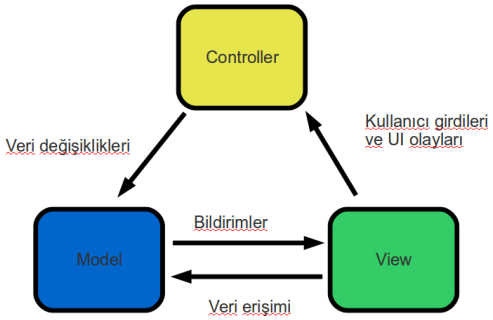
MVC, MVP ve Mediator ile TDD Tecrübeleri

İçindekiler

MVC Nedir?

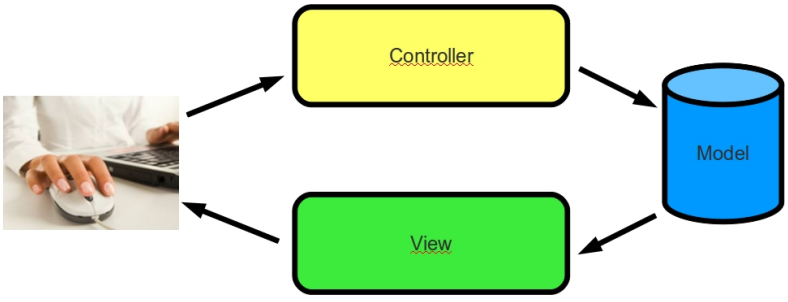
Model View Controller 70'li yılların sonunda Norveçli bilim adamı Tyrgve Reenskaug'un Amerika'daki Xerox lablarını ziyareti sırasında ortaya konulmuş mimarisel bir örüntüdür. Kısaca MVC olarak adlandırılır.



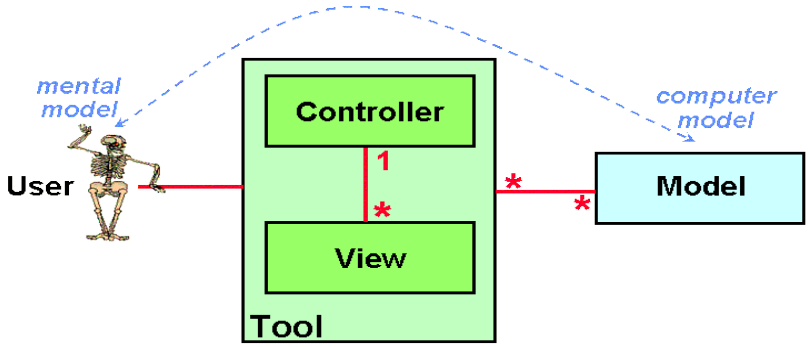
Model, view tarafından görüntülenen veriyi ifade eder. Örneğin, bir checkbox bileşenin on/off state bilgisi, yada bir textfield bileşeninin metin verisi gibi. View ihtiyaç duyduğu veriye model üzerinden erişir ve bu veriyi kullanarak GUI render işlemini gerçekleştirir. Controller ise kullanıcı inputundan (mouse hareketler, click, keyboard input vb) eventler ile model üzerinde değişikliğe gidilmesini sağlar. Model'deki değişiklik de notifikasyonlar vasıtası ile view tarafından algılanarak ekrana yansıtılır.

MVC'nin Problemi Ne?

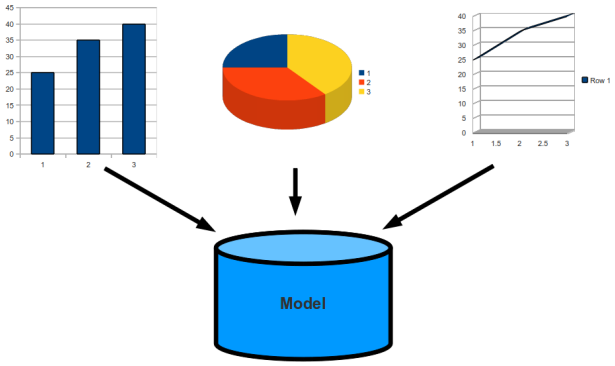
Güncel pek çok dokümanda MVC'nin amacı olarak "iş mantığının GUI kodundan ayrılması" olarak anlatılır. Bu sayede view katmanında herhangi bir değişiklik yapmak istersek, bunu iş mantığında herhangi bir probleme veya değişikliğe yol açmadan kolaylıkla yapabileceğimiz vurgulanır.



