# The Pragmatic Programmer Özet

## Bölüm 1- Pragmatik Bir Felsefe

1. Bu senin hayatın
   * Bu senin hayatın, sahibi sensin ve alacağın her aksiyon senin sorumluluğunda olacak.
   * Seçim yapma özgürlüğüne sahipsin.
2. Kedi Kaynak Kodumu Yedi
   * Sorumluluk almak
     + Saçma bahaneler yerine seçenekler sunun.
3. Yazılım Entropisi
   * "Kırık pencereleri" (kötü tasarımlar, yanlış kararlar veya zayıf kod) onarmadan bırakmayın.
4. Taş Çorbası ve Haşlanmış Kurbağa
   * Değişim için katalizör olun.
   * Büyük resmi hatırla (Sadece kişisel olarak ne yaptığınızı değil, etrafınızda olup bitenleri sürekli gözden geçirin).
5. Yeterince İyi Yazılım
   * Kullanıcılarını pazarlıklara dahil et
   * Ürettiğiniz sistemin kapsamı ve kalitesi, o sistemin gereksinimlerinin bir parçası olarak belirtilmelidir.
   * Kaliteyi gereksinim sorunu haline getirin
   * Ne zaman duracağınızı bilin (iyi bir durumda olan programı aşırı süslemeye çalışarak bozmaya gerek yok.).
6. Bilgi Portföyünüz
   * Bilgi Portföyünüz
     + Bilgi birikimine yatırım her zaman en iyi faiz öder.
     + Ciddi yatırımcılar, bir alışkanlık olarak düzenli olarak yatırım yapar.
     + Çeşitlendirme, uzun vadeli başarının anahtarıdır.
     + Akıllı yatırımcılar yüksek-riskli ve az-riskli yatırımlar arasında denge gözetirler.
     + Yatırımcılar maksimum kazanç için düşük alıp yüksek satmaya çalışırlar.
     + Portföyler düzenli şekilde gözden geçirilip tekrar düzenlenmelidir.
   * Portföyünüzü Oluşturmak
     + Düzenli olarak yatırım yapın (Finansal yatırım gibi bilgi portföyüne de düzenli yatırım yapmalısın.)
     + Çeşitlendirin. (Ne kadar farklı şey bilirsen, o kadar değerli olursun.)
     + Riski yönetin
     + Düşükten alın yüksekten satın. ( Yeni ortaya çıkan teknolojileri öğrenip kullanmak çok zor bir iştir ama bir o kadar da getirisi olan bir şeydir.).
     + Gözden geçirin ve tekrar düzenleyin. (Geçenlerde kullandığın yeni teknoloji şimdi çok eski olabilir.
   * Bilgi portföyünüze düzenli olarak yatırım yapın.
   * Hedefler
     + Her yıl en az bir yeni dil öğrenin.
     + Her çeyrekte bir teknik kitap okuyun.
     + Teknik olmayan kitapları da okuyun.
     + Ders alın.
     + Yerel kullanıcı gruplarına katılın.
     + Farklı ortamlarla denemeler yapın.
     + Güncel kalın.
     + Aktif olun.
   * Eleştirel Düşünme
     + Okuduklarınızı ve duyduklarınızı eleştirel olarak analiz edin.
7. İletişim kurmak
   * Ne söylemek istediğinizi bilin.
     + Şimdi ne demek istiyorsun. Ne söylemek istediğini planla. Bir taslak yazın.
     + Kendi kendine sor, “Bunlar söylemek istediğim şeyleri karşıya aktarıyor mu?
     + Aktarmıyorsa sadeleştir.
   * Kitlenizi tanıyın
     + Ne istiyorlar?
     + İlgileri nedir?
     + Ne kadar sofistikeler?
     + Ne kadar detay istiyorlar?
     + Bilgiye kimin sahip olmasını İstiyorsunuz?
     + Onları dinlemeye nasıl motive edebilirsiniz?
   * Anınızı seçin: Hedef kitlenizin bilgilerinizi ne zaman duyması gerektiğini anlayın.
   * Bir stil seçin: (Yalnızca gerçekler, geniş ciltli raporlar, basit bir not.)
   * İyi görünmesini sağlayın: Önemli fikirlerinize güzel görünen bir araç ekleyin ve hedef kitlenizin ilgisini çekin.
   * Kitlenizi dahil edin: Geri bildirimlerini alın ve akıllarını karıştırın.
   * Dinleyici olun: Soru sorarak insanları konuşmaya teşvik edin.
   * İnsanlara geri dönün: Daha sonra insanları bilgilendirin.
   * Ne söylediğin kadar nasıl söylediğin de önemlidir.

