Clean Code

Bu Kitap iyi programlama hakkındadır. İyi kodun nasıl yazılacağı ve kötü kodun nasıl iyi koda dönüştürüleceği ile ilgilidir.

Kod, gereksinimlerin ayrıntılarını temsil eder ve ayrıntılar göz ardı edilemez veya soyutlanamaz. Gereksinimlere daha yakın diller oluşturabiliriz. Bu gereksinimleri ayrıştırmamıza ve resmi yapılarda birleştirmemize yardımcı olacak araçlar oluşturabiliriz. Ancak gerekli hassasiyeti asla ortadan kaldırmayacağız.

Her deneyimli programcının kendi temiz kod tanımı vardır, ancak net olan bir şey var, temiz kod kolayca okuyabileceğiniz bir koddur. Temiz kod, halledilen koddur.

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.

İsimler yazılımın her yerindedir. Dosyalar, dizinler, değişkenler, fonksiyonlar vs. Çünkü çok şey yapıyoruz. Bunu iyi yapsak iyi olur.

İsimlerin niyeti ortaya koyduğunu söylemek kolaydır. İyi isimler seçmek zaman alır, ancak gerekenden daha fazlasını kazandırır. Bu yüzden isimlerinize dikkat edin ve daha iyilerini bulduğunuzda değiştirin.

Bir değişkenin, işlevin veya sınıfın adı, tüm büyük soruları yanıtlamalıdır. Size neden var olduğunu, ne işe yaradığını ve nasıl kullanıldığını anlatmalıdır. Bir isim yorum gerektiriyorsa, o zaman isim amacını açıklamaz.

Programcılar, kodun anlamını gizleyen yanlış ipuçları bırakmaktan kaçınmalıdır. Yerleşik anlamı, kastedilen anlamımızdan farklı olan kelimelerden kaçınmalıyız.

Tek harfli adlar ve sayısal sabitler, bir metin gövdesi boyunca kolayca bulunamadıkları için belirli bir soruna sahiptir.

Okuyucular, adlarınızı zaten bildikleri diğer adlara zihinsel olarak çevirmek zorunda kalmamalıdır.

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.

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.

Yöntemler, postPayment, deletePage veya save gibi fiil veya fiil tümcesi adlarına sahip olmalıdır. Erişimciler, mutatörler ve yüklemler, değerlerine göre adlandırılmalı ve javabean standardına göre get, set ve is ile önek eklenmelidir.

Daha kısa isimler, net oldukları sürece genellikle daha uzun olanlardan daha iyidir. Bir ada gereğinden fazla bağlam eklemeyin.

Fonksiyonların ilk kuralı, küçük olmaları gerektiğidir. Fonksiyonların ikinci kuralı, bundan daha küçük olmaları gerektiğidir.

Bildirimler, başlatma vb. bölümlere ayrılmış bir işleviniz varsa, bu, işlevin birden fazla şey yaptığının bariz bir belirtisidir. Tek bir şey yapan işlevler makul bir şekilde bölümlere ayrılamaz.

İşlevlerimizin "tek bir şey" yaptığından emin olmak için, işlevimizdeki tüm ifadelerin aynı soyutlama düzeyinde olduğundan emin olmamız gerekir.

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 ustaca kullanma yeteneğine sahip olsaydık, yorumlara çok fazla ihtiyaç duymazdık belki de hiç.

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, bu pisliği temizlemek için harcayın.

Bazı yorumlar gerekli veya faydalıdır. Ancak gerçekten iyi olan tek yorum, yazmamanın bir yolunu bulduğunuz yorumdur.

Kod formatı önemlidir. Görmezden gelinemeyecek kadar önemli ve dindarca davranılmayacak kadar önemlidir. Kod biçimlendirme, iletişimle ilgilidir ve iletişim, profesyonel geliştiricinin ilk işidir.