Oysa MVC'nin mucidi Reenskaug, MVC'yi anlattığı makalesinde asıl amacın aşağıdaki şekilde de görüldüğü üzere kullanıcıların zihinlerindeki mental model ile bilgisayar sistemlerindeki sayısal model arasındaki boşluğu dolduran genel bir çözüm oluşturmak olduğunu vurgular. Bu çözüm ile domain verisi, başka bir deyişle model doğrudan kullanıcı tarafından erişilebilir, incelenebilir ve güncellenebilir hale gelecektir. (http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html)



Uygulamayı modüler bir yapıya büründürmek ve farklı görevleri farklı katmanlara ayrıştırmak MVC için ilk hedef olmamıştır. Model, controller ve view bölümleri çözüm içerisinde vardır, ancak bunlar yukarıda bahsettiğimiz asıl amaca yönelik olarak şekillenen kısımlardır. Orijinal MVC makalesinde "Seperation of Concern" bir amaç değil sonuçtur.

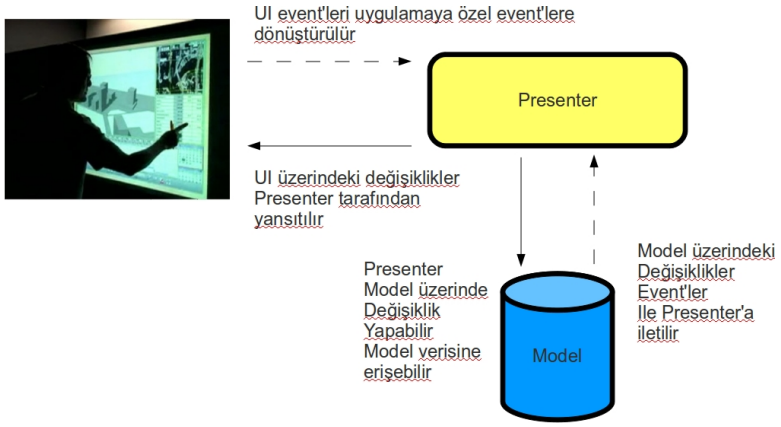


Uygulamanın modüler biçimde geliştirilebilmesi ve katmanların diğer katmanlardan bağımsız biçimde görevlerini yerine getirebilmesi amacı ile MVC örüntüsü üzerinde bir uyarlamaya gidilmesi söz konusudur. Bunun temel nedenlerinden birisi de view içindeki presentasyon ile ilgili kodun ve iş

mantığının genellikle iç içe girmeleridir. İki katmanı birbirlerinden daha net biçimde ayıracak bir yapıya ihtiyaç vardır.

MVP Mimarisi ve İşleyişi

MVP'nin özü view sınıfı içerisinde yer alan GUI kodunu (sayfalar arasındaki akış, gui içerisindeki işleyiş vb) view sınıfı içerisinden çıkararak ayrı bir Presenter sınıfına taşımaktır. Böylece presentasyon ile ilgili kod GUI oluşturulması ve render edilmesi işlemlerinden bağımsız biçimde çalıştırılarak test edilebilmektedir.

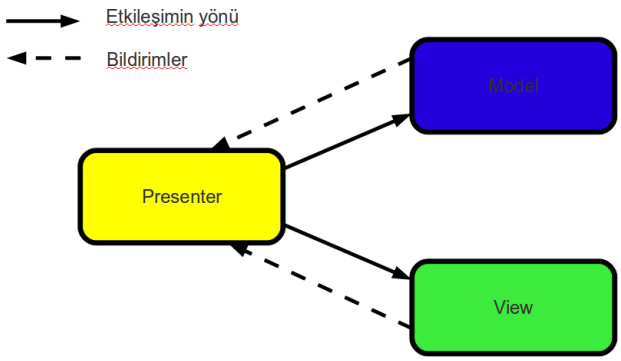


Presenter view tarafından kullanıcı input'unu elde ederek ilgili iş mantığını yürütmesi için işi model katmanına havale eder. Model tarafında işletilen davranış sonucu model üzerinde pek muhtemelen bir takım state değişiklikleri söz konusu olacaktır. Bu state değişiklikleri de yine presenter'a event'ler vasıtası ile haberdar edilir. Presenter'da bu state değişikliklerini uygun metotları kullanarak view tarafına yansıtır.