## Bölüm 2-Pragmatik Bir Yaklaşım

1. İyi Tasarımın Özü
   * İyi tasarım kötü tasarımdan daha kolay değiştirilir.
   * Bir şey, onu kullanan insanlara uyum sağlıyorsa iyi tasarlanmış demektir.
   * Easier to change.(Değiştirmek daha kolay)
   * ETC bir kural değil, değerdir.
   * Kendinize “şimdi yaptığım şey genel sistemi değiştirmeyi kolaylaştırdı mı yoksa zorlaştırdı mı?” diye sorun.
   * Yazdıklarınızı değiştirilebilir hale getirmeye çalışın.
   * Kodu ayrıştırılmış ve uyumlu tutmayı her zaman aklınızda tutun.
2. Kopyalamanın Kötülükleri
   * Kendinizi Tekrar Etmeyin
     + Dayatılan çoğaltma: Geliştiricilerin çoğaltmadan başka seçeneği olmadığını düşündüğü zaman ortaya çıkar.
     + Yanlışlıkla yapılan çoğaltma: Geliştiriciler çoğaltma yaptığının farkında olmaz.
     + Sabırsız çoğaltma: Geliştiriciler daha kolay göründüğü için tembelleşir ve kopyalar.
     + Geliştiriciler arası çoğaltma: Bir ekipteki (veya farklı ekiplerdeki) birden çok kişi bir bilgi parçasını çoğaltır.
   * Yeniden kullanımı kolaylaştırın.
   * Kötü kod çok sayıda yorum gerektirir.
   * Kısa yollar uzun geciktirmeler yaratır.
   * İnsanların kendileri yazmalarındansa, var olan şeyleri tekrar bulup kullanmalarını teşvik edecekleri bir ortam kur.
3. Dikeylik (ortogonallik)
   * Birindeki değişiklikler diğerlerini etkilemiyorsa iki veya daha fazla şey dikeydir.
   * İlgisiz şeyler arasındaki etkileri ortadan kaldırın
   * Biz kendi kendine yeten, bağımsız, tek ve iyi-tanımlanmış amacı olan bileşenler tasarlamak istiyoruz.
   * Faydaları:
     + Verimlilik Kazanın
       - Değişiklikler sınırlandırılır.
       - Yeniden kullanımı teşvik eder.
       - Dikey bileşenleri birleştirdiğin zaman verimde çok muazzam bir kazanç elde edersin. Birbirine dikey olan, M farklı şeyi yapan bir bileşen ile N farklı şeyi yapan bir bileşeni birleştirdiğinde M x N şey ortaya çıkar.
     + Azaltılmış Risk
       - Hastalıklı bölümler veya kod izole edilir.
       - Dikey bir sistem daha iyi test edilecektir.
       - Bir ürüne veya platforma bağlı değil
   * İşyerinizde uygulayabileceğiz dikey ilkeler.
     + Proje Ekipleri: İşlevsellik bölünmüştür
     + Tasarım: Komple bir projeyi bileşenleri aracılığıyla tasarlamak daha kolaydır.
     + Araç Setleri ve Kütüphaneler: Dikeyliği korumak için akıllıca seçimler yapın.
     + Kodlama: Kod eklerken dikeyliği korumak için şunları yapın:
       - Kodunuzu ayrı tutun.
       - Küresel verilerden(Global Değişken) kaçının.
       - Benzer fonksiyonlardan kaçının.
     + Test etme: Dikey sistemlerin test edilmesi daha kolaydır.
4. Tersine Çevrilebilme
   * Değişikliklere hazırlıklı olun.
   * Nihai karar yoktur.
5. İzli Mermiler
   * Yeni projelerde, kullanıcılarınızın gereksinimleri belirsiz olabilir. Yeni algoritmaların, tekniklerin, dillerin veya bilinmeyen kitaplıkların kullanımı gelecek. Ve çevre, siz işiniz bitmeden zamanla değişecektir. Bizi bir gereksinimden son sistemin bir yönüne hızlı, görünür ve tekrarlanabilir bir şekilde götüren bir şey arıyoruz.
   * Hedefi bulmak için izleyici mermileri kullanın.
   * Avantajları
     + Kullanıcılar bir şeyin erken çalıştığını görebilir.
     + Geliştiriciler çalışmak için bir yapı oluşturur.
     + Bir entegrasyon platformunuz vardır.
     + Gösterecek bir şeyiniz vardır.
     + İlerlemeyi daha iyi gözlemlersiniz.
   * İzli mermiler her zaman hedeflerini vurmaz.
   * Bir prototip ile nihai sistemin belirli yönlerini keşfetmeyi hedefliyorsunuz.
   * İzleme kodu, uygulamanın bir bütün olarak nasıl bir arada kaldığını bilmek için kullanılır.
   * Prototip oluşturma, tek kullanımlık kod üretir.
   * İzleme kodu yalındır ancak eksiksizdir ve nihai sistemin iskeletinin bir parçasını oluşturur.
6. Prototipler ve Post-it Notlar
   * Riski değerlendirmek, ortaya çıkarmak ve önemli ölçüde daha düşük maliyetle düzeltme fırsatları sağlamak için yazılım prototipleri oluşturuyoruz.
   * Detaylardan vazgeçemediğiniz bir çevrede kendinizi bulmuşsanız, kendinize gerçekten prototip oluşturmaya ihtiyacınız olup olmadığını sormalısınız.
   * Prototiplenecek Şeyler
     + Mimari.
     + Var olan sistemdeki yeni özellikler.
     + Dış verinin yapısı veya içeriği.
     + Üçüncü parti araçlar veya bileşenler.
     + Performans konuları.
     + Kullanıcı arayüz tasarımı.
   * Öğrenmek için prototipleyin
   * Ayrıntılardan kaçının
     + Doğruluk.
     + Bütünlük.
     + Sağlamlık.
     + Stil.
   * Prototipleme Mimarisi
     + Ana bileşenlerin sorumlulukları iyi tanımlanmış ve uygun mu?
     + Ana bileşenler arasındaki iş birlikleri iyi tanımlanmış mı?
     + Bağlantı en aza indirildi mi?
     + Potansiyel çoğaltma kaynaklarını belirleyebilir misiniz?
     + Arayüz tanımları ve kısıtlamaları kabul edilebilir mi?
     + Her modülün yürütme sırasında ihtiyaç duyduğu verilere erişim yolu var mı?
   * Uygun şekilde kullanılırsa, prototip, çok fazla zaman, para, acı ve emek gibi etkenlerden sizi kurtarabilir.
7. Alan(Domain) Dilleri
   * Dilin sınırları, kişinin dünyasının sınırları kadardır. ( Ludwig Wittgenstein)
   * Problem alanına yakın programlama yapın.
8. Tahmin
   * Sürprizlerden kaçınmak için tahmin edin.
   * Ne kadar doğru, yeterince doğru mu?
     + Yüksek doğruluğa mı ihtiyaçları var yoksa kaba hesap mi arıyorlar?
     + Zaman tahminlerini doğru şekilde ölçeklendirin.

