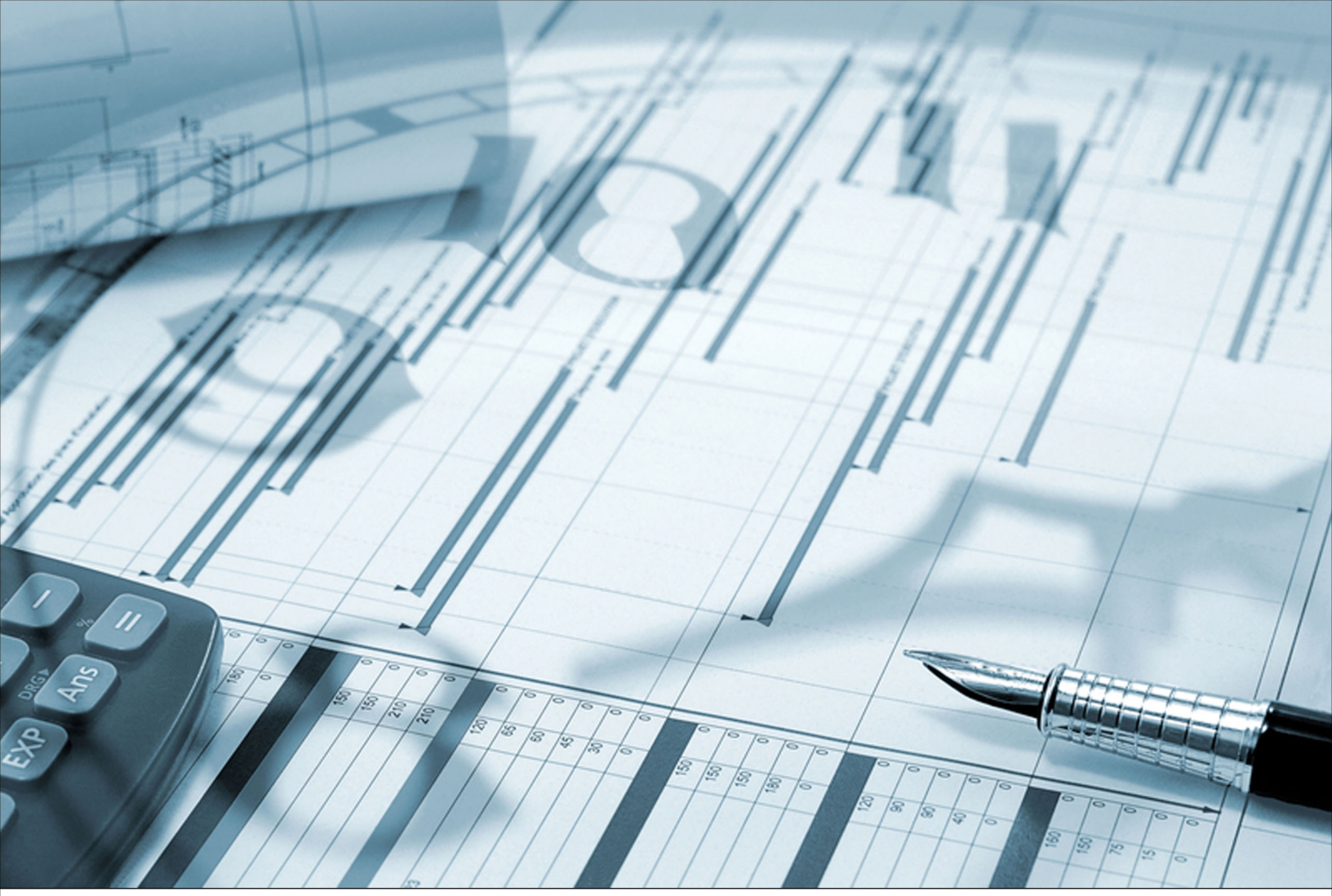

Getting Stuff Done *with Laravel 4*

PHP'nin en popüler yeni Framework'ü ile uygulama tasarımı ve geliştirme üzerine bir yolculuk

Yazarlar Chuck Heintzelman ve Sinan Eldem

Getting Stuff Done with Laravel 4 (TR)

PHP'nin en popüler yeni Framework'ü ile uygulama tasarımı ve geliştirme üzerine bir yolculuk

Chuck Heintzelman ve Sinan Eldem

Bu kitap şu adreste satılmaktadır http://leanpub.com/gsd-laravel-tr

Bu versiyon şu tarihte yayımlandı 2013-12-28

Kitabı tweetleyin!

Chuck Heintzelman ve Sinan Eldem'a kitabını şu adresten Twitter tanıtarak yardımcı olun!

Kitap için önerilen hashtag #LaravelGSD.

Kitap için diğerleri ne demiş merak ediyorsanız bağlantıya tıklayarak hashtagları arayabilirsiniz:

https://twitter.com/search?q =#LaravelGSD

İçindekiler

Teşekkürler

Bu kitabı aldığınız için içtenlikle teşekkür ediyorum. Umarım onu çekici, eğlenceli ve en önemlisi yararlı bulursunuz.

Laravel Öğrenilecek Diğer Yerler

- Laravel websitesi[1]. Bir şey aramak için ilk durağım her zaman budur. Oradaki forumlara göz atın. Bilgilerle dopdoludur.
- NetTuts[2]. Bu sitede bazı güzel Laravel dersleri mevcut.
- Laravel Testing Decoded[3]. Jeffery Way tarafından yazılmış bu kitap Laravel kodunuzu nasıl test edeceğiniz hususunda müthiş bir kaynaktır.
- Code Bright[4]. Dayle Rees tarafından yazılan bu kitap hem eğlenceli hem de bilgilendiricidir.
- From Apprentice to Artisan[5]. Laravel'in geliştirici Taylor Otwell tarafından yazılmış ... fazla söze gerek yok.
- Implementing Laravel[6]. Chris Fidao tarafından yazılan bu kitap Laravel'le proje uygulanmasına odaklanmıştır. Yapıları ve sık kullanılan desenleri anlatır. Büyük bir kitaptır.
- Laravel 4 Cookbook[7]. Christopher Pitt tarafından yazıldı. Bu kitap Laravel 4'te inşa edilen çeşitli projeler içermektedir.

