Clean Code Kitap Raporu

Can Ateş ODAK-GIS

Kitap hakkında

Kitap iyi ve kötü kod ile dolu, kodlara sağından solundan, bazen önünden bazen arkasından bazen de içinden yani muhtemel her açıdan bakarak iyi kod nasıl olur nasıl yazılır onu anlatmaya çalışıyor. Kitabı okuduktan sonra iyi kod ve kötü kod arasında farkı, iyi kod yazmayı ve kötü kodu iyi koda çevirmeyi öğreniyorsunuz

Bölüm 1

“THERE WILL BE CODE”

İlerde koda gerek kalmayacak ya da programcılar bütün kodları şablonlar halinde alarak kodlayacak diye düşünenler olabilir, yazar bu teorinin asla gerçekleşmeyeceğine ve her zaman birilerinin bir iş yerinin kod ile uğraşacağına kodun asla tamamen kodla üretilmeyeceğini savunuyor ve gerekli, işleri kolaylaştıran araçlar yazılsa da kaçınılmaz incelikleri gene programcılar yapacağı için “kod olacak” (“There will be code”) görüşündedir.

“BAD CODE”

Her programcı belki öğrenim belki iş hayatında içinden çıkılmaz ne olduğu anlaşılmaz sadece kendinin anlamlandırabildiği kod yazmıştır. Sebebi belki ödevi son gün yetiştirme çabası, üşengeçlik, “patronum işi 1 günde istiyor” gibi bir sürü şey olabilir . Böyle bir durumda elimizdeki çalışan kocaman kaosa bakıp çalışması hiç olmamasından iyidir diye düşünüp sonra düzeltirim dediğimiz günler hepimizin olmuştur. O günlerde bilmediğimiz şey ise Leblanc’ın sonra = asla yasasıdır. Geriye dönüp baktığımda benimde okul zamanında yazıp bir kenara attığım bugün yarın bakarım dediğim çalışan ama nasıl çalıştığını sadece benim anladığım kodlarım mevcut. Bu kitabı o zamanlar okumuş olsaydım çok daha farklı kodlar çıkardı ortaya. O zaman üşengeçlikten ötürü kodu uzatıyor diye düşündüğüm şeyler aslında kodu değil kodun ömrünü uzattığını anlıyorum. Şimdi açıp eski kodumu okusam ne olduğunu anlamam uzun bir zaman alacak kötü yazıldığı için.

Peki nedir bu güzel yani temiz kod? Temiz kodun kesin olarak budur denecek bir örneği yok programcıdan programcıya ilkeler yaklaşımlar değişiyor.Temiz kod Bjarne Stroustrup (c++’ın yaratıcısı) a göre temiz kod zarif ve etkili olmalı iken Grady Booch (obje odaklı analiz ve dizayn kitabının yazarı) a göre ise basit ve doğrudan olmalı. Hangi tecrübeli programcıya sorarsanız farklı bir temizlik duyacaksınız. Bana göre ise yazdığınız kod başka insanların işini kolaylaştırıp onlar tarafından minimal sürede anlaşılıp kavranıyorsa temizdir.

Bölüm 2

“MEANİNGFULL NAMES”

İsimler kodun kaçınılmazlarıdır. Yazdığınız her şeye bir isim vermek zorundasınız. Ve bunu yaparken “Uf şimdi kim aylıkOrtalamaGonderilenVeriBuyuklugu yazacak” diye düşünerek dahiyane bir şekilde x yazarak geçiştirilen isimler yapmamalısınız (kim zamanında yapmamıştır ki 😬). Bunun gibi kötü isimleri sadece yazan o kodu yazdığı kısa süre zaafında anlar, 1 sene sonra dönüp baksa “bu x de neyin nesi” diye düşünecektir. Bu nedenden ötürü yarattığımız şeylere iyi isim vermeliyiz. Peki iyi isim nasıl verilir?

