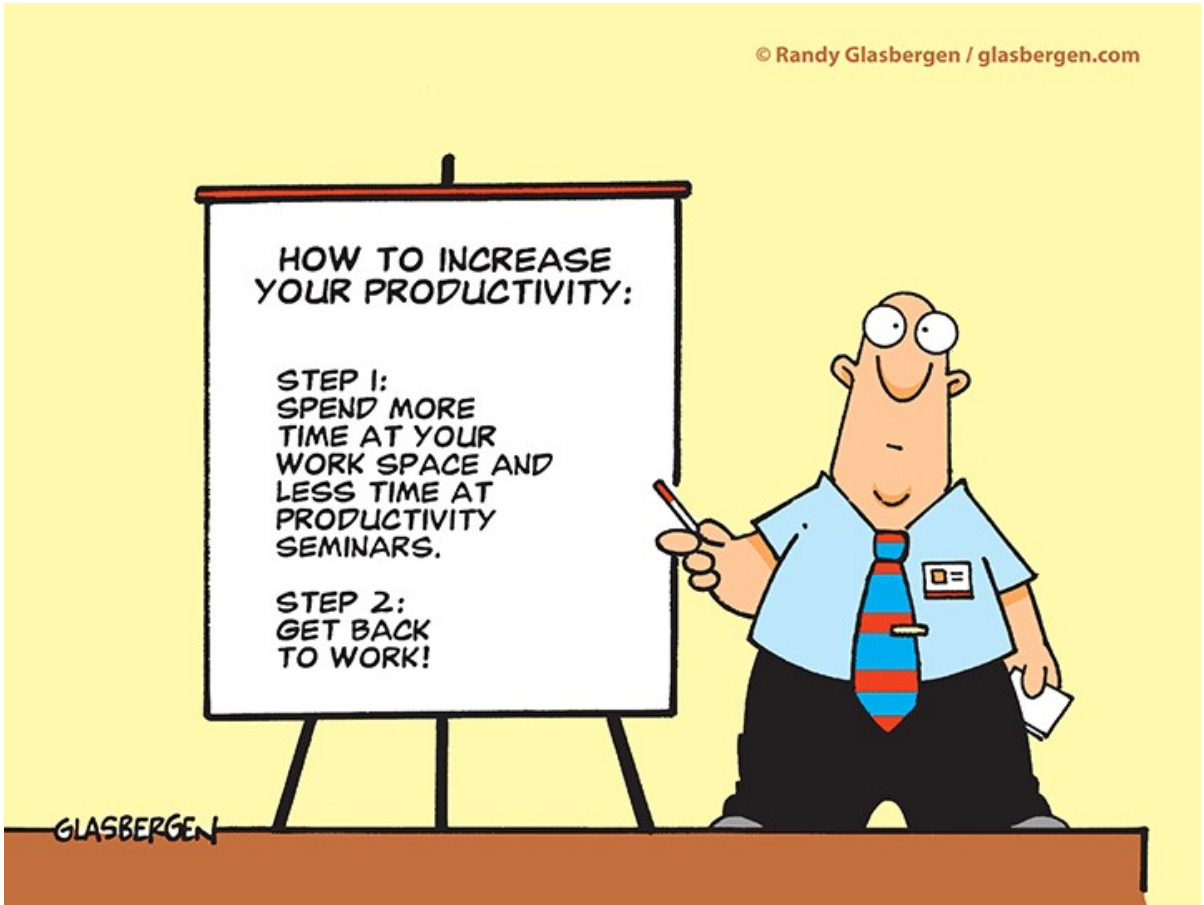
Kural #1: Tüm testleri çalıştırın .....	97
Kural #2: Tekrarlanmış kodlar yazmayın.....	98
Kural #3: Açıklayıcı olun .....	100
Kural #4: Sınıf ve metot sayısını en aza indirin.....	101

Bu kitap iyi programlamayı anlatıyor. Bir sürü kodlama örneği olacak. Bitirdiğimiz zaman ise iyi kod ve kötü kod arasındaki farkı anlayabileceğiz. Nasıl iyi kod yazabileceğimizi ve kötü yazılmış bir kodu iyi bir koda nasıl dönüştürebileceğimizi öğreneceğiz.

20 yıl sonra şirketin ilk çalışanlarından biri ile karşılaştım ve ona ne olduğunu sordum. Cevabı korkularımı doğruladı. Ürünü markete erkenden sürebilmek için çok acele etmiş ve kodda çok büyük bir kargaşaya sebep olmuşlardı. Daha fazla özellik ekledikçe, kod daha da kötü bir hal almış ve o kadar kötü hale gelmişti ki, artık kodu yönetemiyorlardı. Böylece kötü kod şirketin kapanmasına sebep olmuştu.

Hepimiz zaman zaman geri dönüp kodumuzu temize çekeceğimizi söylemişizdir. Ancak o zamanlar LeBlanc'ın şu kuralını bilmiyorduk: [“Sonra asla demektir \(Later equals never\).”](#)

Kod karmaşıklığı arttıkça takımların verimliliği düşer ve sıfıra yaklaşır. Verimlilik düştükçe de yöneticiler yapabildikleri tek şeyi yaparlar; verimliliği artırması umudu ile projeye daha çok insan kaynağı eklerler. (*İnsan kaynak mıdır yoksa değer mi?*) Takımdaki herkes verimliliği artırmak için büyük baskı altındadır. Öyle ki verimliliği sıfıra daha da yaklaştıracak şekilde kod karmaşası yaratmaya devam ederler.



## Peki Koda Ne Oldu?

Ne oldu da iyi kod bu denli bir hızla kötü koda dönüştü? Bunun için bir çok sebep sıralayabiliriz. Gereksinimlerin (requirements) çok fazla değiştiğinden şikayet edebiliriz. Teslim tarihlerinin (deadline) çok sıkı olduğundan da yakınabiliriz. Beceriksiz yöneticilere ya da hoşgörüsüz müşterilere de püskürebiliriz. Ancak hata tamamen bizde. Bizler profesyonel değiliz!

Kabul etmesi zor, hata nasıl bizde olabilir? Diğerleri, onların hiç suçu yok mu? Hayır. Yöneticiler taahhüt vermek için bizden birşeyler duymayı beklerler. Beklemedikleri zaman bile onlara ne düşündüğümüzü söylemekten kaçınmamalıyız. Proje yöneticileri de zamanlama için bizden birşeyler duymayı beklerler. Bu yüzden, proje planlaması ve başarısızlıklar konusunda epey suçluyuz!

“Eğer yapamazsam kovulurum!” diyorsun fakat büyük ihtimalle kovulmayacaksın. Çoğu yönetici istiyormuş gibi yapmasa bile gerçeği ister. Çoğu yönetici iyi kod ister, zamanlama konusunda takıntılı olduklarında bile. Zamanı ve gereksinimleri tutkuyla savunabilirler, ancak bu onların işidir. Senin için ise aynı tutku ile kodunu savunmaktır.

Temiz kod yazabilmek, temizlik (cleanliness) duygusuyla uygulanmış sayısız küçük teknik yöntemlerin disiplinli bir şekilde kullanımını gerektirir. Bazılarımız bu duyguya doğuştan sahiptirler. Bazılarımız ise elde edebilmek için savaşmak zorundadırlar. Bu duygu sadece iyi ya da kötü kodu ayırt etmemizi sağlamaz, aynı zamanda kötü kodu temiz koda (clean code) dönüştürebileceğimiz stratejiyi de bize gösterir. Bu duygudan yoksun bir yazılımcı karmaşık bir modüle baktığında karmaşıklığı tanır ancak onunla ne yapacağı hakkında en ufak bir fikri yoktur. Bu duyguya sahip bir yazılımcı ise bu karmaşık koda bakar ve seçenekleri görür.

## Temiz Kod Nedir?

Yapılmış birçok tanımı var. Ben de çok iyi bilinen bazı yazılımcılara ne düşündüklerini sordum:

**Bjarne Stroustrup (C++'ın mucidi):** Kodumun şık ve temiz olmasını seviyorum. Kodda mantık, hataların saklanması zorlayacak kadar düz; bağımlılıklar (dependency) bakımı kolaylaştıracak kadar minimal olmalı. Tüm istisnai durumlar (exceptions) ele alınmalı, performans optimale yakın olmalı.

**Grady Booch (Object Oriented Analysis and Design with Applications kitabının yazarı):** Temiz kod basit ve açıktır. Temiz kod, iyi yazılmış bir düzyazı gibidir. Temiz kod, asla tasarımcının niyetini gizlemez, daha çok berrak soyutlamalarla ve düz kontrol satırlarıyla doludur.

**Dave Thomas (OTI Labs'ın kurucusu):** Temiz kod, onu geliştiren yazılımcı dışında başka geliştiriciler tarafından da okunabilir ve iyileştirilebilir. Birim ve kabul testleri vardır. Anlamlı isimlendirmeleri vardır. Bir şeyin yapılması için tek bir yol vardır. Çok az bağılılığı vardır ve temiz bir API sağlar.



**Michael Feathers (Working Effectively with Legacy Code kitabının yazarı):** *Temiz kod için bildiğim birçok özelliği sıralayabilirim; ancak bir tanesi diğer tüm özellikleri kapsıyor. Temiz kod her zaman ona değer veren biri tarafından yazılmış gibi görünür.*

**Peki Robert C. Martin olarak ben ne düşünüyorum?** Bu kitap size tam da bunu anlatacak; bir değişken, sınıf ya da metod adının temiz olabilmesi için ne düşündüğümü yazacağım. Elbette bu kitaptaki önermelerin çoğu tartışmaya açık. Büyük ihtimalle bazılarını katılmayacaksınız, bazılarını şiddetle karşı çıkacaksınız. Sorun değil. Sadece şunu bilmelisiniz ki, bu yöntemleri onlarca yıllık tecrübeler sonucunda, birçok deneme ve yanılmamalarla öğrendim.

## Bölüm 2 - Anlamlı İsimlendirmelerle *Takım* Arkadaşlarınıza Küçük Sürprizler Yapın

İyi isim seçmek zaman alır, ancak uzun vadede daha çok zaman kazandırır.

Her şeye bir isim veririz; değişkenlerimize, fonksiyonlarımıza, argümanlarımıza, sınıflarımıza ve paketlerimize. O kadar çok isimlendirme yaparız ki, bunu iyi yapsak iyi olur aslında. :)

İlk örneğimizle başlayalım:

```
int d; //elapsed time in days (gün cinsinden geçen süre)
```

`d` burada hiçbir şeyi açıklamıyor, günlerle ya da zaman ile alakalı hiçbir şey uyandırmıyor. Daha anlamlı isimler seçmeliyiz, şunlar gibi:

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Peki aşağıdaki kodun amacı nedir? Ne yaptığını açıklamak neden bu kadar zor? Karışık işlemler yok, boşluklar ve girintiler (indentation) de mantıklı. Burada problem kodun basitliği değil, kapalı oluşu.

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

1. `theList` ne içeriyor?
2. `theList`'in ilk elemanın önemi nedir?
3. 4 değerinin önemi nedir?
4. Dönen `list1` listesini nasıl kullanmalıyım?

Bu soruların cevapları kodda değil, yazarın programcının zihninde. Ancak şöyle olsa:

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)
```

```

        flaggedCells.add(cell);
    return flaggedCells;
}

```

Kodun basitliğinin değişmediğini farkedin. Hala aynı sayıda işlemci ve sabit var. Ama kod daha açık hale geldi. Daha da ileri giderek, `int[]` dizisi kullanmak yerine, `cell`'leri içeren bir `Cell` sınıfı yapabiliriz.

Yaptığımız bu basit değişikliklerle kodda neler olduğunu anlamak hiç de zor değil.

Kötü isim örneklerinden biri de küçük l (Lüleburgaz'ın L'si) ya da büyük O kullanmaktır. Çünkü görünüşte bir ve sıfıra benzerler.

```

int a = 1;
if (0 == 1)
    a = 01;
else
    l = 01;

```

Yazılımcılar kod yazarken problemleri kendileri yaratırlar. Örneğin aynı ismi aynı kapsamdaki (scope) 2 farklı objeye veremediğimizden, birini değiştirme yoluna gideriz. Sayı eklemek yeterli olmadığında, bunu birinin bir harfini eksilterek yaparız. Ancak, isimlerin farklı olması gerekiyorsa anlamları da farklı olmalıdır.

`a1`, `a2`, `a3`, ... gibi isimlendirmeler kesinlikle anlamlı değildir. Bu tür isimler yazarın amacı hakkında en ufak bir ipucu bile vermezler. Burada `a1` ve `a2` yerine `source` (kaynak) ve `destination` (hedef) kullanılması çok daha anlamlıdır:

```

public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}

```

`Product` isimli bir sınıfınız olduğunu hayal edin. Bir `ProductInfo` ve `ProductData` sınıflarımız olsun. "Info" ve "Data" birbirine yakın çağrışımlı, dolayısıyla etkisiz kelimelerdir ve aynı kapsamda kullanılmaları belirsizlik yaratır. Örneğin yeni eklenecek bir alanın (field) hangisine ekleneceği tamamen belirsizdir.

Şu şekilde metotlarımız olsun:

```

getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();

```

Geliştiriciler bu metotlardan hangisini çağıracağını nasıl bilebilir?

Benzer durum değişkenlerde de karşımıza çıkabilir.

Örneğin `moneyAmount` değişkeni `money` değişkeninden, `customerInfo` `customer`'dan ya da `accountData` `account`'tan ayırt edilemez.

## Telafluz Edilebilir İsimler Kullanın

Eğer bir değişkenin adını telaffuz edemiyorsanız onu kullanmayın. Yakın zamanda karşılaştığım bir örnek: “yeterli” anlamında, `adequate` kelimesinin yerine, daha yaygın kullanılan ve dolayısıyla telaffuzu daha iyi bilinen `enough` kelimesi tercih edilmelidir.

## Aranabilir İsimler Kullanın

Tek harfli isimler ya da sayı sabitleri arama dostu değildir.

Örneğin `MAX_CLASSES_PER_STUDENT` sabitini çok rahatça bulabiliriz. Ancak 7'yi bulmak oldukça zahmetlidir. Aynı şekilde `e` de çok kötü bir seçimdir. İngilizce'deki en yaygın harftir ve neredeyse her yerde karşımıza çıkacaktır.

Kişisel tercihim, tek harfli isimleri yalnızca kısa metotlarda lokal değişken olarak kullanmaktır. Bir ismin uzunluğu, o değişkenin kapsamının genişliği ile eşdeğer olmalıdır. Eğer bir değişken ya da sabit bir kod bloğunda birden fazla kullanılacaksa, arama dostu bir isim vermek en iyisidir.

Şu iki kodu karşılaştıralım:

```
for (int j = 0; j < 34; j++) {
    s += (t[j] * 4) / 5;
}int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j = 0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

`sum` çok kullanışlı bir isim değil ancak en azından aranabilir. Benzer şekilde `WORK_DAYS_PER_WEEK` sabitinin kullanımlarını bulmak, 5 sayısını bulmaktan kat be kat daha kolaydır.

## Arayüzler ve Gerçekleştirimler (Interfaces - Implementations)

Bir arayüz ve onu gerçekleştirecek somut (concrete) bir sınıf yazacağınızı düşünün. Bu iki sınıfa ne isim verirdiniz? `IShapeFactory` ve `ShapeFactory` mi? Ben arayüz sınıflarını sade bırakmayı tercih ediyorum. Başa `I` harfi eklemek yaygın bir pratik, ancak dikkat dağıtmakta

bir numara ve bilgi vermek konusunda da sonuncu. Eğer arayüz ya da gerçekleştirim sınıflarından birini belirtmem gerekiyorsa, gerçekleştirim sınıfını seçiyorum; `ShapeFactoryImpl` şeklinde. Arayüz sınıfını belirtmekten çok daha iyi.

Sınıf isimleri `Customer`, `WikiPage`, `Account` ya da `AddressParser` gibi isimlerden/isim tamlamalarından oluşmalıdır. Sınıf isimleri asla bir fiil olmamalıdır.

Metot isimleri `postPayment()`, `deletePage()` ya da `save()` gibi fiillerden/fiil tamlamalarından oluşmalıdır. Erişimci (getters), mutator (setters) ya da doğrulayıcı (predicate) (true/false dönen; `isEmpty()` gibi) metotların başlarına, Java standartlarına göre `get`, `set`, `is` eklenmelidir.

```
String name = employee.getName();
customer.setName("Mike");
if (paycheck.isPosted())
...
```

Farklı parametrelere sahip kurucular (constructors) oluşturmak yerine, “static” yapıcı (builder) metotlar kullanılmalıdır.

İlk satırdaki kod örneği, ikincisinden çok daha iyidir:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
Complex fulcrumPoint = new Complex(23.0);
```

## Kelime Oyunu Yapmaktan Kaçının

İki farklı amaç için aynı kelimeyi kullanmaktan ya da aynı amaçlar için farklı kelimeleri kullanmaktan kaçının. Örneğin `Controller`, `Manager` ya da `Driver` kelimelerini aynı kapsamda farklı sınıflar için kullanmak iyi bir kullanım örneği değildir. Birini seçin ve onunla devam edin.

Örneğin birileri sizden önce `add` metodu yazmış olsun ve bu metot da iki değeri birbirine birleştiriyor (concat) olsun. Bizim de bir listeye değer ekleyen bir metota ihtiyacımız olsun. Bu metoda `add` mi demeliyiz? Hayır. Bu durumda yeni metodumuza `insert` ya da `append` demeliyiz. Yeni bir `add` metodu yazmak, kelime oyunu yapmaktır.

## Gereksiz Bağlamlardan Kaçının

`firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` ve `zipcode` ismi mli değişkenlerimiz olduğunu düşünelim. Birlikte alınca bir adresin detayları olduğunu çok çabuk anlayabiliyoruz. Ancak sadece `state` değişkenini görürsek, gene de adrese ait olduğunu düşünebilir miyiz?

Önekler (prefix) kullanarak bağlam (context)

sağlayabilirsiniz; `addrFirstName`, `addrLastName`, `addrState` vb. En azından okuyucular bu değişkenlerin daha büyük bir yapının parçası olduğunu anlayabileceklerdir. Elbette daha iyi bir çözüm `Address` isimli bir sınıf yaratmaktır.

Aşağıdaki örnekte ise metod oldukça uzun ve bir sürü değişkene sahiptir:

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
    String guessMessage = String.format(
        "There %s %s %s%s", verb, number, candidate,
pluralModifier);
    print(guessMessage);
}
```

Bu metodu daha küçük parçalara bölebilmek için `GuessStatisticsMessage` isimli bir sınıf oluşturmali ve içine üç adet değişken koymalıyız:

```
public class GuessStatisticsMessage {    private String number;
private String verb;    private String pluralModifier;

    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format(
            "There %s %s %s%s",
            verb, number, candidate, pluralModifier);
    }    private void createPluralDependentMessageParts(int
count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }
}
```

```

    }
}   private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
}   private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
}   private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
}}

```



Son örneğimize geçelim: Gas Station Deluxe isimli bir uygulamamız olsun. Bu uygulamada her sınıfın başına GSD öneki koymak kötü bir fikir. Örneğin GSD'nin hesap modülüne bir MailingAddres sınıfı eklediğinizi ve ismine GSDAccountAddres dediğinizi düşünelim. Daha sonra müşteri modülü için de bir MailingAddres sınıfına ihtiyaç duyduğunuzda GSDAccountAddres sınıfını kullanır mısınız? Doğru isim mi sizce?

Burada 10 karakter (GSDAccount) tamamen gereksizdir. Bu nedenle kısa isimler, açık ve net oldukları müddetçe uzun isimlerden her zaman daha iyidir.

İsimplendirme demişken...

## Bölüm 3 - Fonksiyonlar

Fonksiyonlar bir programın içindeki organizasyonun ilk satırlarıdır.

### REFACTORING IS KEY





Aşağıdaki koda bir bakalım. Sadece uzun değil, içinde tekrarlanmış (duplicate) kodlar, bir sürü ne olduğu belirsiz metin sabitleri ve net olmayan veri tipleri var. 3 dakika içinde metodu ne kadar anlayabileceğinize bir bakın:

```
public static String testableHtml(PageData pageData, boolean
includeSuiteSetup) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_SETUP_NAME,
wikiPage);
            if (suiteSetup != null) {
                WikiPagePath pagePath =
suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName =
PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName).append("\n");
            }
        }
        WikiPage setup =
PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName).append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath =
wikiPage.getPageCrawler().getFullPath(teardown);
            String tearDownPathName =
PathParser.render(tearDownPath);
            buffer.append("\n")
                .append("!include -teardown .")
                .append(tearDownPathName).append("\n");
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown =
PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME,
wikiPage);
```

```

        if (suiteTeardown != null) {
            WikiPagePath pagePath =
suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName =
PathParser.render(pagePath);
            buffer.append("!include -teardown .")
                .append(pagePathName).append("\n");
        }
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}

```

Muhtemelen 3 dakika içinde neler döndüğünü çözemediniz.

Bir kaç metot sadeleştirilmesi, bir kaç yeniden adlandırma ve yeniden yapılandırmadan (refactoring) sonra kodu şu şekle getirebildim:

```

public static String renderPageWithSetupsAndTeardowns(PageData
pageData, boolean isSuite) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}

```

Son dokunuşlardan sonra ise şu hale geldi:

```

public static String renderPageWithSetupsAndTeardowns(PageData
pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}

```

**Bir Metot Ne Kadar Kısa Olmalı?**

Fonksiyonların ilk kuralı küçük, kısa olmaları gerektiğidir. İkinci kural ise daha da kısa olmaları gerektiğidir.

80'lerde bir fonksiyonun tüm ekrandan daha büyük olmaması gerektiği gibi bir kural vardı. Bunu söylediğimiz zamanlarda VT100 ekranlarımız 80 kolona 24 satırdı.



Eski bir VT100 ekran

Bugünlerde normal bir yazı tipiyle (font) ve güzel büyük bir monitörle bir satıra 150 karakter ve bir ekrana 100 satırdan fazlasını sığdırabilirsiniz. Satırlar 150 karakter, fonksiyonlar da 100 satır filan olmamalı.

Aslına bakarsanız, fonksiyonlar 20 satırı zar zor bulmalı.

1991'de [Kent Beck](#)'i evinde ziyarete gitmiştim. Oturduk ve birlikte biraz kod yazdık. Bir ara bana *Sparkle* adında küçük bir *Java/Swing* programı gösterdi. Her fonksiyon 2, 3 ya da 4 satır uzunluğundaydı. Her biri çok şeffaftı, her biri bir şeyler anlatıyordu. İşte fonksiyonların ne kadar kısa olmaları gerektiğinin cevabı buydu!

## Sadece Bir Şey Yapmalı

Yukarıdaki kodun ilk halinin birden fazla şey yaptığı aşikar. **Buffer**'lar oluşturuyor, sayfayı indiriyor ve **HTML** dokümanı oluşturuyor. Diğer taraftan, son hali ise sadece bir şey yapıyor. Çünkü fonksiyonlar bir şey yapmalı. Onu çok iyi yapmalı ve sadece onu yapmalı. Buradaki problem “bir şey”i tanımlamanın zor oluşu. Örnek kodumuzun son versiyonu sadece bir şey mi yapıyor?

```
public static String renderPageWithSetupsAndTeardowns (PageData
pageData, boolean isSuite) throws Exception {
    if (isTestPage (pageData))
        includeSetupAndTeardownPages (pageData, isSuite);
    return pageData.getHtml ();
}
```

Aslında şu 3 şeyi yaptığını söyleyebiliriz:

1. Sayfanın bir test sayfası olup olmadığına bakıyor.
2. Eğer öyleyse **includeSetupAndTeardownPages** metodunu çağırıyor.
3. **HTML** dokümanını oluşturuyor.

