# 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.

**Bölüm 4 – Yorumlar**

* Hiçbir şey iyi yerleştirilmiş bir yorum kadar yardımcı olamaz.
* Hiçbir şey bir modülü anlamsız dogmatik yorumlardan daha fazla karıştıramaz.
* Hiçbir şey yalanları ve yanlış bilgileri yayan eski bir yorum kadar zarar verici olamaz.
* Programlama dillerimiz yeterince ifade edici olsaydı ya da niyetimizi ifade etmek için bu dilleri kurnazca kullanma yeteneğine sahip olsaydık, yorumlara çok fazla ihtiyaç duymazdık, belki de hiç.
* Yorumlar kötü kodu telafi etmez
  + Az yorum içeren açık ve anlamlı kod, çok sayıda yorum içeren karmaşık ve karmaşık koddan çok daha üstündür.
  + Vaktinizi yaptığınız pisliği açıklayan yorumlar yazarak harcamak yerine, o pisliği temizlemekle geçirin.
* Kendinizi kodla açıklayın.
  + // Check to see if the employee is eligible for full benefits
  + if ((employee.flags & HOURLY\_FLAG) && (employee.age > 65))
  + <======================================================>
  + if (employee.isEligibleForFullBenefits())
* 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 ve durumları gereklidir ve her kaynak dosyanın başına böyle bir yorum koymak mantıklıdır.
  + Bilgilendirici yorumlar
    - Bazen bir yorum ile temel bilgileri sağlamak yararlıdır.
    - Örneğin, soyut bir yöntemin dönüş değerini açıklayan şu yorumu göz önünde bulundurun:
      * // Returns an instance of the Responder being tested.
      * protected abstract Responder responderInstance();
    - Bunun gibi bir yorum bazen yararlı olabilir, ancak mümkün olduğunda bilgiyi iletmek için fonksiyonun adını kullanmak daha iyidir. Örneğin, bu durumda, fonksiyon yeniden adlandırılarak yorum gereksiz hale getirilebilir: “answererBeingTested”.
  + Niyet açıklaması
    - Bazen bir yorum, uygulama hakkında yararlı bilgilerin ötesine geçer ve bir kararın arkasındaki amacı sağlar.
  + Aydınlatma
    - Bazen belirsiz bir argümanın anlamını çevirmek veya değeri okunabilir bir şeye çevirmek yardımcı olur.
    - Genel olarak, bu argümanı veya dönüş değerini kendi başına netleştirmenin bir yolunu bulmak daha iyidir; ancak standart kitaplığın bir parçasıysa veya değiştiremeyeceğiniz bir koddaysa, açıklayıcı bir yorum faydalı olabilir.
  + Sonuçlara karşı uyarı
    - Bazen diğer programcıları belirli sonuçlar konusunda uyarmak yararlı olabilir.
  + Yapılacaklar (TODO) Yorumlar
    - 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.
  + Amplifikasyon
    - Aksi takdirde önemsiz görünebilecek bir şeyin önemini artırmak için bir yorum kullanılabilir.
  + Genel API'lerde Javadoc'lar
    - İyi tanımlanmış bir genel API kadar faydalı ve tatmin edici başka bir şey yoktur.
    - Standart Java kitaplığı için javadoclar buna bir örnektir.
    - Onlar olmadan Java programları yazmak en iyi ihtimalle zor olacaktır.
* Kötü yorumlar
  + Çoğu yorum bu kategoriye girer.
  + Eğer gerçekten yorum yazmaya karar verdiyseniz, zamanınızı yazabileceğiniz en iyi yorumu yazmak için harcadığınızdan emin olun.
  + Mırıldanma
    - Sırf yapman gerektiğini düşündüğün için ya da süreç gerektirdiği için bir yoruma girmek bir hiledir.
    - Bir yorum yazmaya karar verirseniz, bunun yazabileceğiniz en iyi yorum olduğundan emin olmak için gerekli zamanı harcayın.

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.
    - Ancak tüm varsayılanları kim yükler?
  + Gereksiz yorumlar

// 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");

}

}

* + - Bu yorum hangi amaca hizmet ediyor?
    - Kesinlikle koddan daha bilgilendirici değil.
    - Kodu doğrulamaz veya niyet veya mantık sağlamaz.
    - Kodu okumaktan daha kolay değil.
    - Aslında, koddan daha az kesindir ve okuyucuyu gerçek anlayış yerine bu kesinlik eksikliğini kabul etmeye ikna eder.
  + Yanıltıcı yorumlar
    - Bazen, tüm iyi niyetiyle, bir programcı yorumlarında doğru olamayacak kadar kesin olmayan bir açıklama yapar.
    - Bir an için önceki bölümün örneğini düşünün. this.closed true olduğunda yöntem geri dönmez.
    - this.closed true ise döner; aksi takdirde, bir kör zaman aşımı bekler ve ardından this.closed hala doğru değilse bir istisna atar.
  + Zorunlu yorumlar
    - Her işlevin bir javadoc'u olması gerektiğini veya her değişkenin bir açıklaması olması gerektiğini söyleyen bir kuralın olması tamamen aptalca.
    - Bunun gibi yorumlar yalnızca kodu karıştırır, yalanları yayar ve genel kafa karışıklığına ve düzensizliğe yol açar.
  + Günlük yorumlar
    - Bazen insanlar bir modülü her düzenlediklerinde başlangıcına bir yorum eklerler.

\* 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);

* + - * Bugün kaynak kodu kontrol sistemlerimiz var, bu tür günlüklere ihtiyacımız yok.
  + Gürültü Yorumları
    - Aşağıdaki örneklerdeki yorumlar yeni bilgi sağlamaz.

/\*\*

\* Default constructor.

