# Clean Code Özet

* Bu kitabı iki nedenle okuyorsunuz.
* Birincisi, sen bir programcısın.
* İkincisi, daha iyi bir programcı olmak istiyorsunuz.
* Güzel. Daha iyi programcılara ihtiyacımız var.
* Bu kitap iyi programlamayı anlatıyor.
* 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.

**Bölüm 1- Clean Code**

* Sonra asla demektir.
* 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. Takımdaki herkes verimliliği artırmak için büyük baskı altındadır. Öyle ki verimliliği sıfıra daha da yaklaştıracak şekilde kod karmaşası yaratmaya devam ederler.
* Ne oldu da iyi kod bu denli bir hızla kötü koda dönüştü?
* 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!
* Temiz kod yazabilmek, temizlik (cleanliness) duygusuyla uygulanmış sayısız küçük teknik yöntemlerin disiplinli bir şekilde kullanımını gerektirir.
* 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?
  + **Bjarne Stroustrup** (C++'ın mucidi ve The C++ Programlama Dili'nin yazarı):
    - Kodumun şık ve temiz olmasını seviyorum.
    - Kodda mantık, hataların saklanmasını 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.
  + **Büyük Dave Thomas (Eclipse stratejisinin manevi babası OTI'nin 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 bağlılığı 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.
  + Ron Jeffries, Extreme Programming Installed ve Extreme Programming Adventures in C# kitaplarının yazarı:
    - Öncelik sırasına göre basit kod:
      * Tüm testleri çalıştırır;
      * Kopya içermez;
      * Sistemdeki tüm tasarım fikirlerini ifade eder;
      * Sınıflar, yöntemler, işlevler ve benzeri varlıkların sayısını en aza indirir.
  + **Ward Cunningham**, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming… :
    - Okuduğunuz her rutin beklediğiniz gibi çıktığında temiz kod üzerinde çalıştığınızı bilirsiniz.
    - Kod aynı zamanda dilin sorun için yapılmış gibi görünmesini sağladığında buna güzel kod diyebilirsiniz.
  + Düşünce okulları
    - Peki ya ben (Bob Amca)? Temiz kodun ne olduğunu 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 bazılarına katılmayacaksınız, bazılarına şiddetle karşı çıkacaksınız.
    - Sorun değil.
    - Sadece şunu bilmelisiniz ki, bu yöntemleri onlarca yıllık tecrübeler sonucunda, birçok deneme ve yanılmamalarla öğrendim.
* Kodu iyi yazmak yetmez.
* Kodun zaman içinde temiz tutulması gerekir.
* Zaman geçtikçe kodun çürüdüğünü ve bozulduğunu hepimiz gördük.
* Dolayısıyla bu bozulmanın önlenmesinde aktif rol almalıyız.
* Amerika İzcilerinin mesleğimize uygulayabileceğimiz basit bir kuralı var.
* Kamp alanını bulduğunuzdan daha temiz bırakın.5
* Hepimiz kodumuzu teslim ettiğimizden biraz daha temiz bir şekilde teslim edersek, kod basitçe bozulamazdı.
* Temizlemenin büyük bir şey olması gerekmez.
  + Bir değişken adını daha iyi hale getirin,
  + Biraz fazla büyük olan bir işlevi ayırın,
  + Küçük bir tekrarı ortadan kaldırın,
  + Bir bileşik if ifadesini temizleyin.