Peki hangisi? Metot sadece 1 iş mi yoksa 3 iş mi yapıyor? Metodu aşağıdaki şu tek cümle ile tanımlayabiliriz:

“Bir sayfanın test sayfası olup olmadığını kontrol ederiz ve eğer öyleyse **Setup** ve **Teardown** sayfalarını dahil ederiz. Her iki durumda da dokümanı oluştururuz.”

Eğer bir fonksiyon sadece isminde belirtilen adımları yapıyorsa, o fonksiyon bir şey yapıyordur.

Örnek kodumuzun ilk versiyonu kesinlikle birden fazla iş yapıyordu. İkinci versiyonunda bile 2 seviye soyutlama (two-level abstraction) yapıyordu, ki kodu küçülterek son haline getirebiliyor oluşumuz bile bunun kanıtı. Son halinde kodu daha fazla küçültemiyoruz. **if** ifadesini alıp yeni bir metot içerisine yazabilirdik. Ancak bu, koddaki soyutlama seviyesini değiştirmeden kodu yeniden yazmak olurdu. Fonksiyonlarımızın bir şeyi yaptığından emin olmak için fonksiyonumuzun içindeki ifadelerin fonksiyonumuzun adında ifade edilen ile aynı soyutlama seviyesinde olması gerekir.

## Switch İfadeleri

Kısa bir **switch** ifadesi yazmak zordur. Bir şey yapan **switch** de yazmak zordur. Doğaları gereği **switch** ifadeleri  $N$  tane şey yaparlar. Ve ne yazık ki, **switch** ifadelerinden her zaman kaçamıyoruz, ancak her **switch** ifadesinin alt seviye bir sınıfa gömüldüğünden ve tekrarlanmadığından emin olabiliriz. Elbette bunu polimorfizmle (çok biçimlilik) yaparız.

Şu koda bir bakalım, **Employee**'nin tipine bağlı işlemlerden sadece birini gösteriyor:

```
public Money calculatePay(Employee e) throws InvalidEmployeeType
{
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Bu fonksiyonla ilgili bir sürü problem var:

1. Çok büyük ve **Employee** tipleri eklendikçe de büyümeye devam edecek.
2. Kesinlikle birden fazla şey yapıyor.
3. Tek Sorumluluk Kuralı'nı ([SRP](#) — Single Responsibility Rule) ihlal ediyor.
4. Açık/Kapalı Prensibi'ni ([OCP](#) — Open/Closed Principle) de ihlal ediyor.

Bu kodun sorununun çözümü, **switch** ifadesini şu şekilde bir “Abstract Factory” tasarım deseni (design pattern) temeline gömmek ve başka hiç kimsenin görmesine izin vermemektir:

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}-----
--public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType;
}-----
--public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType {
        switch (r.type) {
```

```

        case COMMISSIONED:
            return new CommissionedEmployee(r);
        case HOURLY:
            return new HourlyEmployee(r);
        case SALARIED:
            return new SalariedEmployee(r);
        default:
            throw new InvalidEmployeeType(r.type);
    }
}

```

**EmployeeFactoryImpl**, **Employee** türevlerinin uygun örneklerini (instances) oluşturmak için switch ifadesini kullanacak ve **calculatePay**, **isPayday** ve **deliverPay** gibi çeşitli fonksiyonlar, **Employee** arayüzünden polimorfik olarak gönderilecektir.

Bence **switch** ifadeleri yalnızca şu durumlar için tolere edilebilirler:

- Yalnızca bir kez görüneceklerse
- Sistemin geri kalanının göremeyeceği şekilde bir arayüz ilişkisi arkasına gizleneceklerse
- Polimorfik nesneleri yaratmak için kullanılacaklarsa

#### Açıklayıcı İsimler Kullanın

Yukarıdaki örneğimizde **testableHtml** fonksiyonumuzun ismini **SetupTeardownIncluder.render** olarak değiştirdim. Bu isim çok daha iyi, çünkü fonksiyonun yaptığı işi daha iyi tanımlıyor. **private** metodlarımızın isimlerine de aynı denklkte tanımlayıcı isimler verdim; **isTestable** ya da **includeSetupAndTeardownPages** gibi...

Ward'ın prensibini hatırlayın: "Programın her parçası beklediğiniz gibi çıktığında, temiz kod üzerinde çalıştığınızı anlarsınız." Bu prensibi uygulamanın ilk adımı, tek bir şey yapan küçük fonksiyonlar için iyi isimler seçmektir. Küçük ve daha odaklı bir fonksiyon için açıklayıcı bir isim seçmek daha kolaydır.

Uzun isimler kullanmaktan kaçınmayın. Uzun tanımlayıcı isimler, kısa bilmece gibi isimlerden veya uzun bir yorumdan daha iyidir.

İsim seçmek için zaman harcamaktan korkmayın. Bir sürü farklı isim deneyin ve her biri ile kodu tekrar tekrar okuyun. Açıklayıcı isimler seçmek zihninizdeki tasarımı ortaya çıkaracak ve onu geliştirmenize yardım edecektir.

İsimlerinizde tutarlı olun. Fonksiyon isimlerinizde aynı isimleri, kelime gruplarını, sıfatları ve fiilleri kullanın.

## Argümanlar

Bir fonksiyon için ideal argüman sayısı sıfırdır. **StringBuffer** örneğimizi düşünelim. Onu bir örnek değişken (instance variable) olarak yazmak yerine argüman olarak da geçebilirdik. Ancak okuyucularımız değişkeni her gördüğünde tekrar tekrar yorumlamak zorunda kalacaklardı.

Argüman fonksiyon adından daha farklı bir soyutlama seviyesidir ve o noktada çok önemli olmayan bazı detayları bilmeye zorlar.

Argümanlar test açısından daha da zorlar. Hiç argüman yoksa önemsizdir. Bir tane varsa çok zor değildir. İki argüman biraz daha zorlayıcıdır ancak iki argüman sonrası yıldırıcı olabilir.

Bir fonksiyona tek bir argüman geçmenin iki yaygın sebebi vardır. Bir argüman ile bir soru soruyor olabilirsiniz; **boolean fileExists("My File")** gibi... Ya da belki bu argümanı işleme sokuyor, onu bir şeye dönüştürüyor ve **return** ediyorsunuzdur. Bu iki kullanım şekli kullanıcıların bir fonksiyonda görmek isteyebilecekleri kullanım şekilleridir.

Tek argümanın biraz daha az kullanım şekli ise **event**'tir. Bu kullanım şeklinde girdi var ancak çıktı yoktur. Tüm program bu fonksiyonun çağrısını bir **event** (olay) olarak yorumlar ve argümanı sistemin durumunu değiştirmek için kullanır. Örneğin; **void passwordAttemptFailedNtimes(int attempts)**. Bu kullanımda oldukça dikkatli olunmalıdır ve okuyucuya bunun bir **event** olduğu açıkça belli edilmelidir. İsimler ve bağlamlar dikkatli seçilmelidir.

## Boolean Argümanlar

**Boolean** argümanlar çirkindir. Fonksiyonlara parametre olarak geçmek ise korkunç bir pratiktir. Metodun imzasını karmaşıklaştırır ve "Bu fonksiyon birden fazla şey yapıyor." diye bağırır. Argüman **true** ise bir şey yap, **false** ise başka bir şey yap.

## Nesne Argümanlar

Eğer bir fonksiyon 2 ya da 3 argümandan fazlasına ihtiyaç duyuyorsa, bu argümanlardan birkaçı bir sınıf ile sarmalanmalıdır.

Şu iki örnek çağırma bakalım. Nesneler yaratarak argümanların sayısını azaltmak hile yapmak gibi görünebilir ama değildir:

```
Circle makeCircle(double x, double y, double radius);  
Circle makeCircle(Point center, double radius);
```

## Fiiller ve İsimler

Bir fonksiyon için iyi isim seçmek, fonksiyonun ve değişkenlerin niyetlerini açıklamak konusunda güzel bir başlangıç olabilir. Tek argümanlı (Monad) durumunda, fonksiyon ve argüman çok iyi bir fiil ve isim ikilisi olacak şekilde seçilmelidir. Örneğin **write(name)** imzasına sahip bir fonksiyon oldukça açıklayıcıdır. Hatta **writeField(name)** bize **name**'in bir alan (field) olduğunu ve bir yerlere yazılacağını söyler.

İkinci bir örnek olarak **assertEquals** metodu, **assertExpectedEqualsActual (expected, actual)** olarak yazılsaydı çok daha iyi olabilirdi. Bu yazım şekli, argümanların sırasını hatırlamanın zorunluluğunu ortadan kaldırır.

## Yan Etkiler Oluşturma (Side Effects)

Yan etkiler bizi kandırır. Fonksiyonunuz bir şeyi yapmaya söz verir ama yanında başka gizli şeyler de yapar. Bazen kendi sınıfının değişkenlerinde beklenmedik değişiklikler yapar. Bazen bunu fonksiyonlara geçilen parametreler ile ya da globaller ile yapar.

Şu zararsız koda bir bakalım:

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public boolean checkPassword(String userName, String  
password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase =  
user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase,  
password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
    }  
}
```



```
        return false;
    }
}
```

Bu fonksiyon bir **userName**'i bir **password**'e eşlemek için standart bir algoritma kullanıyor. Eşleşirse **true**, eşleşmezse **false** dönüyor. Ancak bir yan etkisi var, o da **Session.initialize()** çağrısıdır. **checkPassword**, isminde söylendiği gibi şifreyi kontrol eder. Ancak **Session**'ın **initialize** edileceğini söylemiyor. Bu nedenle bu fonksiyonu çağıran, mevcut oturum verisini silme riski taşıyor.

Fonksiyonlar ya bir şeyler yaparlar ya da bir şeylere cevap verirler; ancak ikisini birden yapmazlar. Ya bir nesnenin yapısını değiştirirler ya da o nesne hakkında bilgi dönerler.

Şu örneği inceleyelim:

```
public boolean set(String attribute, String value);
```

Bu fonksiyon **attribute** isimli bir değişkene bir **value** atar ve başarılı ise **true** döner ya da **attribute** diye bir değişken yoksa **false** döner. Bu, şu tür tuhaf durumlara sebep olur:

```
if (set("username", "unclebob"))...
```

Okuyucunun açısından bir düşünelim. Ne anlama geliyor? **username**'in daha öncesinde **unclebob** olarak set edilip edilmediğini mi sorguluyor yoksa **username**'in başarılı bir şekilde **unclebob** olarak set edilip edilmediğini mi Anlamı buradan çıkarmak çok zor, çünkü net değil. Yazar fiil gibi kullanmak istemiş ancak **if** içerisinde bir sıfatmış hissi veriyor.

Bu problemi, fonksiyon ismini **setAndCheckIfExists** şeklinde yeniden düzenleyerek çözebiliriz. Ancak bu da **if**'in okunabilirliği için çok da yardımcı olmuyor. **if**'in okunabilirlik problemini de kodu şu şekilde düzenleyerek halledebiliriz:

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob"); ...
}
```

try/catch Bloklarını Ayırın

**try/catch** blokları doğaları gereği çirkindir. Normal işleyişi ve kodun yapısını karıştırır. Bu nedenle **try** ve **catch** bloklarının gövdelerini kendi fonksiyonlarına ayırmak daha iyidir.

```

public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    } catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws
Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}

```

Bu örnekte **delete** fonksiyonu tamamen hata işleme (error processing) ile ilgilidir. Anlaması kolaydır. **deletePageAndAllReferences** fonksiyonu ise tamamen bir sayfayı silmenin süreçleri ile alakalıdır. Hata işleme yok sayılabilir. Bu kullanım, kodun daha kolay anlaşılması ve düzenlenebilmesi için güzel bir ayrım sağlar.

Fonksiyonlar bir şeyi yapmalıdır, hata işleme ise başlı başına bir şeydir. Bu yüzden hataları işleyen bir metod başka bir şey yapmamalıdır.

Error.java

```

public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}

```

Bunun gibi sınıflar bağımlılık magneti gibidirler; diğer çoğu sınıf da **import** (dahil etmek/ithal etmek) etmeli ve kullanılmalıdır. **Error enum**'ı değiştiği zaman, diğer tüm sınıflar da tekrar derlenmeli ve tekrar **deploy** (dağıtım) edilmelidir. Yazılımcılar yeniden **build** (inşa) etmek ve her şeyi yeniden **deploy** etmek istemediklerinden yeni hata kodları eklemeyiz; mevcut olanlardan kullanırlar.

Hata kodlarındansa, türetilmiş **Error** sınıfları kullanmak daha iyidir. Hiçbir yeri yeniden derlenmeye, **deploy** etmeye zorlamadan eklenip çıkartılabilirler. (bkz: OCP)

Bazı yazılımcılar [Edsger Dijkstra](#)'nın yapısal programlama kurallarını takip ederler. Dijkstra her fonksiyonun bir giriş ve bir çıkışı olması gerektiğini söyler. Yani, bir fonksiyonun sadece bir **return** ifadesi olmalı, bir döngüde **break**, **continue** ya da **goto** ifadeleri asla ve asla olmamalıdır. Fonksiyonlar küçük olduğunda, bu kurallar küçük faydalar sağlar. Sadece büyük metotlarda bu ifadeler büyük faydalar getirir. Yani fonksiyonlarınızı küçük tutarsanız, birden fazla **return**, **continue** ya da **break** ifadesi kodunuza zarar veremez; hatta tek giriş - tek çıkış kuralından (single-entry, single-exit) bile daha açıklayıcı olabilir. Diğer bir taraftan **goto** ifadesi sadece çok büyük fonksiyonlarda bir anlam ifade eder bu nedenle **goto** kullanımından kaçınılmalıdır.

Peki, Böyle Metotları Nasıl Yazabilirsiniz?

Yazılım yapmak, yazmanın bir diğer türevidir. Bir makale hakkında bir şeyler yazdığınızda; önce düşüncelerinizi toplarsınız, daha sonra kulağa iyi gelene kadar düzenleme yaparsınız. İlk taslak acemi ve dağınık olabilir.

Ben kod yazıyorken fonksiyonlarımın ilk halleri uzun ve karışık olur. Bir sürü girintilemeler ve iç içe döngüleri olur. Uzun argüman listeleri olur. Verdiğim isimler keyfidir ve tekrarlanmış (duplicate) kodlar vardır. Ancak bu acemi satırların her birini kapsayan testlerim de vardır.

Daha sonra kodu fonksiyonlara bölerek, isimleri değiştirerek, tekrarlanmış kodları çıkartarak (extract) rafine ederim. Metotları küçültürüm ve tekrar sıraya koyarım. Sonunda fonksiyonlarım, anlattığım tüm bu kurallara uygun olurlar.

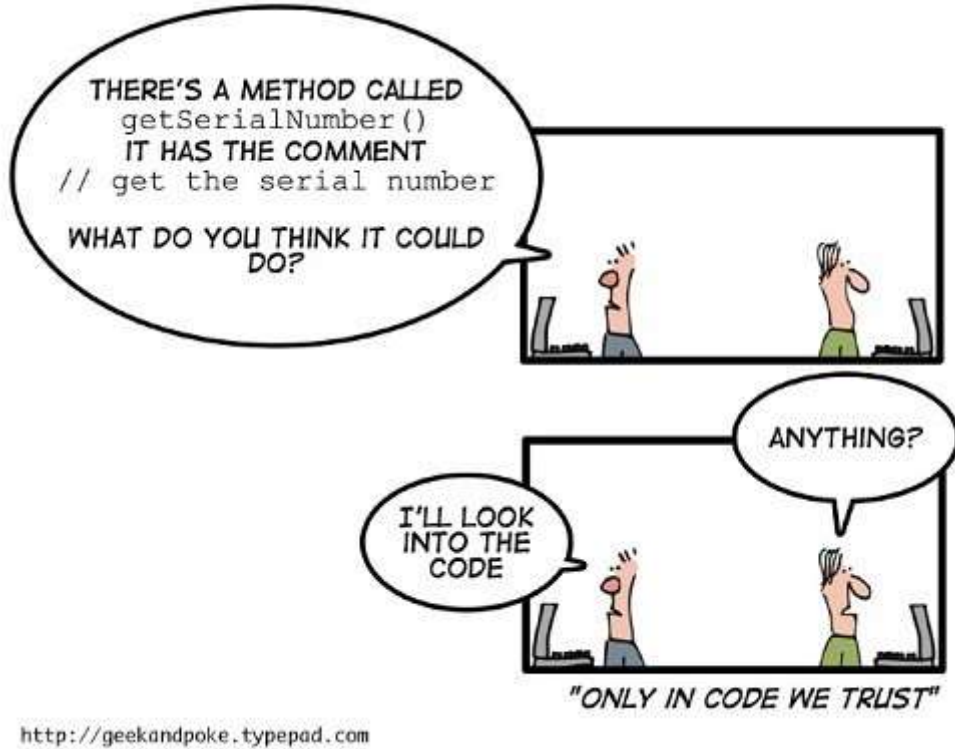
## Bölüm 4 - Yorum Yok

Kötü koda yorum yazmayın, onu yeniden yazın.

Yorumları yazmamızın amacı, kodda kendimizi iyi ifade edemediğimiz noktaları telafi etmektir. Yorumlar her zaman koddaki kusurlarımızdır, çünkü kendimizi onlarsız nasıl ifade edebileceğimizi bilmiyoruz.

Bu yüzden eğer kendimizi yorum yazarken buluyorsak, durup tekrar düşünelim. Kendimizi açıklayabileceğimiz başka yollar olup olmadığına bakalım. Yorumlara neden bu kadar kötü bakıyorum? Çünkü yorumlar yalan söylerler. Her zaman ve bilerek değil, ancak çoğunlukla söylerler.

Bir yorum ne kadar eski olursa, yazıldığı koddan bir o kadar uzak ve bir o kadar yanlış oluyor. Sebebi basit, yazılımcılar yorumları güncellemezler.



Örneğin şu yoruma ve ifade etmek istediği satıra bakalım:

```
MockRequest request;  
private final String HTTP_DATE_REGEX =  
    "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s" +  
    "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";  
private Response response;  
private FitNesseContext context;
```

```
private FileResponder responder;  
private Locale saveLocale;  
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Açıklayıcı satır ve `HTTP_DATE_REGEXP` sabiti arasına muhtemelen daha sonradan değişkenler sokulmuş. Ancak yorum satırı güncellenmeden orada kalmış.

Yanlış yorumlar, hiç yorum olmamasından da kötüdür. Aldatıcılar ve yanlış yönlendiriyorlar. Asla başarılamayacak şeyleri vadediyorlar. Gerçek sadece bir yerdedir, koddadır. Sadece kod size gerçekte ne yaptığını anlatabilir. Sadece kod gerçek ve doğru bilginin kaynağıdır.

## Yorumlarla Kötü Kodu Süslemeyin

Yorum yazmanın en yaygın motivasyonlarından biri de kötü koddur. Bir modül yazar ve onun kafa karıştırıcı, dağınık olduğunu biliriz. Bu yüzden kendimize *“Yorum yazsam iyi olacak.”* deriz. Aslında, temiz yazsak iyi olacak.

Bir kaç yorumu olan temiz ve açıklayıcı kod, bir sürü yorumu olan karmaşık ve darmadağın bir koddan daha iyidir. Yarattığın karmaşayı açıklamak için yazacağın yoruma harcadığın enerjiyi, bu karmaşayı temizlemek için harcamalısın.

Örneğin bu mu:

```
// Check to see if the employee is eligible for full benefits  
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

Yoksa şu mu:

```
if (employee.isEligibleForFullBenefits())
```

Koddaki asıl niyeti açıklayabilmek sadece bir kaç saniye alıyor, ancak çoğu durumda, yazmak istediğin yorumdaki aynı şeyi söyleyebilen bir metot yazmak meseledir.

Gerçekten iyi bir yorum, onu yazmamanın bir yolunu bulduğunuz yorumdur.

## İyi Yorumlar

Bazı yorumlar zorunludur ve faydalıdır.

## Yasal Yorumlar

Bazen bizim kurumsal kodlama standartlarımız yasal sebeplerden ötürü bizi kesin yorumlar yazmaya zorlar. Örneğin telif hakkı (copyright) ve yazarlık (authorship) durumları gereklidir ve her kaynak dosyanın başına böyle bir yorum koymak mantıklıdır.

Burada örneğin, zamanında çalıştığım bir projede her kaynak dosyasının başına koyduğumuz standart bir yorum başlığı var:

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.  
// Released under the terms of the GNU General Public License  
version 2 or later.
```

Tüm şartlar ve koşulları yorum içine koymaktansa, mümkünse standart bir lisans ya da dış bir dokümana referans tercih edilmelidir.

## Bilgilendirici Yorumlar

Şu yoruma bakalım; `abstract` bir metodun dönüş değerini açıklıyor. Fonksiyon ismini `responderBeingTested` şeklinde değiştirerek yorumu gereksiz hale getirebilirdik:

```
// Returns an instance of the Responder being tested.  
protected abstract Responder responderInstance();
```

Daha iyi bir örnek:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w*  
\\d*, \\d*");
```

Yorum yazmak yerine eğer kod, tarihlerin ve zamanların formatını dönüştüren bir sınıfa taşındaydı çok daha açık ve temiz olabilirdi.

Bazen diğer programcılar kesin sonuçlar hakkında uyarmak iyi olabilir. Örneğin burada, belli bir test metodunun neden kapatıldığını açıklayan bir yorum var:

```
// Don't run unless you have some time to kill.  
public void _testWithReallyBigFile() {  
    writeLinesToFile(10000000);  
    response.setBody(testFile);  
    response.readyToSend(this);  
    String responseString = output.toString();  
    assertSubString("Content-Length: 1000000000",  
responseString);  
    assertTrue(bytesSent > 1000000000);  
}
```

Bugünlerde bir testi, açıklama da ekleyerek `@Ignore` ile kapatabiliriz. Ancak JUnit 4'ten öncesinde, metodun başına bir `_` koymak yaygın bir adetti.

## TODO Yorumları

Kodda yapmak istediklerimizi `//TODO` şeklinde yorum olarak yazabiliriz.

Sıradaki örnekte `TODO` yorumu, fonksiyonun neden bozuk bir gerçekleştirimi olduğunu ve fonksiyonun gelecekte nasıl olması gerektiğini söylüyor:

```
//TODO-MdM these are not needed  
// We expect this to go away when we do the checkout model  
protected VersionInfo makeVersion() throws Exception {  
    return null;  
}
```

**TODO**'lar programcıların yapmaları gereken ancak o anda bazı sebeplerden ötürü yapmadıkları işler içindir. Kullanımdan kaldırılan (deprecated) bir özelliği silmek için hatırlatıcı olabilir, birilerinin daha iyi bir isim düşünmesini isteyebiliriz ya da başka birilerinin probleme göz atması için bir rica olabilir. **TODO** ne olursa olsun, sistemde kötü kod bırakmak için bahane değildir.

## Javadoc'lar

İyi tanımlanmış bir API kadar yardımcı olan başka bir şey yoktur. Eğer başkalarının kullanımına sunacağınız bir API yazıyorsanız, kesinlikle iyi **Javadoc**'lar da yazmalısınız. Ancak **Javadoc**'lar da diğer tüm yorumlar gibi yanıltıcı ve aldatıcı olabilir.

## Kötü Yorumlar

Çoğu yorum bu kategoriye girer. Eğer gerçekten yorum yazmaya karar verdiyseniz, zamanınızı yazabileceğiniz en iyi yorumu yazmak için harcadığınızdan emin olun.

