Clean Code

Kitap Özeti

Clean Code

21/10/2021

Robert C. Martin Kitap Özeti

Bölüm 1 – Temiz Kod

- Kitap genel olarak iyi programlama pratiklerini, temiz kod, adı altında, belirli başlıklarla anlatıyor,
- Planlama veya herhangi bir aşamada, yönetimi durum ile ilgili bilgilendirip, doğruları söylemek geliştiricilerin de sorumluluğu,
- Temiz kod odaklı olmalıdır. Her fonksiyon, her sınıf, her modül, çevredeki ayrıntılar tarafından tamamen dikkati dağılmayan ve kirletilmeyen tek fikirli bir tavrı ortaya koyar.
- Temiz kod basit ve direktir. Temiz kod tasarımcının amacını gölgelemez, aksine daha net soyutlamalar ve basit kontroller ile doludur.
- Okuduğunuz her fonksiyon beklediğiniz gibi çıktığında, temiz kod üzerinde çalıştığınızı bilirsiniz. Kod, dilin sorun için yapılmış gibi görünmesini sağladığında da buna güzel kod diyebilirsiniz.
- Kod zamanla da temiz tutulmalıdır,
- «Kamp yerini bulduğunuzdan temiz bırakın» prensibi,
- Bir değişkenin ismini daha iyi olması için değiştirin, büyük bir fonksiyonu daha küçük parçalara bölün, küçük de olsa tekrarlardan birini giderin ve birleşik if'lerden birini temizleyin.

Bölüm 2 – Anlamlı İsimlendirme

- Bir değişkenin, fonksiyonun veya sınıfın adı tüm büyük soruları yanıtlamalıdır. Size neden var olduğunu, ne yaptığını ve nasıl kullanıldığını anlatmalıdır. Bir isim bir yorum gerektiriyorsa, o zaman isim onun amacını ifşa etmez.
- Programcılar, kodun anlamını gölgeleyen yanlış ipuçları bırakmaktan kaçınmalıdır. Yerleşik anlamları, kastettiğimiz anlamdan farklı olan kelimelerden kaçınmalıyız.
- Ayrımlar anlamlı yapılmalıdır,
- Telaffuz edilebilir isimler kullanın,
- Aranabilir adlar kullanın, tek harfli adlar ve sayısal sabitler, bir metin gövdesinde bulunmalarının kolay olmaması nedeniyle sıkıntı yaratır,
- Benim kişisel tercihim, tek harfli isimlerin SADECE kısa metotlarda yerel değişkenler olarak kullanılabilmesidir. Bir adın uzunluğu, tanımlı olduğu kapsamın boyu ile orantılı olmalıdır. Bir değişken veya sabit, bir kod gövdesinde birden çok yerde görülebiliyor veya kullanılıyorsa, ona arama dostu bir ad vermek zorunludur.
- Akıllı bir programcı ile profesyonel bir programcı arasındaki bir fark, profesyonelin netliğin kral olduğunu anlamasıdır. Profesyoneller güçlerini iyilik için kullanırlar ve başkalarının anlayabileceği kodlar yazarlar.
- Sınıf isimleri
  - Sınıflar ve nesneler, Müşteri, Sayfa, Hesap ve AderesAyıklayıcısı gibi isim veya isim tümcesi adlarına sahip olmalıdır. Sınıf adında Yönetici, İşlemci, Veri veya Bilgi gibi sözcüklerden kaçının. Bir sınıf adı bir fiil olmamalıdır.
- Fonksiyon isimleri
    - Fonksiyonlar, ödemeYap, sayfaSil veya kaydet gibi fiil veya fiil tümcesi adlarına sahip olmalıdır. Erişimciler, mutatörler ve yüklemler değerlerine göre adlandırılmalıdır.
- Soyut bir kavram için bir kelime seçin ve ona bağlı kalın. Örneğin, farklı sınıfların eşdeğer yöntemleri olarak getirme ve alma kafa karıştırıcıdır.
- İyi isimler seçmenin en zor yanı, iyi bir tanımlayıcı beceriler ve ortak bir kültürel geçmiş gerektirmesidir. Bu, teknik, ticari veya yönetimle ilgili bir sorundan ziyade bir öğretim sorunudur. Sonuç olarak, bu alandaki birçok insan bunu çok iyi yapmayı öğrenmiyor.