MVP ile “seperation of concern” hedefi daha kolay biçimde hayata geçirilebilir olmaktadır. Ayrıca uygulamaya ait davranışın da view'dan bağımsız biçimde kolay biçimde test edilebilir hale gelmesi sağlanmaktadır. MVP, yazılım ekiplerinin büyük bir uygulamayı fonksiyonel olarak gruplara ayırarak aynı anda birden fazla grubun beraber çalışarak geliştirmelerine de yardımcı olacak bir mimarisel altyapı sunmaktadır.

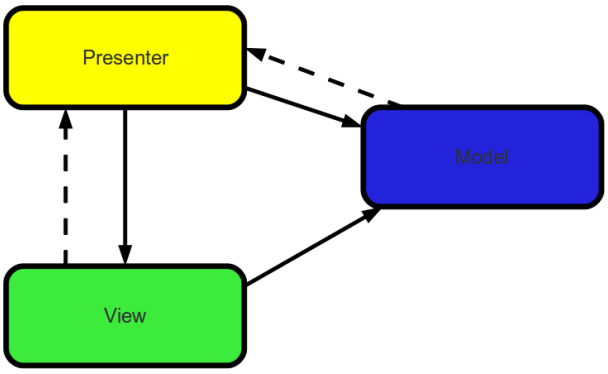
MVP Uyarlamaları

Passive View



Bu varyasyonun MVC'den en büyük farkı view model'den tamamen bağımsız ve bi-haber vaziyettedir. Model ile view arasındaki koordinasyonu Presenter veya Controller nesneleri gerçekleştirir. Presenter UI event'lerini ele alarak gerekli işlemleri yürüttükten sonra değişiklikleri View tarafına yansıtmakla görevlidir.

Supervising Controller



View – model arasındaki ilişki sadece “data binding” ile sınırlıdır. Modeldeki değişiklikler data binding ile view tarafına yansıtılabilir. Daha kompleks davranışlar ise yine Presenter üzerinden gerçekleştirilir.

Nereden Başlamalı? Nasıl Kodlamalı?

Modelden başlanması kullanıcının ilk etapta görmeyeceği veya etkileşimde bulunmayacağı kısımlara odaklanmaya neden olabilir. Alttan üste doğru geliştirme söz konusudur. Tam olarak domain kavranmadan modelin geliştirilmesi söz konusu olabilir. Modelin geliştirilmesi kullanıcı senaryolarının bir süre birikip sistemin fonksiyonaliteleri hakkında daha geniş bir fikir sahibi oluncaya kadar ertelenirse daha yararlı olabilir.

View'dan başlanması da sıklıkla görülmektedir. Sonuç olarak kullanıcı senaryoları bir takım fonksiyonları tarif etmektedir ve bu fonksiyonların gerçekleştirilebilmesi ve kullanıcının da bunları kısa zaman içinde görebilmesi, kullanarak geri bildirimde bulunabilmesi için view'dan başlanabilir. Ancak geliştirme sürecinin daha ilk evresinde kullanıcı arayüzüne odaklanılmasına neden olacaktır. Kullanıcıların gereğinden fazla kullanıcı arayüzlerine odaklaması sıklıkla arayüzlerin değişmesine neden olacak, bu da geliştirici ekibi daha önemli kısımlara odaklanmaktan alı koyacaktır. Diğer bir risk ise view katmanına gereğinden fazla iş mantığının yığılması ihtimalidir. Ayrıca GUI arayüzlerinin kolay biçimde test edilebilir olmaması ve TDD sürecinin sekteye uğratması da diğer bir dezavantajdır.