- Amacına uygun isimler yazılmalı: int x değil de int degistirilmeTarihi
- Yanlış bilgilendirme yapılmamalı: elinizde bir array’e tutup da sayıList demek
- Benzer kelimeler kullanılsa da farklılık açık olmalı
- Rahat okunabilir kelimeler kullanılmalı: hsptklnpra nın hesaptaKalanPara olduğunu anlamak zor oluyor.
- Rahat aranabilir kelimeler kullanılmalı
- Şifrelemeden kaçınılmalı
- Class adları fiil olmamalı ve minimal bir biçimde tanımlanmalı
- Metod adları ise fiil olmalı ve yaptığı işi açık bir biçimde özetlemeli: iş başına bir kelime seçilerek bu elde edilebilir.

Bölüm 3

“FUNCTIONS”

Fonksiyonlar koddaki ilk organizasyonlardır. Doğru yazılmazlar ise olduğundan daha da anlaşılmaz daha da kötü yapacaklardır kodunuzu. Gelelim fonksiyonların olmazsa olmazlarına:

- Fonksiyonlar kısa olmalıdırlar: 100 satırlık fonksiyon artık fonksiyonluğunu yitirmiştir en fazla 20 satır arası olmalıdırlar.
- İflerin içi elselerin içi döngülerin içi eğer 1 satır uzunluğunda oluyorsa orası da fonksiyon olmalı.
- Fonksiyonlar sadece bir şey yapmalı, onu güzel yapmalı, ve sadece onu yapmalı.Bu kural benim eskiden en çok ihlal ettiğim kural olabilir. “Bütün main in içini fonksiyon yapayım kodum güzel gözüksün “diye düşündüğüm zamanlar vardı ancak bu düşüncenin pekte temiz bir çözüm olmadığını eski kodlarıma baktığımda anlıyorum.
- Birbiri ile kullanılan fonksiyonlar peş peşe yazılmalı
- İç içe switch ler yerine çok biçimlilik(polymorphism) kullanılmalı
- İsim kurallarına uyulmalı

- Argümanların ideal sayısı 0 olmakla beraber 1 ve 2 kabul edilebilir 3 belki 4 çok çok nadiren ama 5 asla olmamalıdır.
- Kodunuza bakıp ta “ben bu satırı daha öncede yazmıştım” dediğiniz yerler mutlaka vardır o kısımlarda fonksiyon olmalıdır. Tekrar ve çoğaltma koddaki kötülüklerin kaynaklarından biridir.

Bölüm 4

“COMMENTS”

İyi bir yorumun bütün bir modülü kolayca anlatabileceği gibi kötü ve yanlış bilgilendiren yorumda okuyanın kodu anlamasını bir denli zorlaştırır. Yorumlar kendimi kod ile ifade edemediğimiz yerde devreye girer. O yüzden gerekli kötülüklerdir. Her yorum yazılışında yorumda yazılan detayların kodla ifade edilip edilemeyeceği düşünülmelidir. Yasal patent yorum satırları gerekli yorumlara bir örnektir. Yorum satırları basit bilgiler vermelidir. Sırf yazılmalı diye yazılan yorumların çoğu kötü yorumlar kategorisine girerler.

```
    //invalid menu input
    default:
        Console.WriteLine("Invalid menu input");
        break;
}
```

Kendi yazdığım bir kötü yorum örneği

```
//veriables
bool loop = true;
string input;
double score;
double percentege  ;
```

Başka bir gene bana ait kötü yorum örneği

Bölüm 5

“Formatting”

Başka kişiler sizin kodunuzu açtığında gördükleri temizlikten ve düzenden etkilensin isteriz. Bunun içinde kodu biçimlendirmemiz gerekir bu biçimlendirmeyi bir set halindeki basit kurallara karar verip onları standart olarak kabul ederek elde edebiliriz. Mesela her “fonksiyondan sonra bir boşluk”,” farklı konseptler arası bir boşluk” gibi standartlar belirlenip onlara uyulursa ortaya çıkan kodun da bir düzeni ve tertibi olacaktır. Ancak dikkat edilmesi gereken şeylerden biri aşırıya kaçılmamasıdır. Her şeye boşluk koyulursa dikey yoğunluk azalacağından gözükecek görüntüde kötüleşecektir. Dikkat edilmesi gereken kurallardan biride birbiri ile ilgili metotların birbirine yakın yazılmasıdır.