Zamanında çalıştığım bir projeden bulduğum bir örnek, bir yorumun faydalı olabildiği bir örnek. Ancak yazarın acelesi varmış ya da çok dikkatini vermemiş:

```
public void loadProperties() {  
    try {  
        String propertiesPath = propertiesLocation + "/" +  
        PROPERTIES_FILE;  
        FileInputStream propertiesStream = new  
        FileInputStream(propertiesPath);  
        loadedProperties.load(propertiesStream);  
    } catch (IOException e) {  
        // No properties files means all defaults are loaded  
    }  
}
```

`catch` bloğundaki yorum ne demek? Yazara bir şey ifade ettiği belli. Eğer bir `IOException` alırsak, bu demektir ki `*.properties` dosyası yok ve bu durumda tüm varsayılanlar (default) yüklenmiş.

Peki tüm varsayılanları kim yükledi? `loadProperties.load` çağrılmadan önce de yüklenmişler miydi? Ya da `loadProperties.load` istisnayı yakaladı, varsayılanları yükledi ve bizim için görmezden mi geldi? Ya da `loadProperties.load`, dosyayı yüklemeye başlamadan önce varsayılanları yükledi mi? Yoksa yazar `catch` bloğunu boş bıraktığı için kendini mi rahatlatmaya çalışıyordu? Ya da en kötüsü, yazar buraya tekrar geleceğini ve varsayılanları yükleyen kodu yazacağını söyleyen notu mu bıraktı kendine?

Bize yardım edebilecek tek şey, bu kodu test etmektir.

Ne anlama geldiğini anlamamız için diğer modüllere bakmanızı gerektiren hiçbir yorum, tükettiği bitlere değmeyecektir.

## Gereksiz Yorumlar

Aşağıdaki örnekte kodun kendisini okumak, yorumu okumaktan daha kısa sürecektir:

```
// Utility method that returns when this.closed is true.  
// Throws an exception if the timeout is reached.  
public synchronized void waitForClose(final long timeoutMillis)  
    throws Exception {  
    if (!closed) {  
        wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("MockResponseSender could not be  
closed");  
    }  
}
```

Burada ise **Apache Tomcat**'in kaynak kodlarından alınmış, tamamıyla gereksiz ve faydasız bir yorum örneği var. Bu yorumlar sadece karmaşaya ve belirsizliğe sebep olur, belgelemeye ise hiçbir faydası olmaz:

```
public abstract class ContainerBase implements Container,  
Lifecycle, Pipeline, MBeanRegistration, Serializable {  
    * The processor delay for this component.  
    */  
    protected int backgroundProcessorDelay = -1; /**  
    * The lifecycle event support for this component.  
    */  
    protected LifecycleSupport lifecycle = new  
LifecycleSupport(this); /**  
    * The container event listeners for this Container.
```



```
*/  
protected ArrayList listeners = new ArrayList();.....
```

## Zorunlu Yorumlar

Her fonksiyonun bir **Javadoc**'u ya da her değişkenin bir yorumu olması gerektiğini söyleyen kurallar saçmadır. Örneğin şu koddaki yorumlar mevcut koda hiçbir şey katmaz:

```
/**  
 * @param title           The title of the CD  
 * @param author          The author of the CD  
 * @param tracks           The number of tracks on the CD  
 * @param durationInMinutes The duration of the CD in minutes  
 */  
public void addCD(String title, String author,  
                  int tracks, int durationInMinutes) {  
    CD cd = new CD();  
    cd.title = title;  
    cd.author = author;  
    cd.tracks = tracks;  
    cd.duration = duration;  
    cdList.add(cd);  
}
```

## Günlük Gibi Yazılan Yorumlar

Bazı kişiler bir modülü her düzenlemeye başladıklarında, başladıkları tarihi yorum olarak eklerler. Bu yorumlar bir günlük gibi birikir:

```
* Changes (from 11-Oct-2001)  
* -----  
* 11-Oct-2001 : Re-organised the class and moved it to new package  
*               com.jrefinery.date (DG);  
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate  
*               class (DG);  
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate  
*               class is gone (DG); Changed getPreviousDayOfWeek(),  
*               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct  
*               bugs (DG);  
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);  
* 29-May-2002 : Moved the month constants into a separate interface  
*               (MonthConstants) (DG);  
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);  
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);  
* 13-Mar-2003 : Implemented Serializable (DG);  
* 29-May-2003 : Fixed bug in addMonths method (DG);  
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);  
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Uzun bir zaman önce bu logları yaratmak için iyi bir sebebimiz vardı; çünkü bunu bizim için yapabilecek **kaynak kod kontrol sistemlerimiz** (source kod management systems) yoktu. Ancak bugünlerde bu yorumlar sadece kalabalık yapıyorlar.

Aşağıdaki ilk yorum yerinde görünüyor, `catch` bloğunun neden yok sayıldığını açıklıyor. Ancak ikinci yorum tamamen etkisiz. Görünen o ki programcı bu fonksiyonda `try/catch` yazmaktan o kadar yılmış ki, ara vermeye ihtiyaç duymuş :-)

```
private void startSending() {
    try {
        doSending();
    } catch (SocketException e) {
        // normal. someone stopped the request.
    } catch (Exception e) {
        try {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        } catch (Exception e1) {
            //Give me a break!
        }
    }
}
```

Değersiz ve etkisiz bir yorum yazmaktansa, programcı problemi kodun yapısını geliştirerek çözebileceğinin farkına varabilmeliydi. Şu şekilde ***try/catch*** bloğunu ayrı metoda çıkarmalıydı:

```
private void startSending() {
    try {
        doSending();
    } catch (SocketException e) {
        // normal. someone stopped the request.
    } catch (Exception e) {
        addExceptionAndCloseResponse(e);
    }
}

private void addExceptionAndCloseResponse(Exception e) {
    try {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    } catch (Exception e1) {
        [ ]
        [SEP]
    }
}
```

**Javadoc**'lar da etkisiz olabilir. Aşağıdaki **Javadoc**'lar çok iyi bilinen açık kaynak bir kütüphaneden alındı:

```

/**
 * The name.
 */
private String name;/**
 * The version.
 */
private String version;/**
 * The licenceName.
 */
private String licenceName;/**
 * The version.
 */
private String info;

```

Yorumlardaki kes-yapıştır hatasını görebiliyor musunuz? Eğer yazar, yorumlar yazılırken (ya da yapıştırılırken), dikkatini vermiyorsa, okuyucunun bundan fayda sağlaması nasıl beklenebilir?

## Bir Fonksiyon ya da Değişken Kullanabilecekken Yorum Yazmayın

Aşağıdaki koda bir bakalım:

```

// does the module from the global list <mod> depend on the
// subsystem we are part of?
if
(smodule.getDependSubsystems().contains(subSysMod.getSubSystem())

```

Yorum olmadan şu şekilde olabilirdi:

```

ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))

```

Yazar benim yaptığım gibi kodu yeniden yapılandırmalıydı. Böylece yorum kaldırılabilirdi.

## Kapama Parantezlerine Koyduğumuz Yorumlar

Bazen yazılımcılar kapama parantezlerine özel yorumlar eklerler, aşağıdaki örnekte olduğu gibi. Uzun fonksiyonlarda bir anlam ifade etse de, küçük fonksiyonlarda sadece kalabalık sağlar. Bu nedenle, eğer bir kapama parantezine yorum ekliyorsanız, bunun yerine fonksiyonunuzu kısaltmayı denemelisiniz:

```

public class WC {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));

```

```

String line;
int lineCount = 0;
int charCount = 0;
int wordCount = 0;
try {
    while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
    } //while
    System.out.println("wordCount = " + wordCount);
    System.out.println("lineCount = " + lineCount);
    System.out.println("charCount = " + charCount);
} // try
catch (IOException e) {
    System.err.println("Error:" + e.getMessage());
} //catch
} //main
}

```

## Kapatılmış Kodlar

Aşağıdaki şu kodu görenler onu silmeye cesaret edemeyecek, orada bir sebeple durduğunu ve çok önemli olduğunu düşüneceklerdir:

```

InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));

```

**Apache Commons**'tan alınan şu örneğe bakalım:

```

this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if(writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
} else {
    this.pngBytes = null;
}
return this.pngBytes;

```

Neden o iki satır kapalı? Önemliler mi? Ya da sadece birileri yıllar önce kapatmış ve temizlemeye değer mi görmemişler?

Kodları kapatmanın önemli olduğu zamanlar vardı, 60'lar. Ancak şimdi versiyon kontrol sistemlerimiz (SVN, Git vb.) var ve bu sistemler kodu bizim için akıllarında tutuyorlar.

## Yersiz Bilgi

Eğer bir yorum yazmanız gerekiyorsa, onun yakınında olduğu kodu tanımladığından emin olun. Kısmi bir yorum içerisinde, tüm sisteme ait bir bilgi vermeyin.

Aşağıdaki **Javadoc** yorumuna bakalım örneğin:

```
/**
 * Port on which fitnessse would run. Defaults to <b>8082</b>.
 *
 * @param fitnesssePort
 */
public void setFitnesssePort(int fitnesssePort) {
    this.fitnesssePort = fitnesssePort;
}
```

Yorum kesinlikle fonksiyonu tanımlamıyor. Gereksiz bir bilgiyi, varsayılan port ayarını bize söylüyor. Ve elbette ayar değiştiğinde, bu yorumun da değişeceğinin hiçbir garantisi yok.

Bir yorum ve tanımladığı kod arasındaki bağlantı net olmalıdır. Eğer bir yorum yazmak için sıkıntı çekecekseniz, en azından okuyucunun yorumu ve koda baktığında ne demek istediğini anlayabilmesini istersiniz:

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Bir yorumun amacı kendi kendisini açıklamakta yetersiz olan kodu açıklamaktır. Ancak burada yorumun kendisi de bir açıklamaya ihtiyaç duyuyor.

Aşağıdaki modülü ilk **XP Immersion** için yazmıştım. Nasıl kötü kod ve yorum yazılıra bir örnek gibi. Daha sonra **Kent Beck** bu kodu onlarca öğrencinin önünde çok daha hoş bir hale getirdi:

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
```

```

* d. c. 194, Alexandria. The first man to calculate the
* circumference of the Earth. Also known for working on
* calendars with leap years and ran the library at Alexandria.
* <p>
* The algorithm is quite simple. Given an array of integers
* starting at 2. Cross out all multiples of 2. Find the next
* uncrossed integer, and cross out all of its multiples.
* Repeat until you have passed the square root of the maximum
* value.
*
* @author Alphonse
* @version 13 Feb 2002 atp
*/
public class GeneratePrimes {

    /**
     *
     * @param maxValue is the generation limit.
     */
    public static int[] generatePrimes(int maxValue) {
        if (maxValue >= 2) {
            int s = maxValue + 1; // size of array
            boolean[] f = new boolean[s];
            int i;
            for (i = 0; i < s; i++)
                f[i] = true;
            f[0] = f[1] = false;
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++) {
                if (f[i]) { // if i is uncrossed, cross its
multiples.
                    for (j = 2 * i; j < s; j += i)
                        f[j] = false; // multiple is not prime
                }
            }

            // how many primes are there?
            int count = 0;
            for (i = 0; i < s; i++) {
                if (f[i])
                    count++; // bump count.
            }

            int[] primes = new int[count];

            // move the primes into the result

            for (i = 0, j = 0; i < s; i++) {
                if (f[i]) // if prime
                    primes[j++] = i;
            }
        }
    }
}

```

```

        return primes; // return the primes
    } else // maxValue < 2
        return new int[0]; // return null array if bad input.
    }
}

```

Aşağıdaki kod ise yeniden düzenlenmiş versiyonu. Yorumların kullanımının önemli ölçüde sınırlandırılmış olduğuna bakın; koca modülde sadece 2 tane yorum var:

```

/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until there are no more multiples
 * in the array.
 */
public class PrimeGenerator {

    private static boolean[] crossedOut;
    private static int[] result;

    public static int[] generatePrimes(int maxValue) {
        if (maxValue < 2) return new int[0];
        else {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }

    private static void uncrossIntegersUpTo(int maxValue) {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }

    private static void crossOutMultiples() {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i)) crossOutMultiplesOf(i);
    }

    private static int determineIterationLimit() {
        // Every multiple in the array has a prime factor that
        // is less than or equal to the root of the array size,
        // so we don't have to cross out multiples of numbers
        // larger than that root.
    }
}

```

```

        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }

    private static void crossOutMultiplesOf(int i) {
        for (int multiple = 2 * i;
            multiple < crossedOut.length; multiple += i)
            crossedOut[multiple] = true;
    }

    private static boolean notCrossed(int i) {
        return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult() {
        result = new int[numberOfUncrossedIntegers()];
        for (int j = 0, i = 2; i < crossedOut.length; i++)
            if (notCrossed(i)) result[j++] = i;
    }

    private static int numberOfUncrossedIntegers() {
        int count = 0;
        for (int i = 2; i < crossedOut.length; i++)
            if (notCrossed(i)) count++;
        return count;
    }
}

```

İlk yorumun gereksiz olup olmadığını tartışabiliriz, çünkü tıpkı fonksiyonu okur gibi yazılmış. Gene de okuyucunun algoritmayı okumasına yardım edeceğini düşünüyorum, bu nedenle yorumu tutmakta kararlıyım.

İkincisi ise hemen hemen zorunlu bir açıklama. Karekökün döngü limiti gibi kullanılmasının ardındaki gerekçeyi açıklıyor. Ne daha basit bir değişken adı ne de bu kadar temiz bir kodlama stili bulamazdım.

Kitabın bu bölümünden sizlerle paylaşmak istediğim notlar bu kadardı. Aşağıdakiler ise **Stack Overflow**'dan bulduğum komik bir kaç yorum :) :

```

//When I wrote this, only God and I understood what I was doing
//Now, God only knows

// Autogenerated, do not edit. All changes will be undone.

// Magic. Do not touch.

```





```
/**
 * Always returns true.
 */
public boolean isAvailable() {
    return false;
}
```

## Bölüm 5 - Formatlama

Kodunuzun düzgün formatta (biçimde) olmasına dikkat etmelisiniz. Kodunuzun formatını belirleyen bir takım kurallar benimsemeli ve o kurallara her zaman uymalısınız. Eğer bir takımda çalışıyorsanız, takım olarak belli kuralları kabul etmeli ve tüm üyeler bu konuda hem fikir olmalıdır.

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] == $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



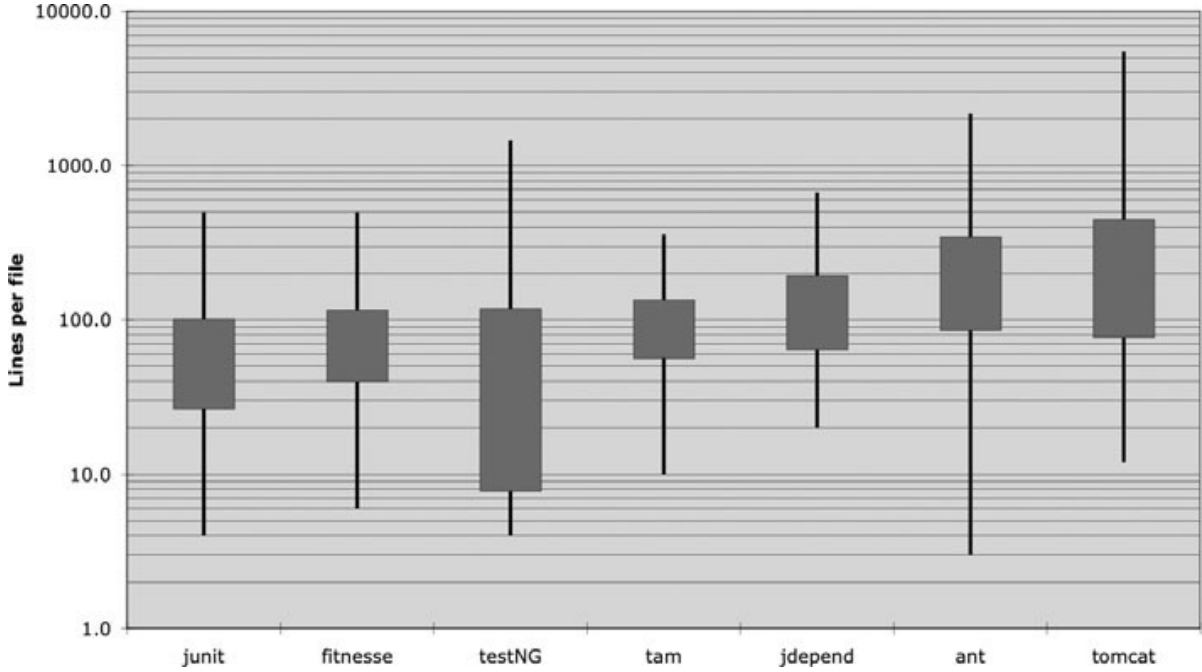
### Formatlamanın Amacı

Açık olalım; formatlama yani kodun biçimsel düzeni önemlidir. Kodu formatlamak iletişimle alakalıdır ve iletişim profesyonel bir geliştiricinin işinin ilk adımıdır. Belki de profesyonel bir geliştiricinin en önemli işinin “kodun çalışması” olduğunu düşündünüz. Umuyorum bu kitap, bu düşüncenizi değiştirmiştir.

Bugün eklediğiniz yeni bir fonksiyonla, muhtemelen bir sonraki sürümde tekrar değişecektir ve kodunuzun okunabilirliği, yapacağınız tüm değişiklikler için köklü bir etkiye sahip olacaktır. Kodlama tarzı ve okunabilirliği, orijinal kod değiştikten çok çok sonra da bakımını etkileyecektir. Disiplininiz ve kodlama tarzınız orada durmaya devam edecektir. Peki, iletişim kurmamıza yardım eden en iyi formatlama konuları neler?

### Dikey Formatlama

Dikey boyut ile başlayalım. Bir kaynak dosyası ne kadar büyük olmalı? Örneğin **Java** kaynak dosyaları en fazla ne kadar büyük olabiliyorlar? Aşağıdaki grafikte 7 farklı proje gösteriliyor: **JUnit**, **Fitness**, **TestNG**, **Time and Money**, **JDepend**, **Ant** ve **Tomcat**. Kutuların etrafındaki çizgiler, her projedeki minimum ve maksimum dosya boyutlarını gösteriyor. Kutu, dosyaların yaklaşık olarak üçte birini, kutunun ortası ise yaklaşık ortalamayı gösteriyor.



Yani **Fitness** projesindeki dosyalar ortalama 65 satır civarı ve dosyaların üçte biri 40 ve 100+ satır arası. **Fitness**'teki en büyük dosya 400 ve en küçüğü 6 satır.

**JUnit**, **Fitness** ve **Time and Money** küçük dosyalardan oluşmuş. Hiçbiri 500 satırın üzerinde değil ve bu dosyaların bir çoğu 200 satır. Diğer bir taraftan **Tomcat** ve **Ant** binlerce satırlık dosyalara sahip ve yarısından fazlası 200 satırın üzerinde.

Bu grafik bize ne anlatıyor? Çok önemli sistemleri, projeleri (maksimum limitimizi 500 tutarak) 200 satırlık dosyalardan oluşturabiliriz. Çünkü küçük dosyalar büyük dosyalardan her zaman daha anlaşılırdır.

## Gazete Metaforu

İyi yazılmış bir makale düşünelim. Onu dikey şekilde okursunuz. En başında yazının ne hakkında olduğunu söyleyen bir başlık beklersiniz ve ona göre yazıyı okuyup okumayacağınıza karar verirsiniz. İlk paragraf tüm hikayenin özetini verir. Aşağılara devam ettikçe detaylar artar.

Kaynak kodun gazete makalesi gibi okunmasını isteriz. İsmi basit ama açıklayıcı olmalı. İsmi kendisi, bize doğru modülde olup olmadığını söyleyebilecek kadar net olmalı. Detaylar aşağılarda olmalı.

Bir gazete de bir çok makaleden oluşur ve bu makalelerin çoğu küçüktür. Bir sürü makale sadece bir sayfaya sığabilir. Eğer gazeteler sayfalarca yazılmış tek bir makaleden oluşsaydı, muhtemelen kimse o gazeteyi okumazdı.

## Kavramlar Arası Dikey Açıklık

Hemen hemen tüm kodları soldan sağa ve yukarıdan aşağıya okuruz. Her satır bir açıklamayı ve satırlar bütünü bir düşünceyi temsil eder. Bu düşünceler birbirlerinden boşluk satırlarıyla ayrılmalıdırlar.

Aşağıdaki örneğe bakalım; paket tanımını, **import** ifadesini ve her bir fonksiyonu birbirinden ayıran boşluklar var. Kodda görselliğin en temel kuralı budur. Her boş satır yeni ve ayrı bir kavramın işaretidir.

Örneğin şu şekildedir:

```
package fitnessse.wikitext.widgets;import java.util.regex.*;public
class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?''';
    private static final Pattern pattern =
Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws
Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Şurada ise boşlukları silinmiş versiyonu; okunabilirliği ciddi derecede etkiliyor:

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?''';
    private static final Pattern pattern =
Pattern.compile("'''(.+?)'''",
```

```

        Pattern.MULTILINE + Pattern.DOTALL
    );
    public BoldWidget(ParentWidget parent, String text) throws
Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}

```

## Dikey Yoğunluk

Eğer açıklık (openness), kavramların ayırt edilmelerini sağlıyorsa; dikey yoğunluk da yakın ilişkiyi temsil ediyor olmalı. Öyle ki; birbirine sıkıca bağlı kod satırları dikey olarak yoğun şekilde görünür. Aşağıdaki gereksiz yorumların, 2 örnek değişken arasındaki ilişkiyi nasıl böldüğüne bakalım:

```

public class ReporterConfig {

    /**
     * The class name of the reporter listener
     */
    private String m_className;

    /**
     * The properties of the reporter listener
     */
    private List < Property > m_properties = new ArrayList <
Property > ();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
}

```

Şu kodun ise okunması çok daha kolay. Koda bakarız ve bunun iki değişkeni ve bir metodu olan bir sınıf olduğunu anlarız:

```

public class ReporterConfig {
    private String m_className;
    private List < Property > m_properties = new ArrayList <
Property > ();
}

```

```
public void addProperty(Property property) {
    m_properties.add(property);
}
```

## Dikey Uzaklık

Birbirine yakın kavramlar, dikey olarak birbirlerine yakın şekilde tutulmalıdırlar. Sonrasında bu ilişkili kavramlar -çok iyi bir sebebimiz olmadıkça- ayrı parçalara ayrılmamalıdırlar. Okuyucuların kodumuzu okurken sayfadan sayfaya geçmelerini istemeyiz.

Değişkenler, kullanıldıkları yere en yakın şekilde tanımlanmalıdır, şu şekilde:

```
private static void readPreferences() {
    InputStream is = null;
    try {
        is = new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is != null)
                is.close();
        } catch (IOException e1) {}
    }
}
```

Döngüler için kullandığımız kontrol değişkenleri ise döngüyle beraber tanımlanmalıdır. Şu fonksiyonda olduğu gibi:

```
public int countTestCases() {
    int count = 0;
    for (Test each: tests)
        count += each.countTestCases();
    return count;
}
```

## Örnek Değişkenler (Instance Variables)

Sınıfın en tepesinde tanımlı olmalıdır, ancak gene de bu durum örnek değişkenlerin dikey uzaklığını artırmamalıdır. Çünkü iyi tasarlanmış bir sınıfta örnek değişkenler sınıfın metotları tarafından defalarca kullanılırlar.