Application, table

Description automatically generated

* + - Tahminler nereden geliyor?
    - Geçmişte benzer bir durumda olan birine sorun.
    - Ne sorulduğunu anlayın.
    - Sistemin bir modelini oluşturun.
    - Modeli bileşenlere ayırın.
    - Her parametreye bir değer verin.
    - Cevapları hesaplayın.
    - Tahmin becerilerinizi takip edin.
  + Proje Takvimlerini Tahmin Etme
    - Bir projenin zaman çizelgesini belirlemenin tek yolu, aynı projede deneyim kazanmaktır. Aşağıdaki adımları tekrarlayarak artımlı geliştirme uygulayın:
      * Gereksinimleri kontrol edin.
      * Riski hesaplayın.
      * Tasarlayın, uygulayın ve entegre edin.
      * Kullanıcılarla beraber doğrulayın.
  + Programınızı kodla yineleyin.
  + Birisi sana bir tahminde bulunmanı istediğinde ne diyeceksin?
    - Size geri döneceğim.
  + Tahminlerinizin kaydını tutmaya başlayın. Her biri için ne kadar isabetli olduğunuzu takip edin. Hatanız %50'den büyükse, tahmininizin nerede yanlış gittiğini bulmaya çalışın.

Bölüm 3-Temel Araçlar

* Araçlar yeteneklerinizi artırır.
* Daha iyi araçlara sahip olursanız, onları daha iyi kullanmayı bilirsiniz ve daha üretken olursunuz.

1. Düz Metnin Gücü
   * Bilgiyi düz metinde tutun.
   * Metnin Gücü
     + Eskimeye karşı sigortalıdır.
     + Mevcut araçlardan yararlanılabilir.
     + Daha kolay test edilebilir.
2. Shell Games (Kabuk Oyunları - Terminal)
   * Terminalin gücünü kullanın
   * Terminale aşinalık kazanın ve eninde sonunda üretkenliğinizin arttığını göreceksiniz.
3. Güçlü Düzenleme
   * Editörde akıcı olun.
   * “AKICI” ne demek?
     + Metin düzenlerken karakter, kelime, satır ve paragraf bazında seçim yapabilmek.
     + Değişikliklerin ardından kodun girintisini ayarlayın.
     + Bir kod parçasını yoruma alın ya da çıkarın.
     + Değişikliği geri alın ya da ileri götürün.
     + Belirli bir satır numarasına gidin.
     + Editörünüzü belli pencerelere bölün.
     + Proje içerisinde arama yapın.
     + Derleme hatalarını gösterin.
     + Projenin testlerini çalıştırın.
     + Seçili satırları sıralayın.
4. Versiyon Kontrol
   * Her zaman versiyon kontrol sistemi kullanın.
5. Hata Ayıklama
   * Hata ayıklamanın sadece problem çözmek olduğu gerçeğini benimseyin ve ona bu şekilde saldırın.
   * Hata ayıklamanın psikolojisi
     + Suçu değil sorunu çözün.
   * Hata ayıklama zihniyeti
     + Panik yapmayın.
   * Nereden başlamalısınız?
     + Başlamadan önce kodun hatasız ve uyarısız bir şekilde derlenebildiğinden emin olun.
     + Herhangi bir sorunu çözmeye çalışırken, ilgili tüm verileri toplamanız gerekir.
   * Hata ayıklama stratejileri
     + Bir hatayı düzeltmeye başlamanın en iyi yolu onu tekrarlanabilir kılmaktır.
     + Kodu düzeltmeden önce başarısız test yap.
     + Hata mesajını okuyun.
     + Varsaymayın, kanıtlayın.

1. Metin Manipülasyonu
   * Bir metin manipülasyon dili öğrenin.
2. Mühendislik Günlükleri
   * Günlük tutmanın 3 temel yararı vardır
     + Hafızanızdan daha güvenilirdir.
     + Eldeki görevle alakalı olmayan fikirleri depolamak için bir yer sağlar.
     + Bir tür lastik ördek görevi görür.( Bir not almaya başlayabilir ve sonra aniden yaptığınız şeyin, notun konusunun tamamen yanlış olduğunu fark edebilirsiniz)