Bölüm 3 - Fonksiyonlar

- Peki, bir fonksiyonun okunmasını ve anlaşılmasını kolaylaştıran nedir? Bir işlevin amacını aktarmasını nasıl sağlayabiliriz? Sıradan bir okuyucunun içinde yaşadıkları program türünü sezmesine izin verecek işlevlerimize hangi özellikleri verebiliriz? Küçük olmalıdırlar!
- Fonksiyonların ilk kuralı, küçük olmalarıdır. Fonksiyonların ikinci kuralı, bundan daha küçük olmaları gerektiğidir,
- Fonksiyonlar bir şey yapmalıdırlar, fonksiyonlar bir şey yapmalıdır, iyi yapmalıdır ve sadece bunu yapmalılar,
- Bir işlev, içerisinde bulunduğu seviyenin bir alt seviyesine ait işler yapıyor olsa da yine bir işlev yapıyor kabul edilebilir,
- Bir işlevin "bir şeyden" fazlasını yaptığını bilmenin başka bir yolu, ondan yalnızca uygulamasının yeniden ifadesi olmayan bir adla başka bir işlev çıkarabilmenizdir. Eğer bunu yapabiliyorsanız, bu fonksiyonu gözden geçirmenizde fayda var,
- İşlev Başına Bir Soyutlama Düzeyi,
- Kodu Yukarıdan Aşağıya Okumak: Adım Atma Kuralı,
- Switch deyimleri için genel kuralım, yalnızca bir kez göründüklerinde, polimorfik nesneler oluşturmak için kullanıldığında ve sistemin geri kalanının onları görememesi için bir kalıtım ilişkisinin arkasına gizlenmelerinde tolere edilebilir olmalarıdır [G23]. Elbette her durum benzersizdir ve bu kuralın bir veya daha fazla bölümünü ihlal ettiğim zamanlar vardır.
- Tanımlayıcı adlar kullanın,
- Bir ismi uzatmaktan korkmayın. Uzun açıklayıcı bir ad, kısa gizemli bir adtan daha iyidir,
- Bir isim seçmek için zaman harcamaktan korkmayın,
- Bir fonksiyon için ideal argüman sayısı sıfırdır (niladik). Ardından bir (monadik) gelir, ardından iki (dyadik) yakından gelir. Mümkünse üç argümandan (üçlü) kaçınılmalıdır. Üçten fazlası (poliadik) çok özel gerekçeler gerektirir ve bu durumda yine de kullanılmamalıdır,
- Çıktı argümanlarını anlamak, girdi argümanlarını anlamaktan daha zordur. Bir fonksiyonu okuduğumuzda, bilginin fonksiyona argümanlar yoluyla girip, geri dönüş değeri yoluyla dışarı çıkması fikrine alışkınız. Genellikle bilgilerin argümanlar yoluyla dışarı çıkmasını beklemiyoruz. Bu nedenle çıktı argümanları genellikle iki kez almamıza neden olur.
- Bayrak Argümanları: Bayrak argümanları çirkindir. Boole değerini bir fonksiyona geçirmek gerçekten korkunç bir uygulamadır.
- Bir fonksiyon iki veya üçten fazla argümana ihtiyaç duyduğunda, bu argümanlardan bazılarının kendi sınıflarına sarılması muhtemeldir.