Çevirenin Notu

- Laravel Türkiye websitesi[8]. Türkçe destek alabileceğiniz hızla büyüyen Laravel Türkiye Ailesi.
- Ne kadar şanslısınız ki, elinizdeki bu kitaptan başka, bu listedeki son dört kitabın Türkçelerine de ulaşabilirsiniz. Laravel 4 Türkçe Kitapları[9]

---

[1] http://laravel.com
[2] http://net.tutsplus.com/
[3] http://leanpub.com/laravel-testing-decoded
[4] https://leanpub.com/codebright-tr
[5] https://leanpub.com/laravel-4-tr
[6] https://leanpub.com/implementinglaravel-tr
[7] https://leanpub.com/laravel4cookbook-tr
[8] http://laravel.gen.tr
[9] http://leanpub.com/u/sineld

Revizyon Geçmişi

Güncel sürüm: 1.1

Sürüm	Tarih	Notlar
1.1	28-Kas-2013	Yazım hataları ve genel temizlik.
1.0	2-Kas-2013	Birçok yazım hatası düzeltildi ve Leanpub'da yayımlandı.
0.9	27-Eki-2013	Tamamlandı ve Leanpub'da yayımlandı.
0.8	20-Eki-2013	4 bölüm eklendi, Leanpub'da yayımlandı.
0.7	13-Eki-2013	3 bölüm, iki ek eklendi. Leanpub'da yayımlandı.
0.6	6-Eki-2013	8 bölüm eklenerek Kısım 3 tamamlandı. Leanpub'da yayımlandı.
0.5	29-Eyl-2013	Kısım 3'e 7 bölüm eklendi ve Leanpub'da yayımlandı.
0.4	22-Eyl-2013	Kısım 3'e 7 bölüm eklendi ve Leanpub'da yayımlandı.
0.3	15-Eyl-2013	Kısım 2'ye kadar olan taslaklar temizlendi. Leanpub'da ilk sürümü yayımlama kararı verildi.
0.2	8-Eyl-2013	Kısım 2'nin ilk taslağı bitirildi
0.1	31-Aug-2013	Hoş Geldiniz ve Kısım 1'in ilk taslakları bitirildi
0.0	3-Aug-2013	İlk taslak yazılmaya başlandı

Özel bir teşekkür

Bu sayfalardaki yazım hatalarını ve diğer sorunları bulmak suretiyle bana yardımcı olan isimsiz-olmayan insanların bir listesi:

- Peter Steenbergen
- Jeremy Vaught

Size çok teşekkür ediyorum!

Kapak Görüntüsü

[10] http://www.dreamstime.com/kemaltaner_info
[11] http://www.dreamstime.com/

Örnek Sürüm Hakkında

Bu kitabın "Örnek" sürümünü indirdiğiniz için, her şeyi almış olmuyorsunuz.

Hoş Geldiniz Kısımından kendi bütünlüğündeki sekiz bölümü veriyorum. Artı **SOLID Nesne Tasarımı** bölümünü. Bu size yazdığım şeyin lezzetini tattıracak ve bu kitabın nasıl yararlı olacağını açıklayacaktır.

Ücretli Sürümün İçeriği

Leanpub Sayfasına[12] bakarsanız İçindekiler tablosunun tüm içeriği incelemeniz için sunulmuştur.

Oraya gidin. Kontrol edin. Bu kitabın tam sürümünü satın almanıza karar vermenize yardımcı olacak bir şeyler görebilirsiniz.

[12] https://leanpub.com/gsd-laravel-tr

Hoş Geldiniz

Getting Stuff Done with Laravel 'e hoş geldiniz. Kitabın **Hoş Geldiniz** Kısmı bu kitaptan ne elde edeceğinizi açıklamaktadır.

Genel olarak şu şekilde organize edilmiştir.

Hoş Geldiniz
: Kitabın ilk kısmı bu kitaptan ne elde edeceğinizi açıklamaktadır.

Kısım 1 - Tasarım Felsefeleri ve İlkeleri
: Bu kısım uygulama oluştururken izleyeceğimiz genel tasarım ilkelerinden bahseder.

Kısım 2 - Uygulama Tasarımı
: Bu, gerçek uygulama tasarladığımız kısımdır.

Kısım 3 - Konsol Uygulaması
: Daha sonra, konsolda kullanılabilir uygulama yapıyoruz.

Kısım 4 - Web Uygulaması
: Şimdi yaptığımızı alacağız ve ona bir web postu giydireceğiz. **Mwaa**, **haa**, **ha**.

Ekler
: Tamamlayıcı bilgiler. Composer'ın nasıl yükleneceği gibi.

Bölüm 1 - Bu kitabın amacı

Bu kitap sizi Laravel boyunca bir yolculuğa çıkaracak. Umuyorum ki, hiç gitmediğiniz yerlere gideceksiniz ve hiç görmediğiniz şeyleri göreceksiniz. Bu bir tür seyahatnamedir. Varacağımız kesin bir yer var (oluşturduğumuz uygulama) ve yol boyunca ben şaşırtıcı bazı manzaralar göstereceğim. Sonuna geldiğiniz zaman bana bir not bırakın chuckh@gmail.com[13]. Yolculukla ilgili düşüncenizle çok ilgileniyorum.

Bu kitap yaşanmak içindir. Kullanılmalıdır. Lütfen bölüm bölüm takip ve inşa edin. Her bölüm bir sonrakine götürür. Bir bölüm içindeki kesimler ileriye doğru akar. Kitabın her kısmı önceki üzerine inşa edilir.

Her bölümdeki kesimleri şehirler olarak düşünün. Bu durumda bölümlerin kendileri ülkeler oluyor ve kitabın kısımları kıtalardır ve … *Tamam, yeterince yorucu bir yolculuk benzetmesi.*

Kitap boyunca odak noktamız Laravel 4 kullanarak adım adım bir uygulama oluşturulmasıdır.

Bu kitap tipik bir teknik el kitabı değildir

Ben tasarım ve geliştirme konusunda mümkün olduğunca gerçeği taklit etmeyi deneyeceğim. Bunun anlamı yanlış başlama, tasarım değişiklikleri ve yol boyunca yeniden düzenleme demektir.

Uyarıldınız <sırıtış>.

Bu kitapta neler yoktur

- Laravel'in her yönü. Bu, bütün framework üzerine bir başvuru kitabı değildir.
- Caching, Events veya Logging. Önbellekleme, olaylar ve günceye yazma önemli konulardır, fakat bizim oluşturacağımız uygulama bunları gerektirmez.
- Queues, Authentication, Cookies veya Sessions. Aynı şekilde kuyruklar, kimlik doğrulaması, çerezler ve oturumlar önemli şeylerdir, ancak onlara ihtiyacımız yok.
- Database. Evet, bunu itiraf etmek bana çok acı geliyor. Laravel'in en büyük yönlerinden birisi onun *Fluent Query Builder* ve *Eloquent ORM* sidir. Ne büyük isimler olduğunu anlıyorum. Tatbikatını tam karşılayan isimler. Maalesef bu konuya temas etmiyorum, çünkü … tahmin ettiniz onu … oluşturacağımız uygulama onu gerektirmiyor.

[13] mailto:chuckh@gmail.com

Bu kitapta neler vardır

Çoğunlukla, uygulama oluştururken ne yapıyorum, neden yapıyorum diye gevezelik ederim. Kimi zaman benimle aynı fikirde olursunuz. Kimi zaman bana karşı çıkabilirsiniz. Bazen benim tam bir aptal olduğumu düşünebilirsiniz. Umarım bazen de *"Ha evet. İyi biri"* diye düşünürsünüz. Fakat sonunda, kullanabileceğiniz gerçek bir sistem oluşturmanın esaslarını ve pratik yönlerini elde edeceksiniz.