Örnek değişkenlerin nereye konulması gerektiğine dair çok tartışmalar yapıldı. **C++**'ta, sözde [makas \(scissors\) kuralı](#)ni takip ettik, bu kural tüm örnek değişkenlerin alta tanımlanması gerektiğini söyler. **Java**'daki genel kullanım şekli ise sınıfın üst kısmına koymaktır. Aslına bakarsanız kullanım şekillerinden birini diğerine tercih etmek için bir sebep

göremiyorum. Önemli olan örnek değişkenlerin iyi bilinen bir yere tanımlanmasıdır. Böylece herkes tanımları bulabilmek için nereye bakması gerektiğini bilir.

Aşağıdaki **TestSuite** sınıfında tanımlanmış iki değişken göreceksiniz. Onları *saklamak için* daha iyi bir yer bulunamazdı:

```
public class TestSuite implements Test {
    static public Test createTest(Class << ? extends TestCase >
theClass,
        String name) {
        ...
    }    getTestConstructor(Class << ? extends TestCase >
theClass)
        throws NoSuchMethodException {
        ...
    }    public static Test warning(final String message) {
        ...
    }    private static String exceptionToString(Throwable t) {
        ...
    }    private String fName;    private Vector < Test > fTests
= new Vector < Test > (10);    public TestSuite() {}    public
TestSuite(final Class << ? extends TestCase > theClass)
{
    ...
}
    public TestSuite(Class << ? extends TestCase > theClass,
String name) {
        ...
    }
    .....
}
```

## Bağımlı (Dependent) Fonksiyonlar

Eğer bir fonksiyon başka bir fonksiyonu çağırıyorsa, bu fonksiyonlar birbirine yakın olmalı ve eğer mümkünse çağırılan çağırılanın üstünde olmalıdır. Bu durum programa doğal bir akış sağlar.

Aşağıdaki koda bakalım; en üstteki fonksiyonun aşağıdaki fonksiyonu nasıl çağırdığına ve onun da diğerlerini nasıl çağırdığına bakın. Bu akış, çağırılan fonksiyonu bulmayı ve tüm modülün okunuşunu kolaylaştırıyor:

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
```

```

protected PageCrawler crawler;

public Response makeResponse(FitNesseContext context, Request
request) throws Exception {
    String pageName = getPageNameOrDefault(request,
"FrontPage");
    if (page == null)
        return notFoundResponse(context, request);
    else
        return makePageResponse(context);
}

private String getPageNameOrDefault(Request request, String
defaultPageName) {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
        pageName = defaultPageName;
    return pageName;
}

protected void loadPage(String resource, FitNesseContext
context) throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new
VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
        pageData = page.getData();
}

private Response notFoundResponse(FitNesseContext context,
Request request) throws Exception {
    return new NotFoundResponder().makeResponse(context,
request);
}

private SimpleResponse makePageResponse(FitNesseContext
context) throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);
    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
}

...

```

**Bir satır ne kadar geniş olmalı?**



Programcılar genelde kısa satırları tercih ederler. Benim kuralım ise bir satırda asla sağa kaydırma (scroll) yapma gereksinimi duymamak. Ancak bugünlerde monitörlerimiz çok geniş ve genç programcılarımız yazı tipini öyle küçük tutuyor ki, ekrana 200 karakter sığdırabiliyorlar. Bu iyi değil. Bir satır 120 karakterden uzun olmamalıdır.

## Yatay Açıklık ve Yoğunluk

Yatay boşluğu, güçlü ilişki içerisinde olanları birleştirmek ve daha zayıf olanları ayırt etmek için kullanırsınız.

Şu fonksiyona bakalım:

```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

Atama ifadelerinin iki ayrı tarafı olur: sağ ve sol. İşte boşluklar bu ayrımı net kılar. İşlemcileri (operators) vurgulamak için onlardan önce bir boşluk bıraktım. Diğer bir taraftan fonksiyon isimleri ile açılış parantezi arasına ise boşluk koymadım. Çünkü fonksiyon ve onun argümanları birbiriyle çok yakından alakalıdır. Ancak argümanların ayrı olduğunu vurgulamak için onları fonksiyon açılış parantezinden ayırdım.

## Girintileme

Bir sınıf hiyerarşilerden oluşur. Sınıf içerisinde metotlar, metotlar içerisinde bloklar... Bu hiyerarşinin her bir seviyesi isimlerini tanımlayabildiğimiz kapsamlardır ve kapsamların bu hiyerarşilerini görünür yapabilmek için konumlarına göre satırları girintilendirmemiz gerekir.

Bir sınıf içerisindeki metotlar sınıfın bir seviye sağına, metotların gerçekleştirimleri bu metot tanımlarının bir seviye sağına, metot içerisindeki bloklar ise her bir bloğun sağına girintilidir. Girintileme olmadan programlar insanlar tarafından neredeyse okunmaz olurdu.

### *Girintilemeyi Bozmak*

Kısa `if` ifadeleri, kısa `while`'lar ya da kısa fonksiyonlar için girintilemeyi bozmak bazen cazip gelir. Ancak bunu ne zaman yapsam, tekrar girintiliyorum ve ifadelerimi tek satıra indirgemekten kaçınıyorum. Şunun gibi hizalamayı ve genişletmeyi tercih ediyorum:

```
public class CommentWidget extends TextWidget {
    public static final String REGEXP =
    "^#[^\\r\\n]*(?: (?:\\r\\n|\\n|\\r) ?";
```

```
public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
}

public String render() throws Exception {
    return "";
}
}
```

## Takım Kuralları

Her programcının kendi formatlama kuralları vardır. Ancak bir takımda çalışıyorsa, orada takımın kuralları geçerli olur. Geliştiricilerden oluşan bir takım kendi basit formatlama tarzı üzerinde anlaşmalıdır ve takımın her bir üyesi bu tarzı kullanmalıdır. Yazılımımızın anlaşmazlık yaşamış bir grup insan tarafından yazılmış gibi değil de, tutarlı görünmesini isteriz.

2002'de bir projeme başladığımda, takımla beraber kodlama tarzımız üzerinde çalışmak için bir araya geldik. Araçları nereye koyacağımıza, sınıf, değişken, metot isimlerimizin nasıl olması gerektiğine karar verdik. Daha sonra bu kuralları geliştirme aracımızın (IDE) **Code Formatter**'ına (geliştirme araçlarının sunduğu otomatik kod biçimlendirme araçları) tanımladık ve hep ona bağlı kaldık. Sadece benim tercihim değil, takımın da kararını verdiği kurallardı bunlar. Takımın bir üyesi olarak, ben de hep bu kuralları takip ettim.

İyi bir yazılımın, iyi okunabilen dokümanlardan oluştuğunu unutmayın. Tutarlı ve akıcı tarzları olmalı. Okuyucu bir dosyada gördüğü formatlama tarzının, diğer tüm dosyalarda da olduğunu düşünebilmeli.

## Uncle Bob'un Formatlama Kuralları

Burada Bob amcamız kişisel olarak kullandığı formatlama kurallarını koda dökerek anlatmış. Ancak sınıf isminden hemen sonra (değişkenlerden önce), sınıf değişkenleri arasında ve sınıf bitmeden hemen önce (en son metottan sonra) koymadığı boşlukların beni üzdüğünü belirtmeden geçemedim <:) Sahi o boşlukları neden koymamış, fikri olan var mı?

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }
}
```

```

    public static List < File > findJavaFiles(File
parentDirectory) {
        List < File > files = new ArrayList < File > ();
        findJavaFiles(parentDirectory, files);
        return files;
    }

    private static void findJavaFiles(File parentDirectory, List
< File > files) {
        for (File file: parentDirectory.listFiles()) {
            if (file.getName().endsWith(".java"))
                files.add(file);
            else if (file.isDirectory())
                findJavaFiles(file, files);
        }
    }

    public void analyzeFile(File javaFile) throws Exception {
        BufferedReader br = new BufferedReader(new
FileReader(javaFile));
        String line;
        while ((line = br.readLine()) != null)
            measureLine(line);
    }

    private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
    }

    private void recordWidestLine(int lineSize) {
        if (lineSize > maxLineWidth) {
            maxLineWidth = lineSize;
            widestLineNumber = lineCount;
        }
    }

    public int getLineCount() {
        return lineCount;
    }

    public int getMaxLineWidth() {
        return maxLineWidth;
    }

    public int getWidestLineNumber() {
        return widestLineNumber;
    }

```

```

public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
}

public double getMeanLineWidth() {
    return (double) totalChars / lineCount;
}

public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width: sortedWidths) {
        cumulativeLineCount += lineCountForWidth(width);
        if (cumulativeLineCount > lineCount / 2)
            return width;
    }
    throw new Error("Cannot get here");
}

private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
}

private Integer[] getSortedWidths() {
    Set < Integer > widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new
Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
}
}

```

## Bölüm 6 - Nesneler ve Veri Yapıları

Değişkenlerimizi **private** yapmamızın bir sebebi var: kimsenin onlara bağımlı (dependent) olmasını istemiyoruz. Gerçekleştirmelerini ya da tiplerini değiştirme özgürlüğünü elimizde tutmak istiyoruz. Peki, programcılar neden **getter** ve **setter** metotlarını otomatik olarak ekler ve değişkenlerini sanki **public** değişkenlermiş gibi açığa çıkarır?

### Veri Soyutlama

Aşağıdaki iki kodun farkına bakalım. Her ikisi de kartezyen çarpımındaki **nokta** (point) kavramını temsil ediyor. Ancak birisi gerçekleştirimini açığa çıkarırken, diğeri onu tamamen saklıyor:

1.

```
public class Point {  
    public double x;  
    public double y;  
}
```

2.

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

İkinci örneğimizde gerçekleştirmenin bir dikdörtgen ya da polar koordinatlar içinde olup olmadığını anlamının bir yolu yok. İkisi de olabilir. Ve birden fazla veri yapısını temsil ediyor. İçindeki metotlar ise bir erişim politikasını (access policy) zorluyor. Burada koordinatları atomik şekilde düşünmek zorundayız.

İlk örneğimiz ise çok açık şekilde bir dikdörtgenin koordinatları ve bizi bu koordinatları bağımsız şekilde manipüle etmeye zorluyor. Değişkenler **private** olsaydı ve biz de **getter/setter** metotları kullanıyor olsaydık, bu durum gene de gerçekleştirmeyi açığa çıkarırdı.

Gerçekleştirmeyi gizlemek, yalnızca değişkenler ile fonksiyonlar arasına bir katman koyma meselesi değildir. Gerçekleştirmeleri gizlemek, soyutlama ile ilgilidir. Bir sınıf, değişkenlerini **getter** ve **setter** metotlar aracılığı ile dışarı açmaz; aksine gerçekleştirmeyi bilmelerine gerek olmadan veriyi değiştirmelerine izin veren arayüzleri açar.

Şimdi aşağıdaki şu iki koda bakalım:

1.

```
public interface Vehicle {  
    double getFuelTankCapacityInGallons();  
    double getGallonsOfGasoline();  
}
```

2.

```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

Burada ikincisini tercih etmeliyiz. Verimizin detaylarını vermek istemeyiz. Aksine verilerimizi soyut ifadelerle ifade etmek istiyoruz. Bu sadece arayüzleri veya **getter/setter** metotları kullanarak gerçekleşmez. Gerçek niyetimizi, nesnenin içerdiği veriyi en iyi şekilde temsil edebilecek şekle sokmalıyız. En kötü seçenek ise **getter/setter** metotları eklemektir.

## Veri/Nesne Anti-Simetrisi

Nesneler verilerini soyutlamalar arkasında saklarlar ve bu verileri işleyecek fonksiyonları açarlar. Veri yapıları ise nesnelerini dışa açarlar ve anlamlı hiçbir fonksiyonları yoktur. Bu ikisi birbirine sanal olarak tamamen zıttır.

Aşağıdaki örneğe bakalım:

```
public class Square {  
    public Point topLeft;  
    public double side;  
}  
public class Rectangle {  
    public Point topLeft;  
    public double height;  
    public double width;  
}  
public class Circle {  
    public Point center;  
    public double radius;  
}  
public class Geometry {  
    public final double PI = 3.141592653589793;  
    public double area(Object shape) throws NoSuchElementException  
{  
        if (shape instanceof Square) {
```

```

        Square s = (Square) shape;
        return s.side * s.side;
    } else if (shape instanceof Rectangle) {
        Rectangle r = (Rectangle) shape;
        return r.height * r.width;
    } else if (shape instanceof Circle) {
        Circle c = (Circle) shape;
        return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
}
}

```

**Geometry** sınıfı 3 tane *şekil* (shape) sınıfı üzerinde işlemler yapıyor. *Şekil* sınıfları hiçbir davranış sergilemeyen basit veri yapılarıdır. Tüm davranış **Geometry** sınıfı içerisinde.

**Geometry** sınıfına **perimeter()** isimli bir fonksiyon eklediğimizi düşünelim. *Şekil* sınıfları değişmemiş olacaktı. Bağlı olan diğer sınıflar da etkilenmemiş olacaktı. Diğer bir taraftan yeni bir *şekil* eklersem **Geometry** sınıfındaki tüm fonksiyonların değişmesi gerekecekti.

Şimdi aşağıdaki nesne yönelimli çözüme bakalım:

```

public class Square implements Shape {
    private Point topLeft;
    private double side;    public double area() {
        return side * side;
    }
}
public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;    public double area() {
        return height * width;
    }
}
public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;    public double
area() {
    return PI * radius * radius;
    }
}
}

```

**area()** polimorfik bir metot. **Geometry** isimli bir sınıfa ihtiyaç yok. Yani eğer yeni bir *şekil* eklersek, mevcut fonksiyonların hiçbirisi etkilenmeyecek. Ancak yeni bir fonksiyon eklersek, tüm *şekiller* değişmek zorunda kalacak.

İşte burada nesneler ve veri yapıları arasındaki ayrımı görebilirsiniz:

*Prosedürel kod -veri yapılarını kullanan kod-, mevcut veri yapılarını değiştirmeden yeni fonksiyonlar eklemeyi kolaylaştırır. Nesne yönelimli kod ise mevcut fonksiyonları değiştirmeden yeni sınıflar eklemeyi kolaylaştırır.*

*Prosedürel kod yeni veri yapıları eklemeyi zorlaştırır çünkü tüm fonksiyonlar değişmelidir. Nesne yönelimli kod ise yeni fonksiyonlar eklemeyi zorlaştırır çünkü tüm sınıflar değişmelidir.*

Yani, nesne yönelimli için zor olan şeyler prosedürel için kolay, prosedürel için zor olanlar da nesne yönelimli için kolaydır.

## Demeter Kuralı ([Law of Demeter](#))

**Demeter Kuralı** der ki: Bir modül, değiştirdiği bir nesnenin içini bilmemelidir. Nesneler, verilerini saklar ve işlemlerini açarlar. Yani, bir nesne iç yapısını erişimciler aracılığıyla açmamalıdır.

**C** isimli bir sınıfımız ve bunun **f** isimli bir fonksiyonu olsun. Bu fonksiyon sadece şunların metotlarını çağırmalıdır:

- **C**'ye ait metotlar
- **f** tarafından yaratılmış bir nesnenin metotları
- **f** fonksiyonuna argüman olarak geçilen bir nesneye ait metotlar
- **C** içerisinde örnek değişken olarak tutulan bir nesneye ait metotlar

Fonksiyon, izin verilen fonksiyonlardan herhangi biri tarafından dönen nesneler üzerindeki metotları çağırmamalıdır. Diğer bir deyişle: “Arkadaşlarınla konuş, yabancılarla değil (*talk to friends, not to strangers*).”





**Apache**'de bir yerlerde bulduğum aşağıdaki kod bu kuralı ihlal ediyor.

Çünkü `getOptions()`'dan dönen nesne

ile `getScratchDir()` metodunu, `getScratchDir()`'den dönen değer ile  
de `getAbsolutePath()` metodunu çağırıyor.

```
final String outputDir =  
ctxt.getOptions().getScratchDir().getAbsolutePath();
```

## Tren Enkazları

Uç uca geçmiş bir dizi tren vagonu gibi birbirini çağıran fonksiyonlar için bu benzetme yapılır.  
Bu tür kodları parçalara ayırmak en iyisidir.

Şu koda bir bakalım:

```
Options opts = ctxt.getOptions();  
File scratchDir = opts.getScratchDir();  
final String outputDir = scratchDir.getAbsolutePath();
```

Bu koddaki ikinci ve üçüncü satırlar **Demeter Kuralı**'nı ihlal ediyorlar mı? Kesinlikle. Bu kodun **Demeter Kuralı**'nı ihlal edip etmediği; `ctxt`, `opts` ve `scratchDir`'in nesne ya da veri yapısı olup olmamasına bağlıdır. Eğer nesneyseler, o zaman iç yapıları saklanmalıdır. Yani iç yapılarının biliniyor oluşu açıkça bir ihlaldir. Diğer bir taraftan hiçbir davranış sergilemeyen veri yapıları iseler, o zaman doğal olarak iç yapılarını açık ediyorlar yani **Demeter Kuralı**'nı ihlal ediyorlardır.

Erişimci fonksiyonların kullanımı meseleyi karıştırır. Eğer kod şu şekilde yazılmış olsaydı, muhtemelen **Demeter Kuralı**'nı sorgulamayacaktık:

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Ya `ctxt`, `opts` ve `scratchDir` gerçek davranışlı nesneler olsalardı? O zaman iç yapılarını gizlemek zorunda olduklarından, bu nesneler aracılığı ile başka yerlere yönlenecezdik. O halde `scratchDir`'e ait `absolutePath` nesnesini nasıl alabileceğiz?

Bu:

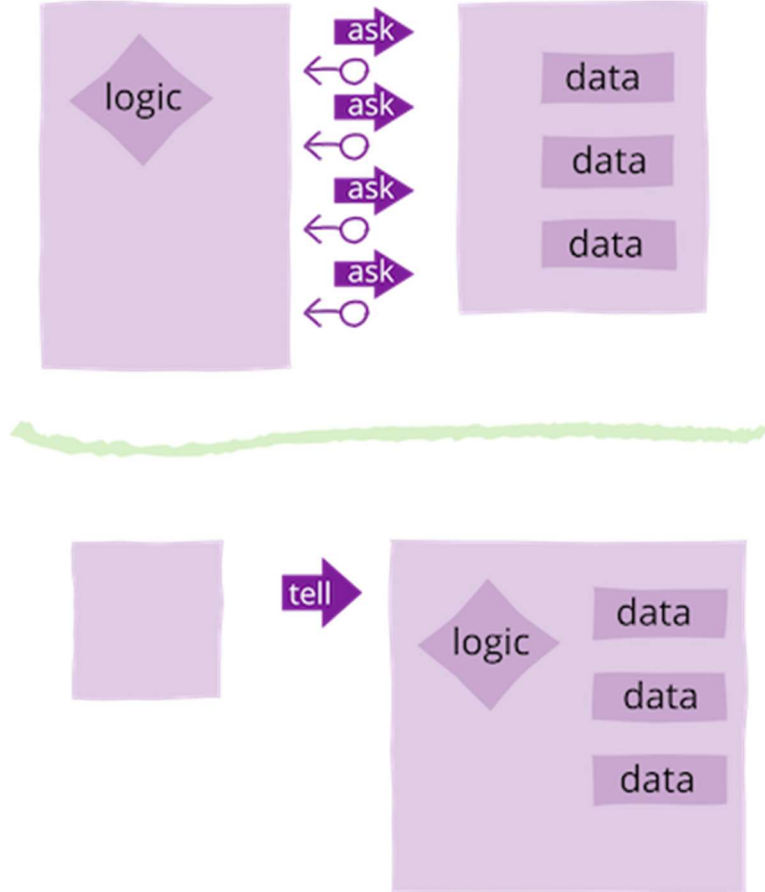
```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

Ya da şu:

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

İkinci seçenek `getScratchDirectoryOption()` metodunun bir nesne değil bir veri yapısı olduğunu varsayıyor ve burada iki seçenek de iyi durmuyor.

Eğer *ctxt* bir nesne ise, ona bir şeyler yapmasını söylemeliyiz, ona içindekileri sormamalıyız. (**Tell Don't Ask prensibi**) O zaman neden `scratchDirectory`'den `absolutePath`'i istiyoruz? Onunla ne yapacağız?



Şu koda bakalım; noktalar, eğik çizgiler (slash), dosya uzantıları ve **File** nesneleri birlikte bu kadar dikkatsiz kullanılmamalıdır:

```
String outFile = outputDir + "/" + className.replace('.', '/') +
".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Burada **"absolute path"**'i almak istememizin sebebi, verilen isimde bir **"scratch file"** yaratmak istememiz. Yani, *ctxt*'ye bunu yapacağını söylemek isteseydik, şöyle olmalıydı:

```
BufferedOutputStream bos =
ctxt.createScratchFileStream(className);
```

Bir nesnenin yapması makul olan bir iş bu. Bu durum **ctxt**'nin iç yapısını gizlemesine izin verir. Ve mevcut fonksiyonun bilmemesi gereken nesneler aracılığıyla yönlendirilerek **Demeter Kuralı**'nı ihlal etmesini de önler.

## Veri Aktarım Nesneleri (Data Transfer Objects)

Veri yapısını en kısa şekilde özetleyecek olursak, **public** değişkenleri olan ve hiç fonksiyonu olmayan sınıftır. Buna bazen **DTO** da deriz; ki bu nesneler özellikle veritabanı vb. yerlerle haberleşiyorken oldukça kullanışlıdır.

Biraz daha yaygını **Bean** formudur. **Bean**'lerin **getter/setter**'larca değiştirilmiş **private** değişkenleri vardır. Bazı **Object Oriented puristlerini** (pürist: dilin kullanımında aşırı dikkatli kişi) iyi hissettirebilir ancak başka hiçbir faydası yoktur. Örnek bir **Bean**:

```
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;
    public Address(String street, String streetExtra,
        String city, String state, String zip) {
        this.street = street;
        this.streetExtra = streetExtra;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }
    public String getStreet() {
        return street;
    }
    public String getStreetExtra() {
        return streetExtra;
    }
    public String getCity() {
        return city;
    }
    public String getState() {
        return state;
    }
    public String getZip() {
        return zip;
    }
}
```

## Melez Yapılar (Hybrids)

Yarısı nesne yarısı veri yapısı olan melez yapılar bazen karışıklığa sebep olur. Önemli işler yapan fonksiyonları, **private** değişkenleri **public** yapan erişimcileri (getters/setters) ve **public** değişkenleri vardır. Bu tür karmaşık yapılar yeni fonksiyonlar eklemeyi zorlaştırır. Ve hatta yeni veri yapıları eklemeyi de zorlaştırır. Melez yapılar oluşturmaktan kaçınmalıyız. Bu tür yapılar, başka tiplerden ya da fonksiyonlardan korunma ihtiyacı olup olmadığından emin olunamayan sistemlerin göstergesidir.

## Aktif Kayıt (Active Record)

Aktif kayıtlar **DTO**'ların özel formlarıdır. **Public** değişkenleri olan veri yapılarıdır; ancak **save** ve **find** gibi yönlendirici metotları vardır. Bu aktif kayıtlar genellikle veritabanı tablolarından ya da diğer veri kaynaklarından doğrudan çeviridir. Fakat geliştiricilerin aktif kayıtları belli iş kuralları içerisine koyarak, onlara nesnelermiş gibi davranmaya çalıştıklarına sık sık denk geliyoruz. Çözüm ise elbette bir aktif kayda veri yapısıymış gibi davranmak ve iş kurallarını içeren ayrı nesneler yaratarak iç yapıyı saklamaktır.