Bölüm 3 – Fonksiyonlar - Devam

- Fonksiyon yan etkileri sıkıntılıdır. İşleviniz bir şeyi yapmayı vaat ediyor, ancak aynı zamanda başka gizli şeyler de yapıyor ise bu ileride başınızı ağrıtabilir. Bazen kendi sınıfının değişkenlerinde beklenmedik değişiklikler yapacaktır.
- Parametreler doğal olarak bir fonksiyonun girdileri olarak yorumlanır. Genel olarak çıktı argümanlarından kaçınılmalıdır.
- Fonksiyonlar ya bir şey yapmalı ya da bir şeye cevap vermelidir, ancak ikisini birden değil.
- Hata kodlarını döndürmek yerine istisnaları tercih edin,
- Komut işlevlerinden hata kodlarının döndürülmesi, komut sorgusu ayrımının ince bir ihlalidir,
- Kendinizi Tekrar Etmeyin (DRY prensibi),
- Yazılımdaki tüm kötülüklerin kaynağı duplikasyon olabilir.

- Fonksiyon yazdığımda uzun ve karmaşık oluyorlar. Çok sayıda girinti ve iç içe döngüleri vardır. Uzun argüman listeleri var. Adlar isteğe bağlıdır ve yinelenen kod vardır. Ama aynı zamanda, bu beceriksiz kod satırlarının her birini kapsayan bir dizi birim testim var. Sonra bu kodun üzerinden geçip, düzeltiyorum, işlevleri bölüyorum, isimleri değiştiriyorum, tekrarlamayı ortadan kaldırıyorum. Yöntemleri küçültüp yeniden sıralarım. Bazen tüm sınıfları kırıyorum, bu arada testleri geçiyorum (bir fonksiyonun refaktör edilmesi için izlenebilecek yöntem).
- Fonksiyonlar o dilin fiilleridir ve sınıflar isimlerdir.
- Usta programcılar, sistemleri yazılacak programlardan ziyade anlatılacak hikayeler olarak düşünürler.

Bölüm 4 – Yorumlar

- -Güzel bir söz: “Kötü kodu yoruma almayın, yeniden yazın.” —Brian W. Kernighan and P. J. Plaugher
- Yorumların doğru kullanımı, kendimizi kodla ifade edememezi, başarısızlığımızı telafi etmektir. Başarısızlık kelimesini kullandığıma dikkat edin. Onu kastetmiştim. Yorumlar her zaman başarısızlıktır. Onlara sahip olmalıyız çünkü onlarsız kendimizi nasıl ifade edeceğimizi her zaman çözemeyiz, ancak kullanımları bir kutlama nedeni değildir.
- Nedeni basit. Programcılar onları gerçekçi bir şekilde koruyamaz.
- Yorumlar kötü kodu düzeltmez
- Kodta kendinizi açıklayın,

- İyi Yorumlar
  - Yasal/lisanssal yorumlar
  - Bilgilendirici yorumlar
  - Amacın açıklanması
  - Açıklığa kavuşturma
  - Sonuçlara karşın uyarma
  - ToDo yorumları
  - Önemini Güçlendirme
- Kötü Yorumlar
  - Mırılandma ya da gereksin kod yorumları
  - Yanıltıcı yorumlar
  - Otomot her unsur için her zaman yorum ekleme yaklaşımı
  - Günlükvari yorumlar
  - Fonksiyon ya da değişken yerine bir yorum kullanma
  - Yer belirteçleri
  - Parantez başlangıç/bitiş işaretleri
  - Kişilere atıflar
  - Yorumlanmış kodlar
  - HTML yorumlar
  - Tanımlandığı yerlerden uzak yorumlar
  - Gereksiz bilgi

Bölüm 5 – Formatlama

- Dikey Formatlama
    - Gazete Metaforu: Kod genelde gazete gibi yukarıdan aşağı ilerlemeli
    - Kavramlar Arası Dikey Açıklık: Kavramlar arasında boşluk bırakılması
    - Dikey Yoğunluk: İlgili kavramların birbirine yakın olması
    - Dikey Mesafe: İlgili bileşenler aynı ortamda ve çok uzakta olmaması
    - Değişken tanımları: Değişkenler kullanıldıkları yerlere olabildiğince yakın tanımlanmalıdır
    - Bağımlı fonksiyonlar: Birbirine bağımlı fonksiyonlar, dikey olarak birbirine yakın olmalı ve çağıran, cağrılandan önce gelmelidir,
- Yatay Formatlama
    - Yatay Açıklık ve Yoğunluk: Güçlü bir şekilde ilişkili şeyleri ilişkilendirmek ve daha zayıf ilişkili şeyleri ayırmak için yatay beyaz boşluk kullanırız.
    - Girinti: Bu kapsam hiyerarşisini görünür kılmak için, kaynak kodu satırlarını hiyerarşideki konumlarıyla orantılı olarak girintiliyoruz.
- Takım Kuralları
    - Her programcının kendi favori biçimlendirme kuralları vardır, ancak bir takımda çalışıyorsa, o zaman takım kuralları.