\*/

protected AnnualDateRule() {

}

* + - Javadocs yorumları bu kategoriye girebilir.
    - Çoğu zaman bunlar, belge sağlamak için yersiz bir istekle yazılan gereksiz, gürültülü yorumlardır.
  + Bir fonksiyon ya da değişken kullanabilecekken yorum yazmayın

// does the module from the global list <mod> depend on the

// subsystem we are part of?

if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))  
vs

ArrayList moduleDependees = smodule.getDependSubsystems();

String ourSubSystem = subSysMod.getSubSystem();

if (moduleDependees.contains(ourSubSystem))

* + Konum işaretleyicileri
    - Bu tür yorumlar gürültü yapıyor.

// Actions //////////////////////////////////

* + Kapanış ayracı yorumları

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

* + - Bu tür yorumları kullanmak yerine kodu küçük işlevlerde kırabilirsiniz.
  + Atıflar ve Bylines
    - Example:
    - /\* Added by Rick \*/
    - VCS bunun yerine bu bilgileri yönetebilir.
  + Yorumlanan kod

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()));

* + - Artık ihtiyacınız yoksa, lütfen silin, tekrar ihtiyacınız olursa versiyon kontrol sisteminizle daha sonra geri dönebilirsiniz.
  + HTML yorumları
    - Aşağıdaki kodu okuyarak anlayabileceğiniz gibi, kaynak kod yorumlarındaki HTML iğrençtir.

/\*\*

\* Task to run fit tests.

\* This task runs fitnesse tests and publishes the results.

\* <p/>

\* <pre>

\* Usage:

\* &lt;taskdef name=&quot;execute-fitnesse-tests&quot;

\* classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;

\* classpathref=&quot;classpath&quot; /&gt;

\* OR

\* &lt;taskdef classpathref=&quot;classpath&quot;

\* resource=&quot;tasks.properties&quot; /&gt;

\* <p/>

\* &lt;execute-fitnesse-tests

\* suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;

\* fitnesseport=&quot;8082&quot;

\* resultsdir=&quot;${results.dir}&quot;

\* resultshtmlpage=&quot;fit-results.html&quot;

\* classpathref=&quot;classpath&quot; /&gt;

\* </pre>

\*/

* + Yerel olmayan bilgi
    - Bir yorum yazmanız gerekiyorsa, yanında göründüğü kodu açıkladığından emin olun. Yerel bir yorum bağlamında sistem çapında bilgi sunmayın.
  + Çok fazla bilgi
    - Yorumlarınıza ilginç tarihsel tartışmalar veya alakasız ayrıntılar açıklamaları koymayın.
  + Belirsiz bağlantı
    - Bir yorum ile açıkladığı kod arasındaki bağlantı açık olmalıdır.
    - Bir yorum yazma zahmetine girecekseniz, en azından okuyucunun yoruma ve koda bakabilmesini ve yorumun neden bahsettiğini anlayabilmesini istersiniz.
  + Fonksiyon başlıkları
    - Kısa fonksiyonların fazla açıklamaya ihtiyacı yoktur.
    - Tek bir şey yapan küçük bir fonksiyon için iyi seçilmiş bir ad, genellikle bir yorum başlığından daha iyidir.
  + Genel olmayan kodda javadocs
    - Javadoc'lar genel API'ler içindir, herkese açık olmayan kodda yardımdan çok dikkat dağıtıcı olabilir.

**Bölüm 5 - Formatlama**

* Kod formatı önemlidir.
* Kodunuzun formatını belirleyen bir takım kurallar benimsemeli ve o kurallara her zaman uymalısınız.
* Görmezden gelinemeyecek kadar önemli ve dindarca davranılmayacak kadar önemlidir.
* 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.
* Dikey formatlama
  + 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.
      * İsmin kendisi, bize doğru modülde olup olmadığımızı söyleyebilecek kadar net olmalı.
      * Detaylar aşağılarda olmalı.
      * Bir gazete de birçok 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
      * 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.

package fitnesse.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();

}

}

* + - * Gördüğünüz gibi, ilk örneğin okunabilirliği alttaki örnekten daha fazladır.

package fitnesse.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 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.
      * Değişken tanımlama.
        + Değişkenler, kullanıldıkları yere en yakın şekilde tanımlanmalıdır,
    - Örnek değişkenler,
      * 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.
    - Bağımlı fonksiyonlar
      * Bir fonksiyon diğerini çağırırsa, bunlar dikey olarak yakın olmalı ve mümkünse arayan, arananın üzerinde olmalıdır.
      * Bu, programa doğal bir akış sağlar.
      * Kural güvenilir bir şekilde izlenirse, okuyucular işlev tanımlarının kullanımlarından kısa bir süre sonra geleceğine güvenebileceklerdir.
    - Kavramsal yakınlık
      * Belirli kod bitleri, diğer bitlerin yakınında olmak ister.
      * Belli bir kavramsal yakınlıkları var.
      * Bu yakınlık ne kadar güçlüyse, aralarında o kadar az dikey mesafe olmalıdır.
    - Dikey sıralama
      * Genel olarak, işlev çağrısı bağımlılıklarının aşağı yönü göstermesini istiyoruz.
      * Yani çağrılan bir fonksiyon, çağıran bir fonksiyonun altında olmalıdır.
      * Bu, kaynak kod modülünde yüksek seviyeden düşük seviyeye güzel bir akış oluşturur.
* Yatay formatlama
  + Programcılar genelde kısa satırları tercih ederler.
  + Bir satır 120 karakterden uzun olmamalıdır.
  + Yatay açıklık ve yoğunluk
    - Yatay boşluğu, güçlü ilişki içerisinde olanları birleştirmek ve daha zayıf olanları ayırt etmek için kullanırız.

private void measureLine(String line) {

lineCount++;

int lineSize = line.length();

totalChars += lineSize;

lineWidthHistogram.addLine(lineSize, lineCount);

recordWidestLine(lineSize);