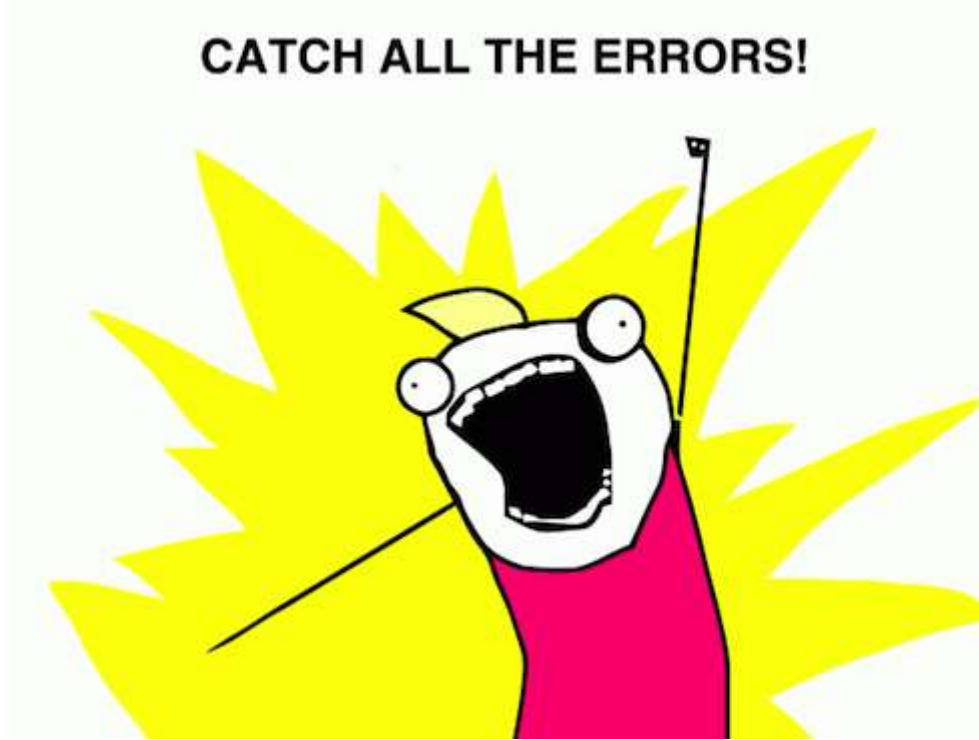
Nesneler davranışını açığa vurur ve verisini saklar. Bu, hiçbir mevcut davranış değiştirmeden yeni nesneler eklemeyi kolaylaştırır. Mevcut nesnelere ise yeni davranışlar eklemeyi zorlaştırır.

Veri yapıları ise verisini açığa vurur ve önemli bir davranış yoktur. Bu, mevcut veri yapılarına yeni davranışlar eklemeyi kolaylaştırır ancak mevcut fonksiyonlara yeni veri yapıları eklemeyi zorlaştırır.

Bazen yeni veri tipleri bazen de yeni davranışlar ekleyebilme esnekliği isteriz. İyi yazılımcılar, bulunduğu kodun durumuna göre en iyi yaklaşımı seçeceklerdir.

## Bölüm 7 - Hata İşleme

Hata işleme (Error handling), kod yazarken yapmak zorunda olduğumuz şeylerden sadece biri. Bir şeyler yanlış gidebilir ve programımız patlayabilir. Biz programcılar olarak kodumuzu böyle durumlara karşı hazırlamaktan sorumluyuz.



### Dönüş Kodları Yerine İstisnaları Kullanın

Eskiden yazdığımız bazı dillerde istisnaları (exception) işleyebileceğimiz ve raporlayabileceğimiz teknikler sınırlıydı. Bunun yerine **flag**ler ya da hata kodları dönüyorduk. Bu yaklaşıma örnek:

```
public class DeviceController {  
    ...  
    public void sendShutDown() {  
        DeviceHandle handle = getHandle(DEV1);  
        // Check the state of the device  
        if (handle != DeviceHandle.INVALID) {  
            // Save the device status to the record field  
            retrieveDeviceRecord(handle);  
            // If not suspended, shut down  
            if (record.getStatus() != DEVICE_SUSPENDED) {  
                pauseDevice(handle);  
                clearDeviceWorkQueue(handle);  
                closeDevice(handle);  
            }  
        }  
    }  
}
```

```

        } else {
            logger.log("Device suspended. Unable to shut
down");
        }
    } else {
        logger.log("Invalid handle for: " +
DEV1.toString());
    }
}
...
}

```

Bu tür yaklaşımlar çağırının kafasını karıştırır. Çağırın, bu çağırımdan hemen sonra hataları kontrol etmelidir, ancak maalesef bu kontroller çok çabuk unutulabiliyor. Bu sebeple, bir hata ile karşılaşıldığında istisna fırlatmak (throw) daha iyidir. Çağırının kodunu daha temiz hale getirir.

Aşağıdaki kod, istisna fırlatmayı seçtiğimiz versiyonu:

```

public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }
    private void tryToShutDown() throws DeviceShutDownError {
        DeviceHandle handle = getHandle(DEV1);
        DeviceRecord record = retrieveDeviceRecord(handle);
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
    }
    private DeviceHandle getHandle(DeviceID id) {
        ...
        throw new DeviceShutDownError("Invalid handle for: "
+ id.toString());
        ...
    }
    ...
}

```

Kod daha temiz çünkü birbirine geçmiş iki işlem de (cihazın kapanması ve hata işleme) şimdi birbirinden ayrı. Bu 2 işleme bakabilir ve onları birbirinden bağımsız olarak algılayabiliriz.

## İlk Önce Try-Catch-Finally Bloklarını Yazın

**try-catch** yazmak programınıza bir kapsam sağlar ve uygulamanın `try` bloğunda bir yerlerde hata alabileceğini ve `catch`'de duracağını belirtirsiniz. `catch` blokları programınızı tutarlı bir durumda bırakmak zorundadırlar. Bu sebeple istisnalar fırlatabileceğiniz bir koda, `try-catch-finally` bloklarını yazmakla başlayın. Kullanıcının koddan ne beklemesi gerektiğini anlamasına yardım edecektir.

Şu örneğe bakalım. Bir dosyaya erişen ve bazı `serialize` edilmiş nesneleri okuyan kodu yazmamız gerekiyor. Dosya bulunamazsa istisna fırlatılacağını söyleyen bir birim test ile başlayalım:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection("invalid - file");
}
```

Test, şu taklit gerçekleştirimi de yazmamızı gerektirecektir:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    // dummy return until we have a real implementation
    return new ArrayList<RecordedGrip>();
}
```

Testimiz patlayacak çünkü bir istisna fırlatmıyor. Şimdi ise kodumuzu geçersiz bir dosyaya erişecek şekilde değiştirelim ve bir istisna fırlattığını görelim:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName)
    } catch (Exception e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Testimiz şimdi geçecek çünkü istisnayı yakaladık. Bu noktada kodu yeniden yapılandırabiliriz. İstisnayı, aslında fırlatılması gereken istisna tipine daraltacağız:

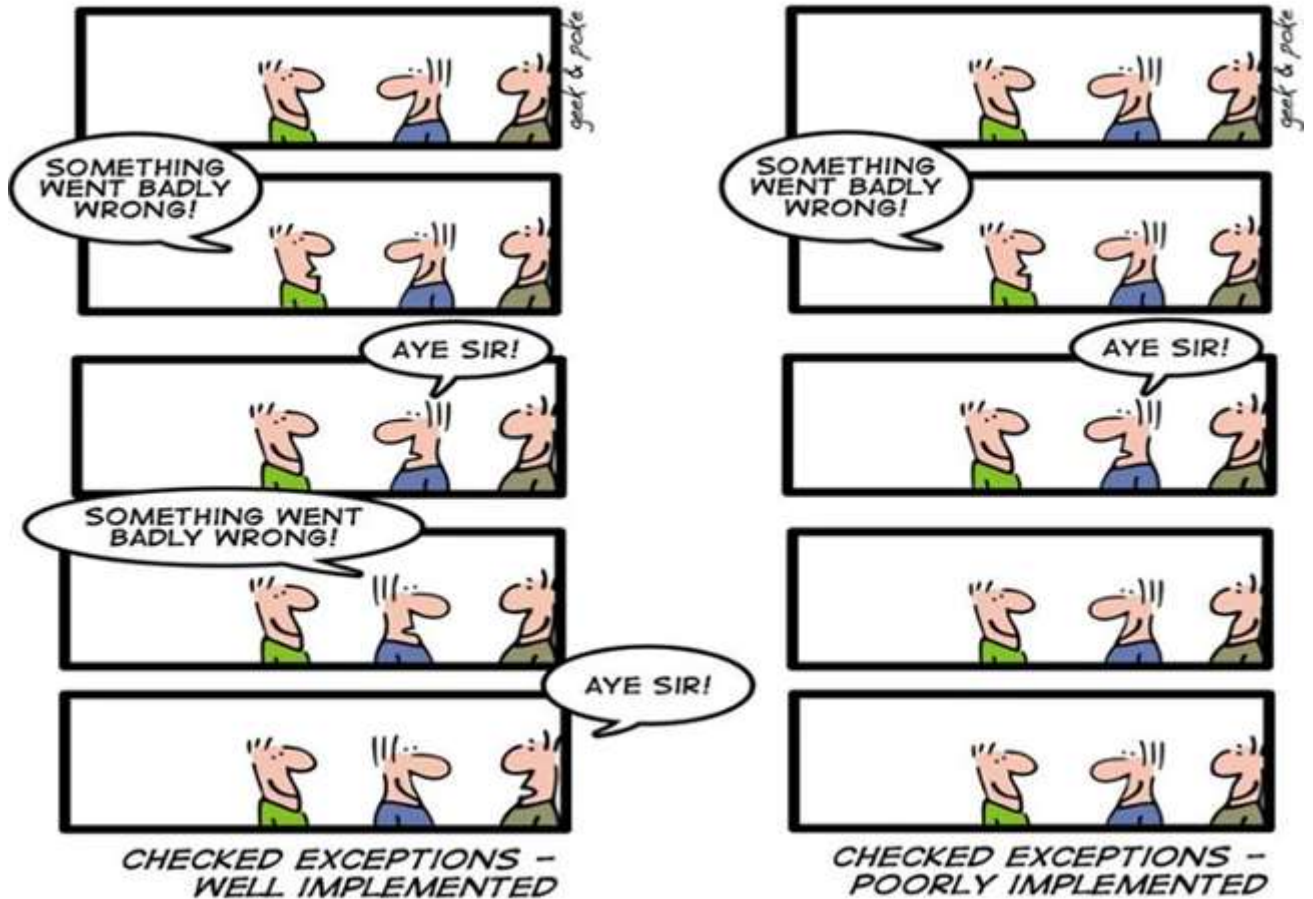
```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new
FileInputStream(sectionName);
        stream.close();
    } catch (FileNotFoundException e) {
        throw new StorageException("retrieval error", e);
    }
}
```

```
return new ArrayList<RecordedGrip>();  
}
```

İhtiyacımız olan mantığı inşa edebilmek için **TDD (Test Driven Development)** kullanabiliriz. İstisnaları zorlayan testler yazmaya çalışmalı ve ardından testlerimize cevap verecek kodu yazmalıyız. Bu, `try` bloğunu önce yazmamızı sağlayacak ve bakım yapmamızı kolaylaştıracaktır.

## KontROLSÜZ (Unchecked) İstisnalar Kullanın

**Java** programcıları yıllarca kontrollü (checked) istisnaların faydalarını tartıştılar. **Java**'nın ilk versiyonunda kontrollü istisnalar tanıtıldığında bize harika bir fikir gibi geldi. Her metodun imzası, çağırının geçebileceği tüm istisnaları listeleyecekti. Daha da fazlası, bu istisnalar metodun tipinin bir parçasıydı.



**C#**'in -cesur teşebbüslerine rağmen- kontrollü istisnaları yok, **C++**'in da, **Python** ya da **Ruby**'nin de. Kontrollü istisnaların bedeline değip değmeyeceğine karar vermemiz gerek.

Bedel nedir? Bedel, **Açık/Kapalı Kuralı**'nın (OCP — Gelişime Açık Değişime Kapalılık Kuralı) ihlalidir. Metodunuzdan kontrollü bir istisna fırlatırsanız ve `catch` üç seviyeden fazlaysa, o istisnayı sizinle `catch` arasındaki her metodun imzasında belirtmeniz gerekir. Bu demektir ki,



düşük seviyede bir değişiklik, imza değişikliklerini daha üst seviyelerde zorlayabilir. Onları ilgilendiren hiçbir şey olmadığı halde, değiştirilen modüller yeniden derlenmeli ve dağıtılmalıdır.

Büyük sistemlerdeki çağırma hiyerarşilerini düşünün. En alt seviye metotlardan birisinin bir istisna fırlatacak şekilde düzenlenmesi durumunda, çağırın tüm metotlar da imzasına bir `throws` eklemek zorundadır. Bu durumda kapsülleme de (encapsulation) bozulmuştur çünkü değişen metotlar da artık bu istisnanın detaylarını biliyor olacaktır.

## İstisnalarla Bağlam Sağlayın

Fırlattığınız her istisna, kaynağa ve hatanın yerine ait yeterli bilgi sağlamalıdır. **Java**'da herhangi bir istisnadan bir iz bulabilirsiniz ya da bazen bu izler bize hiçbir şey söylemez. Bu nedenle fırlattığınız istisnalarda bilgilendirici hata mesajları verin. Başarısız olan işlemde, hatanın tipinden bahsedin. Eğer uygulamanızda loglama yapıyorsanız, `catch` bloğunuza bu hatayı loglayabilecek yeterli bilgiyi geçin.

## Çağırının İhtiyaçlarına Uygun İstisna Sınıfları Tanımlayın

Hataları sınıflandırabilmenin bir sürü yolu vardır; kaynaklarına göre, türlerine göre sınıflandırabiliriz. Cihaz hatası mı, ağ hatası mı ya da programlama hataları mı?

Şu sınıflandırma örneğine bakalım. Üçüncü taraf bir kütüphane çağırımı için `try-catch-finally` yazılmış. Çağrılardan fırlatılabilecek tüm istisnaları kapsıyor:

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```

Bu ifadede çokça tekrarlanmış kod var. Burada yaptığımız işin kabaca aynı olduğunu bildiğimizden çağırdığımız **API**'yi sararak (wrapping) ve ortak bir istisna tipi döndüğünden emin olarak kodumuzu önemli ölçüde basitleştirebiliriz:

```

LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}

```

**LocalPort** sınıfımız, **ACMEPort** sınıfından atılmış istisnaları yakalayan ve çeviren basit bir sarmalayıcıdır (wrapper):

```

public class LocalPort {
    private ACMEPort innerPort;
    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }
    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {

```

Sarmalayıcı sınıflar üçüncü taraf **API**'lerin detaylarını gizlemek için en iyi pratiktir. Belli bir tedarikçinin **API**'sine bağlı olmazsınız ve rahat hissedebileceğiniz bir **API** tanımlayabilirsiniz. Ve ileride farklı bir kütüphaneye geçmek istediğinizde, sarmalayarak minimize ettiğiniz bağımlılıklar ile geçiş yapmak çok daha kolaydır.

## Normal Bir Akış Tanımlayın

Şu örneğe bakalım; masrafları hesaplayan bir kod parçası:

```

try {
    MealExpenses expenses =
expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch (MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}

```

Eğer öğünler masraflandırılmışsa toplama ekliyor, değilse çalışan o gün için o yemek tutarını alıyor. Burada istisna akışı karıştırıyor. Bu özel durum ile ilgilenmemiş olsak, kodumuz çok daha temiz görünürdü:

```
MealExpenses expenses =  
expenseReportDAO.getMeals(employee.getID());  
m_total += expenses.getTotal();
```

**ExpenseReportDAO** sınıfını her zaman bir **MealExpense** nesnesi dönecek şekilde değiştirdiğimizde kodumuzu daha basit hale getirmiş olduk. Eğer hiç masraf yoksa da, günlük bir **MealExpense** nesnesi dönecektir:

```
public class PerDiemMealExpenses implements MealExpenses {  
    public int getTotal() {  
        // return the per diem default  
    }  
}
```

Buna [Special Case Pattern](#) (**Özel Durum Deseni**) denir. Özel durumlarla senin için başa çıkacak bir sınıf yaratır ya da bir nesne ayarlarsın. Bunu yaptığında, ön yüz kodu istisnai durumlarla uğraşmak zorunda kalmayacaktır. Bu davranış özel durum nesnesi ile kapsüllenmiş olur.

## Null Değerler Dönmeyin

Hata işlemeyen bahsediyorken, hataları davet edercesine yaptığımız şeylerden de bahsetmeliyiz. `null` dönmek listedeki ilk şeydir. Örneğin:

```
public void registerItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.getItemRegistry();  
        if (registry != null) {  
            Item existing = registry.getItem(item.getID());  
            if (existing.getBillingPeriod().hasRetailOwner()) {  
                existing.register(item);  
            }  
        }  
    }  
}
```

Bu kod gözünüze normal gelebilir, ancak oldukça kötüdür. `null` döndüğümüzde aslında gene kendimize iş çıkarıyor ve topu fonksiyonumuzu çağıranlara atıyoruz. Uygulamanın kontrolden çıkması için tek bir eksik `null` kontrolü yeterli gibi görünüyor. İlk `if`'ten sonraki satırda `null` kontrolü yok. `persistentStore` nesnesi `null` olmuş olsaydı, çalışma anında çok minnoş bir `NullPointerException` alırdık.

Eğer bir metottan `null` dönecekse, onun yerine istisna fırlatmayı ya da bir **Special Case** nesnesi dönmeyi düşünün. Eğer kullandığınız bir **API**'den `null` dönebilecek bir metot çağırıyorsanız da, bu metodu özel durum nesnesi dönen ya da istisna fırlatan bir metot ile sarmalamayı düşünebilirsiniz.

Çoğu durumda özel durum nesneleri kolay çözümdür. Şunun gibi bir kodunuz olduğunu düşünelim:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for (Employee e: employees) {
        totalPay += e.getPay();
    }
}
```

`getEmployees()` `null` dönebilir ancak gerçekten de dönmek zorunda mı?

Eğer `getEmployees()` metodunu boş liste dönecek şekilde düzenlersek, kodumuzu temizleyebiliriz:

```
List<Employee> employees = getEmployees();
for (Employee e: employees) {
    totalPay += e.getPay();
}
```

Neyse ki **Java**'nın `Collections.emptyList()` metodu var ve boş sabit bir liste dönüyor:

```
public List<Employee> getEmployees() {
    if (...there are no employees...)
        return Collections.emptyList();
}
```

Eğer bu şekilde kodlarsanız, `NullPointerException` alma şansınızı minimize edersiniz.

## Null Argümanlar Geçmeyin

Metotlardan `null` dönmek kötü bir pratiktir ancak metotlara `null` geçmek daha da kötü bir pratiktir. Sizden `null` bekleyen bir **API** ile çalışmadıkça, kodunuzda mümkün mertebe `null` geçmekten kaçınmalısınız.

Nedenini anlamak için şu örneğe bakalım; iki nokta için bir metrik hesaplıyor:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        return (p2.x- p1.x) * 1.5;
    }
    ...
}
```

Birileri `null` bir parametre geçtiğinde ne olacak?

```
calculator.xProjection(null, new Point(12, 13));
```

`NullPointerException` alacağız elbette. Bunu nasıl düzeltebiliriz? Yeni bir istisna oluşturabilir ve bunu fırlatabiliriz:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            throw IllegalArgumentException(
                "Invalid argument for
MetricsCalculator.xProjection");
        }
        return (p2.x- p1.x) * 1.5;
    }
}
```

Diğer bir alternatif:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x- p1.x) * 1.5;
    }
}
```

Belgeme için iyi ancak problemimizi çözmiyor; birileri `null` geçtiğinde yine de hata alacağız.

Çoğu dilde istemeden gönderilmiş `null` değerler ile uğraşabilmenin bir yolu yoktur. Durum böyle olduğundan, burada en gerçekçi yaklaşım `null` değerlerin gelmesini önlemektir.

Temiz kod okunabilirdir ancak güçlü de olmalıdır. Bu ikisi çelişkili hedefler değildirler. Eğer hata işlemeyi bağımsız bir iş olarak görürsek, temiz ve güçlü kodlar yazabilir ve kodumuzun sürdürülebilirliği konusunda büyük adımlar atabiliriz.

## Bölüm 8 - Sınırlar

Sistemlerimizdeki yazılımları nadiren kontrol ederiz. Bazen üçüncü taraf paketler satın alırsınız veya açık kaynak kullanırsınız. Diğer zamanlarda da şirketimizin diğer ekiplerinin bizler için yazdığı bileşenlere bağımlıyızdır. Ve bu yabancı kodları kodumuzla temiz bir şekilde birleştirmek zorunda kalırız.

### Üçüncü Taraf Yazılım (Kod) Kullanmak

Bir arayüz kullanıcısı ile arayüz sağlayıcısı arasında doğal bir gerginlik vardır. Üçüncü taraf paketlerin ve çatıların (framework) sağlayıcıları, yazılımlarının geniş kitlelerce kullanılabilir olmaları ve bir çok ortamda çalışabilir olmaları için çabalarlar. Diğer bir taraftan kullanıcılar ise belirli ihtiyaçlara odaklanmış arayüzlerle çalışmak isterler. İşte gerginlik burada başlar.

Bir örnek olması açısından ***java.util.Map*** sınıfına bakalım:

- `clear()` void - Map
- `containsKey(Object key)` boolean - Map
- `containsValue(Object value)` boolean - Map
- `entrySet()` Set - Map
- `equals(Object o)` boolean - Map
- `get(Object key)` Object - Map
- `getClass()` Class<? extends Object> - Object
- `hashCode()` int - Map
- `isEmpty()` boolean - Map
- `keySet()` Set - Map
- `notify()` void - Object
- `notifyAll()` void - Object
- `put(Object key, Object value)` Object - Map
- `putAll(Map t)` void - Map
- `remove(Object key)` Object - Map
- `size()` int - Map
- `toString()` String - Object
- `values()` Collection - Map
- `wait()` void - Object
- `wait(long timeout)` void - Object
- `wait(long timeout, int nanos)` void - Object

(**Not: Java 6**'da `putAll()` ve `toString()` metotları kaldırılmış. **Java 8**'deki **Map** sınıfı için ise [suraya](#) bakabilirsiniz. Bir çok metot eklenmiş ve `putAll()` geri gelmiş. `toString()` ise, özyinelemeli (recursive) işlemlerde **Map** kendi referansını içerdiğinden ve bu sebeple bir istisna fırlatılabileceğinden eklenmemiş.)

**Map** bir çok şey yapabilen metotlarla dolu bir arayüzdür. Bu güç ve esneklik yeri geldiğinde kullanışlıdır ancak bir çok sorumluluğu da beraberinde getirir. Örneğin uygulamamızda bir **Map** kullanıyor ve onu bir yerlere argüman olarak geçiriyor olalım. **Map**'imizi kullananların içerisinden hiçbir şey silmemesi gerekebilir. Ancak **Map** arayüzü **clear()** metoduna sahiptir ve **Map**'i kullanan herkes bu güce sahiptir. Ya da tasarımıma göre **Map** üzerinde sadece belirli tipte nesneleri tutuyor olabiliriz. Fakat **Map**, içindeki nesnelerin tiplerini güvenilir şekilde sınırlandırmaz. Herhangi bir kullanıcı, herhangi tipte bir nesneyi **Map**'e ekleyebilir.