Uygulamayı gizlemek, yalnızca değişkenler arasına bir işlev katmanı koyma meselesi değildir. Uygulamayı gizlemek soyutlamalarla ilgilidir! Bir sınıf, değişkenlerini basitçe alıcılar ve ayarlayıcılar aracılığıyla dışarı itmez. Bunun yerine, kullanıcılarının, uygulanmasını bilmek zorunda kalmadan verilerin özünü manipüle etmelerine izin veren soyut arayüzleri ortaya çıkarır.

Olgun programcılar, her şeyin bir nesne olduğu fikrinin bir efsane olduğunu bilirler. Bazen, üzerinde çalışan prosedürleri olan basit veri yapılarını gerçekten istersiniz.

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 imkânsız. Hata işleme önemlidir, ancak mantığı engelliyorsa yanlıştır.

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.

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.

Bir yöntemden null döndürmek istiyorsanız, bunun yerine bir istisna atmayı veya özel bir durum 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.

Yöntemlerden null döndürmek kötüdür, ancak yöntemlere null geçirmek daha kötüdür. Null değerini geçmenizi bekleyen bir API ile çalışmıyorsanız, mümkün olduğunca kodunuzda null değerini geçmekten kaçınmalısınız.

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, 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.

Öğrenme testleri hiçbir şeye mal olmaz. Her halükârda 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.

Birinci Yasa Başarısız bir birim testi yazana kadar üretim kodu yazamazsınız.

İkinci Yasa Başarısız olmak için yeterli olandan daha fazla birim testi yazamazsınız ve derleme yapmamak başarısız olur.

Üçüncü Yasa Şu anda başarısız olan testleri geçmek için yeterli olandan daha fazla üretim kodu yazamazsınız.

Testlerinizi temiz tutmazsanız, kaybedersiniz.

Testlerinizi temiz tutmak için okunabilirlik çok önemlidir.

- Hızlı Test hızlı olmalıdır.
- Bağımsız Test birbirine bağlı olmamalıdır.
- Tekrarlanabilir Test Her ortamda tekrarlanabilir olmalıdır.
- Kendi Kendini Doğrulayan Test bir boole çıktısına sahip olmalıdır. Ya geçerler ya da başarısız olurlar.
- Timely Unit testleri, geçmelerini sağlayan üretim kodundan hemen önce yazılmalıdır. Testleri üretim kodundan sonra yazarsanız, üretim kodunu test etmenin zor olduğunu görebilirsiniz.

Değişkenlerimizi ve yardımcı işlevlerimizi küçük tutmayı seviyoruz, ancak bu konuda fanatik değiliz. Bazen, bir testle erişilebilmesi için bir değişkeni veya yardımcı işlevi korumalı hale getirmemiz gerekir.

Sınıfların bir sorumluluğu olmalı, değişmek için bir nedeni olmalı

SRP, OO tasarımındaki en önemli kavramlardan biridir. Aynı zamanda anlaşılması ve uyulması gereken basit kavramlardan biridir.

Sınıflar az sayıda örnek değişkene sahip olmalıdır. Bir sınıfın metotlarından her biri, bu değişkenlerden birini veya daha fazlasını manipüle etmelidir. Genel olarak, bir yöntem ne kadar çok değişken kullanırsa, o yöntem kendi sınıfına o kadar uyumlu olur. Her değişkenin her bir yöntem tarafından kullanıldığı bir sınıf, maksimum düzeyde uyumludur.

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.

Yazılım Sistemleri, uygulama nesneleri oluşturulduğunda ve bağımlılıklar birlikte "kabloladığında" başlatma sürecini, başlatmadan sonra devralan çalışma zamanı mantığından ayırmalıdır.

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.

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 noktalarının isabet ettiği 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, iş birliği yapan birçok küçük bilgisayar gibi görünür. Bu, sistemin anlaşılmasını kolaylaştırabilir ve endişeleri ayırmak için bazı güçlü yollar sunar.

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.