Bölüm 2 - Siz kimsiniz?

Çoğu kitap yazar hakkında bilgi ile başlar ama daha önemli soru aslında “siz kimsiniz”dir.

Ben aşağıdaki varsayımları yapıyorum:

- Siz bilgisayarlar hakkında çoğu insandan daha fazla şey biliyorsunuz.
- Siz bir programcısınız.
- Siz PHP’de nasıl program yapılacağını biliyorsunuz. Belki pek az. Belki pek çok.
- Laravel’i[14] duydunuz. *(Bu anlaşma-bozucu değil, çünkü size onu anlatacağım.)*
- Siz programlamayı seviyorsunuz veya bilgisayarlara teklif ettiğiniz ilk tutkunuzu tekrar elde etmek istiyorsunuz.
- Adınız Taylor Otwell değil, çünkü eğer öyleyse ben buna layık değilim.

Malzemelerin yararlı olması amacıyla, yeni başlayanlara araçları yaklaşılabilir ve cana yakın, ara programcılara hala ilginç ve yeterince derinlemesine yapmak için elimden geleni yapacağım.

En alt satır

Siz Laravel konusunda daha çok şey öğrenmek istiyorsunuz.

[14] http://laravel.com

Bölüm 3 - Ben kimim?

Burası tipik bir rah-rah, ben harika bir bölüm değilim kısmı. Burada Laravel hakkında tek bir şey öğrenmeyeceksiniz. Yapabileceğiniz en akılcı hareket doğruca bir sonraki bölüme atlamak.

Merhaba. Adım Chuck Heintzelman ve bilgisayar programları yazıyorum.

(Kendimi bir destek grubunun karşısında gibi hissettim. Umarım kimse "Hi Chuck" demedi.)

Cidden. Dokuzuncu sınıftayken bir gün hastalık nedeniyle okula gidemeyip ödünç aldığım bir *BASIC Language Reference* el kitabı ile evde kaldığımdan beri program yazıyorum. O gün kağıt üzerinde Asteroids[15] benzeri bir oyun yazdım, uçan asteroidler dışında diğer gemiler size ölüm kusan uzun beyaz bloklar ateşliyordu.

Saatler süren hata ayıklama ve programımı "kütle deposuna" (bir teyp kaseti) yüklemek/kaydetmek için TRS-80 bekledikten sonra oyun nihayet çalıştı. Bu 33 yıl önceydi. Bilgisayar dinazorlarının, iri vahşi canavarların klimalı odaları doldurduğu günlere geri döndüm. Hayır, delikli kartları *fiilen* hiç kullanmadım ama kullanıldığını görmüştüm.

O zamandan bu yana Fortran, COBOL (evet, biliyorum), Assembly Dili, Basic, C, C++, C#, Java, Pascal, Perl, Javascript ve PHP'de programlar yazdım. Diğer birçok, pek çok dille uğraştım ama insanların fiilen kullandığı programlar yazmadım.

Küçük aile dükkanları yanında Fortune 500 şirketleri için de sistemler oluşturdum. Xenix'te çalışan mail order sistemlerinden PHP'de çalışan web uygulamalarına kadar her şey. İnternet günlerinin öncesinde *(İnternet'in **gerçek** başlangıcından önce değil, 90'lı yılların ortasında başlayan heyecandan hemen önce)* birkaç şirkette ve ondan sonra da birkaç nokta komda çalıştım. Ve yaptığım şeylerin hepsi yapmayı sevdiğim şeylerdi–bilgisayar programları yazmak.

Vay be! Peki, ne kadar büyük olduğum konusunda bu kadar yeter.

İşte benim meselem

Kariyerim boyunca programlama konusunda bir kitap oluşturma gereğini şimdiye kadar hiç duymadım. Bunu yazıyor olmamın tek sebebi Laravel yüzündendir.

[15] http://en.wikipedia.org/wiki/Asteroids_(video_game)

Çevirenin Notu

Bu kitap, diğer çevirilerden farklı olarak, revaçta olan teknolojileri bir araya getirme açısından daha özel bir yere sahip. Chuck'ın anlatım üzerindeki hakimiyeti farklı bir havaya sokuyor okuyucuyu ve sayfalar kendiliğinden akıp gidiyor.

Öncelikle sevgili eşim Bilge ve gözümün ışığı kızım Tuana Şeyma'ya teşekkürler. İyi ki varsınız!

Gerek dokümantasyon, gerekse tüm kitapların çevirisinde tüm süreç boyunca yanımda olan ve çok katkı sağlayan değerli Sergin Arı'ya, kattıklarından dolayı minnettarım. Sen olmadan olmazdı!

Bölüm 4 - Laravel Nedir?

Bu tanıdık geliyorsa elinizi kaldırım

> *Şirketinizin mevcut sistemine bir özellik eklemekle görevlendirildiniz. Ne yazık ki, sistem PHP 4'te yazılmış ve orijinal programcılar her kimse, siz onların çok fazla "Wordpress Çıldırmış" videosu izlediğinden kuşkulandınız.*
>
> *Hiçbir sınıfı olmayan, global değişkenlerle tıka basa dolu ve 50.000 parçalı bir yap boz oyunundan farklı olmayan bir yapıdaki bu kod temelini devraldınız.*
>
> *İlk etapta işinize, yönetim ekibinin kısa görüşlülüğüne ve programlamayla para kazanmak istediğiniz için sahip olduğunu her şeye lanet okudunuz.*
>
> *Sonuçta, programlama eğlenceli olmalı. Değil mi?*

Biz hepimiz oradayız.

Laravel'e katılın.

(Tam tam sesleri geliyor: duh-duh duh-duh duh-da-duh)

Laravel programlamayı tekrar eğlenceli yapan bir PHP frameworküdür.

Haydi dostum ... o sadece bir framework

Laravel yeni bir dil değildir. O sadece bir frameworktür. Eğer abartıyı keser ve işin özüne bakarsanız, Laravel sadece bir PHP Frameworküdür.

Ancak ben, Laravel web sitesindeki slogana katılıyorum:

The PHP Framework for Web Artisans (Web Ustalarının PHP Atölyesi).

Ruby On Rails *sadece bir frameworktür.* Ama, arkasındaki hayranlara bakın.

Laravel sizin PHP spagetti kodunuzu sihirli bir biçimde düzeltmeyecek ama bu şeyleri yapmak için size yeni, hızlı ve zarif bir yol sağlar. *(Not, Getting Stuff Done kavramı bu kitapta yinelenen bir temadır.)*

Kısaca, Laravel PHP programlamayı bir eğlence haline getiren bir mimari sağlar. Mevcut kodunuzu etkileyici, zarif ve ileride sürdürülmesi ve genişletilmesi kolay olacak bir yolla yeniden düzenleyebileceksiniz.