Uygulamamızın sensörlerden oluşan bir **Map**'e ihtiyacı olsaydı, muhtemelen şu şekilde tanımlayacaktık:

```
Map sensors = new HashMap();
```

Daha sonra bir yerlerde tek bir sensöre erişmek isteseydik, şu şekilde erişecektik:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Bu çözüm işe yarar ancak kod temiz değildir. Bu kod bir **Map** üzerinden bir **Object**'e erişmenin ve onu doğru tipe dönüştürmenin (type casting) sorumluluğunu taşıyor. Kodun okunabilirliğini [Generic](#)'leri kullanarak büyük oranda geliştirebiliriz:

```
Map<Sensor> sensors = new HashMap<Sensor>();
.....
Sensor s = sensors.get(sensorId);
```

`Map<Sensor>` yapması gerekenden ya da istediğimizden daha fazla şey yapıyor ve bu çözüm de problemimizi çözmiyor.

`Map<Sensor>` örneğini (instance) serbestçe bir yerlere argüman olarak geçirmek demek; **Map** arayüzü her değişiklik yaptığında, düzeltilecek çok yerin olacağı anlamına gelir. Böyle bir değişikliğin pek olası olmadığını düşünebiliriz ancak **Java 5**'te **Generic**'lere destek geldiği zamanı hatırlayın. **Map**'lerin özgür kullanımını telafi etmek için gereken büyük değişiklik nedeniyle **Generic** kullanımını engelleyen sistemler gördük.

**Map**'in daha temiz bir kullanımı aşağıdaki gibidir.

```
public class Sensors {
    private Map sensors = new HashMap();
    public Sensor
    getById(String id) {
        return (Sensor) sensors.get(id);
    }
}
```

```
...  
}
```

İşte burada **Map** arayüzü **Sensors** sınırının (boundary) ardına gizlenmiştir. Uygulamanın geri kalanı üzerinde çok az etki bırakarak değiştirilebilir. **Generic** kullanımı artık büyük bir sorun değil çünkü **tip dönüşümü** ve **tip yönetimi**, **Sensors** sınıfının içinde halledildi. Bu arayüz sadece uygulamanın ihtiyaçlarını karşılaması için oluşturuldu.

**Map**'in her kullanımında bu şekilde kapsüllenmesini önermiyoruz. Bunun yerine, **Map**'leri (veya sınırdaki başka herhangi bir arayüzü) sisteminizde bir yerlere geçirmemenizi tavsiye ediyoruz. **Map** gibi bir sınır arayüzü kullanıyorsanız, onu sınıfın içinde tutun. **Map**'leri return etmekten ve **public** API'lere argüman olarak geçmekten kaçınin.

## Sınırları Keşfetmek ve Öğrenmek

Üçüncü taraf yazılımlar/çatılar daha kısa sürede daha fazla işlevsellik elde etmemize yardımcı olur. Ancak bu paketleri kullanmak istediğimizde test yazmak bizim için en iyi yol olabilir.

Üçüncü taraf kütüphanemizi nasıl kullanacağımızın net olmadığını varsayalım. Doküman okumak ve onu nasıl kullanacağımıza karar vermek için bir veya iki gün (veya daha fazla) harcayabiliriz. Ardından, kodumuzu üçüncü taraf kodu kullanacak şekilde yazabilir ve düşündüğümüz şekilde çalışıp çalışmayacağını görebiliriz. Hatta çoğu zaman kendimizi, aldığımız hataların kodumuzda mı yoksa onların hataları mı olup olmadığını anlamaya çalışırken buluruz.

Üçüncü taraf kodu öğrenmek ve entegre etmek zordur. İkisini birlikte yapmak daha da zordur. Üretim (production) kodumuzda yeni şeyler denemek yerine, üçüncü taraf kodu keşfetmek için bazı testler yazabiliriz. **Jim Newkirk** bu testleri, *öğrenme testleri* (learning tests) olarak nitelendiriyor.

*Öğrenme testlerinde*, üçüncü taraf API'yi uygulamamızda kullanmayı umduğumuz şekilde çağırırız. Aslında, bu API hakkındaki anlayışımızı kontrol eden kontrollü deneyler yapıyoruz. Testler, API'den ne istediğimiz üzerine odaklanır.

## Log4j Öğrenmek

Diyelim ki kendi yazdığımız **logger**ımız yerine, **Apache Log4j** paketini kullanmak istiyoruz. Biraz doküman okuduktan sonra ilk testimizi yazarız ve konsola bir "hello" yazmasını bekleriz:

```
@Test  
public void testLogCreate() {  
    Logger logger = Logger.getLogger("MyLogger");  
    logger.info("hello");  
}
```



Testi çalıştırdığımızda **logger, Appender** adı verilen bir şeye ihtiyacımız olduğunu belirten bir hata üretir. Biraz daha okuduktan sonra bir **ConsoleAppender** olduğunu farkediyoruz. **ConsoleAppender** da ekledikten sonra kod şu şekile geliyor:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("hello");
}
```

Bu kez, **Appender**'ın çıktı üretmediğini görüyoruz. **Google**'dan küçük bir yardım aldıktan sonra aşağıdakileri deneyelim:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("hello");
}
```

İşe yaradı ve konsola "hello" yazdırdık. **ConsoleAppender**'a *konsola yazdığını* söylemek zorundayız.

İlginçtir ki, **ConsoleAppender.SYSTEM\_OUT** bağımsız değişkenini kaldırdığımızda, "hello" ifadesinin yine de yazdırıldığını görüyoruz. Fakat **PatternLayout**'u çıkardığımızda, gene hata alıyoruz. Bu çok garip bir davranış. Dokümantasyona biraz daha dikkatli baktığımızda, varsayılan

**ConsoleAppender** kurucusunun "yapılandırılmamış" olduğunu görüyoruz. Bu **Log4j**'de hata veya en azından bir tutarsızlık gibi görünüyor.

En sonunda aşağıdaki basit birim testini kodladık:

```
public class LogTest {
    private Logger logger;    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }
    @Test
```

```

public void addAppenderWithStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("addAppenderWithStream");
}
@Test
public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n")));
    logger.info("addAppenderWithoutStream");
}
}

```

Artık basit bir konsol **logger**ını nasıl başlatacağımızı biliyoruz. Bu bilgiyi de kendi **logger** sınıfımıza koyarak, uygulamanın geri kalanının **Log4j** sınır arayüzünden izole edilmesini sağlayabiliriz.

Öğrenme testlerinin maliyeti yoktur. API'yi zaten öğrenmek zorundayızdır ve bu testleri yazmak bu bilgiyi elde etmenin kolay ve izole bir yoludur. Öğrenme testleri uzun vadede de pozitif bir getiri sağlar; üçüncü taraf yazılımların yeni sürümleri çıktığında, davranış farklılıklarının olup olmadığını görmek için öğrenme testlerini kullanırız.

Entegre olduktan sonra, üçüncü taraf kodun ihtiyaçlarımızla uyumlu kalacağına dair bir garantimiz yoktur. Yazarları, kodlarını kendi ihtiyaçlarını karşılamak üzere değiştirmek zorunda kalırlar, hataları düzeltirler veya yeni yetenekler eklerler. Kullananlara ise her **release** ile birlikte yeni riskler gelecektir. Testlerimize uyumlu olmayan değişiklikleri öğrenme testlerimiz sayesinde hemen bulabiliriz.

Öğrenme testlerinin sağladığı bilgiye ihtiyacınız olsun veya olmasın, üretim kodunun yaptığını yapan testleriniz tarafından uygulamanızda temiz bir sınır çizilmelidir.

Birkaç yıl önce, bir radyo iletişim sistemi için yazılım geliştiren bir takım içindeydim. Hakkında çok az bildiğimiz **Transmitter** isimli bir alt sistem vardı ve bu alt sistemden sorumlu kişiler arayüzünü tanımlama noktasına gelmemişlerdi. Engellenmek istemediğimizden, kodun bilinmeyen kısmından uzaklaşmaya başladık.

Dünyamızın nerede bittiğini ve yenisinin nerede başladığını oldukça iyi biliyorduk. Çalışırken bazen bu sınıra karşı koyduk. Sisler ve cehalet bulutları sınırların ardını gizlese de, çalışmalarımız sınır arayüzünün nasıl olmasını gerektiğini farketmemizi sağladı. **Transmitter**'dan şöyle bir beklentimiz vardı:

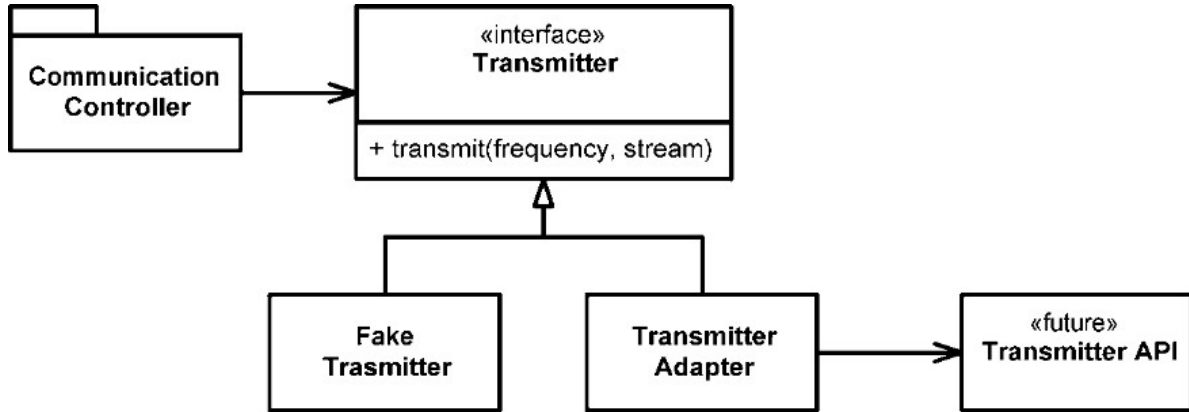
*“Vericiyi verilen frekansta tuşla ve bu akıştan gelen verilerin analog bir temsilini yayınla.”*

API'nin henüz tasarlanmamış olması nedeniyle nasıl yapılacağını bilmiyorduk. Bu yüzden detayları daha sonra çözmeye karar verdik.

Kendi arayüzümüzü tanımladık, adına **Transmitter** benzeri bir şey koyduk. Bu arayüze, sıklığı (frequency) ve veri akışı (data stream) alan bir metot ekledik. İstedığımız arayüz buydu.

Bu arayüz bizim kontrolümüz altındaydı. İstemci kodunu daha okunaklı tutmaya ve sadece yapmak istediğiniz şeye odaklanmanıza yardımcı oluyordu.

Aşağıdaki şekilde, kontrolümüz dışında olan ve tanımlanmamış olan API'den **CommunicationsController** sınıflarını yalıtığımızı görebilirsiniz. Kendi uygulamamıza özgü arayüzü kullanarak, **CommunicationsController** sınıfımızı daha temiz ve verimli hale getirmiş olduk. **Transmitter** API'si tanımlandıktan sonra, boşluğu kapatmak için **TransmitterAdapter** yazdık. **ADAPTER2**, API ile olan etkileşimi sarmalıyor ve API geliştikçe değiştirmemiz gereken tek yer burası olmuş oluyordu:



Bu tasarım aynı zamanda test kodunda da uygun bağlantı noktası sağlar. Uygun bir **FakeTransmitter** kullanarak, **CommunicationsController** sınıflarını test edebiliriz. Ayrıca, API'yi doğru kullandığımızdan emin olduğumuz **Transmitter API**'ye sahip olduğumuzda, sınır testleri de oluşturabiliriz.

## Temiz Sınırlar

Sınırlarda değişimler olur. İyi yazılımlar, büyük yatırımlar ve yeniden çalışmalar olmadan değişikliklere uyum sağlarlar. Kontrolümüz dışında olan kodları kullandığımızda, yatırımımızı korumak için özel bir dikkat göstermeli ve gelecekte yapılacak değişikliklerin çok pahalı olmayacağından emin olmalıyız.

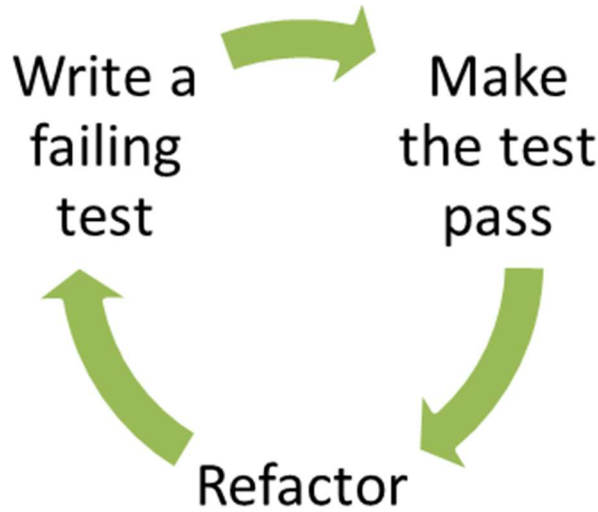
Sınırlardaki kod, beklentileri tanımlayan kesin ayırımlara ve testlere ihtiyaç duyar. Kodumuzun üçüncü taraf yazılımların ayrıntılarıyla ilgili çok fazla şey bilmesini önlemeliyiz. Kontrol etmediğimiz bir şeyden çok, kontrol ettiğimiz bir şeye güvenmek daha iyidir, çünkü sonunda o bizi kontrolü altına alacaktır.

## Bölüm 9 - Birim Testleri

Mesleğimiz son yıllarda uzun bir yol kat etti. 1997'de hiç kimse **Test Driven Development** (TDD) kavramını bilmiyordu. Büyük çoğunluğumuzun birim testleri, programlarımızın çalışıp çalışmadığından emin olmak için yazdığı kodlardı. Özenle sınıflarımızı ve metotlarımızı yazar ve sonra bunları test etmek için bazı özel kodlar yazardık.

Bugünlerde, yazdığım her satır kodun beklediğim gibi çalıştığından emin olmak için test yazıyorum. Geçen testlerimi yazdıktan sonra ise, yazdığım kodla sonradan çalışacak herkesin de çalıştırmasına uygun olacak şekilde olduklarından emin oluyorum.

Agile ve TDD hareketleri birçok programcıyı otomatik birim testleri yazmaya teşvik etti ve her geçen gün daha fazla programcı bu hareketlere katılıyorlar. Ancak daha gidilecek çok yol var. Kodlama disiplini test yazmayı eklemekte acele edince, birçok programcı iyi testler yazmanın incelikli ve önemli noktalarından bazılarını kaçırdılar.



### TDD'nin 3 Kuralı

Herkes **TDD**'nin üretim kodundan önce birim testleri yazmamızı istediğini bilir. Ancak bu kural buzdağının sadece görünen kısmıdır:

1. Geçmeyen bir birim testi yazmadan, üretim (uygulama) kodu yazmamalısın.
2. Aynı anda birden fazla geçmeyen birim testi yazmamalısın. Derleme hatası da geçmeyen test demektir.
3. O andaki geçmeyen testi geçirecek üretim kodundan başka üretim kodu yazmamalısın.

Bu üç yasa sizi otuz saniye uzunluğunda bir döngüye sokar. Testler ve üretim kodu birlikte yazılmışlardır. Sadece, testler üretim kodundan bir kaç saniye öncedir.

Bu şekilde çalışırsak, her gün düzinelerce, her ay yüzlerce ve her yıl binlerce test yazmış oluruz. Ve testlerimiz hemen hemen tüm üretim kodumuzu kapsar. Üretim kodunun boyutuna rakip olabilecek test kodu, yıldırıcı bir yönetim problemi oluşturabilir.

## Testlerimizi Temiz Tutmak

Birkaç yıl önce, test koduna üretim koduyla aynı kalite ve standartlarda bakım yapılması gerekmediğine karar veren bir ekibe koçluk yapmam istendi. Değişkenlerin iyi adlandırılması, test metotlarının kısa ve açıklayıcı olması gerekmiyordu. Test kodlarının iyi tasarlanmış ve üzerinde düşünülmüş bir şekilde bölünmesine de ihtiyaç yoktu. Testler çalıştığı ve üretim kodunu kapsadığı sürece sorun yoktu.

Hiç test olmamasındansa kirli testlerin olmasının daha iyi olduğunu söyleyebilirsiniz. Ancak bu ekibin farkında olmadığı şey, kirli testlerin bulunmasının -daha kötü olmasa da- hiç test olmaması ile eşdeğer olmasıydı. Sorun şuydu ki; üretim kodları geliştikçe testler de değişmeliydi. Ve testler ne kadar kirliyse, değiştirmesi de o kadar zor oluyordu. Üretim kodunu değiştirirken eski testler geçmemeye başlar, ve test kodundaki karışıklık bu testlerin tekrar geçirilmesini zorlaştırır. Böylece testler gittikçe artan bir yükümlülük gibi görülür.

Sürüm çıkardıkça testleri yazma maliyeti arttı ve nihayetinde geliştiricilerin tek şikayet ettiği şey haline geldi. Yöneticiler tahminlerin (estimations) neden bu kadar arttığını sorduğunda, testleri suçluyorlardı. Sonunda testleri tamamen çıkarmaya zorlandılar. Ancak testler olmadan, sistemlerinin bir kısmındaki değişikliklerin diğer kısımları bozmadığından emin olamazlardı. Bu nedenle hata (defect/bug) oranı artmaya başladı. Hata sayısı arttıkça da, üretim kodunda değişiklikler yapmaya çekinmeye başladılar. Çünkü iyilikten çok zararı olmasından korkuyorlardı. Üretim kodları çürümeye başladı. Sonunda elimize hiç testi olmayan, karışık ve hatalarla dolu üretim kodu, sinirli müşteriler ve test çabalarının başarısız olduğuna dair hisler kaldı.

Bir bakıma haklılardı. Onların test yazma çabaları başarısız olmuştu. Ancak bu başarısızlığın sebebi, testlerin dağınık olmasına izin verme kararlarıydı. Testlerini temiz tutmuş olsaydılar, test yazma çabaları başarısız olmazdı.

Hikayenin özü basit:

Test kodu, üretim kodu kadar önemlidir ve üretim kodu kadar da temiz tutulmalıdır. Testler ikinci sınıf vatandaş mualemesi görmemelidir. Test yazmak düşünce, tasarım ve dikkat gerektirir.

Testlerinizi temiz tutmazsanız, onları kaybedersiniz. Ve onlar olmaksızın, üretim kodunu esnek tutan şeyleri de kaybedersiniz. Evet, kodumuzu esnek, bakımlı ve tekrar kullanılabilir tutan şey birim testlerimizdir. Testleriniz varsa, kodda değişiklikler yapmaktan korkmazsınız. Testleriniz olmadan, her değişiklik muhtemel bir hatadır ve mimariniz ne kadar esnek olursa olsun değişiklik yapmaktan korkarsınız.

Testlerinizi yazdıkça ve test kapsam yüzdesi yükseldikçe değişiklik yapmaktan daha az korkarsınız. Hatta korkmadan mimarinizi ve tasarımınızı bile geliştirebilirsiniz. Tasarımınızı ve mimarinizi olabildiğince temiz tutmanın anahtarı üretim kodunu kapsayan otomatik (automated) birim testlere sahip olmaktır.

## Temiz Testler

Bir testi şu 3 şey temiz yapar:

1. Okunabilirlik
2. Okunabilirlik
3. Okunabilirlik

Okunabilirliğin birim testlerde olması belki de üretim kodunda olmasından daha önemlidir. Peki testleri ne okunabilir yapar? Tüm kodu okunabilir kılan aynı şeyler: açıklık, basitlik ve ifade yoğunluğu.

Şu koda bir bakalım. Üç test de anlaması zor.

Öncelikle `addPage` ve `assertSubString` metoduna yapılan çağrılarda çok sayıda tekrarlanmış kod var. Daha da önemlisi, bu kod testin ifade edilebilirliğine müdahale eden ayrıntılarla dolu:

```
public void testGetPageHierarchyAsXml() throws Exception {
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));
    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();
    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}

public void
testGetPageHierarchyAsXmlDoesntContainSymbolicLinks()
throws Exception {
    WikiPage pageOne = crawler.addPage(root,
    PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));
    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
}
```

```

        WikiPageProperty symLinks =
properties.set(SymbolicPage.PROPERTY_NAME);
        symLinks.set("SymPage", "PageTwo");
        pageOne.commit(data);
        request.setResource("root");
        request.addInput("type", "pages");
        Responder responder = new SerializedPageResponder();
        SimpleResponse response =
            (SimpleResponse) responder.makeResponse(
                new FitNesseContext(root), request);
        String xml = response.getContent();
        assertEquals("text/xml", response.getContentType());
        assertSubString("<name>PageOne</name>", xml);
        assertSubString("<name>PageTwo</name>", xml);
        assertSubString("<name>ChildOne</name>", xml);
        assertNotSubString("SymPage", xml);
    }public void testGetDataAsHtml() throws Exception {
        crawler.addPage(root, PathParser.parse("TestPageOne"), "test
page");
        request.setResource("TestPageOne");
        request.addInput("type", "data");
        Responder responder = new SerializedPageResponder();
        SimpleResponse response =
            (SimpleResponse) responder.makeResponse(
                new FitNesseContext(root), request);
        String xml = response.getContent();
        assertEquals("text/xml", response.getContentType());
        assertSubString("test page", xml);
        assertSubString("<Test", xml);
    }
}

```

Örneğin, **PathParser** çağrılarına bakalım. **String**leri, tarayıcılar tarafından kullanılan **PagePath** örneklerine (instance) dönüştürüyorlar. Bu dönüşüm, testle tamamen alakasız. **Responder**'ı yaratmak ve cevabı toplama ve aktarma ile ilgili ayrıntılar da sadece gürültü. Sonuç olarak, kod okunmak için tasarlanmamış.

Şimdi ise yeniden düzenlediğim, çok daha temiz ve açıklayıcı olan şu versiyonuna bakalım:

```

public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>",
        "<name>ChildOne</name>"
    );
}public void testSymbolicLinksAreNotInXmlPageHierarchy() throws
Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");
}

```

```

        addLinkTo(page, "PageTwo", "SymPage");
        submitRequest("root", "type:pages");
        assertResponseIsXML();
        assertResponseContains(
            "<name>PageOne</name>", "<name>PageTwo</name>",
            "<name>ChildOne</name>"
        );
        assertResponseDoesNotContain("SymPage");
    } public void testGetDataAsXml() throws Exception {
        makePageWithContent("TestPageOne", "test page");
        submitRequest("TestPageOne", "type:data");
        assertResponseIsXML();
        assertResponseContains("test page", "<Test>");
    }
}

```

BUILD-OPERATE-CHECK deseni, testlerin bu yapısıyla açıkça görülüyor. Testlerin her biri üç kısma ayrılmış. İlk kısım test verisini oluşturuyor, ikinci kısım bu test verisini işliyor ve üçüncü kısım ise işlemin beklenen sonuçları verip vermediğine bakıyor.

## Çifte Standart

Test API'si içindeki kod, üretim kodundan farklı mühendislik standartlarına sahiptir. Basit, öz ve açıklayıcı olmalıdır, ancak üretim kodu kadar verimli olmasına gerek yoktur. Sonuçta, üretim ortamında değil test ortamında çalışır ve bu iki ortamın çok farklı ihtiyaçları vardır.