Bölüm 4- Pragmatik Paranoya

* + Mükemmel Yazılım Yazamazsınız

1. Sözleşme ile tasarım
   * Meyer ‘in fonksiyonlardan beklenenleri anlamak için göz önünde bulundurmamız gereken maddeleri:
     + Ön koşullar
     + Son koşullar
     + Sınıf değişkenleri
   * Ne olursa olsun, sözleşmeye uymamanın bir hata olduğu konusunda hemfikir olun.
   * Sözleşme ile tasarlayın.
   * Başlamadan önce neleri kabul edeceğiniz konusunda katı olun ve karşılığında mümkün olduğunca az söz verin.
2. Ölü programlar yalan söylemez.
   * Tüm hatalar size bilgi verir.
   * Kendinizi hatanın olamayacağına ikna edebilir ve görmezden gelmeyi seçebilirsiniz.
   * Bunun yerine pragmatik programcılar kendilerine bir hata varsa, çok, çok kötü bir şey olduğunu söylerler.
   * Hata mesajını okumayı unutmayın.
   * Ölü bir program normalde sakat bir programdan çok daha az hasar verir.
3. İddialı Programlama
   * İmkansızı Önlemek İçin İddiaları Kullanın
   * Performans sorunlarınız olsa bile, yalnızca sizi gerçekten etkileyen kontrolleri kapatın.
4. Kaynaklar nasıl dengelenir?
   * Başladığınız işi bitirin.
   * Kaynakları, tahsis ettiğiniz sıranın tersi sırayla serbest bırakın.
   * Bu şekilde, bir kaynak diğerine referanslar içeriyorsa kaynakları tek bırakmazsınız.
   * Aynı kaynak grubunu kodunuzda farklı yerlere tahsis ederken, onları her zaman aynı sırayla tahsis edin.
   * Bu, kilitlenme olasılığını azaltacaktır.
   * Bir kaynağı kim tahsis ederse, onu serbest bırakmaktan sorumlu olmalıdır.
5. Farlarınızı Geçmeyin
   * Daima küçük adımlar atın.
   * Geri bildirimleri kontrol edin ve ayarlamalar yapın.
   * Asla "çok büyük" bir adım veya görev almayın.
   * Geri bildirim ile tam olarak ne demek istiyoruz?
     + REPL’deki sonuçlar, API’leri ve algoritmaları anlamanız hakkında geri bildirim sağlar.
     + Birim testleri, son kod değişikliğiniz hakkında geri bildirim sağlar.
     + Kullanıcı demosu ve konuşma, özellikler ve kullanılabilirlik hakkında geri bildirim sağlar.
   * Çok büyük bir görev nedir?
     + Aylar sonraki tamamlanma tarihlerini tahmin etme.
     + Gelecekteki bakım veya genişletilebilirlik için bir tasarım planlayın.
     + Kullanıcının gelecekteki ihtiyaçlarını tahmin edin.
     + Gelecekteki teknoloji kullanılabilirliğini tahmin edin.

Bölüm 5- Bükül ya da Kır

* Günümüzün çılgına yakın değişim hızına ayak uydurabilmek için, mümkün olduğu kadar gevşek, esnek kod yazmak için her türlü çabayı göstermemiz gerekiyor.
* Aksi takdirde, kodumuzu hızlı bir şekilde güncelliğini yitirebilir veya düzeltilemeyecek kadar kırılgan hale getirebilir ve nihayetinde geleceğe yönelik çılgınca bir koşuda geride kalabiliriz.

1. Ayrışma
   * Herhangi bir şeyi kendi başına seçmeye çalıştığımızda, onun Evrendeki diğer her şeye bağlı olduğunu görürüz. ( John Muir, My First Summer in the Sierra)
   * Ayrıştırılmış kodu değiştirmek daha kolaydır.
   * (Train wrecks ) Tren kazaları—yöntem çağrıları zincirleri
   * (Globalization) Küreselleşme—durağan şeylerin tehlikeleri
   * (Inheritance) Kalıtım—alt sınıflandırma neden tehlikelidir?
   * Tren kazaları
     + Söyle, Sorma
     + Fonksiyon çağrılarını zincirlemeyin.
     + Bir kısma erişirken birden fazla “.” bulundurmamaya çalışın.
   * Küreselleşmenin Kötülükleri
     + Global verilerden kaçının.
     + Global olmak için yeterince önemliyse, API kullanın.
2. Gerçek Dünyayı Oynamak
   * Events
     + Sonlu Durum Makineleri (Finite State Machines)
     + Gözlemci Modeli (Observer Pattern)
     + Yayınla/Abone Ol (Publish/Subscribe)
     + Reaktif Programlama ve Akışlar
       - Finite State Machines
         * Durum makinesi, temel olarak olayların nasıl ele alınacağının bir göstergesidir.
         * Her durum için önemli olan olayları listeleriz.
         * Bu olayların her biri için sistemin yeni mevcut durumunu tanımlarız.
       - Observer Pattern
         * Gözlemci deseninde bir nesnenin durumu değiştiğinde diğer nesnelere haber verilir ve haberi alan nesneler kendilerini günceller.
         * Ancak gözlemci modelinin bir sorunu vardır: Gözlemcilerin her birinin gözlemlenebilir olana kaydolması gerektiğinden, bağlılığı (coupling) getirmektedir.
       - Publish/Subscribe
         * Bu modelde yayıncılarımız ve abonelerimiz vardır.
         * Bunlar kanallar aracılığıyla birbirine bağlanır.
         * Kanallar ayrı bir kod gövdesinde uygulanır: bazen bir kütüphane, bazen bir süreç ve bazen de dağıtılmış bir altyapı.
         * Her kanalın bir adı vardır.
         * Aboneler, bu adlandırılmış kanallardan bir veya daha fazlasına ilgi gösterir ve yayıncılar bunlara olaylar yazar.
         * Gözlemci modelinden farklı olarak, yayıncı ve abone arasındaki iletişim, kodunuzun dışında gerçekleştirilir ve potansiyel olarak eşzamanlı değildir.
       - Reaktif Programlama ve Akışlar
         * Bir hücre, ikinci bir hücreye başvuran bir formül içeriyorsa, o ikinci hücrenin güncellenmesi birincinin de güncellenmesine neden olur.
         * Değerler, kullandıkları hücredeki değerler değiştikçe tepki verir.
3. Dönüşen Programlama
   * Yaptığınız işi tanımlayamıyorsanız, ne yaptığınızı bilmiyorsunuz demektir.” — W. Edwards Deming,
   * Programlama kodla ilgiliyken programlar verilerle ilgilidir.
4. Kalıtım vergisi
   * Kalıtımı zorunlu kılmayın.
   * Bir daha asla kalıtımı kullanmaya gerek duymayacağınız anlamına gelen üç teknik önerelim:
     + Arayüzler ve protokoller (Interfaces and protocols).
       - Polimorfizmi ifade etmek için arayüzleri & protokolleri tercih edin.
       - Arayüzler ve protokoller bize kalıtım olmadan polimorfizm verir.
     + Delegasyon (Delegation).
     + Karışımlar ve özellikler (Mixins and traits).
       - İşlevselliği paylaşmak için karışımları kullanın.
5. Yapılandırma
   * Harici yapılandırmayı kullanarak uygulamanızı parametrelendirin
   * Kod, uygulama yayınlandıktan sonra değişebilecek değerlere dayandığında, bu değerleri uygulamanın dışında tutun.
   * Uygulamanız farklı ortamlarda ve potansiyel olarak farklı müşteriler için çalışacağı zaman, ortama ve müşteriye özel değerleri uygulamanın dışında tutun.
   * Bu şekilde uygulamanızı parametrelendirirsiniz; kod çalıştığı yerlere uyum sağlar.
   * Yapılandırma verilerinize muhtemelen aşağıdakileri dahil etmek isteyeceksiniz:
     + Dış hizmetler için kimlik bilgileri (veritabanı, üçüncü taraf API’leri vb.).
     + Kayıt seviyeleri ve hedefler.
     + Uygulamanın kullandığı bağlantı noktası, IP adresi, makine ve küme adları.
     + Ortama özel doğrulama parametreleri.
     + Vergi oranları gibi harici olarak ayarlanan parametreler.
     + Siteye özel biçimlendirme ayrıntıları.
     + Lisans anahtarları.