En iyi başlangıç noktası Presenter kısmıdır. Geliştirmeye kullanıcı senaryolarından herhangi biri seçilerek Presenter sınıfının implementasyonundan başlanır. Kullanıcı senaryolarındaki kullanıcı ifadeleri Presenter'daki metodun yapısına yön verir. Bu nedenle senaryolardaki kullanıcı ifadeleri olabildiğince korunarak Presenter metotları oluşturulmaya çalışılır. Bu sayede kullanıcıların talep ettiği fonksiyonel gereksinimlerin kod içerisinde birebir takibini yapmak da kolaylaşır.

Presenter sınıfı implement edilirken, ihtiyaç duyduğu model ve view sınıflarına karşılık gelen arayüzlerden mock nesneler oluşturulur. Bu sayede model ve view arayüzlerindeki davranışlar kullanıcı senaryoları implement edildikçe şekillenecektir. Senaryolara karşılık gelen birim testleri tamamlandıktan sonra model ve view arayüzlerine karşılık gelen gerçek sınıflar implement edilerek kullanıcı senaryosunun tam olarak çalışır hale gelmesi sağlanır.

Önce Presenter

Bu şekilde özellikle GUI içeren uygulamaların geliştirilmesine “Önce Presenter” (Presenter

First) yaklaşımı adı verilmektedir. GUI uygulamaları da çoğu zaman herhangi bir uygulama davranışı kullanıcının bir aksiyonu ile tetiklenmektedir. Bu nedenle kullanıcı senaryolarındaki kaydet butonuna tıklandığında..., sorgu sonuçları arasından bir kayıt seçildiğinde...,kayıt silindiği zaman..., gibi ifadeler bu yaklaşımda anahtar ifadelerdir. Presenter sınıflarındaki metotların neler yapması gerektiğini, hangi model ve view arayüzleri ile etkileşimde olacaklarını işaret ederler.

Kullanıcıların GUI üzerinde gerçekleştirdikleri işlemler bir takım event'leri tetikler. Bu eventler Presenter tarafından ele alınarak gerekli davranış hayata geçirilir ve sonuç olarak yine GUI tarafında birtakım değişiklikler, sonuçlar vs. kullanıcıya yansıtılır. Bu event'lerin çıktığı yerler view sınıflarıdır. Presenter nesneleri bu eventler hakkında haberdar edildiklerinde devreye girerler. View sınıflarından Presenter'da doğru olan iletişim her zaman için event'ler üzerinden gerçekleştirilir. Presenter'ın devreye girmesi sonucu ortaya çıkan durum değişiklikleri de yine view sınıflarının sunduğu metotlar aracılığı ile GUI'ye yansıtılır.

View ve Model sınıfları arasında herhangi bir bağlantı söz konusu değildir. View'daki herhangi bir değişiklik event'ler aracılığı ile Presenter'a iletilir. Presenter bu değişilikle ilgili gerekli yansıtmaları model'e yapar. Aynı şekilde Model'deki bir hangi bir değişiklik de Presenter'a yine event'ler aracılığı ile iletilir. Gerekli değişiklikler View'a yine Presenter üzerinden yansıtılır.

View ve model sınıflarının sahip olmaları gereken davranışlar Presenter sınıflarının geliştirilmesi sonucu kendiliğinden ortaya çıkacaktır. Bu arayüzler kullanıcı senaryoları için bir spesifikasyon görevi de görmektedirler. View arayüzleri çıkmaya başladıktan sonra genel hatları ile çalışan view'lar geliştirilerek kullanıcılardan geri bildirim alınabilir.

View sınıflarının tek görevi herhangi bir değişiklikten Presenter'ı derhal haberdar etmektir. Bunun dışında View sınıflarında herhangi bir davranış söz konusu değildir. View sınıfları bu nedenle GUI bileşenlerinin bir araya getirilmesi ve render edilmesi dışında herhangi başka bir fonksiyonaliteye sahip değillerdir.