}

* + - * Atama ifadelerinin iki farklı ve ana öğesi vardır: sol taraf ve sağ taraf.
      * Boşluklar bu ayrımı belirgin hale getirir.
  + Yatay hizalama
    - Modern dillerde bu tür hizalama kullanışlı değildir.
    - Hizalama yanlış şeyleri vurguluyor gibi görünüyor ve gözümü gerçek niyetten uzaklaştırıyor.
  + Girinti
    - Girinti önemlidir çünkü görünür bir hiyerarşiye ve iyi tanımlanmış bloklara sahip olmamıza yardımcı olur.
    - 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.
  + Takım kuralları
    - Her programcının kendi favori biçimlendirme kuralları vardır, ancak bir ekipte çalışıyorsa, o zaman ekip kuralları koyar.
    - Bir geliştirici ekibi tek bir biçimlendirme stili üzerinde anlaşmalı ve ardından o ekibin her üyesi bu stili kullanmalıdır.
    - Yazılımın tutarlı bir stili olmasını istiyoruz.
    - Bir grup aynı fikirde olmayan kişi tarafından yazılmış gibi görünmesini istemiyoruz.

**Bölüm 6 - Nesneler ve Veri Yapıları**

* Değişkenlerimizi gizli tutmamızın bir nedeni var.
* Başka kimsenin onlara bağımlı olmasını istemiyoruz.
* Türlerini veya uygulamalarını bir kapris veya dürtüyle değiştirme özgürlüğünü korumak istiyoruz.
* Öyleyse neden bu kadar çok programcı nesnelerine otomatik olarak alıcılar ve ayarlayıcılar ekleyerek özel değişkenlerini herkese açıkmış gibi gösteriyor?
* Veri soyutlama
  + Gerçekleştirimi gizlemek, yalnızca değişkenler ile fonksiyonlar arasına bir katman koyma meselesi değildir.
  + Gerçekleştirimleri gizlemek, soyutlama ile ilgilidir.
  + Bir sınıf, değişkenlerini *getter*ve *setter*metotlar aracılığı ile dışarı açmaz; aksine gerçekleştirimi bilmelerine gerek olmadan veriyi değiştirmelerine izin veren arayüzleri açar.
* Veri/Nesne Anti-Simetrisi
  + Bu iki örnek, nesneler ve veri yapıları arasındaki farkı göstermektedir.
  + Nesneler, verilerini soyutlamaların arkasına gizler ve bu veriler üzerinde çalışan işlevleri ortaya çıkarır.
  + Veri yapısı, verilerini açığa çıkarır ve anlamlı işlevleri yoktur.

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 NoSuchShapeException {

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 sınıfı üzerinde işlemler yapıyor.
  + Şekil sınıfları hiçbir davranış sergilemeyen basit veri yapılarıdır.
  + Tüm davranış Geometry sınıfı içerisindedir.
  + 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.

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 hiçbiri 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 kanunu
  + Bir modülün manipüle ettiği nesnelerin iç kısımlarını bilmemesi gerektiğini söyleyen Demeter Yasası adlı iyi bilinen bir buluşsal yöntem vardır.
  + *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.”.
* Tren enkazları
  + Uç uca geçmiş bir dizi tren vagonu gibi birbirini çağıran fonksiyonlar için bu benzetme yapılır. Bu tür kodları parçalara ayırmak en iyisidir.
* Melez yapılar
  + 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.
* Veri aktarım nesneleri
  + Bir veri yapısının özlü biçimi, genel değişkenleri olan ve işlevleri olmayan bir sınıftır.
  + Buna bazen veri aktarım nesnesi veya DTO denir.
  + DTO'lar, özellikle veritabanlarıyla iletişim kurarken veya soketlerden gelen mesajları ayrıştırırken vb. çok kullanışlı yapılardır.
  + Bir veritabanındaki ham verileri uygulama kodundaki nesnelere dönüştüren bir dizi çeviri aşamasında genellikle ilk olurlar.
  + Biraz daha yaygını Bean formudur.
  + Bean’lerin getter/setter’larca değiştirilmiş private değişkenleri vardır.
  + Bazı Object Oriented püristlerini iyi hissettirebilir ancak başka hiçbir faydası yoktur.
  + Aktif Kayıt
    - 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.