Laravel her derde deva bir ilaç değildir. Eğer elinizdeki mevcut kod berbatsa, **o şimdi nerede** den **o nerede olmalı** ya kadar gelmek bir azap olacaktır. Bizim sektörün doğası bu.

Fakat basit anlatımcılığa imkan veren bir frameworke geçmek istiyorsanız *(hatta bir kelimeyle mi?)* o zaman cevap Laravel'dir.

Bölüm 5 - Laravel nasıl savunulur

İşte problem (veya bir problem) ...

> *Şirketinizin koyduğu sınırlandırmalar altında çalışmak zorundasınız, Yani, mevcut yazılımı desteklemek ve mevcut sistemlerinizle iyi oynayan yeni kod geliştirmek zorundasınız. Orada .NET, bir miktar Java'nın bir karışımı var, fakat mevcut kodun çoğu PHP'dir.*

Son zamanlarda Laravel'i keşfettiniz ve onu sevdiniz ve yeni geliştirmenizde onu kullanmak istiyorsunuz.

Laravel'e geçişi nasıl savunabilirsiniz?

Bir an için dedektif şapkamızı takalım.

Hmmm. Dedektiflerin şüphelileri ve nedenleri ararken parayı takip ettiklerini biliyorum (tabi ki TV'den). Öyleyse parayı takip edelim ...

Müşteriler mal ve hizmet karşılığında işletmelere para verirler. Ürün ne kadar iyiyse ve ne kadar çok müşteri bu ürünü gerçekten talep ederse, işletmeye o kadar çok para öderler.

Yöneticiler işletmeyi geliştirmek ister. Onlar mümkün olan en sıklıkta mümkün olan en çok parayı verecek mümkün olan en çok müşteri isterler.

Yönetimin bakış açısından düşünün ...

- Müşterilerim mutlu olsun istiyorum.
- Yeni müşteriler istiyorum.
- Müşterilerin mutluluğu beklentilerinin karşılanmasına eşittir.
- Programcılarımın gereksinimleri zamanında teslim edebilmesini istiyorum.
- Programlama ekibinin çevik (agile) olmasını istiyorum. *(Anlamı her neyse ... aşağıdaki kutuya bakınız.)*
- Müşterilerimin isteklerini zamanında kolaylaştırmak istiyorum.
- Büyük ürünler teslim eden büyük geliştiriciler istiyorum.

Çevik ne anlama geliyor?

Bir kelimeyi çok sık söyler veya yazarsanız anlamını kaybetmez mi? Tıpkı Şirinler gibi … her şey şirinleniyor, şirinlenebilir, şirinimsidir. Çevik bu kelimelerden birine benziyor. Kelimenin geçmişte geçirdiği evreleri. Her şey Çevik bu, çevik şu. İnsanlar yinelemeli yazılım sürecinden mi bahsediyor, yoksa başka bir şeyden mi? Büyülü bir şey mi? Gerçekten bilmiyorum.

Eğer yukarıdaki liste yönetimin bakış açısıysa, bu durumda Laravel kolaylıkla savunulabilir:

- Müşteriler gereksinimleri ele alınıp karşılandığı zaman mutlu olurlar.
- Müşteriler beklentileri aşıldığı zaman daha da mutlu olurlar.
- Laravel şunları sağlayan bir frameworktür…
  - İşlevselliğin genişletilmesini kolay bir hale getirir.
  - Tasarımda en iyi uygulamalar desenini izler.
  - Çok sayıda programcının verimli bir işbirliğine imkan verir.
  - Programcıları mutlu yapar. *(Yöneticileri unutmayın: mutlu bir programcı üretken bir programcıdır.)*
  - Daha hızlı stuff get done'a izin verir.
  - Test yapmayı her uygulamanın çekirdek bir bileşeni olarak kabul ederek unit testini teşvik eder.

Laravel yöneticiler için programcılarının daha çok, daha hızlı işler yapabilmesini sağlar ve web geliştirmenin özünde bulunan engellerin birçoğunu ortadan kaldırır. İleride bunu açacağım.

Oldukça kolay bir savunma, değil mi?

Bölüm 6 - Programcılar Laravel'i Neden Sever

Sadede gelelim ... Neden siz, bir programcı olarak, bir framework olarak Laravel kullanmak istiyorsunuz?

Bırakın biraz Framework Envy hakkında konuşayım.

> *(Burada bir terapistle görüştüğümü düşünüyorum. Başını bilgece sallıyor, piposundan bir fırt çekiyor ve diyor ki, "zee framework envy hakkında konuşun.")*

Bana, yazılmış PHP projeleri verildi. Bunlar iyice şişmiş, sahip olduğu **"sınıf"** kavramı sadece okulda geçilen bir şey olan bir geliştirici tarafından yazılmış PHP 4 projeleri idi. Ve ben sokak boyunca Ruby geliştiricilerine bakıyor ve onların bina seviyeleri için sessizce doğal afetler–deprem, fırtına, hatta yıldırım–diliyorum.

Bu beni kötü bir insan yapar mı?

Bütün bunlar Ruby'nin tümden parlak ve yeni olduğu bir zamanda oldu. Ruby'yi harika yapan şey dilin kendisi değildi (dilin çok güzel yönleri olmasına karşın). Hayır, Ruby'yi harika yapan şey Ruby on Rails (RoR) idi.

Bütün geliştiriciler Ruby on Rails'e akın ediyordu.

Neden ona akın ediliyordu?

Çünkü, o eğlenceli olan bir geliştirme yolu vaad etmişti. Ve eğlenceli deyince, ben güçlü, anlamlı ve uygulanması kolay anlıyorum. Programlama yapmakta zevkli bir atmosfer oluşturması konusunda RoR'u tekrar şükranla anıyorum. RoR tarafından aşılanan kodlama keyfi, bizim hepimizin programcı olma istememizdeki ilk ivme ile tam aynı duygudur.

PHP dünyasında saplanıp kalmış olmamız ne kadar üzücüydü? Bir Wordpress kurulumunu hackleyebildikleri için oradaki her Tom, Dick ve Henrietta bir "PHP Programcısı" olmuştu.

(Sonraki bölüme bakınız: Wordpress - İyi, Kötü ve Çirkin)

Fakat, hayır, projelerimizin PHP'de olması şartları ile sıkışmıştık. Tüm Ruby geliştiricilerinin olduğu gibi serin, harika çocuklar olamazdık. Onlar en öndeydi. Onlar kendilerine bir isim yapan, sınırları zorlayan birileriydiler.

Laravel'e gelince. Ruby on Rails'in en iyilerini alır ve onu PHP dünyasına getirir. Aniden, bir PHP geliştirici tek tek scriptler yerine controllerler için rotalarla uğraşmaya başlar. DRY (Don't Repeat Yourself [Kendinizi Tekrar Etmeyin]) gibi kavramlar şimdi daha anlamlıdır. Aniden, tek rüyamız

Smarty Şablonları gibi bir yolla, PHP'nin özüne katıştırılmış bir "Blade" şablon motorumuz olur. Biz, kelimenin tam anlamıyla, PHP Nirvana potansiyeline sahibiz.