Bölüm 6 – Nesneler ve Veri Yapıları

- Veri Soyutlama
  - İmplementasyonu gizlemek, yalnızca değişkenler arasına bir işlev katmanı yerleştirme meselesi değildir. Uygulamayı gizlemek soyutlamalarla ilgilidir! Bir sınıf, değişkenlerini alıcılar ve ayarlayıcılar aracılığıyla basitçe dışarı itmez. Bunun yerine soyut arayüzleri ortaya çıkarır.
  - Verilerimizin ayrıntılarını ifşa etmek istemiyoruz. Bunun yerine verilerimizi soyut terimlerle ifade etmek istiyoruz.
- Veri/Nesne Anti-Simetirisi
  - Nesneler, verilerini soyutlamaların arkasına saklar ve bu veriler üzerinde çalışan işlevleri ortaya çıkarır. Veri yapısı, verilerini açığa çıkarır ve anlamlı işlevleri yoktur.
  - Prosedürel kod (veri yapılarını kullanan kod), mevcut veri yapılarını değiştirmeden yeni işlevler eklemeyi kolaylaştırır. OO kodu ise mevcut işlevleri değiştirmeden yeni sınıflar eklemeyi kolaylaştırır.
  - Prosedürel kod, tüm fonksiyonların değişmesi gerektiğinden yeni veri yapıları eklemeyi zorlaştırır. OO kodu, tüm sınıfların değişmesi gerektiğinden yeni işlevler eklemeyi zorlaştırır.
- Demeter Kuralı
  - Bir modülün manipüle ettiği nesnelerin iç kısımlarını bilmemesi gerektiğini söyleyen Demeter Yasası adlı iyi bilinen bir buluşsal yöntem vardır. Son bölümde gördüğümüz gibi, nesneler verilerini gizler ve işlemleri açığa çıkarır. Bu, bir nesnenin iç yapısını erişimciler aracılığıyla ifşa etmemesi gerektiği anlamına gelir, çünkü bunu yapmak iç yapısını gizlemek yerine ifşa etmektir.
- Veri Transfer Nesneleri (Data Transfer Objects)
  - Bir veri yapısının en önemli biçimi, genel değişkenleri olan ve hiçbir işlevi olmayan bir sınıftır. Buna bazen veri tranfer nesnesi veya VTN denir. VTN'lar, özellikle veritabanları ile iletişim kurarken veya soketlerden mesajları ayrıştırırken vb. çok kullanışlı yapılardır. Genellikle bir veritabanındaki ham verileri uygulama kodundaki nesnelere dönüştüren bir dizi çeviri aşamasında ilk olurlar.

Bölüm 7 – Hataların Kotarılması

- Hataların kotarılması önemlidir, fakat eğer asıl kodu gölgelemeye başlıyorsa, bu yanlıştır,
- Exception'lar, program içerisinde bir kapsam tanımlarlar ve ilgili yakalama ifadesi sonrasında uygulamanın, tutarlı bir durumda bırakılması önemlidir,
- İş mantığı ile hataların kotarılması arasındaki ayrıma dikkat edilmesi ve vakit ayrılmalıdır,
- Null dönmek her ne kadar başta masum görünse de, hem çağıran hem de bizler için gereksiz yük oluşturmaktadır. Bu sebeple olabildiğinde null dönmemeye çalışın,
- Benzer şekilde, eğer özellikle ilgili API bunu kullanmıyorsa, fonksiyonlara null geçirmekten kaçının,
- Genel olarak hata kodunun, uygulama mantığından başarılı bir şekilde ayrılması, hem kodun idamesini hem de olası hataların önüne geçmemizi sağlayacaktır.