Bölüm 6

“Objects and Data Structures”

Değişkenlerin private olmasının sebebi yazdığımız classı kullanan insanların o değişkenlere bağlılığını ortadan kaldırmaktır. Bir objenin hangi kısımlarının gizlenip hangilerinin erişime açılacağı konusunda iyice düşünülmesi gerekir. En kötü yöntem olarakta bütün verilere getter setter yazmak olarak geçer. Ben okulda yazdığım kodlarda maalesef en kötü yöntemi uyguluyordum bütün değişkenler private ve hepsinin getter setter ı var. Bunun gibi durumların yapılmaması için üretilmiş bir yasa vardır , “Law Of Demeter”. Annelerimizden de defalarca duyduğumuz gibi bu yasa derki “Yabancılarla konuşmayın”

Yabancılarla konuşmaktan nasıl kaçınırız? sorusu ise 4 basit adımda anlatılır. Bir nesneye ait bir metot olsun o metodun çağrılabileceği 4 yöntem vardır.

- Nesnenin kendisine ait metodlar

```
public class LoDOrnekSınıf
{
    public void KendiMethodunuCagır()
    {
        birseylerYap(); // Nesnenin kendi metodunu çağırıyor, LoD 1
    }
    private void birseylerYap()
    {
    }
}
```

- Metoda parametre olarak gelen nesnenin metotları

```
public class LoDOrnekSınıf
{
    public void biseylerYap(BaskaSınıf baskaSinif)
    {
        baskaSinif.baskaBiseylerYap(); // Parametre ile gelen nesnenin metodunu çağırıyor, LoD 2
    }
}
```

- Metodun içinde oluşturulmuş bir nesneye ait metotlar

```
public class LoDOrnekSınıf
{
    public void biseylerYap()
    {
        BaskaSinif baskaSinif = new BaskaSinif();
        baskaSinif.baskaBiseylerYap(); // Metod içerisinde oluşturulmuş nesnenin metodunu çağırıyor, LoD 3
    }
}
```

- Nesnenin direk erişimi olan nesnelere ait metotlar

```
public class LoDOrnekSınıf
{
    BaskaSinif baskaSinif = new BaskaSinif();
    public void birseylerYap()
    {
        baskaSinif.baskaBirseylerYap(); // Nesnenin direk erişimi olan nesnenin metodunu çağırıyor, LoD 4
    }
}
```

İçinde sadece public veri olan ve baska fonksiyon olmayan sınıflara DTO denir. DTO lar database ile etkileşime geçen kodlarda çok işe yararlar.Ham veriyi database e aktarırken DTO lar ile taşırız.

Bölüm 7

“Error Handling”

Program çalışırken meydana gelebilecek hataları ön görüp onların önlemini almak her kodun içinde vardır. Ancak yukarıdaki diğer konularda da olduğu gibi bunun da örfleri adetleri vardır. “Hata olursa -1 döndüreyim” gibi düşünmek yerine try-catch blokları kullanılmalıdır. Yapılan en genel hatalardan biri (ben de dahilim bu hatayı yapan kişilere)

Try-catch bloğundaki finally anahtar kelimesini en son yazmaktır. Önce finally kısmı yazılıp sonra catch blokları ile oluşabilecek hatalar ayıklanmalıdır. Oluşabilecek hatalara göre önceden tanımlanmamışsa hata sınıfları yazılmalıdır. Temiz kod okunaklı olmalıdır ama aynı zamanda güçlü olmalıdır , gücünü de hata ayıklamadan alır. Bu nedenden ötürü hata

ayıklamayı kodun ana mantığından ayırarak başka bir problemmiş gibi incelemeliyiz. Temiz hata ayıklama kodları yazmanın yöntemi de budur.