**Bölüm 7 - Hata Yönetimi**

* Birçok kod tabanına tamamen hata işleme hakimdir.
* Domine ettiğimde, yaptıkları tek şeyin hata işleme olduğunu kastetmiyorum.
* Demek istediğim, tüm dağınık hata işleme nedeniyle kodun ne yaptığını görmek neredeyse imkansız. Hata işleme önemlidir, ancak mantığı engelliyorsa yanlıştır.
* Dönüş kodları yerine istisnaları kullanın
  + Uzak geçmişte, istisnaları olmayan birçok dil vardı.
  + Bu dillerde hataları ele alma ve raporlama teknikleri sınırlıydı.
  + Ya bir hata bayrağı ayarladınız ya da arayanın kontrol edebileceği bir hata kodu gönderdiniz.
* İk önce Try-Catch-Finally bloklarını yazın.
  + Try blokları bir bakıma işlemler gibidir.
  + Avınız, denemede ne olursa olsun, programınızı tutarlı bir durumda bırakmalıdır.
  + Bu nedenle, istisnalar oluşturabilecek bir kod yazarken try-catch-finally ifadesiyle başlamak iyi bir uygulamadır.
  + Bu, denemede yürütülen kodda ne ters giderse gitsin, o kodun kullanıcısının ne beklemesi gerektiğini tanımlamanıza yardımcı olur.
* Kontrolsüz (Unchecked) istisnalar 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, çağıranın geçebileceği tüm istisnaları listeleyecekti.
  + Daha da fazlası, bu istisnalar metodun tipinin bir parçasıydı.
  + 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, çağıran tüm metotlar da imzasına bir throws eklemek zorundadır.
  + Bu durumda kapsülleme de (encapsulation) bozulmuştur çünkü değişen metotlar da artık bu istisnanın detaylarını biliyor olacaktır.
* Bağlamı istisnalarla sağlayın
  + Attığınız her özel durum, bir hatanın kaynağını ve konumunu belirlemek için yeterli bağlamı sağlamalıdır.
  + Bilgilendirici hata mesajları oluşturun ve bunları istisnalarınızla birlikte iletin.
  + Başarısız olan işlemden ve hatanın türünden bahsedin.
  + Uygulamanızda oturum açıyorsanız, yakalamanızdaki hatayı günlüğe kaydedebilmek için yeterli bilgiyi iletin.
* Çağıranın ihtiyaçlarına göre istisna sınıflarını tanımlayın
  + Hataları sınıflandırabilmenin bir sürü yolu vardır; kaynaklarına göre, türlerine göre sınıflandırabiliriz.
  + Cihaz hatası mı, ağ hatası mı ya da programlama hataları mı?
  + 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
  + 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
  + Bir yöntemden null döndürmek istiyorsanız, bunun yerine bir istisna atmayı veya bir ÖZEL CASE nesnesi döndürmeyi düşünün.
  + Bir üçüncü taraf API'sinden boş değer döndüren bir yöntem çağırıyorsanız, bu yöntemi bir istisna oluşturan veya özel durum nesnesi döndüren bir yöntemle sarmalamayı düşünün.
* Null 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.
* 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 tüm yazılımları nadiren kontrol ederiz.
* Bazen üçüncü taraf paketleri satın alırız veya açık kaynak kullanırız.
* Diğer zamanlarda, bizim için bileşenler veya alt sistemler üretmeleri için kendi şirketimizdeki ekiplere güveniriz.
* Bir şekilde bu yabancı kodu kendi kodumuzla temiz bir şekilde entegre etmeliyiz.
* Üçüncü taraf kodu kullanma
  + Bir arayüzün sağlayıcısı ile arayüzün kullanıcısı arasında doğal bir gerilim vardır.
  + Üçüncü taraf paketlerin ve çerçevelerin sağlayıcıları, birçok ortamda çalışabilmeleri ve geniş bir kitleye hitap edebilmeleri için geniş uygulanabilirlik için çabalar.
  + Kullanıcılar ise kendi özel ihtiyaçlarına odaklanan bir arayüz isterler.
  + Bu gerilim, sistemlerimizin sınırlarında sorunlara neden olabilir.

Map sensors = new HashMap();

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

**VS**  
public class Sensors {

private Map sensors = new HashMap();

public Sensor getById(String id) {

return (Sensor) sensors.get(id);

}

//snip

}

* + İlk kod, Haritadaki dökümü gösterirken, ikincisi, uygulamanın geri kalanını çok az etkileyerek gelişebilir.
  + Döküm ve tip yönetimi, Sensörler sınıfı içinde gerçekleştirilir.
  + Arayüz ayrıca, uygulamanın ihtiyaçlarını karşılamak için uyarlanmış ve sınırlandırılmıştır.
  + Anlaşılması daha kolay ve kötüye kullanılması daha zor olan bir kodla sonuçlanır. Sensors sınıfı, tasarım ve iş kurallarını uygulayabilir.
* Sınırları keşfetmek ve öğrenmek
  + Üçüncü taraf kodu, daha kısa sürede teslim edilen daha fazla işlevsellik elde etmemize yardımcı olur.
  + Bazı üçüncü taraf paketlerini kullanmak istediğimizde nereden başlayacağız?
  + Üçüncü taraf kodunu test etmek bizim işimiz değil, ancak kullandığımız üçüncü taraf kodu için testler yazmak bizim çıkarımıza olabilir.
  + Bir üçüncü taraf kodunun nasıl kullanılacağını öğrenmek ve anlamak için bir test yazmak iyi bir fikirdir.
  + Newkirk bu tür testleri öğrenme testleri olarak adlandırır.
* Öğrenme testleri ücretsizden daha iyi
  + Öğrenme testleri hiçbir şeye mal olmaz.
  + Her halükarda API'yi öğrenmemiz gerekiyordu ve bu testleri yazmak, bu bilgiyi elde etmenin kolay ve izole bir yoluydu.
  + Öğrenme testleri, anlayışımızı artırmaya yardımcı olan kesin deneylerdi.
  + Öğrenme testleri ücretsiz olmasının yanı sıra olumlu bir yatırım getirisine sahiptir.
  + Üçüncü taraf paketinin yeni sürümleri olduğunda, davranışsal farklılıklar olup olmadığını görmek için öğrenme testleri çalıştırıyoruz.
* Henüz var olmayan kodu kullanmak
  + Bazen geliştirilmekte olan başka bir modüle bağlanacak bir modülde çalışmak gerekir ve API henüz tasarlanmadığı için bilgilerin nasıl gönderileceği hakkında hiçbir fikrimiz yoktur.
  + Bu durumlarda, bekleyen modül ile iletişimi kapsüllemek için bir arabirim oluşturmanız önerilir.
  + Bu şekilde modülümüzün kontrolünü elimizde tutuyoruz ve ikinci modül henüz mevcut olmasa da test edebiliyoruz.
* Temiz sınırlar
  + Sınırlarda ilginç şeyler oluyor.
  + Değişim de bunlardan biridir.
  + İyi yazılım tasarımları, büyük yatırımlar ve yeniden çalışma gerektirmeden değişime ayak uydurur.
  + Kontrolümüz dışında bir kod kullandığımızda, yatırımımızı korumak ve gelecekteki değişikliklerin çok maliyetli olmamasını sağlamak için özel dikkat gösterilmelidir.