**Bölüm 2- Anlamlı İsimler**

* İsimler yazılımın her yerindedir.
* Değişkenlerimizi, fonksiyonlarımızı, argümanlarımızı, sınıflarımızı ve paketlerimizi isimlendiririz.
* Kaynak dosyalarımızı ve onları içeren dizinleri adlandırıyoruz.
* Niyet açığa çıkaran isimler kullanın
  + int d; //elapsed time in days
  + *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;
* Dezenformasyondan kaçının.
* Anlamlı ayrımlar yapın.
  + *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:
* Telaffuz edilebilir isimler kullanın.
  + İsimlerinizi telaffuz edilebilir yapın.
  + Telaffuz edemiyorsanız, aptal gibi konuşmadan tartışamazsınız.
* Aranabilir isimler kullanın.
  + Tek harfli adlar ve sayısal sabitler, bir metin gövdesi boyunca kolayca bulunamadıkları için belirli bir soruna sahiptir.
  + MAX\_CLASSES\_PER\_STUDENT için kolayca grep yapılabilir, ancak 7 rakamı daha zahmetli olabilir.
* Kodlamalardan kaçının
  + Tip veya kapsam bilgisinin adlara kodlanması, yalnızca fazladan bir deşifre etme yükü ekler.
  + Kodlanmış adlar nadiren telaffuz edilir ve yanlış yazılması kolaydır.
* Zihinsel haritalamadan kaçının.
  + Genel olarak programcılar oldukça akıllı insanlardır.
  + Zeki insanlar bazen zihinsel hokkabazlık yeteneklerini sergileyerek zekalarını göstermeyi severler.
  + Ne de olsa, r'nin ana bilgisayar ve düzen kaldırılmış url'nin küçük harfli versiyonu olduğunu güvenilir bir şekilde hatırlayabiliyorsanız, o zaman kesinlikle çok akıllı olmalısınız.
  + Akıllı bir programcı ile profesyonel bir programcı arasındaki farklardan biri, profesyonelin netliğin kral olduğunu anlaması.
  + Profesyoneller güçlerini iyilik için kullanırlar ve başkalarının anlayabileceği kodlar yazarlar.
* Sınıf isimleri
  + Sınıflar ve nesneler, Customer, WikiPage, Account ve AddressParser gibi isim veya isim tamlaması adlarına sahip olmalıdır.
  + Bir sınıf adına Yönetici, İşlemci, Veri veya Bilgi gibi sözcüklerden kaçının. Bir sınıf adı bir fiil olmamalıdır.
* Fonksiyon isimleri
  + Yöntemler, postPayment, deletePage veya save gibi fiil veya fiil tümcesi adlarına sahip olmalıdır.
  + Erişimciler, mutatörler ve yüklemler, değerlerine göre adlandırılmalı ve önlerine get, set ve javabean standardına göre eklenmelidir.
* Sevimli olma.
  + İsimler çok zekiceyse, sadece yazarın mizah anlayışını paylaşan insanlar için ve bu insanlar espriyi hatırladıkları sürece akılda kalırlar.
  + HolyHandGrenade adlı işlevin ne yapması gerektiğini bilecekler mi?
* Konsept başına bir kelime seçin.
  + Bir soyut kavram için bir kelime seçin ve ona bağlı kalın.
  + Örneğin, getirme, geri alma ve alma işlemlerinin farklı sınıfların eşdeğer yöntemleri olması kafa karıştırıcıdır.
  + Hangi yöntem adının hangi sınıfla gittiğini nasıl hatırlıyorsunuz?
  + Ne yazık ki, hangi terimin kullanıldığını hatırlamak için, kütüphaneyi veya sınıfı hangi şirketin, grubun veya bireyin yazdığını sık sık hatırlamanız gerekir.
  + Aksi takdirde, başlıklara ve önceki kod örneklerine göz atarak çok fazla zaman harcarsınız.
* Kelime oyunu yapma.
  + 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.
* Çözüm etki alanı adlarını kullanın.
  + Kodunuzu okuyan kişilerin programcı olacağını unutmayın.
  + Öyleyse devam edin ve bilgisayar bilimi (CS) terimlerini, algoritma adlarını, kalıp adlarını, matematik terimlerini vb. kullanın.
  + Her adı sorunlu alandan çekmek akıllıca değildir çünkü iş arkadaşlarımızın, kavramı zaten farklı bir adla bildikleri halde her ismin ne anlama geldiğini sormak için müşteriye gidip gelmesini istemeyiz.
* Sorunlu alan adlarını kullanın
* Anlamlı bağlam ekle
  + firstName, lastName, street, houseNumber, city, state ve zipcode isimli 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; addrsFirstName, 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.
* Nedensiz bağlam eklemeyin
  + 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.