Laravel'in ne kadar harika olduğunu anlatabildim mi? Umarım öyledir.

Bölüm 7 - Wordpress: İyi, Kötü ve Çirkin

Wordpress bloglama devrimi yaptı. Bloglamayı kitlelere taşıdı. Tabii, blogger ve livejournal gibi diğer platformlar da var fakat Wordpress'in yaptığı şey PHP'de yazılmış büyük, popüler bir sistemi kamu alanına koymaktı.

Wordpress'in çıkışıyla birlikte, herkes yapmak istediklerini gerçekleştiren bloglama platformu yapmak için bu PHP scriptlerini hackleyebiliyordu.

> "Büyük güç büyük sorumluluk getirir." – Ben Amca(Örümcek Adam'dan)

Ne yazık ki, Wordpress'in gücü büyük sorumlulukla karşılanmadı. Scriptler genel tasarım veya kullanılabilirlik düşüncesi olmadan hacklendi. Daha da kötüsü, Wordpress, dilin gerçek programcıların sürdürülebilir sistemler inşa etmelerine imkan vermediği bir zamanda, PHP 4 günlerinde başlamıştı.

Wordpress PHP'nin başına gelmiş en iyi şeydi ama aynı zamanda dilin başına gelen en kötü şey oldu.

Çok az zanaatkarın ellerinde çok fazla başarılı bir vakadır.

Bu durum PHP'ye bir damga yapıştırdı.

Softwarati[16]

> Diller üzerine yorum yapan kendi kendine yok olan programlama aydınları.

Bir göz atmanız için ... Softwarati tarafından söylenmiş sık duyulan bir alıntı:

> "Ah. PHP bir varoş dilidir. Çirkindir, hemen hiç sürdürülebilir değildir, ama çalışır ... çoğu zaman"

Burnu kalkmış bu Softwarati'ye tekme atarak gelen Laravel'e teşekkürler.

[16] Evet, bu tamamen benim ürettiğim bir kelimedir.

Bölüm 8 - Bu Kitapta Kullanılan Düzenler

Bu kitap boyunca çeşitli düzenler kullanılmıştır.

Kodlar 2 boşluk girintilidir

Genelde ben kodları 4 boşluk girintilerim ama bu kitap değişik eBook biçimlerinde olduğu için daha küçük yatay ekranlara sığsın istedim.

```
1  for ($i = 0; $i < 10; $i++)
2  {
3    echo $i, " sayısına kadar sayabiliyorum", "\n";
4  }
```

Bu bir ipucudur

Özellikle yararlı bir bilgi parçasını vurgulamak için kullanılır.

Bu bir uyarıdır

Dikkatli olunması gereken bir şeyler hakkında sizi uyarmak için kullanılır.

Bu bir bilgi bloğudur

Önemli bir bilgi parçasını yinelemek için kullanılır.

Bu, yapılacak bir şeydir

Yapmanız gereken kod veya eylem olduğu zaman önüne hep bu sembol getirilmiştir.

Açma tagı kullanıldığı zaman kapatma tagı ?> kullanılmıştır.

Kod yazarken, ben bir dosyada kapatma ?> tagını her zaman için atlarım. Fakat bu kitabı yazdığım editör öyle yaptığım zaman her şeyi sakat bir görünüme sokuyor. Bu yüzden, bu kitap içinde eğer bir PHP bloğunu PHP tagıyla açarsam, her zaman için kapatma tagını da kullanıyorum. Örneğin:

```
<?php
class SomethingOrOther {
    private $dummy;
}
?>
```

PHP Açma ve Kapatma Tagları

Kod örneklerinde bazen gerekli değilken (örneğin bir dosyanın bir kısmını gösterirken) PHP açma tagı (`<?php`) kullanılmıştır. Bazen gerekli olmayan durumlarda PHP kapatma tagı ('?>') kullanılmıştır.

```
<?php
  function somethingOrOther()
  {
    $this->callSetup();
  }
?>
```

Gerçek PHP Kodunda ben dosyanın sonundaki kapatma tagını *her zaman* atlarım. Tagların gerekip gerekmediği kararını size bırakıyorum. Kod örneklerindeki açma ve kapatma taglarının aynen alınmaması gerektiğinin farkında olun.

Ben hangi işletim sistemi kullanıyorum?

Bu kitabı, kodu v.b. Debian ve Ubuntuya dayalı Linux Mint 14[17] kullanarak yazıyorum. Bu Ubuntu 12.10[18] ile temel olarak aynıdır.

[17] http://www.linuxmint.com/
[18] http://www.ubuntu.com/

Kısım 1 - Tasarım Felsefeleri ve İlkeleri

Kitabın bu kısmında çok fazla kod yok. Üzgünüm, buradaki her şey tasarımla ilgili. Burada, uygulama inşa ederken kullanılan genel tasarım ilkelerini tartışacağım.

"Beni direkt koda götür" diye düşünebilirsiniz. Çoğunlukla ben de aynı fikirdeyim. Sıklıkla en hızlı ve en kolayı hemen koda geçmek ve yaparak öğrenmektir. Eğer şu kavramları biliyorsanız (SOLID Nesne Tasarımı, Sözleşme Olarak Interface'ler, Bağımlılık Enjeksiyonu, Ayrık Tutma ve Kontrolün Tersine Çevrilmesi), o zaman uygulama tasarlamaya başlamak için Kısım 2'ye atlayabilirsiniz.

Bölüm 16 - SOLID Nesne Tasarımı

Nesne yönelimli tasarımda **SOLID** adı verilen bir ilkeler kümesi bulunmaktadır. Şunları ifade eden bir kısaltmadır:

- **[S]**ingle Responsibility Principle (Tek Bir Sorumluluk İlkesi)
- **[O]**pen/Closed Principle (Açık/Kapalı İlkesi)
- **[L]**iskov Substitution Principle (Liskov İkame İlkesi)
- **[I]**nterface Segregation Principle (Interface Ayırma İlkesi)
- **[D]**ependency Inversion Principle (Bağımlılığı Tersine Çevirme İlkesi)

Hep birlikte, en iyi uygulamalar kümesini temsil ediyorlar ve bu ilkelere uyulduğu zaman, geliştirdiğiniz yazılımın geçen zamanla daha sürdürülebilir ve genişletilebilir olmasını sağlarlar.

Bu bölüm her bir ilkeyi daha ayrıntılı açıklamaktadır.

Bana bir SOLID yapar mısınız?

Eğer bir programcı size gelir ve "Bana bir SOLID yapabilir misiniz?" derse, ne sorulduğunu anladığınızdan emin olun.