**Bölüm 9- Birim Testleri**

* Test Driven Development
  + The three laws of TDD
    - Geçmeyen bir birim testi yazmadan, üretim (uygulama) kodu yazmamalısın.
    - Aynı anda birden fazla geçmeyen birim testi yazmamalısın. Derleme hatası da geçmeyen test demektir.
    - O andaki geçmeyen testi geçirecek üretim kodundan başka üretim kodu yazmamalısın.
* Testleri temiz tutmak
  + 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.
* Testler -ilities'i etkinleştirir
  + 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 kapsamı yüzdesi yükseldikçe değişikliklerden daha az korkarsınız.
  + Hatta korkmadan mimarinizi ve tasarımınızı bile geliştirebilirsiniz.
  + Tasarımınızı ve mimarinizi hücrelerini temiz tutmanın anahtarı üretim defterini kapsayan otomatik (otomatik) birim testlere sahip olmaktır.
* Temiz testler
  + Testlerinizi temiz tutmak için okunabilirlik çok önemlidir.
  + Tüm kodu okunabilir kılan aynı şeyler: açıklık, basitlik ve ifade yoğunluğu.
* Etki alanına özgü test dili
  + Programcıların sistemi manipüle etmek için kullandıkları API'leri kullanmak yerine, bu API'leri kullanan ve testleri yazmayı ve okumayı daha kolay hale getiren bir dizi işlev ve yardımcı program oluşturuyoruz.
  + Bu işlevler ve yardımcı programlar, testler tarafından kullanılan özel bir API haline gelir.
  + Programcıların testlerini yazmalarına ve daha sonra bu testleri okuması gerekenlere yardımcı olmak için kullandıkları bir test dilidir.
  + Bu test API'si önceden tasarlanmamıştır; bunun yerine, kafa karıştıran ayrıntılarla fazlasıyla lekelenmiş olan test kodunun sürekli olarak yeniden düzenlenmesinden gelişir.
* Çift 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.
  + Üretim ortamında asla yapmayacağımız fakat test ortamında uygulayabileceğimiz şeyler vardır.
  + Genellikle bu şeyler bellek veya CPU verimliliği meseleleridir.
* Test başına bir onay
  + Test başına yalnızca bir iddianın sürdürülmesi önerilir çünkü bu, her testin anlaşılmasını kolaylaştırmaya yardımcı olur.
* Test başına tek konsept
  + Bu kural, kısa işlevleri korumanıza yardımcı olacaktır.
  + Doğrulamanız gereken her kavram için bir test yazın
* F.I.R.S.T
  + Hızlı, Test hızlı olmalıdır.
  + Bağımsız, Test birbirine bağlı olmamalıdır.
  + Tekrarlanabilir, Test her ortamda tekrarlanabilir olmalıdır.
  + Kendini Doğrulayan, Test'in bir boole çıktısı olmalıdır ya geçerler ya da başarısız olurlar.
  + Zamanında, Birim testleri, geçmelerini sağlayan üretim kodundan hemen önce yazılmalıdır. Testleri üretim kodundan sonra yazarsanız, üretim kodunu test etmenin zor olduğunu görebilirsiniz.

**Bölüm 10 – Sınıflar**

* Sınıf organizasyonu
  + Değişkenlerden sonra public fonksiyonlar gelmelidir.
  + public fonksiyonlardan hemen sonra ise public bir fonksiyon tarafından çağrılmış yardımcı private metotlar gelebilir.
  + Kapsülleme
    - Değişkenlerimizi ve yardımcı fonksiyonlarımızı küçük tutmayı seviyoruz, ancak bu konuda fanatik değiliz.
    - Bazen, bir testle erişilebilmesi için bir değişkeni veya yardımcı fonksiyonu korumalı hale getirmemiz gerekir.
  + Sınıflar küçük olmalıdır
    - First Rule: Classes should be small.
    - Second Rule: Classes should be smaller than the first rule.
  + Tek sorumluluk ilkesi
    - 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.
    - 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.
  + Uyum
    - 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.
  + Birçok küçük sınıfta uyum sonuçlarını sürdürme
    - 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 için düzenleme
    - Çoğu sistem için değişim süreklidir.
    - Her değişiklik bizi, sistemin geri kalanının artık amaçlandığı gibi çalışmaması riskine maruz bırakır.
    - Temiz bir sistemde sınıflarımızı değişiklik riskini azaltacak şekilde düzenliyoruz.
  + Değişimden yalıtmak
    - İhtiyaçlar değişeceği için kod da değişecektir.
    - Uygulama ayrıntılarını (kod) içeren somut sınıflar ve yalnızca kavramları temsil eden soyut sınıflar olduğunu OO 101'de öğrendik.
    - Somut ayrıntılara bağlı bir müşteri sınıfı, bu ayrıntılar değiştiğinde risk altındadır. Bu ayrıntıların etkisini yalıtmaya yardımcı olmak için arayüzler ve soyut sınıflar sunabiliriz.

**Bölüm 11 -Sistemler**

* Bir sistem Kurmayı Kullanmaktan Ayırın
  + Yazılım Sistemleri, uygulama nesneleri oluşturulduğunda ve bağımlılıklar birlikte "kablolandığında" başlatma sürecini, başlatmadan sonra devralan çalışma zamanı mantığından ayırmalıdır.
  + Ana parçadan ayırma
    - Konstrüksiyonu kullanımdan ayırmanın bir yolu, basitçe konstrüksiyonun tüm yönlerini ana modüllere veya ana tarafından çağrılan modüllere taşımak ve sistemin geri kalanını tüm nesnelerin uygun şekilde inşa edildiğini ve kablolandığını varsayarak tasarlamaktır.
    - Soyut Fabrika Kalıbı, bu tür bir yaklaşım için bir seçenektir.
  + Bağımlılık Enjeksiyonu
    - Yapıyı kullanımdan ayıran güçlü bir mekanizma, Kontrolün Ters Çevirilmesinin (IoC) bağımlılık yönetimine uygulanması olan Dependency Injection'dır (DI).
    - Kontrolün tersine çevrilmesi, ikincil sorumlulukları bir nesneden amaca tahsis edilmiş diğer nesnelere taşır ve böylece Tek Sorumluluk İlkesini destekler.
    - Bağımlılık yönetimi bağlamında, bir nesne bağımlılıkları kendisi başlatma sorumluluğunu almamalıdır.
    - Bunun yerine, bu sorumluluğu başka bir "yetkili" mekanizmaya devrederek kontrolü tersine çevirmelidir.
    - Kurulum küresel bir sorun olduğundan, bu yetkili mekanizma genellikle "ana" yordam veya özel amaçlı bir kapsayıcı olacaktır.