Bölüm 9 – Birim Testler

- TDD (Test Driven Development)'ın üç temel kuralı:
    - Birinci Kural: Geçmeyen bir birim test yazmadan, kod yazamazsınız,
    - İkinci Kural: Başarısız olmak için yeterli olan ve derlemenin başarısız olduğu bir birim testinden daha fazlasını yazmamalısın.
    - Üçüncü Kural: O an hata alan testi geçmek için yeterli olacak kadar ya da daha fazla ürün kodu yazabilirsin.
- Burada önemli olan husus: birim testlerinin ve kodların birlikte geliştirilmesi,
- Birim testleri de en az kodlar kadar temiz tutulmalı ve ikinci plana atılmamalıdır. Onlar için de refaktör ve tasarım hususları göz önünde bulundurulmalıdır,
- Testler olmadan, kodlara yapılan değişikliklerin uygulamanın beklendiği gibi çalışıp/çalışmadığını kontrol etme yeteneğini kaybetmemize sebep olur,
- Otomatik bir şekilde çalışan birim testler, uygulamanın, tasarım ve mimarisini temiz tutmak için anahtar hususlardan biridir,
- Testler için (BUILD-OPERATE-CHECK) örüntüsü izlenebilir. İlk kısımda test verisi hazırlanır, ikinci kısımda operasyon gerçekleştirilir ve son kısımda da operasyon sonucu kontrol edilir,
- Her ne kadar bu pek mümkün olmasa da, her test fonksiyonu için tek bir kontrol gibi, JUnit de bir öğreti vardır. Bu her ne kadar mümkün olmasa da, amaç bu sayıyı olabildiğince minimum tutmaktır,
- Bu konuda bir diğer yaklaşım da, her bir kavram için bir test fonksiyonu yazmaktır,
- Testler F.I.R.S.T. Kuralına uymalıdır. Yani:
    - Fast(hızlı): Testler hızlı çalışmalıdır.
    - Independent(bağımsız): Testler birbirinden bağımsız olmalıdır.
    - Repeatable(tekrar edilebilir): Her ortamda tekrar edilebilir olmalıdır.
    - Self Validating(kendini doğrular): Testlerin boole, testin kaldığını veya geçtiğini söyleyen, sonuçları olmalıdır.
    - Timely(zamanında): Üretim kodundan hemen önce yazılmalıdır.

Bölüm 10 – Sınıflar

- Üst seviye kod organizasyonuna gerekli hassasiyeti göstermezsek, temiz koda sahip olamayız,
- Enkapsülasyonu sağlayacağım diye, her şeyi gizlemek bir çözüm olmayabilir. En azından testler için bir takım esneklik sağlanabilir,
- Sınıflar olabildiğince küçük olmalıdır. Bu bağlamda sınıfın üstlendiği sorumluluklar, bizler için referans olabilir. Ayrıca sınıfa kolay bir şekilde isim bulamıyorsak bu da bize bir sıkıntı olduğunu ifade eder,
- Processor, Manager ya da Super gibi isimler ile birden fazla kabiliyetin yönetilip/yönetilmediğine dikkat etmeliyiz,
- SRP: Single Responsibility Rule, bir sınıfın sadece ama sadece bir sebeple ile değişmesi gerektiğini ifade eder. Bu bağlamda, uygulamalarımız ya da sistemlerimizi büyük birkaç sınıftan oluşturmaktansa, küçük bir çok sınıftan oluşturmayı hedeflemeliyiz,
- Cohesion (Uyuşma, Bağlılık). Sınıflar, az sayıda üye nesnelere sahip olmalıdır ve sınıfın fonksiyonları, bu sınıfın bir ya da birden fazla değişkeni değiştiriyor olmalıdır,
- Bir diğer ifade ile, «Cohesion», bir sınıfın ya da modülün içindeki öğelerin birbirine ait olma derecesini ifade eder. Bir anlamda, bir sınıfın fonksiyonları ve verileri ile o sınıfın hizmet ettiği birleştirici amaç veya kavram arasındaki ilişkinin gücünün bir ölçüsüdür.
- Bir diğer önemli prensip de, Open Closed Principle (OCP)'dir. Yani sınıflar, genişlemeye açık, fakat modifikasyona kapalı olmalıdır, prensibidir. Bu sayede bir çok küçük ve birbirinde bağımsız sınıflarımız olacak ve değişikliklerden etkilenen sınıf sayısı oldukça azalacaktır,
- Bunu sağlamanın bir diğer yöntemi de, «Dependency Inversion Principle (DIP)» ile ifade edilen ve sınıfların, somut sınıflara bağımlı olmaktansa, arayüzlere bağımlı olmalarını ifade eder.