Bölüm 8

“Boundaries”

Bazen kendi yazdığımız sistemi başka bir yazılımla birleştirmemiz yada iç içe kullanmamız gerekebilir. Böyle durumlarda sınırlar güzel belirlenmelidir. Mesela şirket için yazdığım “taşınmaz” projesinde kendi projemi birkaç open-source üçüncü şahıs uygulaması ile birleştirdim. Bu başka hazır yazılımların kullanılması durumunda ise dikkat edilmesi gereken adımlar vardır. İndirdiğiniz ya da yüklediğiniz üçüncü şahıs kodunu test etmek sizin göreviniz değildir, ancak birkaç küçük test yazmak işinizi kolaylaştıracaktır. Genelde dökümantasyonu bulunur indirilen yazılımın ancak kopyalayıp yapıştırmak ve anlayarak kullanmak arasında dağlar kadar fark vardır. Küçük testler ise sizi bu dağların ötesine taşıyacaktır. “Bu neden çalışıyor” , “ Şunu silsem nasıl çalışır “ , “Peki ya şöyle çalışmasını istesem” gibi sorular sizi sadece kopyalayıp yapıştıran insanlardan ayırıp araştırmaların sonucunda daha iyi bir yazılımcı yapacaktır. Bu bölümün çok az kısmı 3. şahıs uygulamalarının eklenildiği zaman ilk aşamada nelere dikkat edileceği ile ilgili . Diğer kısımları eklenildikten sonra varsa 2 uygulama arası sınırların nasıl olması gerektiği ile ilgili ancak ben hiç o tarzda bir uygulama yazmadığım için kendimden örnek veremeyeceğim.

Bölüm 9

“Unit Test”

Yazdığınız kodun çalışıp çalışmadığı yapılan testler sonucu ortaya çıkar. Siz karşınıza verdiğiniz talimatlara harfiyen uyan kişiler çıkacağı yanılsamasına girmeyin diye yazdığınız kodları testlerden geçirmelisiniz. Bob amcamız da bu yazılan kod ile unit tesler arasındaki etkileşimde dikkat edilmesi gereken şeyleri 3 kurala indirgemiştir

- Hata testi kodunu yazmadan kontrol edeceğiniz kodu yazmayın
- Gereğinden fazla şeyleri kontrol eden unit test kodu yazmayın
- Hata testi kodunun kontrol ettiğinden fazla kapsamlı kod yazmayın

Ancak ben bu bölümü okuduğum zaman kural 3 ün aslında kural 1 ima ettiğini düşündüm benim kendi dikkat edip de hafızamda tutmaya çalışacağım kurallar ise şöyle olacaktır

- Gereğinden fazla şeyleri kontrol eden unit test kodu yazmayın
- Hata testi kodunun kontrol ettiğinden fazla kapsamlı kod yazmayın

Hafızadan ne kadar yer kurtarılsa kardır değil mi ? 😬

Yazılan testler ise temiz sayılabilmesi için önce hızlı olması gerekir. Yavaş çalışan testler yapılan bir hatayı geç bulacağından o hatanın çözülmesi zaman ilerledikçe o kadar da kolay olmayabilir. Örneğin bir kötü çalışan kod parçasının geç tespitinde ortaya çıkabilecek bir durum şirketteki herkesin o kod parçasını kopyalayıp yapıştırıp bir yerlerde kullanması ondan bir şeyler üretmesi gibi bir senaryoda karşımıza çıkar. Test hızlı çalışsa idi bu durum yaşanmadan kurtarılırdı. Yazılan testler bağımsız olmalıdır. Bir test başka bir testin çalışması için gereksinim olmamalıdır. Testler kendi başlarına çalışmalıdır. Başka bir kural ise tekrar edilebilir olmalıdır. Bir kere çalışan testin kullanana pekte bir faydası olmayacaktır. Testlerin boolean çıktıları olmalıdır geçti ya da kaldı teker teker log dosyalarını gezip de yada uzun çıktıların içinde geçip kaldığını aramak ile vakit kaybedilmemelidir. Son olarak testlerin yazılma zamanı ile bir kuralda zamanlamaları testler büyük bir projenin içinde en başta yazılınıp sonradan projeye geçinilmemelidir. Üreten kod ne zaman yazılacak ise ondan hemen önce yazılmalıdırlar.