Bölüm 6- Eşzamanlılık

1. Geçici bağlanmanın kırılması
   * Zamanın bizim için önemli olan iki yönü vardır:
     + Eşzamanlılık (aynı anda olan şeyler)
     + Sıralama (nesnelerin zaman içindeki göreli konumları).
   * Birçok projede, tasarımın bir parçası olarak uygulama iş akışlarını modellememiz ve analiz etmemiz gerekiyor.
   * Aynı anda neler olabileceğini ve kesin bir sırayla neler olması gerektiği öğrenmek istenir.
   * Bunu yapmanın bir yolu, aktivite diyagramı gibi bir gösterim kullanarak iş akışını yakalamaktır.
   * Eş zamanlılığı yakalamak için iş akışınızı analiz edin.
2. Paylaşılan durum yanlış durumdur.
   * Rastgele hatalar genellikle eş zamanlılık sorunlarıdır.
   * Paylaşılan bir kaynak ortamında eş zamanlılık zordur ve bunu kendiniz yönetmek daha da zordur.
3. Aktörler ve süreçler
   * Aktör: Kendi yerel durumuna sahip bağımsız bir sanal işlemcidir.
   * Süreç: Bir süreç, genellikle eşzamanlılığı kolaylaştırmak için işletim sistemi tarafından uygulanan daha genel amaçlı bir sanal işlemcidir.
   * Paylaşılan durum olmadan eş zamanlılık için aktörleri kullanın.
4. Kara tahtalar
   * Kaza mıydı yoksa cinayet miydi?
   * Kara tahta yaklaşımının bazı özellikleri şunlardır:
     + Dedektiflerden hiçbirinin başka bir dedektifin varlığından haberdar olması gerekmez - yeni bilgiler için tahtayı izlerler ve bulgularını eklerler.
     + Dedektifler farklı disiplinlerde eğitim görmüş olabilir, farklı eğitim ve uzmanlık seviyelerine sahip olabilir ve hatta aynı bölgede çalışmayabilirler. Vakayı çözme arzusunu paylaşıyorlar ama hepsi bu.
     + Süreç boyunca farklı dedektifler gelip gidebilir ve farklı vardiyalarda çalışabilirler.
     + Tahtaya ne konulabileceği konusunda herhangi bir kısıtlama yoktur. Resimler, cümleler, fiziksel kanıtlar vb. olabilir.
   * Bir iş akışı sistemi kullanarak her olası kombinasyonu ve durumu ele almayı deneyebilirsiniz.
   * Düzenlemeler değiştikçe iş akışı yeniden düzenlenmelidir: insanlar prosedürlerini değiştirmek zorunda kalabilir ve kodun yeniden yazılması gerekebilir.
   * İş akışını kontrol etmek için kara tahtaları kullanın.