Bölüm 11 – Sistemler

- “Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.” —Ray Ozzie, CTO, Microsoft Corporation
- Sistem seviyesinde, temiz nasıl kalınabilir o irdeleniyor,
- Yapım aşaması kullanım aşamasından oldukça farklı bir süreçtir,
- Yazılım sistemleri, uygulama nesneleri oluşturulduğunda ve bağımlılıklar birbirine "bağlandığında" başlatma sürecini, başlatmadan sonra devralan çalışma zamanı mantığından ayırmalıdır,
- Bu noktada «seperation of concerns» önemli bir kavram olarak karşımıza çıkıyor,
- Dependency Injection bu anlamda kullanılabilecek mekanizmalardan birisi olarak karşımıza çıkıyor,
- Bağımlılık yönetimi bağlamında, bir nesne, bağımlılıkların kendisini başlatma sorumluluğunu almamalıdır. Bunun yerine, bu sorumluluğu başka bir “yetkili” mekanizmaya devretmeli, böylece kontrolü tersine çevirmelidir,
- Java Spring Framework’ü bu anlamda mekanizmalar sunmaktadır,
- Yazılım sistemleri fiziksel sistemlere kıyasla benzersizdir. Hususların (Concerns) uygun şekilde ayrılmasını sağlarsak, mimarileri kademeli olarak büyüyebilir.
- Optimal bir sistem mimarisi, her biri Plain Old Java (veya diğer) Nesneleri (POJO) ile uygulanan modülleştirilmiş ilgi alanlarından oluşur. Farklı alanlar, minimal düzeyde istilacı Unsurlar veya Unsur benzeri araçlarla birlikte entegre edilmiştir. Bu mimari, tıpkı kod gibi test odaklı olabilir,
- Eğer uygulamanızın mantığını POJO’ya uygun bir şekilde geliştirirseniz, yeni kabiliyetleri kolayca ekleyebilirsiniz,
- Bazı kararları olabildiğince geç vermek faydalı olabilir. POJO buna olanak sağlamaktadır,
- Standartlar, fikirleri ve bileşenleri yeniden kullanmayı, ilgili deneyime sahip kişileri işe almayı, iyi fikirleri kapsamayı ve bileşenleri birbirine bağlamayı kolaylaştırır. Bununla birlikte, standart oluşturma süreci bazen endüstrinin beklemesi için çok uzun sürebilir ve bazı standartlar, hizmet etmeyi amaçladıkları benimseyenlerin gerçek ihtiyaçlarıyla temasını kaybeder,
- Etki Alanına Özgü Diller (Domain Specific Languages), uygulamadaki tüm soyutlama düzeylerinin ve tüm etki alanlarının, yüksek düzeyli politikadan düşük düzeyli ayrıntılara kadar POJO'lar olarak ifade edilmesine izin verir,
- Sonuç olarak sistem ya da tekil modüller tasarlarken, unutulmaması gereken en önemli husus, çalışacak en basit yaklaşımı uygulamaktır.