Tek Bir Sorumluluk İlkesi

SOLID kısaltmasının ilk harfi olan 'S' sorumluluğun tek olmasını ifade ediyor. Bu ilke der ki:

> *"Her sınıfın tek bir sorumluluğu olmalıdır ve bu sorumluluk o sınıf tarafından tamamen yerine getirilmelidir. Onun tüm hizmetleri bu sorumluluk içerisine daraltılmalıdır."*

Bununla ilgili düşünülecek başka bir yol, bir sınıfı değiştirmek için bir ve yalnız bir sebep olmalıdır şeklindedir.

Örneğin amacı bir pazarlama raporunu derlemek ve bunu kullanıcıya email göndermek olan bir sınıfımız varsa, eğer teslim etme yöntemini değiştirirsek ne olur? Kullanıcının raporu metin aracılığıyla veya web üzerinde alması gerekirse veya raporun biçimi değişirse ne olur? Bu örnek için bunların hepsi de sınıfın değiştirilmesi için muhtemel sebep türleridir.

Sınıf Amacını VE Olmadan İfade Etmek

Tek bir sorumluluk ilkesini implemente etmek için basit bir yol sınıfın ne yaptığını VE kelimesi kullanmadan tanımlayabilmektir.

Bunu bir miktar kodla gösterelim ...

```
1  // Sınıf Amacı: Bir pazarlama raporunu derlemek ve bir kullanıcıya e postalamak.
2  class MarketingReport {
3    public function execute($reportName, $userEmail)
4    {
5      $report = $this->compileReport($reportName);
6      $this->emailUser($report, $userEmail);
7    }
8    private function compileReport($reportName) { ... }
9    private function emailUser($report, $email) { ... }
10 }
```

Bu örnekte, sınıfı iki sınıfa bölebilir ve hangi rapor ve kime/nasıl gönderileceği kararını besin zincirinde daha yukarıda olana itebiliriz.

```
1  // Sınıf Amacı: Bir pazarlama raporunu derlemek.
2  class MarketingReporter {
3    public function compileReport($reportName) { ... }
4  }
5
6  interface ReportNotifierInterface {
7    public function sendReport($content, $destination);
8  }
9
10 // Sınıf Amacı: Bir kullanıcıya bir raporu Email göndermek
11 class ReportEmailer implements ReportNotifierInterface {
12   public function sendReport($content, $email) { ... }
13 }
```

Orada bir interface'i nasıl gizlice yerleştirdiğimi fark ettiniz mi? Evet, Ben onlar için interface'lerden daha güvenilmez bir adam olabilirim.

Daha Sağlam Sınıflar

Tek Bir Sorumluluk İlkesini izlemekle sonunda sağlam sınıflara ulaşacaksınız. Tek bir konuya odaklanacağınız için, bu sınıf içinde yapılan herhangi bir kod değişikliğinin bu sınıfın dışındaki işlevselliği bozma ihtimali daha az olacaktır.

Açık/Kapalı İlkesi

SOLID kısaltmasının ikinci harfi bu ilkeyi (Open/Closed) temsil ediyor. Bu ilke der ki:

> *"Yazılım antiteleri (sınıflar, modüller, fonksiyonlar v.b.) genişletmeler için açık ama değişiklikler için kapalı olmalıdır."*

Diğer bir deyişle, bir sınıfı kodladıktan sonra bu kodu hataları düzeltmek dışında asla değiştirmemelisiniz. Eğer ek işlevsellik gerekirse, bu durumda sınıf genişletilebilir.

Bu ilke sizi "bu nasıl değiştirilebilir?" diye düşünmeye zorlar. Deneyim burada en iyi öğretmendir. Siz önceki durumlarda sıklıkla kullandığınız desenleri kurarak yazılım tasarlamayı öğrenirsiniz.

Çok Katı Kurallı Olmayın

Ben bu ilkeye sıkı yapışmanın aşırı tasarlanmış yazılımla sonuçlandığını buldum. Bu, programcılar bir sınıfın kullanılabileceği her potansiyel yolu düşünmeye çalıştıklarında meydana geliyor. Bu düşüncenin bir miktar olması iyi bir şeydir ama çok fazla olursa, gerçekte karmaşık olması gerekmediği halde daha karmaşık sınıf veya sınıf gruplarıyla sonuçlanır.

Bu ilkenin fanatik hayranlarını rahatsız ettimse özür dilerim, ama bakın ben sadece ne görürsem onu söylüyorum. O bir ilkedir, bir kanun değil.

Bununla birlikte, yaygın bir örnek verebiliriz. Diyelim ki, bir hesap faturası kesimi üzerinde, özellikle iadeleri işleyen kısmıyla çalışıyorsunuz.

```
1  class AccountRefundProcessor {
2    protected $repository;
3
4    public function __construct(AccountRepositoryInterface $repo)
5    {
6      $this->repository = $repo;
7    }
8    public function process()
9    {
10     foreach ($this->repository->getAllAccounts() as $account)
11     {
12       if ($account->isRefundDue())
13       {
14         $this->processSingleRefund($account);
15       }
16     }
```

```
17    }
18  }
```

Tamam, yukarıdaki koda bir bakalım. Bağımlılık Enjeksiyonu kullanıyoruz ve böylece hesapların deposunu kendi repository sınıfından ayırıyoruz. Güzel.

Ne yazık ki, bir gün işe vardığınızda patronunuzu üzgün buldunuz. Anlaşılan, yönetim büyük iadeler nedeniyle hesapların elle gözden geçirilmesine karar vermiş. Uggh.

Peki, yukarıdaki kodda yanlış olan ne ki? Onu değiştirmek zorundasınız, çünkü o değişikliklere kapalı değildir. Onu bir alt sınıfla genişletebilirdiniz ama bu durumda `process()` kodunun çoğunu tekrar etmek zorunda olacaksınız.

Yani, sonuç olarak iade işlemcisini yeniden düzenleyeceksiniz.

```
1   interface AccountRefundValidatorInterface {
2     public function isValid(Account $account);
3   }
4   class AccountRefundDueValidator extends AccountRefundValidatorInterface {
5     public function isValid(Account $account)
6     {
7       return ($account->balance < 0) ? true : false;
8     }
9   }
10  class AccountRefundReviewedValidator extends AccountRefundValidatorInterface {
11    public function isValid(Account $account)
12    {
13      if ($account->balance < 1000)
14      {
15        return $account->hasBeenReviewed;
16      }
17      return true;
18    }
19  }
20  class AccountRefundProcessor {
21    protected $repository;
22    protected $validators;
23
24    public function __construct(AccountRepositoryInterface $repo,
25      array $validators)
26    {
27      $this->repository = $repo;
28      $this->validators = $validators;
29    }
```

```
30    public function process()
31    {
32      foreach ($this->repository->getAllAccounts() as $account)
33      {
34        $refundIsValid = true;
35        foreach ($this->validators as $validator)
36        {
37          $refundIsValid = ($refundIsValid and $validator->isValid($account));
38        }
39        if ($refundIsValid)
40        {
41          $this->processSingleRefund($account);
42        }
43      }
44    }
45  }
```

Şimdi `AccountRefundProcess` sınıfı oluşturulma sırasında bir validatorler dizisi alıyor. İş kurallarınızın değiştiği bir sonraki sefer yeni bir validator çekebilecek ve sınıf konstrüksiyonuna onu ekleyebileceksiniz ve siz altınsınız.