Bölüm 7- Kodlama Yaparken

1. İç sesinize kulak verin.
   * Rahatsız edici bir şüphe hissettiğinizde veya bir görevle karşılaştığınızda biraz isteksizlik hissettiğinizde, sizinle konuşmaya çalışan edindiğiniz tecrübeleriniz olacaktır.
   * Tam olarak neyin yanlış olduğunu belirleyemeyebilirsiniz. Ancak, kendinize zaman verin ve şüpheleriniz eninde sonunda muhtemelen daha sağlam bir şeye anlam verecektir.
   * İçgüdülerinizin performansınıza katkıda bulunmasına izin verin.
   * Bir hata yapacağınızdan korkabilirsiniz ve bu makul bir korkudur.
   * Bu projenin bizi aştığını düşünebiliriz.
   * İç sesinizi dinleyin.
   * İlk önce, yaptığınız şeyi bırakın. Beyninizin kendini düzenlemesine izin vermek için kendinize biraz zaman ve alan verin.
   * Kodu düşünmeyi bırakın ve bir süre klavyeden uzakta başka şeyler yapın.
   * Yürüyüşe çıkın, öğle yemeği yiyin, biriyle sohbet edin. Hatta biraz uyuyabilirsiniz bile.
   * Bırakın fikirler beyninizin katmanları arasında kendiliğinden süzülsün.
   * Onu zorlayamazsınız. Sonunda bilinçli seviyenize gelebilir ve “[a ha!](https://www.youtube.com/watch?v=kWYlB5ktwzQ" \t "_blank)” anlarından birini yaşayabilirsiniz.
   * Kodlama yaparken iç sesinizi dinlemeyi öğrenmek, geliştirme yapmak için önemli bir beceridir.
   * Bazen bir tasarım size yanlış geliyor olabilir veya bazı gereksinimler sizi tedirgin ediyordur.
   * Durun ve bu duyguları analiz edin.
   * Destekleyici bir ortamdaysanız, bunları yüksek sesle ifade edin.
   * Onları keşfedin.
2. Tesadüfen programlama
   * Tesadüflere güvenmiyoruz, değil mi?
   * Bazen güvenmek isteyebiliriz.
   * Bu düşünce tarzına aldanmak kolaydır.
   * Neden çalışan bir şeyle uğraşma riskini almalısınız?
   * Pekâlâ, birkaç neden düşünebiliriz:
     + Gerçekten çalışmıyor olabilir- öyleymiş gibi görünebilir.
     + Güvendiğiniz sınır koşulu sadece bir tesadüf olabilir. Farklı durumlarda (farklı ekran çözünürlüğü, daha fazla CPU çekirdeği), farklı davranabilir.
     + Belgelenmemiş davranış, kitaplığın bir sonraki sürümüyle değişebilir.
     + Ek ve gereksiz çağrılar, kodunuzu yavaşlatır.
     + Ek çağrılar, kendilerine ait yeni hatalar getirme riskini artırır.
   * ***Tesadüfen çalışan kodlara güvenmeyin.***
   * **Kod üretmek için daha az zaman harcamak, geliştirme döngüsünün mümkün olduğunca erken aşamalarında hataları yakalayıp düzeltmek ve başlangıçta daha az hata oluşturmak istiyoruz.**
   * Bilinçli bir şekilde nasıl programlanır.
     + Her zaman ne yaptığınızın farkında olun.
     + Kodu daha genç bir programcıya ayrıntılı olarak açıklayabilir misiniz? Değilse, belki de tesadüflere güveniyorsunuzdur.
     + Karanlıkta kodlama yapmayın.
     + Tam olarak anlamadığınız bir uygulama oluşturun veya anlamadığınız bir teknoloji kullanın ve muhtemelen tesadüfler tarafından ısırılacaksınız.
     + Neden çalıştığından emin değilseniz, neden başarısız olduğunu bilemezsiniz.
     + İster kafanızın içinde olsun, ister kokteyl peçetesinin arkasında veya beyaz tahtada olsun, bir plandan devam edin.
     + Yalnızca güvenilir şeylere güvenin.
     + Varsayımlara bağlı kalmayın.
     + Bir şeyin güvenilir olup olmadığını anlayamıyorsanız, en kötüsünü varsayın.
     + Varsayımlarınızı belgeleyin.
     + Sözleşmeye Göre Tasarım, varsayımlarınızı kendi zihninizde netleştirmenize ve bunları başkalarına iletmenize yardımcı olabilir.
     + Sadece kodunuzu test etmeyin, aynı zamanda varsayımlarınızı da test edin.
     + Tahmin etme; aslında deneyin.
     + Varsayımlarınızı test etmek için bir iddia yazın.
     + İddianız doğruysa, kodunuzdaki belgeleri geliştirdiniz.
     + Varsayımınızın yanlış olduğunu keşfederseniz, kendinizi şanslı sayın.
     + Çabalarınıza öncelik verin.
     + Önemli yönlere zaman ayırın; büyük olasılıkla, bunlar zor kısımlardır.
     + Tarihin kölesi olmayın.
     + Mevcut kodun gelecekteki kodu dikte etmesine izin vermeyin.
     + Artık uygun değilse tüm kodlar değiştirilebilir.
     + Tek bir programda bile, daha önce yaptıklarınızın bir sonraki işinizi kısıtlamasına izin vermeyin; yeniden düzenlemeye hazır olun.
3. Algoritma hızı
   * Programınızın 1.000 kayıtla ne kadar sürdüğünü biliyorsunuz, ancak 1.000.000'a nasıl ölçeklenecek?
   * Kodun hangi bölümlerinin optimize edilmesi gerekiyor?