Presenter ve Test Driven Development (TDD)

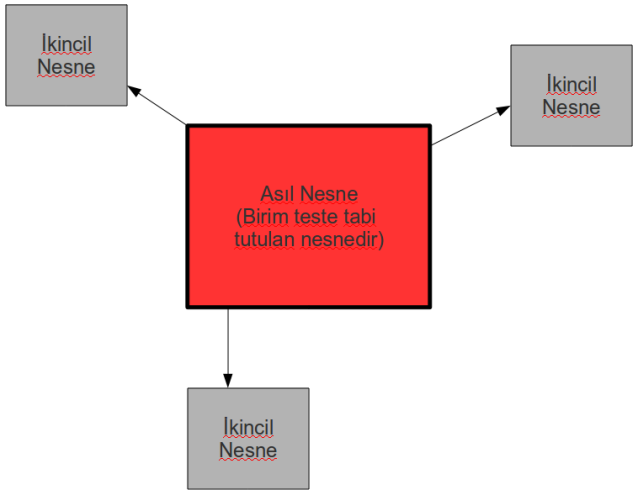
"Önce Presenter" yaklaşımı ile TDD pratiğini de uygulama geliştirmede hayata geçirmek çok daha kolay olmaktadır. Bu yaklaşımda view, model ve ihtiyaç duyulan diğer servis bileşenlerinin mock

türevleri oluşturularak Presenter nesnesine verilir. Presenter bu şekilde view, model ve servis katmanından bağımsız biçimde geliştirilebilmektedir.

TDD pratiğinde asıl nesnenin birim testlerinin oluşturulmasında genel olarak iki yaklaşımın kullanıldığı gözlemlenmiştir. Bunlar

1. Etkileşim tabanlı yaklaşım
2. Durum tabanlı yaklaşım

TDD pratiğinde birim teste tabi tutulan asıl nesnenin çalışması için ihtiyaç duyduğu diğer nesnelere ikincil nesneler adı verilmektedir.



Etkileşim tabanlı yaklaşımda ikincil mock nesneler üzerinde test edilen davranışla ilgili metotların uygun sayıda ve şekilde asıl nesne tarafından çağrılıp çağrılmadığı kontrol edilir. İkincil nesnelerin mock türevlerinin oluşturulmasının pek çok farklı nedeni olabilir.

- Gerçek implementasyonları hazır olmayabilir .
- Hazır olsa bile test ortamında yaratılması çalıştırılması zor olabilir , ya da çok yavaş çalışabilir, network veya dosya sistemi ile ilişkisi olabilir .
- GUI bağlantısı söz konusu olabilir .

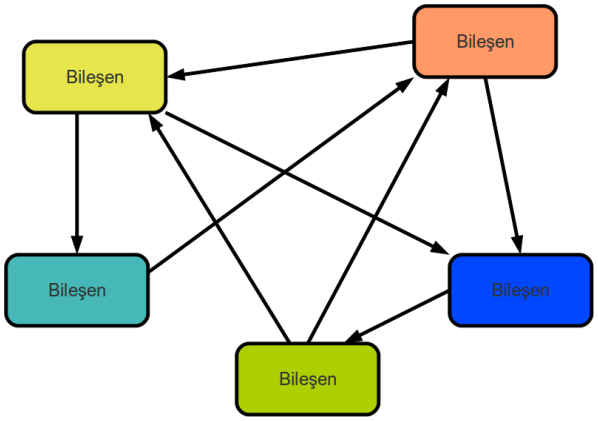
Bu ve benzeri nedenlerle ikincil nesnelerin asılları yerine sahteleri kullanılır . Bunlara da

"mock" nesneler adı verilir.