**Bölüm 3- Fonksiyonlar**

* Fonksiyonlar bir programın içindeki organizasyonun ilk satırlarıdır.
* Küçük
  + Fonksiyonların ilk kuralı, küçük olmaları gerektiğidir. Fonksiyonların ikinci kuralı, bundan daha küçük olmaları gerektiğidir.
* Bloklar ve Girinti
  + Bu, if deyimleri, else deyimleri, while deyimleri vb. içindeki blokların bir satır uzunluğunda olması gerektiği anlamına gelir.
  + Muhtemelen bu satır bir işlev çağrısı olmalıdır.
  + Bu, çevreleyen işlevi küçük tutmakla kalmaz, aynı zamanda blok içinde çağrılan işlev güzel bir şekilde açıklayıcı bir ada sahip olabileceğinden belgesel değeri de ekler.
  + Bu aynı zamanda fonksiyonların iç içe geçmiş yapıları tutacak kadar büyük olmaması gerektiği anlamına gelir.
  + Bu nedenle, bir fonksiyonun girinti seviyesi bir veya ikiden büyük olmamalıdır.
  + Bu, elbette, işlevlerin okunmasını ve anlaşılmasını kolaylaştırır.
* Bir şey yap
  + Fonksiyonlar bir şey yapmalıdır. Bunu İyi yapmalıdır. Sadece bunu yapmalıdır.
  + İşlevler içindeki bölümler
    - Bildirimler, başlatma vb. bölümlere ayrılmış bir fonksiyon varsa, bu, fonksiyonun birden fazla şey yaptığının bariz bir belirtisidir.
    - Bir şeyi yapan fonksiyonlar makul bir şekilde bölümlere ayrılamaz.
* Fonksiyon başına bir düzey soyutlama
  + Fonksiyonlarımızın "tek bir şey" yaptığından emin olmak için, fonksiyonumuzdaki tüm ifadelerin aynı soyutlama düzeyinde olduğundan emin olmamız gerekir.
  + Kodu Yukarıdan Aşağıya Okumak: Düşürme Kuralı
    - Kodun yukarıdan aşağıya bir anlatım gibi okumasını istiyoruz.
    - 5 Her fonksiyonun bir sonraki soyutlama seviyesindekiler tarafından takip edilmesini istiyoruz, böylece programı okuyabilir, fonksiyonlar listesini okurken her defasında bir soyutlama seviyesine inebiliriz.
    - Bunu farklı bir şekilde söylemek gerekirse, programı, her biri mevcut soyutlama düzeyini tanımlayan ve bir sonraki seviyedeki sonraki TO paragraflarına atıfta bulunan bir TO paragrafları setiymiş gibi okuyabilmek istiyoruz.
* Anahtar (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 yaparız.
  + Açıklayıcı İsimler Kullanın
    - Her rutin beklediğiniz gibi çıktığında temiz kod üzerinde çalıştığınızı bilirsiniz.
    - Bu ilkeye ulaşmanın yarısı, tek bir şey yapan küçük fonksiyonlar için iyi adlar seçmektir.
    - Bir fonksiyon ne kadar küçük ve odaklanmışsa, tanımlayıcı bir ad seçmek o kadar kolay olur.
    - Uzun bir isim yapmaktan korkmayın.
    - Uzun, tanımlayıcı bir ad, kısa ve esrarengiz bir addan daha iyidir.
    - Uzun, açıklayıcı bir ad, uzun, açıklayıcı bir yorumdan daha iyidir.
    - İşlev adlarında birden çok sözcüğün kolayca okunmasına izin veren bir adlandırma kuralı kullanın ve ardından işleve ne yaptığını söyleyen bir ad vermek için bu birden çok sözcüğü kullanın.
    - Tanımlayıcı isimler seçmek, modülün kafanızdaki tasarımını netleştirecek ve geliştirmenize yardımcı olacaktır.
    - İyi bir isim aramanın, kodun olumlu bir şekilde yeniden yapılandırılmasıyla sonuçlanması hiç de alışılmadık bir durum değildir.
  + Fonksiyon argumanları