   * Bu soruların genellikle içgüdü, bazı analizler ve Big-O notasyonu adı verilen yaklaşımları yazmanın bir yolu kullanılarak cevaplanabileceği ortaya çıktı.
   * Big-O notasyonu, yaklaşımlarla uğraşmanın matematiksel bir yoludur.
   * Basit döngüler:
     + Basit bir döngü 1'den n’ye kadar çalışıyorsa, algoritma muhtemelen O(n) olacaktır; zaman n ile doğrusal olarak artar.
   * İç içe geçmiş döngüler:
     + Bir döngüyü diğerinin içine yerleştirirseniz, algoritmanız O(m x n) olur, burada m ve n iki döngünün limitidir.
     + Bu tür sıralama algoritmaları O (n2) olma eğilimindedir.
   * İkili parçalama:
     + Algoritmanız döngü etrafında her seferinde dikkate aldığı şeyleri yarıya indirirse, o zaman logaritmik olması muhtemeldir, O(lg(n)).
   * Böl ve fethet:
     + Girdilerini bölen, iki yarı üzerinde bağımsız olarak çalışan ve ardından sonucu birleştiren algoritmalar O(n lg(n)) olabilir.
     + Klasik örnek, verileri iki yarıya bölerek ve her birini yinelemeli olarak sıralayarak çalışan hızlı sıralamadır.
     + Teknik olarak O(n 2) olmasına rağmen, davranışı sıralanmış girdi ile beslendiğinde bozulduğu için, hızlı sıralamanın ortalama çalışma süresi O(n lg(n))’dir.
   * Kombinatorik
     + Algoritmalar şeylerin permütasyonlarına bakmaya başladığında, çalışma süreleri kontrolden çıkabilir.
     + Bunun nedeni, permütasyonların faktöriyelleri içermesidir.
     + Bir kombinatorik algoritmayı beş öğe için zamanlayın: altı öğe için çalıştırmak altı kat, yedi öğe için 42 kat daha uzun sürer.
   * Algoritmalarınızın sırasını tahmin edin.
   * Kodunuzun ne kadar süreceğinden veya ne kadar bellek kullanacağından emin değilseniz, onu çalıştırmayı, giriş kayıt sayısını değiştirmeyi veya çalışma zamanını etkilemesi muhtemel olanı deneyin.
   * Yakında eğrinin şekli hakkında iyi bir fikir edineceksiniz.
   * Girdi boyutu arttıkça yukarı doğru mu kıvrılıyor, düz bir çizgi mi oluyor yoksa düzleşiyor mu?
   * Üç veya dört nokta size bir fikir verecektir.
   * Tahminlerinizi test edin.
   * En hızlısı her zaman iş için en iyisi değildir.
   * Küçük bir girdi seti verildiğinde “Insertion sort”, “Quick sort” kadar iyi performans gösterecek ve hata ayıklama için daha az zaman alacaktır.
   * Seçtiğiniz algoritmanın kurulum maliyeti yüksek ise dikkatli olmalısınız.
4. Yeniden düzenleme
   * Bir program geliştikçe, daha önceki kararları yeniden düşünmek ve kodun bölümlerini yeniden düzenlemek gerekli hale gelecektir.
   * Kod statik bir parça olmadığından sürekli gelişmesi gerekir.
   * Kod tabanının herkes için ücretsiz, toptan yeniden yazılması yerine, kodun değiştirilmesini kolaylaştırmaya yardımcı olan hedefli, hassas bir yaklaşımdır.
   * Dış davranışın değişmediğini garanti etmek için, kodun davranışını doğrulayan iyi, otomatikleştirilmiş birim testine ihtiyacınız var.
   * Peki ne zaman bir kodu yeniden düzenlersiniz?
     + Çoğaltma (Duplication):
       - DRY ilkesinin ihlal edildiğini keşfettiniz.
     + Ortogonal olmayan tasarım:
       - Daha dik hale getirilebilecek bir şey keşfettiniz.
     + Modası geçmiş bilgi
       - İşler değişir, gereksinimler sürüklenir ve sorun hakkındaki bilginiz artar.
       - Kodu zarifleştirin.
     + Kullanım
       - Sistem gerçek insanlar tarafından gerçek koşullar altında kullanılmaya başladıkça, bazı özelliklerin artık daha önce düşünülenden daha önemli olduğunu ve “olması gereken” özelliklerin belki de olmadığını fark ediyorsunuz.
     + Verim
       - Performansı artırmak için fonksiyonu sistemin bir alanından diğerine taşımanız gerekir.
     + Testler Geçti
       - Yeniden düzenlemenin iyi testlerle desteklenen küçük ölçekli bir faaliyet olması gerektiğini söylemiştik.
   * Erken ve sık yeniden düzenleyin
   * Acıyı yönetin: Şimdi acıyorsa, ancak daha sonra daha da acıyacaksa, buna bir son verebilirsiniz.
5. Kodu test et
   * Test etmeyi düşünmek, kodumuzdaki bağlılığı azaltmamızı ve esnekliği artırmamızı sağlar.
   * Kodun yazarı olarak değil, müşterisiymişiz gibi ona objektif olarak bakabiliriz.
   * Bir test, kodunuzun ilk kullanıcısıdır.
   * Test etmenin muhtemelen en büyük avantajını şudur:
     + Test, kodlamanıza rehberlik eden hayati bir geri bildirimdir.
   * Kodlarınızı test edilebilir hale getirmek, bağlılığı da azaltır.
   * Bir kodu test etmeden önce onu anlamalısınız.