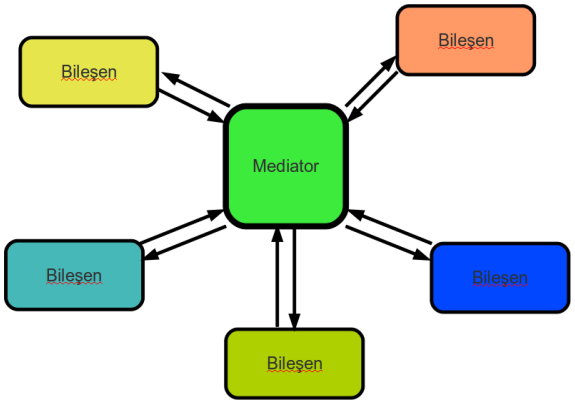
İkinci yaklaşımda ise birincil ve ikincil nesnelerin ilgili davranış sonrasında doğru state değerlerini yansıtıp yansıtmadıkları kontrol edilir. Bu yaklaşımda ise genellikle ikincil nesneler olarak sahte nesne türevleri değil, asıl nesnelerin kendileri kullanılır.

Farklı View-Presenter-Model Üçlülerinin Koordinasyonu (Mediator)

Farklı view-presenter-model üçlülerinin birbirleri ile entegre edilerek daha gelişmiş bir kullanıcı arayüzü geliştirilmesi GUI tabanlı uygulamaların geliştirilmesindeki genel çalışma mantığını oluşturur. Bu aşamada farklı bileşenlerin birbirleri ile iletişim ihtiyacı ortaya çıkar. Bu ihtiyaç sonucu farklı bileşenlerin birbirlerine bağımlı hale gelmesi yaygın bir mimarisel problemdir.



Mediator bir açıdan messenger üzerinden bir grup kişinin birbirleri ile haberleşmesine benzetilebilir. Grup içinden bir üye diğer herhangi bir veya birkaç üyeye herhangi bir mesaj göndermek için Mediator'ı kullanır. Mesaj mediator vasıtası ile diğer grup üyelerine iletilir. Grup üyeleri arasında doğrudan bir ilişki veya bağlantı söz konusu değildir. Grup üyeleri messenger üzerinden o anda iletişimde olanları da bilmezler.



Mediator sonrası bileşenler arasındaki iletişim ağı yukarıdaki gibi bir yapıya dönüşmektedir. Bu sayede bileşenlerin aynı uygulama içerisinde veya farklı uygulamalarda yeniden kullanılmalarının önü açılmaktadır.

Sonuç: "Önce Presenter" Yaklaşımının Faydaları

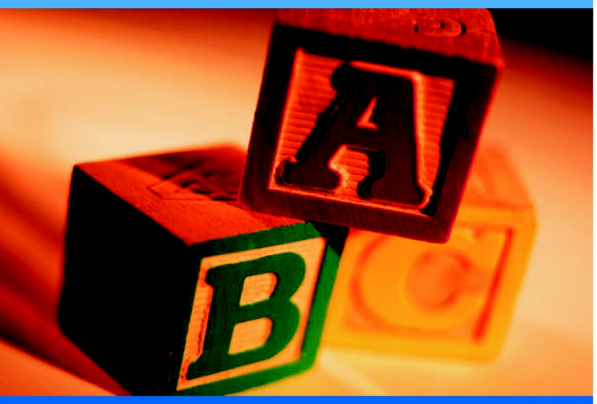
- Model katmanındaki sınıfların mock'lanması ile model verisinin elde edilme zorunluluğu ortadan kalkar. Veritabanı, network gibi bağlantılara, dosya erişimine vs. ihtiyaç duyulmadan geliştirme süreci sürdürülebilir.
- View sınıflarının mock'lanması sayesinde geliştirme sırasında uygulamanın çalıştırılarak test edilmesine, GUI oluşturulmasına gerek kalmaz.
- "Önce Presenter" yaklaşımı sayesinde geliştiriciler GUI bileşenleri üzerinden düşünmek yerine fonksiyonaliteye daha fazla odaklanma şansı bulmaktadırlar.
- Hemen bütün geliştiriciler kullanıcı arayüzü ile iş mantığı kodlarının birbirlerinden bağımsız olması gerektiği konusunda hem fikirdirler. Ancak zaman zaman hepimiz iki tarafın iç içe geçtiğine birbirlerine bağımlı hale geldiklerine şahit olmuşuzdur. "Önce Presenter" yaklaşımı kullanıcı arayüzü ve iş mantığı kısımlarının birbirlerinden ayrı tutulmaları için daha sistematik bir yol sunmaktadır.
- Fonksiyonalite birim testleri ile kontrol altına alındığı için kullanıcı senaryolarında yapılacak

herhangi bir değişiklik, ekleme veya çıkarma sonucunda ortaya çıkacak problemlerin erkenden tespit edilmesi mümkün hale gelir.