    - Bir fonksiyon için ideal argüman sayısı sıfırdır.
    - StringBuffer örneğimizi düşünelim.
    - Onu bir örnek değişken 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.
  + Yaygın monadik formlar
    - 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 şeylere 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.
  + Flag (Etiket) argümanlar
    - Flag argümanlar çirkindir.
    - Fonksiyonlara parametre olarak boolean 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.
  + İkili fonksiyonlar
    - İki bağımsız değişkenli bir fonksiyonu anlamak, monadik bir fonksiyondan daha zordur.
    - Örneğin, writeField(name)'i anlamak writeField(output-Stream, name)'den daha kolaydır
  + Üçlüler
    - Üç bağımsız değişken alan fonksiyonların anlaşılması ikililere göre çok daha zordur.
    - Sıralama, duraklatma ve yok sayma sorunları iki kattan fazla arttı.
    - Bir üçlü oluşturmadan önce çok dikkatli düşünmenizi öneririm.
  + Nesne argümanlar
    - Eğer bir fonksiyon 2 ya da 3 argümandan fazlasına ihtiyaç duyuyorsa, bu argümanlardan birkaçı bir sınıf ile sarmalanmalıdır.
    - Şu iki örnek çağırıma 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);
  + Filler 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.
* Çıktı argümanları
  + Genel olarak çıktı bağımsız değişkenlerinden kaçınılmalıdır.
  + Fonksiyonunuzun bir şeyin durumunu değiştirmesi gerekiyorsa, sahip olduğu nesnenin durumunu değiştirmesini sağlayın.
* Komut Sorgu Ayırma
  + Fonksiyonlar ya bir şey yapmalı ya da bir şeye cevap vermeli, ikisini birden değil.
  + Ya fonksiyonunuz bir nesnenin durumunu değiştirmeli ya da o nesne hakkında bazı bilgiler döndürmeli.
  + Her ikisini birden yapmak genellikle kafa karışıklığına yol açar.
  + Hata Kodlarını Döndürmeye İstisnaları Tercih Et
    - Komut fonksiyonlarından hata kodlarının döndürülmesi, komut sorgu ayrımının incelikli bir ihlalidir.
    - Kendinizi Tekrar Etmeyin
      * Çoğaltma, yazılımdaki tüm kötülüklerin kökü olabilir.
      * Kontrol altına almak veya ortadan kaldırmak amacıyla birçok ilke ve uygulama oluşturulmuştur.
  + Yapılandırılmış programlama
    - Bazı programcılar, Edsger Dijkstra'nın yapısal programlama kurallarına uyar.
    - Dijkstra, her işlevin ve bir işlev içindeki her bloğun bir girişi ve bir çıkışı olması gerektiğini söyledi.
    - Bu kurallara uyulması, bir fonksiyonun yalnızca bir dönüş ifadesi olması, bir döngüde break veya continue ifadeleri olmaması ve hiçbir zaman, hiçbir şekilde goto ifadesi olmaması gerektiği anlamına gelir.
    - Yapılandırılmış programlamanın amaçlarına ve disiplinlerine sempati duysak da, fonksiyonlar çok küçük olduğunda bu kurallar çok az fayda sağlar. Bu tür kurallar yalnızca daha büyük fonksiyonlarda önemli fayda sağlar.
    - Bu nedenle, fonksiyonlarını küçük tutarsanız, ara sıra birden çok dönüş , ara veya devam deyimi zarar vermez ve hatta bazen tek giriş, tek çıkış kuralından daha anlamlı olabilir. Öte yandan, goto yalnızca büyük fonksiyonlarda anlamlıdır, bu nedenle kaçınılmalıdır.