Liskov İkame İlkesi

SOLID'deki "L", Liskov İkame İlkesini ifade eder. Bu ilke der ki:

> *"Bir bilgisayar programında eğer S, T'nin bir alt tipi ise, bu durumda T tipindeki nesneler yerine, programın arzu edilen özelliklerinden herhangi birini (doğruluğu, yaptığı işi v.b.) bozmaksızın, S tipindeki nesneler konabilir (yani, T tipindeki nesneler, S tipindeki nesnelerle ikame edilebilir)."*

Huh? Korkarım SOLID tasarımın bu ilkesi diğerlerinden daha fazla kafa karışıklığına yol açacak.

İkame Edilebilirlik hakkında hepsi bu kadardır fakat bu ikame edilebilirliğin ingilizcesi Substitutability'nin 'S'si kullanılırsa, o zaman kısaltma SOLID yerine SOSID olurdu. Konuşmaları hayal edebiliyor musunuz?

Programcı 1: "Biz sınıf tasarlarken S.O.S.I.D. ilkelerine uyuyoruz."

Programcı 2: "Sosis?"

Programcı 1: "Hayır, SOSID."

Programcı 2: "Sosssu?"

Programcı 1: Telefonla Michael Feathers'i arar... "Hey, daha iyi bir kısaltma gerekiyor."

Basitçe ifade edilirse, Liskov İkame ilkesi programınızın, bir sınıfı kullandığı her yerde o sınıfın alt tiplerini kullanabilmesi anlamına gelir.

Başka bir deyişle, eğer bir Dikdörtgen sınıfınız var ve sizin programınız bir Dikdörtgen sınıfı kullanıyor ve Dikdörtgenden türetilmiş bir Kare sınıfınız varsa. Bu durumda bu program bir Dikdörtgen kullandığı herhangi bir yerde Kare sınıfını kullanabilmelidir.

Hala anlaşılmaz mı? Benim başlangıçta kafam karışmıştı, çünkü "Yaa? Bu oldukça açık" demek istiyordum ama bir şeyleri anlamamış olmamdan korkuyordum. Yani, SOLID tasarımın önemli bir ilkesi niye bu kadar açık olsundu ki?

Bu ilkenin alt tiplerde ön koşulların güçlendirilemeyeceğini ve son koşulların zayıflatılamayacağını söyleyen bazı belirsiz yönleri olduğu ortaya çıkıyor. Ya da bunun tersi mi? Ve üst tipin fırlattığı istisnaların aynısı (veya ondan türetilmiş) olmayan istisnalar atan alt tipleriniz olamaz.

Vay canına! Başka? Liskov ilkesinin burada girmeyeceğim diğer bazı ayrıntıları vardır ve korkarım onunla gerçekten çok fazla zaman harcadım.

Liskov prensibi önemli değil.

Ne?

Evet! Öyle söyledim. (Teknik açıdan bu ilke önemlidir, bu nedenle size yalan söylüyorum ama kendimi bu yalana inandırmaya çalışıyorum.)

PHP'de, eğer aşağıdaki üç kılavuzu izlerseniz, zamanın % 99'unda Liskova'a uymuş olursunuz.

1 - Interfaceler Kullanın

Interfaceler kullanın. Mantıklı olan her yerde onları kullanın. Interface eklemek çok fazla iş yüküne yol açmaz ve sonuç Liskov ikame ilkesine uyduğunuzu neredeyse garantiye alan "saf" bir soyutlamaya sahip olmanızdır.

2 - Implementasyon ayrıntılarını interface'lerin dışında tutun

Interfaceleri kullandığınız zaman, interfaceleri herhangi bir implementasyon ayrıntısına maruz bırakmayın. Depoda boş kalan bir ayrıntıyı sağlayan bir `UserAccountInterface`iniz neden olsun ki?

3 - Tip denetimi kodunuzu gözetleyin.

Belirli tipler üzerinde farklı işler yapan kod yazdığınız zaman, Liskov ikame ilkesini bozabilirsiniz (ve büyük ihtimalle diğer birkaç SOLID ilkesini).

Aşağıdaki kodu ele alın:

```
class PluginManager {
  protected $plugins; // plugins dizisi

  public function add(PluginInterface $plugin)
  {
    if ($plugin instanceof SessionHandler)
    {
      // Sadece kullanıcı giriş yapmışsa ekle.
      if ( ! Auth::check())
      {
        return;
      }
    }
    $this->plugins[] = $plugin;
  }
}
```

Bu kod parçasında yanlış olan nedir? Bu, Liskov ilkesini ihlal etmektedir çünkü yapılan iş nesne tipine bağlı olarak değişmektedir. Bu yeterli bir kod parçasıdır ve küçük bir uygulamada, getting stuff done ruhuyla, onu gösterdiğim için mutluyum.

Fakat... bir `add()` metodu ne eklendiği hakkında neden bilgi alsın ki. Eğer düşünürseniz, `add()` bir parça kontrol ucubesi oluyor. Derin bir nefes almanız ve kontrolü bırakmanız gerekir. *(Hmmm, Inversion of Control)*

Interface Ayrımı İlkesi

SOLID'deki 'I', Interface Ayrımı İlkesini ifade eder. Interface Ayrımı der ki:

> *"Hiçbir istemci kullanmadığı metodlara bağlı olmaya zorlanamaz."*

Düz ifade edersek şişkin interface'leriniz olmasın demektir. Eğer interface'lerinizde "Tek Bir Sorumluluk İlkesine" uyuyorsanız, o zaman bu genellikle bir sorun değildir.

Burada tasarlayacağımız uygulama küçük bir uygulama olacaktır ve bu ilke büyük ihtimalle çok fazla uygulanmayacaktır. Uygulamanız daha büyük bir hale geldiği zaman Interface Ayrımı daha önemli hale gelir.

Bu ilke sıklıkla sürücüler tarafından ihlal edilir. Diyelim ki bir cache sürücünüz var. En basitleştirilmiş haliyle, bir cache gerçekte bir veri değeri için geçici depodan başka bir şey değildir. Bu nedenle bir `save()` veya `put()` metodu ve bunlara karşılık gelen bir `load()` veya `get()` metodu gerekli olacaktır. Fakat çoğu keresinde olan, ekstra davullar ve zurnalar olmasını isteyen ve şuna benzer bir interface tasarlayacak bir tasarımcıdır:

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
  public function getLastAccess($key);
  public function getNumHits($key);
  public function callBobForSomeCash();
}
```

Varsayalım ki, bu cache'yi implemente edeceğiz ve depomuz ona imkan vermediği için `getLastAccess()` metodunu implemente etmemizin imkansız olduğunu fark ettik. Benzer şekilde, `getNumHits()` de problemlidir. Ve biz `callBobForSomeCash()` hakkında hiçbir ipucuna sahip değiliz. *Kim bu Bob denen adam? Ve çağıran birine nakit verir mi?* Bu interface'i implemente ederken sadece istisna fırlatmaya karar veririz.