**Bölüm 12 - Ortaya çıkış**

* Kent Beck'e göre, bu kurallara uyan bir tasarım "basittir".
  + Tüm testleri çalıştır.
    - Kapsamlı bir şekilde test edilen ve tüm testlerini her zaman geçiren bir sistem test edilebilir bir sistemdir.
    - Bir sistemin test edilebilir olması önemlidir.
    - Test edilebilir olmayan sistemler doğrulanabilir de değildir.
    - Doğrulanamayan bir sistem asla dağıtılmamalıdır
    - Burası ayrıca basit tasarımın son üç kuralını uyguladığımız yerdir: tekrarları kaldır, kodunun açıklayıcı olduğundan emin ol, sınıf ve metot sayılarını en aza indir.
  + Tekrarlama içermez.
    - Tekrarlanmış kodlar, iyi tasarlanmış sistemlerin birinci düşmanıdır.
    - Tekrarlanmış kodlar, ek iş, ek risk ve gereksiz karmaşa demektir.
  + Programcıların niyetini ifade eder.
    - Bir yazılım projesinin maliyetinin büyük bir kısmı uzun vadeli bakımdadır.
    - Değişimi başlatırken olası kusurları en aza indirmek için, bir sistemin ne yaptığını anlayabilmek bizim için çok önemlidir.
    - Kod, kusurları azaltmak ve bakım maliyetini azaltmak için yazarının amacını açıkça ifade etmelidir.
    - Bu, iyi isimler seçerek, fonksiyonları ve sınıfları küçük tutarak, standart terminolojiyi kullanarak ve iyi yazılmış birim testleri yazarak yapılabilir.
    - Kişinin yaptığı işten gurur duyması ve kodu başkalarının anlaması için kolaylaştırmaya zaman ayırması önemlidir.
  + Sınıf ve metot sayısını en aza indirir.
    - Yinelemenin ortadan kaldırılması, kod ifadesi ve SRP gibi temel kavramlar bile çok ileri götürülebilir.
    - Sınıflarımızı ve yöntemlerimizi küçültmek için çok fazla küçük sınıf ve yöntem oluşturabiliriz.
    - Dolayısıyla bu kural, işlev ve sınıf sayımızı da düşük tutmamızı önerir. Yüksek sınıf ve yöntem sayıları bazen anlamsız dogmatizmin sonucudur ve daha pragmatik bir yaklaşım lehine direnilmelidir.
    - Amacımız, işlevlerimizi ve sınıflarımızı küçük tutarken genel sistemimizi küçük tutmaktır, ancak bu kural, Basit Tasarımın dört kuralı arasında en düşük önceliğe sahip olanıdır.
    - Sınıf ve işlev sayısını düşük tutmak önemlidir, ancak test yaptırmak, tekrarı ortadan kaldırmak ve kendini ifade etmek daha önemlidir.

**Bölüm 13 - Eşzamanlılık**

* Uyum bir ayrıştırma stratejisidir.
* Neyin bittiğinde neyin işe yaradığını ayırmamıza yardımcı olur.
* Tek iş parçacıklı uygulamalarda ne ve ne zaman o kadar güçlü bir şekilde birleştirilir ki, tüm uygulamanın durumu genellikle yığın geri izlemesine bakılarak belirlenebilir.
* Böyle bir sistemde hata ayıklayan bir programcı, bir kesme noktası veya bir kesme noktaları dizisi ayarlayabilir ve kesme noktalarına isabet eden sistemin durumunu bilebilir.
* Neyi ne zamandan ayırma, bir uygulamanın hem verimini hem de yapısını önemli ölçüde iyileştirebilir.
* Yapısal bir bakış açısından, uygulama tek bir büyük ana döngüden çok, işbirliği yapan birçok küçük bilgisayar gibi görünür.
* Bu, sistemin anlaşılmasını kolaylaştırabilir ve endişeleri ayırmak için bazı güçlü yollar sunar.
* Hayaller ve yanılgılar.
  + Eşzamanlılık her zaman performansı artırır.
  + Eşzamanlılık bazen performansı artırabilir, ancak yalnızca birden çok iş parçacığı veya birden çok işlemci arasında paylaşılabilecek çok fazla bekleme süresi olduğunda.
  + Hiçbir durum önemsiz değildir.
  + Eşzamanlı programlar yazarken tasarım değişmez.
  + Aslında, eşzamanlı bir algoritmanın tasarımı, tek iş parçacıklı bir sistemin tasarımından oldukça farklı olabilir.
  + Neyin ne zamandan itibaren ayrıştırılması genellikle sistemin yapısı üzerinde büyük bir etkiye sahiptir.
  + Eşzamanlılık sorunlarını anlamak, Web veya EJB kapsayıcısı gibi bir kapsayıcıyla çalışırken önemli değildir.
  + Aslında, kapsayıcınızın ne yaptığını ve bu bölümde daha sonra açıklanan eşzamanlı güncelleme ve kilitlenme sorunlarına karşı nasıl korunacağını bilseniz iyi olur.
  + İşte eşzamanlı yazılım yazmaya ilişkin birkaç daha dengeli açıklama:
  + Eşzamanlılık, hem performansta hem de ek kod yazmada bazı ek yüklere neden olur.
  + Doğru eşzamanlılık, basit problemler için bile karmaşıktır.
  + Eşzamanlılık hataları genellikle tekrarlanabilir değildir, bu nedenle gerçek kusurlar yerine genellikle tek seferlik hatalar olarak görmezden gelinirler.
  + Eşzamanlılık genellikle tasarım stratejisinde temel bir değişiklik gerektirir.