   * Hepimiz ne yapmamız gerektiğine dair belirsiz bir anlayışa dayanan bir işe girebiliriz.
   * Ancak bu konuda önce testi düşünürseniz, aklınızın daha berrak olduğunu fark edeceksiniz.
   * Kodlamaya başlamadan önce sınır koşullarını test etmeyi ve bunun nasıl çalışacağını düşünürseniz, mantığınızla işlemi basitleştirecek kalıpları bulabilirsiniz.
   * Test etmeniz gereken hata koşullarını düşünürseniz, işlevinizi buna göre yapılandırabilirsiniz.
   * Testleri önceden düşünmenin tüm faydaları göz önüne alındığında akla gelen önemli bir metodoloji vardır.
   * Buna test odaklı geliştirme veya TDD (Test Driven Development) denir.
   * Basit bir döngüde TDD nedir ona bakalım:
     + Eklemek istediğiniz küçük bir işlevsellik parçasına karar verin.
     + İşlevsellik uygulandığında bunun geçebileceği bir test yazın.
     + Tüm testleri çalıştırın. Yalnızca geçmeyenin az önce yazdığınız test olduğunu doğrulayın.
     + Testi geçmek için gereken en küçük kod miktarını yazın ve testlerin artık temiz çalıştığını doğrulayın.
     + Kodunuzu yeniden düzenleyin. Daha iyi yapabileceğiniz noktalar varsa değiştirmekten çekinmeyin.
   * Birim Testi
   * Yazılım birim testi, bir modülü çalıştıran koddur.
   * Tipik olarak birim testi bir tür yapay ortam oluşturacak ve ardından test edilen modüldeki rutinleri başlatacaktır. Daha sonra döndürülen sonuçları, ya bilinen değerlerle ya da aynı testin önceki çalışmalarından elde edilen sonuçlarla (regresyon testi) karşılaştırarak kontrol edecektir.
   * Dizayn etmek için test edin.
   * Yazdığınız tüm kodlar -siz ve ekibiniz tarafından değilse bile daha sonra nihai kullanıcılar tarafından- test edilecektir.
   * Bu nedenle ürününüzü kapsamlı bir şekilde test etmek önemli bir noktadır.
   * Bu konuda sadece üç seçeneğiniz vardır: En başta test etmek, yolda test etmek veya hiç test etmemek.
   * Yazılımınızı ya siz test edin ya da kullanıcılarınız test edecektir.
   * Test etmek programlamanın bir parçasıdır.
6. Özellik tabanlı test
   * Özellik tabanlı testi, bir kodu farklı girdilerle yüzlerce kez çalıştırdığı için test takımınıza faydalı bir ektir. Bu test frameworkleri, boş listeleri, negatif sayıları ve uzun listeler veya dizeler gibi kodunuzu bozabilecek hemen hemen her girişi kullanır.
   * Değişken tabanlı testler, bir değişkenin her zaman doğru olması gereken yönlerini test etmek için tasarlanmıştır.
   * Test etmek istediğiniz her değer için farklı bir test yazmak yerine, tek bir test içinde bir dizi girdinin programlanıp test edilmesini sağlar.
   * Birim testlerimiz ile kodun birçok önemli yerini test ederken özellik tabanlı testi sayesinde kıyıda köşede bir şüphe bırakmamış oluruz.
7. Orada Güvende Kalın
   * Pragmatik programcılar sağlıklı bir miktarda paranoyaya sahiptir.
   * Hatalarımız ve sınırlamalarımız olduğunu ve dış saldırganların sistemlerimizi tehlikeye atmak için bıraktığımız her açıklığı ele geçireceğini biliyoruz.
   * Her zaman aklınızda bulundurmanız gereken birkaç temel ilke vardır:
     + Saldırı gelebilecek alanı küçültün.
     + En az ayrıcalık ilkesi.
     + Güvenli varsayılanlar.
     + Hassas verileri şifreleyin.
     + Güvenlik güncellemelerini sürdürün.
   * **Basit tutun ve saldırının yüzey alanını azaltın.**
8. Nesneleri Adlandırma
   * Uygulamalar, alt sistemler, modüller, işlevler, değişkenler için adlar oluşturuyoruz.
   * Bu isimler çok önemli, çünkü niyetiniz ve inancınız hakkında çok şey ortaya koyuyorlar.
   * Parçaların kodunuzda oynadıkları role göre adlandırılması gerektiğine inanıyoruz.
   * Bu, bir şey yarattığınız zaman, durup “bunu yaratma motivasyonum nedir?” diye düşünmeniz gerektiği anlamına gelir.
   * Bu güçlü bir sorudur çünkü sizi hemen problem çözme zihniyetinden uzaklaştırır ve daha büyük resme bakmanızı sağlar.
   * Bir değişkenin veya fonksiyonun rolünü düşündüğünüzde, onun hakkında neyin özel olduğunu, neler yapabileceğini ve ne ile etkileşime girdiğini düşünürsünüz.
   * Çoğu zaman, uygun bir isim bulamadığımız için yapmak üzere olduğumuz şeyin hiçbir anlam ifade etmediğini fark ederiz.
   * Bir şeyleri adlandırırken, sürekli olarak ne demek istediğinizi açıklamanın yollarını ararsınız ve bu açıklama eylemi, yazarken kodunuzu daha iyi anlamanıza yardımcı olacaktır.
   * Adlandırmanızı en başta iyi yapın ki yeniden adlandırma yaparken zorlanmayın.