Bölüm 12 – Ortaya Çıkma (Emergence)

- Basit bir tasarıma sahip olmak için takip edilmesi gereken dört kural (Kent Beck):
    - Bütün testleri koştur
    - Tekrar içerme
    - Programcının amacını ifade et
    - Sınıf ve metot sayısını minimize et
- Doğrulanmamış hiçbir yazılım konuşlandırılmamalı. Bu testler ile birlikte düşük bağımlılık ve yüksek kohezyon korunmuş olur,
- Tekrar, bütün iyi tasarlanmış sistemlerin en büyük düşmanıdır,
- Template metodu ile üst seviye tekrar önlenebilir,
- Fonksiyon ve diğer unsurların isimlerinin uyumlu olması, amacı ifade etmek için çok önemlidir, bu nokta fonksiyon ve sınıfların küçük olması, iyi yazılmış birim testler de buna oldukça yardımcı olur.

Bölüm 13 – Eşzamanlılık

- “Objects are abstractions of processing. Threads are abstractions of schedule.” James O. Coplien,
- Birden fazla threadte koşan kod yazmak ve bunu temiz bir şekilde yapmak, tek thread için kod yazmaktan oldukça zordur,
- Eşzamanlılık bir ayrıştırma stratejisidir. Yapılanları, yapıldığından ayırmamıza yardımcı olur. Tek iş parçacıklı uygulamalarda, ne ve ne zaman çok güçlü bir şekilde birleştirilir ki, tüm uygulamanın durumu genellikle yığın geri izlemesine bakılarak belirlenebilir,
- Neyin ne zamandan itibaren ayrıştırılması, bir uygulamanın hem verimini hem de yapılarını önemli ölçüde iyileştirebilir,
- Bu noktada bazı yanlış bilinen hususlara değinmekte fayda var:
    - Eşzamanlılık her zaman performansı artırır.
    - Eşzamanlı programlar yazarken tasarım değişmez. Aslında bu tarz uygulamaların tasarımları, eşzamanlı olmayanlara göre oldukça farklı olabilir,
- Bunun yanında daha makul hususlar şu şekilde sıralanabilir:
    - Eşzamanlılık, hem performans hem de ek kod yazma açısından bir miktar ek yüke neden olur.
    - Eşzamanlılık, hem performans hem de ek kod yazma açısından bir miktar ek yüke neden olur.
    - Doğru eşzamanlılık, basit problemler için bile karmaşıktır.
    - Eşzamanlılık hataları genellikle tekrarlanabilir değildir,
    - Eşzamanlılık genellikle tasarım stratejisinde temel bir değişiklik gerektirir.
- Bu bağlamda sunulan bir takım öneriler şu şekilde sıralanabilir:
    - Eşzamanlılıkla ilgili kodunuzu diğer kodlardan ayrı tutun,
    - Paylaşılabilecek herhangi bir veriye erişimi ciddi şekilde sınırlandırın,
    - Paylaşılan verilerden kaçınmak için veri kopyalamayı düşünebilirsiniz,
    - Thread’ler olabildiğince birbirinden bağımsız olmalıdır,
    - Verileri, muhtemelen farklı işlemcilerde, bağımsız iş parçacıkları tarafından çalıştırılabilecek bağımsız alt kümelere bölmeye çalışın,
    - Temel eşzamanlı algoritmaları (producer/consumer, reader/writer, dining philosophers) öğrenin ve çözümlerini anlayın.
- Thread emniyetli veri yapıları ve bileşenleri kullanmaya özen gösterin,
- Senkronize edilmiş bölümlerinizi mümkün olduğunca küçük tutun.