**Bölüm 14- Ardışık İyileştirme**

* Bu bölümde Ardışık İyileştirme hakkında örnek içeriyor.
* Bunu Nasıl Yaptım?
  + Son birkaç on yılda bir şey öğrendiysek, o da programlamanın bir bilimden çok bir zanaat olduğudur.
  + Temiz kod yazmak için önce kirli kod yazmanız ve ardından temizlemeniz gerekir.
  + Bu sizin için sürpriz olmamalı.
  + Bu gerçeği ilkokulda öğretmenlerimiz (genellikle nafile) bestelerimizin kaba taslaklarını bize yazdırdığında öğrendik.
  + Bize, sürecin kaba bir taslak yazmamız, ardından ikinci bir taslak yazmamız ve ardından nihai versiyonumuz olana kadar birkaç müteakip taslak yazmamız gerektiğini söylediler.
  + Bize söylemeye çalıştıkları temiz kompozisyonlar yazmak, birbirini izleyen bir iyileştirme meselesidir.

**Bölüm 15- JUnit Internals**

* JUnit, tüm Java çerçevelerinin en ünlülerinden biridir.
* Çerçeveler söz konusu olduğunda, tasarım açısından basit, tanım açısından kesin ve uygulama açısından zariftir.
* Bu bölümde, JUnit çerçevesinden alınan bir örnekten bahsediyor.
* Kent Beck ve Eric Gamma, jUnit adlı bir test programı geliştirmeye başladı.
* Harika bir test aracıdır.
* Pek çok geliştirici ve şirket, java kodlarını test etmek için bu aracı kullanır.

**Bölüm 16- Seri Verileri Yeniden Düzenleme**

* Bu bölüm SerialDate kütüphanesinin iyileştirilmesi hakkında örnek içeriyor.
* SerialDate aslında nedir.
  + SerialDate, Java'daki tarih işlevini temsil eden bir sınıftır.
* Bir kodu yeniden düzenlemek, bir proje için çok önemli bir şeydir.
* Temiz olmayan kod herhangi bir sistemde olabilir.
* Onları sık sık yeniden düzenlememiz gerekiyor.
* Yeniden düzenleme için birkaç adım atmamız gerekiyor.
* Tüm kodu test edin ve test hatalarını düzeltin.