Bölüm 10

“Classes”

En başında dikkat edilmesi gereken sınıflar sabitlerle başlamalıdır. Tanımlama sırası ise public’ler önde private’lar en sonda olacak şekilde. Bu yapı göz önüne alınarak public bir metodu kullanacak olan private metot public’in altına yazılmalıdır. Sınıflar kısa olmalıdır ve kısaca özetlerini içeren yorumlar olmalıdır. SRP prensibine göre bir sınıfın sadece bir sorumluluğu olmalıdır. Bir sınıfın bütün değişkenleri bütün metotlar tarafından kullanılıyor ise bu sınıf maksimum uyumludur. Ancak bu uyum oranını korumak değişken sayısı arttıkça karmaşıklığa yol açar, 20 değişkeniniz olduğunu ve 10 metodunuzda o 20 değişkenin hepsinin kullanıldığını düşünün ortaya çıkacak olan sınıf ne küçüktür ne de temizdir.Programcı bu uyum oranını kodun temizlik oranını bozmadan ayarlamalıdır.

Bölüm 11

“Systems”

Sistemler şehirlere en benzeyen yapılardır. Sistem yazılırken de bir şehrin nasıl inşa edilmesi gerektiği göz önüne alınmalıdır. Yapılacak ilk iş inşaat ile kullanım kısımlarını ayırmaktır. Bizim de arka-yüzde kullandığımız gibi startup.cs gibi ilk açılışta yapılar arası bağlantıların yapılacağı bir süreç ile başlamalıdır. Bu süreçten sonrada çalıştırma süresince mantık kısmı devralmalıdır. Sistemler şehirlere benzerler dedik , buradan yola çıkarak genişletilmesi de şehirler gibi olmalıdır . Her şey bir anda inşa edilemez yavaş ve

sistematik bir süreçten geçilmesi gerekir ancak bu sistematiklik ve doğru konsept ayrılıkları kullanılarak istenilen en büyük sistem bile en hatasız şekilde elde edilebilir.

Bölüm 12

“Emergence”

Güzel bir tasarım için aslında sadece 4 basit kural vardır. Bir dizaynın “sade” olması için şu kurallarla tasarlanması gerekir

- Gerekli bütün testler tamamlanmalı.
- Yeniden düzenlenmeli: bir kod ilk defa yazıldığında yukarıda belirtilen kuralların hepsini uygulamaz zor olabiliyor. Gözden kaçırılan ve yapılan hataların belirlenmesi için kodun yeniden düzenlenmek üzere incelenmesi ve nerede nasıl daha iyi sonuçlar elde edilebileceği konusunda çalışmalar yapılması gereklidir. Yeniden düzenlenirken dikkat edilecek konulardan ilki tekrarlardır. Tekrarların kodda yeri yoktur ve ortadan kaldırılmalıdırlar.
- Anlaşılmaz ve kendini ifade etmekte zorlanan değişken, fonksiyon, sınıf, veya modül isimlerinden arındırılmalıdır. Bu isimler hem anlaşılmazlığa sebep olurlar hem de gereksiz yorum satırları oluştururlar.
- Minimal sayıda metot ve sınıf kullanılmalıdır.

Not :

Kitabın bölüm 12’e kadar ki kısmında tecrübelerim olduğu için çıkarımlar yapıp kitabın doğrusunu kendi doğrum ile karşılaştırabildim ve kendi yanlışlarımı görebildim. Ancak bölüm 12’den itibaren bahsi geçen konularda 1 satır dahi kodum olmadığı için sadece ders kitabı gibi okudum.