```
class StupidCache implements CacheInterface {
  public function put($key, $value, $expires) { ... }
  public function get($key) { ... }
  public function clear($key) { ... }
  public function clearAll() { ... }
  public function getLastAccess($key)
  {
    throw new BadMethodCallException('implemente edilmedi');
  }
  public function getNumHits($key)
  {
    throw new BadMethodCallException('implemente edilmedi');
  }
  public function callBobForSomeCache()
  {
    throw new BadMethodCallException('implemente edilmedi');
  }
}
```

Ugh. Çirkin değil mi?

İşte Interface Ayrımı İlkesi. Bunun yerine şöyle daha küçük interfaceler oluşturmalısınız:

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
}
interface CacheTrackableInterface {
  public function getLastAccess($key);
  public function getNumHits($key);
}
interface CacheFromBobInterface {
  public function callBobForSomeCash();
}
```

Mantıklı, değil mi?

Bağımlılığı Tersine Çevirme İlkesi

SOLID'deki son 'D' Dependency Inversion Principle (Bağımlılığı Tersine Çevirme İlkesi) demektir. Bu, iki şey söyler:

> *"A. Yüksek düzeyli modüller düşük düzeyli modüllere dayandırılamaz. Her ikisi de soyutlamalara dayanmalıdır."*
>
> *"B. Soyutlamalar ayrıntılara dayandırılamaz. Ayrıntılar soyutlamalara bağımlı olmalıdır."*

Harika, peki ne anlama geliyor?

Yüksek Düzeyli
: Yüksek düzeyli kod genellikle daha karmaşıktır ve düşük düzeyli kodun işlevselliğine dayanır.

Düşük Düzeyli
: Düşük düzeyli kod dosya sistemlerine erişme veya bir veritabanını yönetme gibi temel, odaklanmış operasyonlar yapar.

Düşük düzey ile yüksek düzey arasında bir yelpaze vardır. Örneğin, ben oturum yönetimini düşük düzeyli kod olarak kabul ederim. Ama oturum yönetimi oturum depolaması için başka düşük düzeyli koda dayanır.

Yüksek düzey ve düşük düzeyi bir diğeriyle ilişkisi bağlamında düşünmek yararlıdır. Oturum yönetimi bir kullanıcının giriş yapması gibi uygulamaya özgü işlevsellikten kesinlikle daha düşüktür, ancak session yönetimi veritabanı erişim katmanı için daha yüksek düzeydir.

Şöyle diyen insanlar duydum ... "Oh, Bağımlılık inversiyonu, Inversion of Controlu implemente ettiğinizde olan şeydir." veya "Hayır. Bağımlılık enjeksiyonu neyse Bağımlılık inversiyonu da odur." Her iki cevap da kısmen doğrudur, çünkü her iki cevap Bağımlılık İnversiyonunu implemente edebilir.

Bağımlılığı Tersine Çevirme İlkesi sınıf bağımlılığının ayrık tutulması tekniklerinden biri olarak daha iyi ifade edilir. Ayrık tutmakla, hem yüksek düzeyli mantık hem de düşük düzeyli nesne birbirine doğrudan dayanmazlar, bunun yerine soyutlamalara dayanırlar.

PHP dünyasında... Bağımlılığı Tersine Çevirme en iyi ne şekilde elde edilir? Tahmin ettiğiniz gibi: interfaceler.

Tüm bu tasarım ilkelerinin birlikte nasıl çalıştığı ilginç değil mi? Ve bu ilkelerin nasıl PHP'nin en kullanılmayan yapısıyla (interface) oyuna katıldıkları?

Bağımlılığı Tersine Çevirmenin Basit Kuralı

Yüksek düzeyli kod kullanan nesneleriniz **her zaman** interfaceler yoluyla olsun. Ve düşük düzeyli kod interfaceleri implemente ediyorsa, bu ilkeyi uyguluyorsunuz demektir.

Bir Örnek

Kullanıcı kimlik doğrulaması iyi bir örnektir. En yüksek düzeyde olan, bir kullanıcının kimliğini doğrulayan koddur.

```
1  <?php
2  interface UserInterface {
3    public function getPassword();
4  }
5
6  interface UserAuthRepositoryInterface {
7    /**
8     * $username'den UserInterface döndür
9     */
10   public function fetchByUsername($username);
11 }
12
13 class UserAuth {
14   protected $repository;
15
16   /**
```

```
17     * Repository bağımlılığı enjekte ediyor
18     */
19    public function __construct(UserAuthRepositoryInterface $repo)
20    {
21      $this->repository = $repo;
22    }
23
24    /**
25     * Eğer $username ve $password geçerli ise true döndürür
26     */
27    public function isValid($username, $password)
28    {
29      $user = $this->repository->fetchByUsername($username);
30      if ($user and $user->getPassword() == $password)
31      {
32        return true;
33      }
34      return false;
35    }
36  }
37  ?>
```

Yani, `UserAuthRepositoryInterface` ve `UserInterface` soyutlamalarına dayanan yüksek düzeyli `UserAuth` sınıfımız var. Şimdi, bu iki soyutlamanın implemente edilmesi oldukça basittir.

```
1   <?php
2   class User extends Eloquent implements UserInterface {
3     public function getPassword()
4     {
5       return $this->password;
6     }
7   }
8   class EloquentUserRepository implements UserAuthRepositoryInterface {
9     public function fetchByUsername($username)
10    {
11      return User::where('username', '=', $username)->first();
12    }
13  }
14  ?>
```

Limon sıkmak kadar kolay.

Şimdi, `UserAuth` sınıfını kullanmak için onu `EloquentUserRepository` ile inşa edebiliriz veya onu otomatik olarak enjekte etmesi için interface'i `EloquentUserRepository` sınıfına bağlayabiliriz.

Bu bölümde kullanılan kimlik doğrulama örneğini Laravel'in `Auth` facade'ı ile karıştırmayın. Bunlar sadece ilkeyi gösteren örneklerdir. Laravel'in implementasyonu, bunlara benzemekle birlikte, çok daha iyidir.

Örneği kontrol ettiğiniz için teşekkürler

Umarım ondan zevk aldınız.

Aslında, Getting Stuff Done with Laravel[19] üzerine tıklamaktan ve geri kalan bölümlerine yatırım yapmaktan hoşlanacağınızı umuyorum.

[19] https://leanpub.com/gsd-laravel-tr