- Kullanıcı arayüzündeki değişiklikler çok daha kolay ve güvenli biçimde gerçekleştirilebilmektedir. Sonuçta burada yapılacak değişikliklerin herhangi bir biçimde işleyişi etkilemeyeceği bilinmektedir.
- Kullanıcı arayüzleri bu konu üzerinde uzmanlaşmış kişiler tarafından geliştirilebilmektedir.

Ürün ve Hizmetlerimiz

Speedy Framework (Model Güdümlü Çevik Yazılım Geliştirme Platformu)
Kurumsal Uygulama Geliştirme
Teknoloji Danışmanlığı ve Koçluk
Kurumsal Java Eğitimleri



Speedy Framework

(Model Güdümlü Çevik Yazılım Geliştirme Platformu)

Tamamen açık kaynak kodlu sistemler üzerine bina ettiğimiz Speedy Framework ile kurumsal yazılım geliştirme sürecini daha sistematik, otomatize, hızlı ve verimli bir hale getiriyor, yazılım sistemlerinin geliştirme, bakım ve idame maliyetlerini önemli ölçüde azaltıyoruz.

Speedy Framework'ün Faydaları

- Orta katman hizmetlerinin hazır olarak sunulması, uygulama geliştirmede ilk andan itibaren mimari yapının oturmuş olması ile yazılım geliştirme sürecinde önemli ölçüde bir hızlanma olur.
- Geliştirilen uygulamaların kalite düzeyinin uygulama genelinde aynı olması sağlanır.
- Uygulamalardaki kalitenin düzeyi uygulama geliştiricilerden bağımsız hale gelir.
- Model güdümlü yazılım geliştirme yaklaşımı, mimarisel yapının hazır olması, tekrar kullanılabilir servisler sayesinde uygulamada ortaya çıkabilecek hata sayısında da hissedilir bir düşüş söz konusu olur.
- Kullanıcı ara yüzleri bütün ekranlarda ve senaryolarda bir standarda sahip olduğu için müşterinin ve son kullanıcıların sisteme adaptasyonu kolaylaşıyor, sistemi öğrenme süreleri oldukça kısalır.
- Projelerin geliştirme sürecinden, bakım ve idame dönemlerine kadar bütün evrelerinde maliyetler azalır.

Daha fazla bilgi için: **http://www.speedyframework.com**

Kurumsal Uygulama Geliştirme

Kurumsal uygulama geliştirme faaliyetlerimizle kurumunuzun ihtiyaç duyduğu her türlü yazılım ihtiyacını karşılayacak, süreçlerinizi daha verimli hale getirecek çözümler üretiyoruz. Uzun yıllar boyunca pekçok kurumsal yazılım projesinde elde ettiğimiz bilgi ve tecrübemizle ihtiyacınız olan çözümleri en uygun şekilde hizmetinize sunuyoruz.

Teknoloji Danışmanlığı ve Koçluk

Java teknolojileri, kurumsal yazılım geliştirme, nesne yönelimli analiz, tasarım ve modelleme, yazılım mimarileri konularında bire bir koçluk ve proje danışmanlığı hizmetlerimizle yazılım geliştirme faaliyetlerinizin her adımında size destek oluyoruz.

Kurumsal Java Eğitimleri

JAVA Eğitimleri

Java Programlama Dili, Spring Application ve Security Framework, Hibernate, Object Oriented Analiz Tasarım, Tasarım Örüntüleri (Design Patterns), Aspect Oriented Programlama konularında verdiğimiz eğitimlerin kurumunuza ve çalışanlarınıza kesinlikle artı değer katacağına eminiz. Kurumlara özel ve genel katılıma açık olarak düzenlediğimiz eğitimlerle bilişim sektörümüze rafine bilgi ve tecrübeyi aktarıyoruz.

Daha fazla bilgi için: **http://www.java-egitimleri.com**