**Bölüm 17 - Kokular ve Buluşsal Yöntemler**

* Bir kod referansı, Martin Fowler'ın Yeniden Düzenleme ve Robert C Martin'in Temiz Kodundan kokuyor.
* Temiz kod bir liste veya değer sisteminden değil, disiplinden gelse de, işte bir başlangıç noktası.
* Yorumlar
  + C1: Uygunsuz Bilgi
    - Kod ve tasarıma atıfta bulunan teknik notlar için yorumları ayırın.
  + C2: Eski Yorum
    - Eski yorumları güncelleyin veya silin.
  + C3: Gereksiz Yorum
    - Gereksiz bir yorum, kendisini yeterince tanımlayabilen bir şeyi tanımlar.
  + C4: Kötü Yazılmış Yorum
    - Yorumlar kısa, öz ve doğru yazılmış olmalıdır.
  + C5: Yorumlanmış Kod
    - Hayalet kodu. Silin.
* Çevre
  + E1: Derleme Birden Fazla Adım Gerektirir
    - Derlemeler, teslim almak için bir komut ve çalıştırmak için bir komut gerektirmelidir.
  + E2: Testler Birden Fazla Adım Gerektirir
    - Testler, bir IDE aracılığıyla tek bir düğme tıklamasıyla veya tek bir komutla çalıştırılmalıdır.
* Fonksiyonlar
  + F1: Çok Fazla Argüman
    - İşlevlerin bağımsız değişkeni olmamalıdır, sonra bir, sonra iki, sonra üç. Üçten fazla değil.
  + F2: Çıktı Argümanları
    - Bağımsız değişkenler girdilerdir, çıktılar değil. Bir şeyin durumunun değiştirilmesi gerekiyorsa, çağrılan nesnenin durumu olsun.
  + F3: Bağımsız Değişkenleri İşaretle
    - Boole bağımsız değişkenlerini ortadan kaldırın.
  + F4: Ölü İşlev
    - Çağrılmayan yöntemleri atın. Bu ölü kod.
* Genel
  + G1: Tek Kaynak Dosyada Birden Çok Dil
    - Bir kaynak dosyadaki dil sayısını en aza indirin. İdeal olarak, sadece bir tane.
  + G2: Bariz Davranış Gerçekleştirilmemiştir
    - Bir fonksiyonun veya sınıfın sonucu sürpriz olmamalıdır.
  + G3: Sınırlarda Yanlış Davranış
    - Her sınır koşulu için testler yazın.
  + G4: Geçersiz Kılınan Emniyetler
    - Güvenlikleri geçersiz kılmak ve manuel kontrol uygulamak, kodun erimesine neden olur.
  + G5: Çoğaltma
    - Yinelenen kod üzerinde soyutlama alıştırması yapın. Tekrarlayan işlevleri çok biçimlilik ile değiştirin.
  + G6: Yanlış Soyutlama Düzeyinde Kod
    - Soyutlanan kodun farklı kaplara ayrıldığından emin olun.
  + G7: Türevlerine Göre Temel Sınıflar
    - Modülerliği uygulayın.
  + G8: Çok Fazla Bilgi
    - Az ile çok şey yapın. Bir sınıfta veya işlevlerde olan şeylerin miktarını sınırlayın.
  + G9: Ölü Kod
    - Yürütülmemiş kodu silin.
  + G10: Dikey Ayırma
    - Değişkenleri ve işlevleri çağrıldıkları yere yakın tanımlayın.
  + G11: Tutarsızlık
    - Bir sözleşme seçin ve onu takip edin. Unutma, sürpriz yok.
  + G12: Dağınıklık
    - Ölü kod.
  + G13: Yapay Kaplin
    - Kullanışlı olmaktansa açık olan kodu tercih edin. Zihinsel eşlemeyi netliğe tercih eden kodu gruplamayın.
  + G14: Özellik Kıskançlığı
    - Bir sınıfın metotları başka bir sınıfın metotlarıyla ilgilenmemelidir.
  + G15: Seçici Argümanlar
    - İşlevlerin sonunda yanlış argümanlar göstermeyin.
  + G16: Gizlenmiş Niyet
    - Kod sihirli veya belirsiz olmamalıdır.
  + G17: Yersiz Sorumluluk
    - Kodunuzu nereye yerleştireceğinize ilişkin geçiş noktaları olarak açık işlev adını kullanın.
  + G18: Uygunsuz Statik
    - İşlevlerinizi statik olmayan yapın.
  + G19: Açıklayıcı Değişkenler Kullanın
    - Açıklayıcı değişkenler yapın ve birçoğunu yapın.
  + G20: İşlev Adları Ne Yaptıklarını Söylemeli
  + G21: Algoritmayı Anlayın
    - Bir işlevin nasıl çalıştığını anlayın. Testleri geçmek yeterli değildir. Bir işlevi yeniden düzenlemek, onun daha iyi anlaşılmasını sağlayabilir.
  + G22: Mantıksal Bağımlılıkları Fiziksel Hale Getirin
    - Kodunuzun ne yaptığını anlayın.
  + G23: Polimorfizmi If/Else veya Switch/Case'e Tercih Et
    - Anahtarın/kutunun kaba kuvvetinden kaçının.
  + G24: Standart Sözleşmeleri Takip Edin
    - Takımınızın konvansiyonunun ne olduğu önemli değil. Sadece sende var ve herkes onu takip ediyor.
  + G25: Sihirli Sayıları Adlandırılmış Sabitlerle Değiştirin
    - Numaraları yazmayı bırak.
  + G26: Kesin Olun
    - Tembel olma Olası sonuçları düşünün, ardından bunların üzerini örtün ve test edin.
  + G27: Gelenek Yerine Yapı
    - Tasarım kararları bir dogma yerine bir yapıya sahip olmalıdır.
  + G28: Koşulluları Kapsülle
    - Koşullarınızı daha kesin hale getirin.
  + G29: Olumsuz Koşullu Cümlelerden Kaçının
    - Olumsuz koşullu önermeleri anlamak, olumlu koşullandırmaya göre daha fazla beyin gücü gerektirir.
  + G31: Gizli Geçici Kaplinler
    - Zamansal eşleşmeyi açık hale getiren argümanlar kullanın.
  + G32: Keyfi Olma
    - Kodunuzun yapısı, yapısının nedenini iletmelidir.
  + G33: Sınır Koşullarını Kapsülleyin
    - +1'leri ve -1'leri kodunuza sızdırmaktan kaçının.
  + G34: İşlevler Yalnızca Bir Soyutlama Düzeyinden Düşmelidir
    - Takip edilmesi en zor buluşsal yöntem. İşlevin açıklanan işleminin altındaki bir düzey soyutlama, kodunuzu netleştirmenize yardımcı olabilir.
  + G35: Yapılandırılabilir Verileri Yüksek Düzeyde Tutun
    - Yüksek seviyeli sabitlerin değiştirilmesi kolaydır.
  + G36: Geçişli Navigasyondan Kaçının
    - Utangaç kodu yaz. Modüller yalnızca komşularını bilmeli, komşularının komşularını bilmemelidir.
* İsimler
  + N1: Açıklayıcı İsimler Seçin
    - Açıklayıcı ve alakalı isimler seçin.
  + N2: Uygun Soyutlama Düzeyinde Adları Seçin
    - Farklı programlarda kullanıldığında kullanıcı tarafından hala anlaşılır olan adları düşünün.
  + N3: Mümkün Olduğunda Standart Terminoloji Kullanın
    - Görevlerini ifade eden isimler kullanın.
  + N4: Açık İsimler
    - Kısalıktansa netliği tercih edin. Uzun, anlamlı bir isim, kısa ve donuk bir isimden daha iyidir.
  + N5: Uzun Kapsamlar İçin Uzun İsimler Kullanın
    - Bir ismin uzunluğu, kapsamı ile ilgili olmalıdır.
  + N6: Kodlamalardan Kaçının
    - İsimleri tür veya kapsam bilgisi ile kodlamayın.
  + N7: İsimler Yan Etkileri Açıklamalı
    - İşlevinizin yan etkilerini göz önünde bulundurun ve bunu adına ekleyin.
* Testler
  + T1: Yetersiz Testler
    - Kırılabilecek her şeyi test edin
  + T2: Bir Kapsam Aracı kullanın!
    - IDE'nizi bir kapsama aracı olarak kullanın.
  + T3: Önemsiz Testleri Atlamayın
  + T4: Yok Sayılan Bir Test, Bir Belirsizlikle İlgili Bir Sorudur
    - Testiniz göz ardı edilirse, kod sorgulanır.
  + T5: Sınır Koşullarını Test Edin
    - Ortası genellikle örtülür. Sınırları hatırla.
  + T6: Yakın Hataları Kapsamlı Bir Şekilde Test Edin
    - Böcekler nadiren yalnızdır. Birini bulduğunuzda, başka bir yakına bakın.
  + T7: Başarısızlık Modelleri Ortaya Çıkıyor
    - İyi düzenlenmiş test senaryoları, başarısızlık kalıplarını ortaya çıkaracaktır.
  + T8: Test Kapsamı Modelleri Açıklayıcı Olabilir
    - Benzer şekilde, bir başarısızlıkta geçirilen veya geçirilmeyen koda bakın.
  + T9: Testler Hızlı Olmalı
    - Yavaş testler çalıştırılmayacak