Aşağıdaki koda bakalım. Bu testi, prototipini yaptığım bir çevre kontrol sisteminin bir parçası olarak yazmıştım. Ayrıntılara girmeden, bu testin, sıcaklık “çok soğuk” olduğunda düşük sıcaklık alarmının, ısıtıcının ve körüğün hepsinin açık olup olmadığını kontrol ettiğini söyleyebiliriz:

```

@Test
public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}

```

Bir sürü detay var. Örneğin `tic()` metodu ne yapıyor? Ya da örneğin gözleriniz önce `heaterState()` metoduna ardından `assertTrue()` 'ya bakıyor, veya `coolerState()` metoduna bakıyor ve `assertFalse()` metodunu kontrol ediyor. Bu durum testi okumayı zorlaştırıyor.

Kodu geliştirerek testin okunabilirliğini şu şekilde arttırdım:



```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

Bir `wayTooCold()` fonksiyonu yazarak `tic()` fonksiyonu çağrısını gizledim. Fakat dikkat etmemiz gereken asıl nokta, `assertEquals`'daki garip *String*. Bu *String*'de büyük harf "açık", küçük harf ise "kapalı" anlamına geliyor ve harfler daima şu sırada bulunuyor: heater, blower, cooler, hi-temp-alarm, lo-temp-alarm. Anlamını bildikten sonra, gözleriniz o *String*'i geçer ve sonuçları çabucak yorumlayabilirsiniz.

Şu koda bakalım ve testleri anlamamanın ne kadar kolay olduğunu görelim:

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
}@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
}@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

`getState` metodu şu şekilde:

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

Çok verimli bir kod değil. Kodu verimli hale getirebilmek için bir ***StringBuffer*** kullanmış olmalıydım. ***StringBuffer***'lar biraz çirkindir ve maliyeti az olsa bile kullanmaktan kaçınırım. Evet, buradaki maliyetin de çok az olduğunu söyleyebilirsiniz. Bu uygulama gömülü gerçek

zamanlı bir sistem ve kaynaklarının çok kısıtlı olması da muhtemel. Ancak test ortamı kaynaklarının kısıtlı olması mümkün değil.

Çifte standardın doğası budur. Üretim ortamında asla yapmayacağımız fakat test ortamında uygulayabileceğimiz şeyler vardır. Genellikle bu şeyler bellek veya CPU verimliliği meseleleridir.

## Test Başına Bir Doğrulama (Assert)

Her testin sadece bir **assert** ifadesi olması çok zorlu görünebilir ancak kodun anlaşılabilirliğini ne kadar kolaylaştırdığını yukarıdaki örneğimizde gördük.

Örneğimizin ilk düzenlenmiş versiyonunu şu şekilde yeniden düzenledim:

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldBeXML();
}public void testGetPageHierarchyHasRightTags() throws Exception
{
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldContain(
        "<name>PageOne</name>", "<name>PageTwo</name>",
        "<name>ChildOne</name>"
    );
}
```

Fonksiyonların adlarını **given-when-then** olarak standardize ettim. Testlerin okunmasını daha da kolaylaştırıyor. Ancak testleri bu şekilde bölmek de bir çok kodun tekrarlanmasına sebep oluyor.

[Template Method](#) desenini kullanarak ve `given/when` kısmını temel (base) bir sınıfa, `then` kısmını da farklı türev sınıflara koyarak kod tekrarlanmasını ortadan kaldırabiliriz. Veya tamamen ayrı bir test sınıfı oluşturabilir ve `given/when` kısmını `@Before` fonksiyonuna ve `then` kısımlarının her birini ayrı birer `@Test` fonksiyonu içine koyabiliriz. Ancak bu, küçük bir mesele için çok büyük değişiklik yapmak demek oluyor. Günün sonunda birden fazla `assert` olan versiyonunu tercih ediyorum.

Bence tek `assert` kuralı iyi bir pratik. Testlerime genellikle tek sayıda `assert` koymaya çalışsam da, bazı zamanlar birden fazlasını koymaktan da çekinmiyorum. Bence, söyleyebileceğimiz en iyi şey; bir testteki `assert` sayısının en aza indirilmesi gerektiğidir.

## Test Başına Tek Bir Konsept

Her test fonksiyonunda tek bir konsepti test etmek istiyoruz. Birinin ardından bir diğer şeyi test eden testler istemiyoruz.

Aşağıda böyle bir teste ait örnek var. Bu test üç kısma ayrılmalı çünkü üç bağımsız şeyi test ediyor. Hepsini aynı fonksiyonda birleştirmek, okuyucuyu, her bölümün neden orada olduğunu ve o bölüm tarafından nelerin test edildiğini anlamaya zorlar:

```
/**
 * Miscellaneous tests for the addMonths() method.
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());    SerialDate d3 =
SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());    SerialDate d4 =
SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Ayrırmak istediğimiz üç test fonksiyonu muhtemelen şu şekilde olmalıdır:

*Elimizde 31 günü olan aylardan birinin son günü olsun (örneğin 31 Mayıs);*

1) 30 günü olan bir ay eklediğimizde (Haziran gibi), tarih ayın 31'i değil 30'u olmalıdır.

2) İki ay eklediğimizde, ikinci ayın 31 günü varsa , tarih ayın 31'i olmalıdır.

*Elimizde 30 günü olan aylardan birinin son günü olsun;*

3) 31 günü olan bir ay eklediğimizde tarih ayın 31'i değil 30'u olmalıdır.

Yani soruna neden olan şey birden fazla **assert** ifadesi değil. Aksine, test edilen birden fazla konseptin olmasıdır. Bu nedenle en iyi kural, konsept başına **assert** sayısını en aza indirmek ve test fonksiyonu başına sadece bir konsept test etmektir.

## F.I.R.S.T Kuralı

Temiz testler, F.I.R.S.T kısaltmasını oluşturan şu beş kuralı takip eder:

**F**ast: Testler hızlı olmalıdır. Testler yavaş çalıştıklarında, sık sık çalıştırmak istemezsiniz.

Testleri sık sık çalıştırmazsanız, problemleri kolayca giderecek kadar erkenden fark edemezsiniz.

**I**ndependent: Testler birbirine bağlı olmamalıdır. Bir test, bir sonraki testin koşullarını

belirlememelidir. Her bir testi bağımsız olarak ve istediğiniz herhangi bir sırada çalıştırabilmelisiniz. Testler birbirine bağımlı olduğunda, hata alan ilki, hiyerarşide aşağılara doğru hatalara sebep olarak ilk hata alınan yerin tespitini zorlaştıracaktır.

**R**epeatable: Testler herhangi bir ortamda çalışabilir olmalıdır. Birim testlerini üretim

ortamında, QA ortamında ve dizüstü bilgisayarınızda veya trende evinize gidiyorken çalıştırabilmelisiniz. Testleriniz herhangi bir ortamda tekrarlanabilir değilse, başarısız olmalarına hep bir mazeretiniz olacaktır.

**S**elf-Validating: Testler ya geçerler ya da başarısız olurlar. Testlerin geçip geçmediğini

anlamak için bir **log** dosyasına bakmamıza gerek olmamalı veya elle iki farklı metin dosyasını karşılaştırmanız gerekmemelidir.

**T**imely: Testlerin zamanında yazılması gerekir. Birim testleri, yazıldığı üretim kodundan

hemen önce yazılmalıdır. Testlerinizi kodunuzu yazdıktan sonra yazarsanız, üretim kodununun test edilmesi zor olabilir. Kodunuzu test edilebilecek şekilde tasarlayamayabilirsiniz.

## Sonuç

Testler, bir projenin sağlığı açısından üretim kodu kadar önemlidir. Belki daha da önemlidir, çünkü testler üretim kodunun esnekliğini, bakımını ve tekrar kullanılabilirliğini korur ve geliştirir. Bu yüzden testlerinizi hep temiz tutun. Onları az ve öz hale getirmek için çalışın. Testlerinizin çürümesine izin verirsiniz, kodunuz da çürür.

## Bölüm 10 — Sınıflar

### Sınıf Düzeni

**Java** kodlama standartlarına (code conventions) göre, bir sınıf değişkenlerle başlamalıdır. Eğer varsa, **public static** sabitler önce gelmelidir. Ardından **private static** değişkenler ve onu takip eden **private instance** (örnek) değişkenleri. **public** bir değişkene sahip olmak için gerçekten iyi bir sebebiniz olmalıdır.

Değişkenlerden sonra **public** fonksiyonlar gelmelidir. **public** fonksiyonlardan hemen sonra ise **public** bir fonksiyon tarafından çağırılmış yardımcı **private** metotlar gelebilir.

### Kapsülleme (Encapsulation)

Değişkenlerimizi ve **util** fonksiyonlarımızı gizli tutmak isteriz, ancak bazen bir değişkeni veya **util** metodu testlerden erişilebilmesi için **protected** yapmamız gerekebilir. (Uncle Bob, bir metodun testinin olmasının o metodun kapsüllenmesinden daha önemli olduğuna vurgu yapıyor.)

### Sınıflar Küçük Olmalıdır

Sınıf tasarımı konusundaki ilk kural sınıfların küçük olmaları gerektiğidir. Buradaki sorumuz, ne kadar küçük?

Fonksiyonların boyutuna, satır sayılarına bakarak karar verdik; ancak sınıfların boyut ölçümünün daha farklı bir birimi vardır: sorumluluklarının sayısı.

Aşağıdaki örneğe ait sınıfta 70 metot var, çok çok büyük:

```
public class SuperDashboard extends JFrame implements
MetadataUser
{
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
}
```

```

public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(MetaObject target, MetaObject pasted,
                          MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPage(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()

```

```

public void setAçowDragging(boolean allowDragging)
public boolean allowDragging()
public boolean isCustomizing()
public void setTitle(String title)
public IdeMenuBar getIdMenuBar()
public void showHelper(MetaObject metaObject, String
propertyName)
    // ... many non-public methods follow ...
}

```

Peki **SuperDashboard** sınıfının sadece aşağıdaki 5 metoda sahip olabileceğini söyleseydik?

```

public class SuperDashboard extends JFrame implements
MetadataUser {
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}

```

Bu durumda bile, az sayıda metoduna rağmen **SuperDashboard**'un sorumlulukları çok fazladır.

Bir sınıfın adı, hangi sorumlulukları yerine getirdiğini tanımlamalıdır. Aslında bir bakıma, adlandırma muhtemelen sınıf boyutunu belirlemeye yardımcı olan ilk şey olacaktır. Bir sınıf için kısa bir isim bulamıyorsak, muhtemelen sınıf çok fazla iş yapıyordur. Sınıf ismi ne kadar belirsiz olursa, sorumlulukları o kadar çok olur. Örneğin, **Processor**, **Manager** veya **Super** gibi kelimeleri içeren sınıflara muhtemelen gereğinden çok sorumluluk yüklenmiştir.

Ayrıca sınıfın kısa bir açıklamasını **eğer**, **ya da** veya **fakat** kelimelerini kullanmadan 25 kelimeyle yazabilmeliyiz. Bu kelimeleri kullanmamız demek, sorumlulukların fazlaşması demektir.

**Single Responsibility Prensibi (SRP)**, bir sınıf veya modülün değiştirilmesi için gereken tek bir sebebi olması gerektiğini söyler. Bu ilke bize hem bir sorumluluk tanımı verir hem de sınıf büyüklüğü için bir rehber olur. Sınıfların sadece bir sorumluluğu olmalıdır.

5 metotlu **SuperDashboard** sınıfı iki sorumluluğa sahiptir. İlki, sürüm bilgisini kontrol etmek; ikincisi ise **Java Swing** bileşenlerini kontrol etmektir.

**Swing** kodunu değiştirirsek sürüm numarasını güncellemek isteyeceğiz, ancak tersi pek de doğru değil. Sürüm bilgisini sistemdeki diğer kod değişikliklerine dayanarak değiştirebiliriz. Sınıftaki sorumlulukları tanımlamaya çalışmak, kodumuzdaki soyutlamaları daha iyi tanımamıza ve oluşturmamıza yardımcı olur.

Sürüm bilgisi ile uğraşan her üç **SuperDashboard** metodunu **Version** adlı ayrı bir sınıfa kolayca çıkarabiliriz. **Version** sınıfı, diğer uygulamalarda yeniden kullanılabilme potansiyeline sahip bir yapıdır:

```
public class Version {  
    public int getMajorVersionNumber()  
    public int getMinorVersionNumber()  
    public int getBuildNumber()  
}
```

**SRP**, nesne yönelimli programlamada en önemli kavramlardan biridir. Anlaşılması ve uygulanması da kolaydır. Ancak bu kavram en çok istismar edilen ilkelerden biridir. Sürekli çok fazla şey yapan sınıflarla karşılaşırız. Peki neden?

Yazılımın çalışması ve yazılımın temiz kalması çok farklı iki etkinliktir. Çoğumuz, kodumuzun organizasyonundan ve temizliğinden daha çok çalışmasına odaklanıyoruz. Sorun şu ki, birçoğumuz program çalıştıktan sonra işimizin bittiğini düşünüyor. Organizasyon ya da temizlik kaygısı duymuyoruz. Geriye dönüp aşırı doldurulmuş sınıfları tek sorumlulukları olan ayrı birimler haline getirmek yerine bir sonraki soruna geçiyoruz. Aynı zamanda birçok geliştirici çok sayıda küçük, tek amaçlı sınıfın daha büyük resmin anlaşılmasını zorlaştırmasından korkuyor. Büyük resmin nasıl tamamlandığını anlamak için sınıftan sınıfa geçmek konusunda endişe duyuyorlar. Birçok küçük sınıfa sahip bir sistem, birkaç büyük sınıfa sahip bir sistemden daha fazla hareketli parçaya sahip değildir.

Büyük sınıflar içerilerinde çok fazla bilgi barındırır. Burada soru şudur: Her biri iyi tanımlanmış ve güzel etiketleri olan bileşenleri içeren birçok küçük çekmeceye mi sahip olmak istersiniz? Yoksa her şeyi içine attığınız birkaç büyük çekmeceye mi?

Her büyük sistem aynı büyüklükte mantık ve karmaşıklık içerecektir. Yeniden ifade etmek gerekirse; sistemlerimizin birkaç büyük sınıftan değil, birçok küçük sınıftan oluşmasını istiyoruz. Her küçük sınıfın, tek bir sorumluluğu ve değiştirilmesi için tek bir nedeni olmalıdır. Ve istenileni verebilmesi için başka sınıflarla birlikte çalışır.

## Birbirine Bağlılık (Cohesion)

Sınıflar az sayıda örnek değişkene (instance variable) sahip olmalıdır. Bir sınıfın her bir metodu, bu değişkenlerden bir veya daha fazlasını değiştirmelidir. Her bir değişkenin her bir metot tarafından kullanıldığı bir sınıf, maksimum düzeyde birbirine bağlılığa sahiptir.

Bu derece bütünleşebilen sınıflar yaratmak ne tavsiye edilir ne de mümkündür. Ancak yine de birbirine bağlılığın yüksek olmasını isteriz.





Aşağıda **Stack**'in gerçekleştirimine ait bir örnek var. Bağımlılığı oldukça yüksek bir sınıf. Sadece `size()` metodu her iki değişkeni kullanmıyor:

```
public class Stack {
    private int topOfStack = 0;
    List < Integer > elements = new LinkedList < Integer > ();
    public int size() {
        return topOfStack;
    }
    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }
    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

Fonksiyonları ve parametre listelerini kısa tutma stratejisi, bazen bir dizi metot tarafından kullanılan örnek değişkenlerin çoğalmasına sebep olabilir. Değişkenleri ve metotları, birbirine daha bağlı olacak şekilde iki veya daha fazla sınıfa ayırmaya çalışmalıyız.

Büyük fonksiyonları daha küçük fonksiyonlara dönüştürme eylemi sınıfların çoğalmasına neden olur. İçinde birçok değişken tanımlanmış büyük bir fonksiyon düşünelim. Diyelim ki, bu fonksiyonun küçük bir bölümünü ayrı bir fonksiyona çıkarmak istiyorsunuz. Bir şekilde ayıklamak istediğiniz kod, fonksiyonda tanımlanan dört değişken kullanıyor olsun. Bu değişkenlerin hepsini argüman olarak yeni fonksiyona aktarmamız gerekir mi?

Hayır. Bu dört değişkeni örnek değişken olarak tanımlamış olsaydık, herhangi bir argüman geçirmeden kodu ayırabilirdik. Ve o zaman fonksiyonun küçük parçalara bölünmesi daha kolay olurdu. Ne yazık ki bu da sınıflarımızın birbirine bağıllığı kaybetmesi anlamına geliyor; çünkü sadece birkaç fonksiyonun paylaşmasına izin vermek için daha fazla örnek değişken eklemek gerekecektir. Ancak bu değişkenleri paylaşmak isteyen sadece birkaç fonksiyon varsa, bu onları belirli bir sınıf yapmaz mı? Elbette yapar. Sınıflar birbirine bağıllığı kaybettiğinde, onları bölün.

Bu nedenle, büyük bir fonksiyonu daha küçük fonksiyonlara dönüştürmek bize genellikle birkaç küçük sınıfa bölme fırsatı verir. Bu, programımıza daha iyi bir organizasyon ve daha şeffaf bir yapı kazandırır. Ancak, daha uzun ve tanımlayıcı isimler kullanacağımız ve programı okunabilir tutmak için boşluk ve biçimlendirme teknikleri uygulayacağımız için programımız çok daha uzun hale gelmiş olur.

## Değişim İçin Düzenleme

Çoğu sistem için değişim sürekli dir. Her değişiklik, sistemin geri kalanının artık amaçlandığı gibi çalışmaması riskini bize yükler. Temiz bir sistemde, değişim riskini azaltmak için sınıflarımızı sürekli yeniden düzenleriz.

Aşağıdaki **Sql** sınıfı SQL cümleleri oluşturmak için kullanılıyor:

```
public class Sql {
    public Sql(String table, Column[] columns)
    public String create()
    public String insert(Object[] fields)
    public String selectAll()
    public String findByKey(String keyColumn, String keyValue)
    public String select(Column column, String pattern)
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[]
columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

Şimdilik **update** işlemini desteklemiyor ancak **update** işlevini de eklememiz gerektiğinde, sınıfı değiştirmemiz gerekecek. Hatta **subselect**leri desteklemesi için **select** ifadesini değiştirmemiz gerektiğinde de değişmesi gerekecektir. Sınıftaki herhangi bir değişiklik, diğer kodları bozma potansiyeline sahip olduğundan, sınıfın en baştan tekrar test edilmesi gerekiyor. Bu sebeple bu sınıf **Single Responsibility** prensibini ihlal ediyor.

**SRP** ihlalini basit bir bakış açısıyla tespit edebiliriz: `selectWithCriteria` gibi yalnızca `select` ifadesine ait `private` metodlar vardır.

Aşağıdaki gibi bir çözüm uyguladım. Koddaki her `public` metodu, **Sql** sınıfının kendi türevinden olan sınıflara, `valuesList` gibi özel metotları ise doğrudan erişilebilecek yerlere taşıdım:

```
abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[]
fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[]
columns)
}public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(String table, Column[] columns,
Criteria criteria)
    @Override public String generate()
}public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(String table, Column[] columns,
Column column, String pattern)
    @Override public String generate()
}public class FindByKeySql extends Sql {
    public FindByKeySql(String table, Column[] columns, String
keyColumn, String keyValue)
    @Override public String generate()
}public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate()
    private String placeholderList(Column[] columns)
}public class Where {
    public Where(String criteria)
    public String generate()
}public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}
```

Kod inanılmaz basit bir hale geldi. Çok kısa sürede neler olup bittiğini anlayabiliyoruz. Bir fonksiyonun bir başkasını bozabilme riski neredeyse ortadan kalktı.

**update** eklememiz gerektiğinde de, mevcut sınıfların hiçbirinin değiştirilmesi gerekmez. **UpdateSql** adlı yeni bir **Sql** alt sınıfı ekleyerek orada güncelleme işlemlerini gerçekleştirebiliriz. Ve bu değişiklik yüzünden sistemdeki başka hiçbir kod kesintiye uğramaz.

Yeni **SqI** sınıfımız iki temel kuralı da destekler: **SRP** (Single Responsibility Principle) ve **OCP** (Open/Closed Principle).

Sınıflar gelişime açık, değişime kapalı olmalıdır.

## Değişimden İzole Etme

İhtiyaçlar değişecek, bu nedenle kod da değişecektir. Gerçekleştirimleri içeren somut (concrete) sınıflarımız, gerçekleştirimleri sadece sunan soyut (abstract) sınıflarımız olduğunu biliyoruz. Somut sınıflarımıza bağımlı istemcilerimiz varsa, bu detaylar değiştiğinde bu istemciler de risk altındadır. Bu etkiyi yalıtmak için arayüzleri (interface) ve soyut sınıfları kullanabiliriz.

Somut sınıflarımızdaki detaylara olan bağımlılıklar, test sistemi için de zorluklar yaratır.

Örneğin bir **Portfolio** sınıfı yaratmak istesek ve o da harici bir **TokyoStockExchange** API'sine bağımlı olsa, testlerimiz her 5 dakikada bir değişen sonuçlardan etkilenecekti. Direkt olarak **TokyoStockExchange** API'sine bağımlı olan bir **Portfolio** sınıfı tasarlamak yerine, **StockExchange** isimli bir arayüz oluşturur ve içine basit bir metot tanımlayabiliriz:

```
public interface StockExchange {
    Money currentPrice(String symbol);
}
```

**TokyoStockExchange** sınıfını bu arayüzün bir gerçekleştirmisi haline getirebilir ve **Portfolio** kurucusunun argüman olarak bir **StockExchange** referansı aldığından da şu şekilde emin olabiliriz:

```
public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}
```

Artık **TokyoStockExchange** sınıfını taklit eden test edilebilir bir **StockExchange** arayüz gerçekleştirmisi yaratabiliriz. Bu test gerçekleştirmisi, testte kullandığımız anlık sembol değerini sabitleyecektir. Örneğin Microsoft'a karşılık gelen "MSFT" sembolü için her zaman \$100 dönmelerini sağlayıp, 5 hisselik bir satın alma işlemi için portföy değerimizin \$500 olmasını bekleyebiliriz:

```
public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;
```

```

@Before
protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
}

@Test
public void GivenFiveMSFTTotalShouldBe500() throws Exception
{
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
}
}

```

Bir sistem bu şekilde test edilebilecek kadar bağımsızsa, aynı zamanda daha esnek olacak ve tekrar kullanılabilirliği daha fazla teşvik edecektir. Daha az bağılılık (coupling), sistemimizin değişimlerden daha izole olması demektir. Bu izolasyon, sistemin her ögesini anlamayı kolaylaştırır.

Bu şekilde bağılılığı en aza indirdiğimizde, sınıflarımız başka bir sınıf tasarım ilkesi olan [Dependency Inversion Principle](#) (DIP) ilkesine de bağlı kalırlar. **DIP** özünde, sınıflarımızın somut ayrıntılar yerine soyutlamalara bağlı olması gerektiğini söyler. **TokyoStockExchange** sınıfının uygulama ayrıntılarına bağımlı olmak yerine **Portfolio** sınıfımız şimdi **StockExchange** arayüzüne bağımlı. **StockExchange**, bir sembolün anlık fiyatını soran soyut konsepti temsil eder. Bu soyutlama, bu fiyatın elde edildiği yer de dahil olmak üzere böyle bir fiyat elde etme ile ilgili tüm ayrıntıları yalıtır.

## Bölüm 11 - Sistemler

Bir şehir yapmak isteseydik, tüm ayrıntıları kendiniz yönetebilir miydiniz? Muhtemelen hayır. Mevcut bir şehri yönetmek bile bir kişi için çok fazladır. Şehirler yine de çalışmaya devam eder, çünkü kentlerin belirli kısımlarını, su sistemlerini, güç sistemlerini, trafiğini, kolluk kuvvetlerini, bina kodlarını vb. yöneten bir takım insanlar vardır. Bazıları bu *büyük resmin* sorumluluğunu üstlenirken diğerleri ayrıntılara odaklanır. Bireylerin ve yönettikleri bileşenlerin *büyük resmi* anlamaksızın etkili bir şekilde çalışmasını mümkün kılan, soyutlama ve modülerlik düzeyleridir.

Temiz kod da, soyutlamanın daha düşük seviyelerinde bunu başarmamıza yardımcı olur.

### Oluşturma Aşamalarının Kullanımdan Ayrılması

*Oluşturma (construction)*, *kullanımdan* çok farklı bir süreçtir. Uygulama nesneleri oluşturulurken ve birbiri ile ilişkilendirilirken, nesnelerin oluşturulma süreci çalışma zamanı mantığından (runtime logic) ayrılmalıdır.

Meselelerin ayrılması (separation of concerns), zanaatımızın (craft) en eski ve en önemli tasarım tekniklerinden biridir. Maalesef çoğu uygulama bu çok faydalı tasarım tekniğini kullanmıyor. Tipik bir örnek:

```
public Service getService() {
    if (service == null)
        service = new MyServiceImpl(...); // Good enough default
    for most cases?
    return service;
}
```

Bu, ihtiyaç halinde oluşturma (lazy initialization/evaluation) deyimidir ve bir çok faydası vardır: Nesneyi kullanmadıkça yapımının (construction) üstesinden gelmek zorunda değiliz ve başlatma sürecimiz çok daha hızlı olabilir. Ayrıca hiçbir zaman `null` dönmediğinden de emin oluruz.

Faydaları olmasına rağmen olumsuz yanları da vardır. Örneğin; `MyServiceImpl` ve yapıcısının (constructor) gerektirdiği her şeye bir bağımlılığımız var. Bağımlılıkları çözmeden kodu derlememiz imkansız. İş mantığı (business logic) ile yapım aşaması birbirine geçmiş olduğundan, testlerimizde hem `null` durumunu, hem de çalışacak olan bloğu test etmeliyiz. Bu sorumlulukların her ikisine birden sahip olması metodun birden fazla şey yaptığını gösterir, bu yüzden bir bakıma *Single Responsibility* prensibini ihlal ediyoruz.

İhtiyaç halinde oluşturma (lazy initialization) deyiminin bir kez kullanımı sorun teşkil etmez ancak uygulamalarda bir çok örneği vardır. Güçlü sistemler oluşturmaya gayret gösteriyorsak, küçük, kullanışlı deyimlerin modülerliği bozmasına asla izin vermemeliyiz. Nesnenin oluşturulması ve birbiri ile ilişkilendirilmesi süreci de bir istisna değildir. Bu işlemleri iş

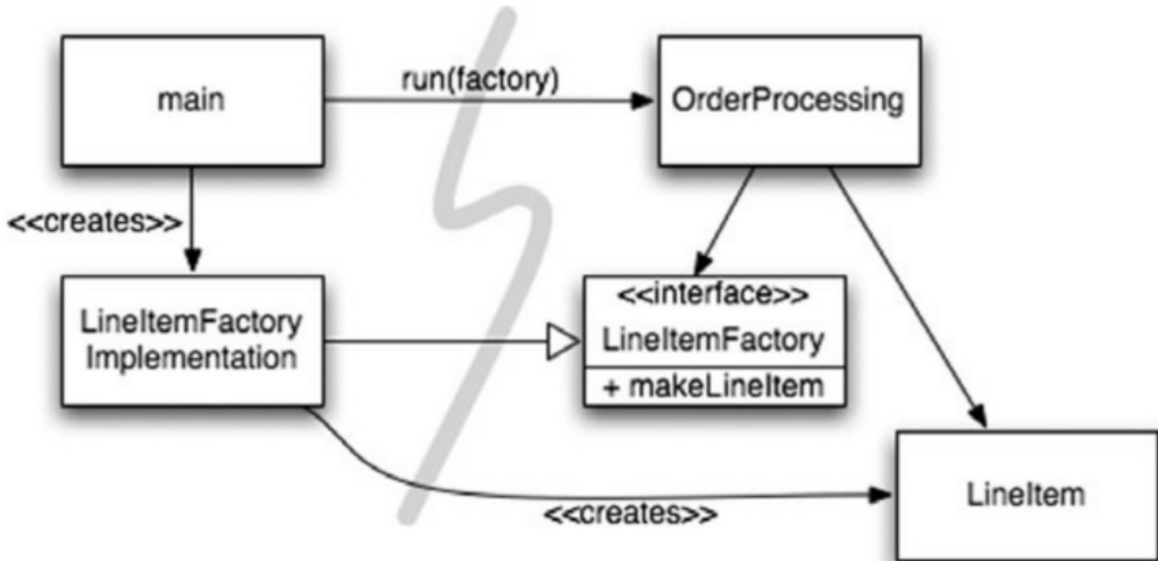
mantığından ayrı olarak modülerize etmeli ve bağımlılıklarımızı çözmek için tutarlı bir stratejimiz olduğundan emin olmalıyız.

## “main”in Ayrılması

**main** fonksiyonu, sistem için gerekli olan tüm nesneleri üretir, ardından oluşturduğu nesneleri onları kullanacak olan uygulamaya geçer. Akış tek yönlü, **main**'den uygulamaya doğrudur. Bu demektir ki, uygulama **main** hakkında hiçbir şey bilmez ve sadece tüm nesnelerin düzgün şekilde oluşturulup oluşturulmadığına bakar.

## Fabrikalar

Bazen bir nesne yaratılmasından uygulamanın sorumlu olmasını isteriz. Örneğin, bir sipariş alma sisteminde uygulama, **Order** nesnesine ekleyebilmek için **LinItem** örneklerini (instances) oluşturmalıdır. Bu durumda **LinItem** örneklerinin oluşturulmasının kontrolünü uygulamaya vermek için **Abstract Factory** desenini kullanabiliriz. Kontrol uygulamada ancak detaylar uygulama kodundan tamamen ayrıdır:



Tüm bağımlılıklar **main**'den **OrderProcessing** uygulamasına doğrudur. Bu, uygulamanın bir **LinItem** nesnesinin nasıl oluşturulacağı ile ilgili detaylardan kopuk olduğu anlamına gelir. Bu detaylar, **main**'in tarafında olan **LinItemFactoryImplementation** sınıfında tutulur. Uygulama, **LinItem** örneklerinin ne zaman oluşturulacağı üzerinde tam kontrole sahiptir, ve hatta uygulamaya özel yapıcı argümanları da sağlayabilir.

## Bağımlılık Enjeksiyonu

Kontrolü tersine çevirme (*Inversion of Control — IoC*) yönteminin bağımlılıkların yönetimi (dependency management) için bir uygulaması olan Bağımlılık Enjeksiyonu (*Dependency Injection — DI*), nesnelerin oluşumunu kullanımdan ayırmada güçlü bir mekanizmadır. **IoC** ikincil sorumlulukları bir nesneden -o işe adanmış- başka nesnelere taşır, ve böylece **Single Responsibility** prensibi desteklenmiş olur. Bağımlılıkların yönetimi konusu özelinde düşünecek olursak, bir nesne kendi bağımlılıklarının örneklerini oluşturma sorumluluğunu almamalıdır. Bunun yerine, bu sorumluluğu başka bir yetkili mekanizmaya, **IoC / DI** mekanizmasına bırakmalıdır.

İyi bir bağımlılık enjeksiyonu mekanizmasında; sınıf, bağımlılıklarını gidermek için doğrudan hiçbir adım atmaz, tamamen pasiftir. Bunun yerine, bağımlılıkları enjekte etmek için kullanılan **setter** metotlar veya yapıcı argümanlar (veya her ikisini) sağlar. Nesnelerin oluşturulma süreçlerinde, **DI** konteyneri gerekli nesneleri örnekler ve bağımlılıkları ilişkilendirmek için sağlanan yapıcı değişkenlerini veya **setter** metotları kullanır. Bağımlılıklar genellikle bir konfigürasyon dosyası aracılığıyla belirtilir. **Spring Framework, Java** için en iyi bilinen **DI** konteynerini sağlar.

## Big Design Up Front

[Big Design Up Front](#) yaklaşımına göre, yazılımın gerçekleştirimi başlamadan önce tasarım tamamlanmalı ya da mükemmelleştirilmelidir. Genellikle yazılım geliştirmedeki şelale (waterfall) modeli ile ilişkilendirilir. **BDUF** kulağa hoş gelse de aslında zararlı bir yaklaşımdır. Önceki eforu çöpe atma konusunda oluşacak olan psikolojik direnç nedeniyle değişime uyum sağlamayı engeller. Bu direnç bazen kişisel, ama çoğu zaman ekonomik sebeplerden kaynaklanır. Eğer uygulama mimarisinde **Seperation of Concerns** sağlanmışsa, radikal değişiklikler dahi ekonomik olarak yapılabilir (feasible) olacaktır. Uygulamaya ait mimari kararların tümünün en başta alınması yerine, olabildiğince basit ancak ayrılmış bir mimari (decoupled architecture) ile başlayarak, çalışan uygulama özelliklerini kısa aralıklarla teslim etmeye odaklanılmalı, mimari iyileştirmeler buna paralel ilerlemelidir.

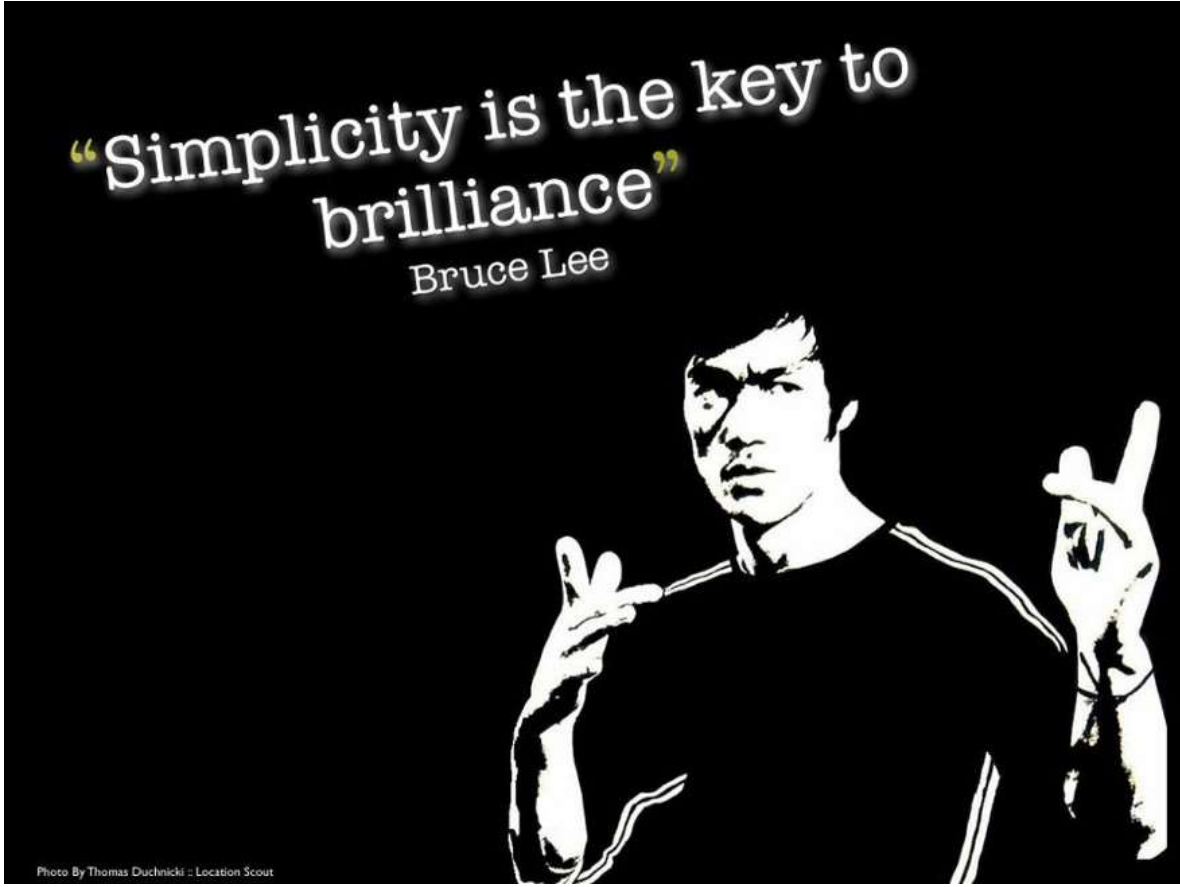
Modülerite ve **Seperation of Concern** dağıtılmış (decentralized) yönetim ve karar vermeyi mümkün kılar. Büyük bir sistemde, bir şehirde ya da bir yazılım projesinde, hiç kimse tüm kararları vermeye yetkin değildir. En iyisi, sorumlulukları en kalifiye kişilere vermek ve kararları mümkün olan son ana kadar ertelemektir. Bu ikisi çelişiyor gibi görünebilir, ancak kararları ertelemek mümkün olan en iyi ve en çok bilgi ile karar vermemizi sağlar. Erken bir karar, optimal olmayan bilgi ile verilmiş bir karardır. Gerçekleştirim kararlarımızı ne kadar erken verirsek, müşteri geri bildirimi ve bu kararlardaki deneyimimiz o kadar az olacaktır.

Sistemler de temiz olmalıdır. Kötü bir mimari iş mantığını (business logic) belirsizleştirir ve çevikliği (agility) olumsuz etkiler. İş mantığı gizlendiğinde, hataların bulunması ve yeni özelliklerin eklenmesi zorlaşır. Çeviklik azalırsa, üretkenlik azalır ve **TDD**'nin faydaları kaybolur.

Soyutlamanın her seviyesinde niyet açık olmalıdır. Sistemler veya bağımsız modüller tasarlarırken, çalışabilecek en basit çözümü uygulamayı unutmayın.



## Bölüm 12 (Final) - Temiz Tasarımın 4 Kuralı



İyi tasarıma sahip sistemler oluşturabilmek için takip edebileceğiniz 4 kural olduğunu söylesem? Bu kuralları izleyerek, kodunuzun yapısını ve tasarımını anlayarak, **SRP** ve **DIP** gibi ilkelerin uygulanmasını kolaylaştıracak olsanız?

**Kent Beck**'in dört basit tasarım kuralı, iyi tasarlanmış yazılımlar yaratmada önemli bir etkindir. Ve Kent Beck'e göre eğer bir tasarım basitse, şu kuralları takip etmelidir:

1. Tüm testleri çalıştırın
2. Tekrarlanmış kodlar yazmayın
3. Açıklayıcı olun
4. Sınıf ve metot sayısını en aza indirin

Kurallar önem sırasına göre. Şimdi sırasıyla bu kuralları inceleyelim:

### Kural #1: Tüm testleri çalıştırın

Bir tasarım, amaçlandığı gibi hareket eden bir sistem üretmelidir. Bir sistem kağıt üzerinde mükemmel bir tasarıma sahip olabilir ancak sistemin amaçlandığı gibi çalıştığını doğrulamanın basit bir yolu yoksa, kağıt üstündeki tüm çabalar boşa olabilir.

Kapsamlı bir şekilde test edilen ve tüm testlerini her zaman geçiren bir sistem **test edilebilir bir sistem**dir. Bir sistemin **test edilebilir** olması önemlidir. Test edilebilir olmayan sistemler doğrulanabilir de değildir. Doğrulanamayan bir sistem asla dağıtılmamalıdır (deploy).

Sistemlerimizi test edilebilir hale getirmek, sınıflarımızın küçük ve tek amacının olduğu bir tasarıma sahip olmamıza yardımcı olur. **SRP**'ye uyan sınıfları test etmek daha kolaydır. Daha çok test yazdıkça, test etmesi daha kolay olan sistemler ortaya çıkarmaya devam ederiz. Dolayısıyla sistemimizin test edilebilir olduğundan emin olmak, daha iyi tasarımlar oluşturmamıza yardımcı olur.

Sıkı bağımlılıklar da (tight coupling) test yazmayı zorlaştırır. Daha çok test yazdıkça; **DIP** gibi ilkeleri ve bağımlılık enjeksiyonu (dependency injection), arayüzler ve soyutlamalar gibi araçları daha çok kullanırız. Tasarımlarımız daha da gelişir.

Bir kere testlerimizi yazdıktan sonra, kodumuzu ve sınıflarımızı temiz tutmak için kendimizi yetkin hissederiz. Bunu, kodumuzu sürekli yeniden yapılandırarak yaparız. Eklediğimiz birkaç kod satırı için durup yeni tasarım üzerine düşünürüz. Testlerimiz geçtiğindeyse hiçbir yeri bozmadığımızdan emin olabiliriz. Testlere sahip olmamız, kodu temizlerken onu bozma korkumuzu ortadan kaldırır.

Düzenlemelerimizi yaparken, temiz tasarım hakkında tüm bildiklerimizi uygularız: birbirine bağlılığı (cohesion) artırırız, bağımlılığı (coupling) azaltırız, meselelerin ayrımını (seperation of concerns) sağlarız, sistemlerimizi modülerize ederiz, fonksiyon ve sınıflarımızı küçültür ve daha iyi isimler seçeriz. Burası ayrıca basit tasarımın son üç kuralını uyguladığımız yerdir: tekrarları kaldır, kodunun açıklayıcı olduğundan emin ol, sınıf ve metot sayılarını en aza indir.

## Kural #2: Tekrarlanmış kodlar yazmayın

Tekrarlanmış kodlar, iyi tasarlanmış sistemlerin birinci düşmanıdır. Tekrarlanmış kodlar, ek iş, ek risk ve gereksiz karmaşa demektir. Örneğin bir *liste* sınıfındaki şu iki metota bakalım:

```
int size() {}  
boolean isEmpty() {}
```

Şu şekilde buradaki tekrarlanmış kodu eleayabiliriz:

```
boolean isEmpty() {  
    return 0 == size();  
}
```

Temiz bir sistem yaratmak, bir kaç satır kod için bile olsa tekrarı azaltma isteği gerektirir. Örneğin:

```

public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01 f); RenderedOp newImage =
ImageUtilities.getScaledImage(image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
}

```

Bu sistemi temiz tutabilmek için, ***scaleToOneDimension*** ve ***rotate*** metotları arasındaki tekrarlanmış kodları elemeliyiz:

```

public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) <
errorThreshold) return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) *
0.01 f);
    replaceImage(ImageUtilities.getScaledImage(image,
scalingFactor, scalingFactor));
}public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
}

```

Bu küçük ortak metodu çıkararak, ***SRP*** ilkesini ihlal ediyor olduğumuzu farkettilik. Böylece yeni çıkarılan metodu başka bir sınıfa taşıyabiliriz. Ve böylece ekipten başka birisine yeni metodu soyut bir sınıfa alma ve onu başka yerlerde kullanma fırsatı vermiş oluruz. Yaptığımız bu küçük değişiklik sistem karmaşıklığının ciddi şekilde azalmasına yardımcı olur.

***Template Method*** deseni, daha üst düzey tekrarları kaldırmak için kullanılan yaygın bir yöntemdir. Örneğin:

```

public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // code to calculate vacation based on hours worked to

```

```

date
    // code to ensure vacation meets US minimums
    // ...
    // code to apply vacation to payroll record
    // ...
}    public void accrueEUDivisionVacation() {
    // code to calculate vacation based on hours worked to
date
    // code to ensure vacation meets EU minimums
    // code to apply vacation to payroll record
    // ...
}
}

```

***accrueUSDivisionVacation*** ve ***accrueEUDivisionVacation*** metodlarının çoğu, yasal minimumları hesaplamak dışında aynı. Algoritmanın bu kısmı çalışan türüne göre değişiyor. **Template Method** desenini uygulayarak bariz tekrarlanışlığı ortadan kaldırabiliriz:

```

abstract public class VacationPolicy {
    public void accrueVacation() {
        calculateBaseVacationHours();
        alterForLegalMinimums();
        applyToPayroll();
    }

    private void calculateBaseVacationHours() { /* ... */ };
    abstract protected void alterForLegalMinimums();
    private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // US specific logic
    }
}

public class EUVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // EU specific logic
    }
}
}

```

Alt sınıflar, tekrarlanmış olmayan diğer bilgiyi sağlayarak ***accrueVacation*** algoritmasındaki “boşluğu” doldurur.

### Kural #3: Açıklayıcı olun

Bir yazılım projesinin maliyetinin çoğu uzun dönem bakımıdır. Kodumuzu değiştirirken hata potansiyelini en aza indirmek için, sistemin ne yaptığını anlamamız önemlidir. Sistemler daha karmaşık hale geldikçe, yazılımcılar olarak anlamamız zorlaşır ve hata yapma riskimiz artar. Bu nedenle, kodumuz yazanın niyetini açıkça belli etmelidir. Yazar kodu daha da açık hale getirdikçe, diğerlerinin anlaması için geçen süre ve bakım süresi daha da azalır.

İyi isimler seçerek kendinizi daha iyi ifade edebilirsiniz. Ayrıca küçük fonksiyon ve sınıfların isimlendirmesi, anlaşılması ve yazılması daha kolaydır.

Standart bir terminoloji kullanarak da kendinizi ifade edebilirsiniz. Örneğin tasarım desenleri (design patterns) büyük ölçüde iletişim ve ifade etme ile alakalıdır. Bu desenleri gerçekleştiren sınıfların isimlerinde bu tür standart desen isimleri kullanarak, **Command** veya **Visitor** gibi, tasarımınızı diğer geliştiricilere kısaca açıklayabilirsiniz.

İyi yazılmış birim testleri de açıklayıcıdır. Testlerin öncelikli amacı, *örnek* dokümantasyon sağlamaktır. Testlerimizi okuyan biri sınıfın neyle ilgili olduğunu hızlı bir şekilde anlayabilmelidir.

Çoğu zaman kodumuzun çalışmasını sağlıyor ve bir sonraki kişinin kodu okumasını kolaylaştırmak için hiç çabalamadan bir sonraki soruna geçiyoruz.

Unutmayın, bu kodu okuyacak bir sonraki kişi büyük ihtimalle siz olacaksınız.

Her bir fonksiyon ve sınıfınızla biraz zaman geçirin. Daha iyi isimler seçin, büyük fonksiyonları daha küçük fonksiyonlara bölün ve yarattığınız şeylere özen gösterin.

## Kural #4: Sınıf ve metot sayısını en aza indirin

Sınıflarımızı ve metotlarımızı küçültmek için çabalarken, küçük küçük bir çok sınıf ve metot yaratabiliriz. Bu kural ise bu sayıyı minimumda tutmamız gerektiğini söylüyor.

Yüksek sayıda sınıf ve metot bazen anlamsız dogmatikliğin bir sonucudur. Örneğin, her sınıf için bir arayüz oluşturmayı ısrarla vurgulayan bir kodlama standardını veya alanların (fields) ve davranışların her zaman veri sınıflarına ve davranış sınıflarına ayrılması gerektiğinde ısrarcı olan geliştiricileri düşünelim. Bu tür dogmalara karşı direnilmeli ve daha pragmatik bir yaklaşım benimsenmelidir.

Bu kuralın, 4 kuraldan en düşük öncelikli olanı olduğunu unutmayın. Bu yüzden sınıf ve fonksiyon sayısını düşük tutmak önemli olsa da, testler yazmak, tekrarları ortadan kaldırmak ve kendimizi açıkça ifade etmek daha önemlidir.