Bölüm 17 – Code Smells

- Refactoring kitabında, Martin Fowlerin belirttiği «Code Smells»'lere bakacağız,
- Yorumlara ilişkin hususlar:
  - Yorumlar uygunsuz/geçersiz bilgi içermemelidir,
  - Geçerliliğini yitirmiş, eski ve yanlış yorumlar silinmelidir,
  - Gereksiz yorumlar, JavaDoc tarzı yorumlar kaldırılmalıdır,
  - Kötü yazılmış yorumlardan uzak durun,
  - Yoruma alınmış kodlardan kurtulun, bunu gördüğünüz anda silin.
- Çevresel hususlar:
  - Yazılım oluşturma adımları açık olmalıdır ve genelde bir adımla bu gerçekleştirilebiliyor olmalıdır,
  - Testler kolay, hızlı ve açık bir şekilde koşturulabiliyor olmalıdır,
- Fonksiyonlara ilişkin hususlar:
  - Çok argüman almıyor olmalılar,
  - Parametre olarak çıktı olabildiğince sunmamalı, bunun yerine çağrıldığı nesneyi güncelleyebilir,
  - Boolean, kontrol girdilerinden kaçının,
  - Çağrılmayan/kullanılmayan fonksiyonlar silinmelidir.
- İsimlendirme ile ilgili hususlar:
  - İsimlerin kendisi açıklayıcı olmalıdır,
  - İsimler ilgili soyutlama seviyesine uygun verilmelidir,
  - Benzer işlevler için standart bir yaklaşım sergilenmelidir (toString, vs.),
  - Açık olmayan isimler kullanılmamalıdır ,
  - Kullanılacağı kapsama göre isim uzunlukları ayarlanmalıdır (Bir ismin uzunluğu, kapsamın uzunluğu ile ilgili olmalıdır. Küçük kapsamlar için çok kısa değişken adları kullanabilirsiniz, ancak büyük kapsamlar için daha uzun adlar kullanmalısınız.)
  - Var ise yan etkilerden bahsetmelidir

Bölüm 17 – Code Smells - Devam

- Testlere ilişkin hususlar:
  - Yetersiz testler,
  - Kod kapsama araçları kullanın,
  - Aşikar testleri yapmaktan kaçınmayın,
  - Göz ardı edilen testlere dikkat edin,
  - Sınır koşullarını test edin,
  - Hata çıkan kod parçalarını ve ilgili yerleri yoğun bir şekilde test edin,
  - Testler olabildiğince hızlı olmalıdır, gereksiz yorumlar, JavaDoc tarzı yorumlar kaldırılmalıdır,
  - Kötü yazılmış yorumlardan uzak durun,
  - Yoruma alınmış kodlardan kurtulun, bunu gördüğünüz anda silin.
- Genel bir takım hususlar:
  - Bir den fazla dile ilişkin kodu aynı dosyada kullanmayın,
  - İsim ile ifade edilen davranış yerine farklı bir davranış sergilenmemeli,
  - Sınır koşulları/durumları kontrol edilmelidir,
  - Geçersiz kılmadan olabildiğince kaçınılmalıdır,
  - Tekrara olabildiğince izin verilmemelidir,
  - Kodların soyutlama seviyesi ve bulunduğu seviyeye dikkat edilmelidir,
  - Üst seviye sınıflar, türetilen sınıflara bağımlı olmamalıdır,
  - Arayüzler gereğinden fazla bilgi vermemelidir,
  - Kullanılmayan kodlar silinmelidir,
  - Değişken ve fonksiyonlar kullanıldıkları yere yakın tanımlanmalıdırlar,
  - Birbirleri ile çelişen davranışlardan kaçınılmalıdır,
  - Gereksiz bağımlılıklardan kurtulunmalıdır,
  - Fonksiyon veya benzeri yapıların amacı açık olmalıdır,
  - Sorumluluklar net olmalıdır,
  - Hangi fonksiyonların statik yapılacağına dikkat edilmelidir,
  - Açıklayıcı Değişken isimleri kullanılmalıdır,
  - Fonksiyon isimleri, ne diyorlarsa onu yapmalılar ve tek bir şey yapmalıdır,
  - Algoritmalara doğru bir şekilde anlaşılmalıdır,
  - Polimorfizmi if/else ve switch/case'lere tercih edin,
  - Standard gösterim ve kullanımları takip edin. Bir kodlama standardınız olsun,
  - Sihirli sabitleri, isimlendirilmiş sabitler ile değiştirin,
  - Birden fazla koşulu ve sınır koşullarını enkapsüle edin,
  - Negatif koşullardan kaçının,
  - Gizli, geçici bağımlılıklardan kaçının,
  - Konfigürasyon verisini olabildiğince üst seviyelerde tutun ve kodun derinliklerine gömmeyin.

İlginiz İçin Teşekkürler

Muhakkak orijinal kitabı da okuyunuz ;)