# NESNE YÖNELİMLİ PROGRAMLAMA 2(Object Oriented Programming 2/OOP)

**Öğr. Gör. Celil ÖZTÜRK**
Marmara Üniversitesi
Teknik Bilimler Meslek Yüksekokulu

# İçerik

- ✓ Yazılım Tasarımı
- ✓ Tasarım Prensipleri
- ✓ Tasarım Kalıpları
- ✓ Yazılım Kalitesi

# Yazılım Tasarımı

- Yazılım tasarımcıları da temelde diğer disiplinlerdeki tasarımcıların yaptığı işi yapar.
- Tasarlanan şey bir yazılım ürünüdür.
  - Yazılım tasarımı, müşterinin gereksinim ve isteklerini karşılayan yazılım ürününün doğasını ve bileşimini belirleme etkinliğidir.

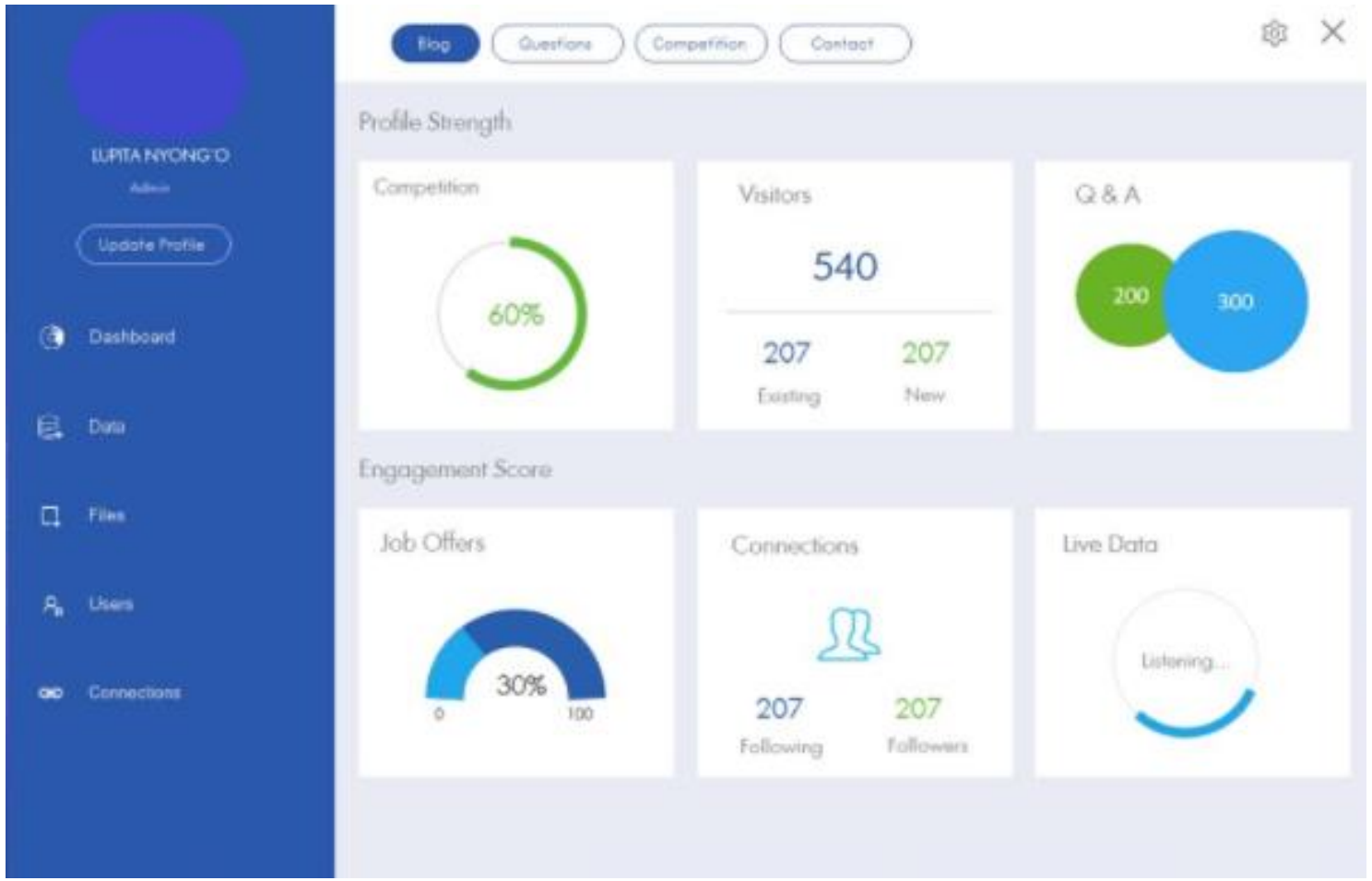
# Soyutlama

- Özellikle yazılım tasarımında Soyutlama son derece önemli bir problem çözme tekniğidir.
  - Soyutlama, problemin anlaşılmasını ve çözümünü kolaylaştırmak üzere nesnelerin, olayların veya durumların bazı özelliklerinin bilinçli ve kasıtlı olarak görmezden gelinmesidir.
  - Soyutlama, problemi (kısmen) çözerken problemin en önemli yönlerine odaklanma olanağı sağlar.

# Yazılım Ürün Tasarımı

- Yazılım ürün tasarımı, müşterinin gereksinim ve isteklerini karşılamak üzere yazılım ürününün özellikleri (feature), yetenekleri, ve arayüzlerinin belirlenmesi etkinliğidir.
- Kullanıcı arayüzü ve etkileşim tasarımı, iletişim, endüstriyel tasarım, ve pazarlama gibi beceriler gerektirir.

# Yazılım Ürün Tasarımı



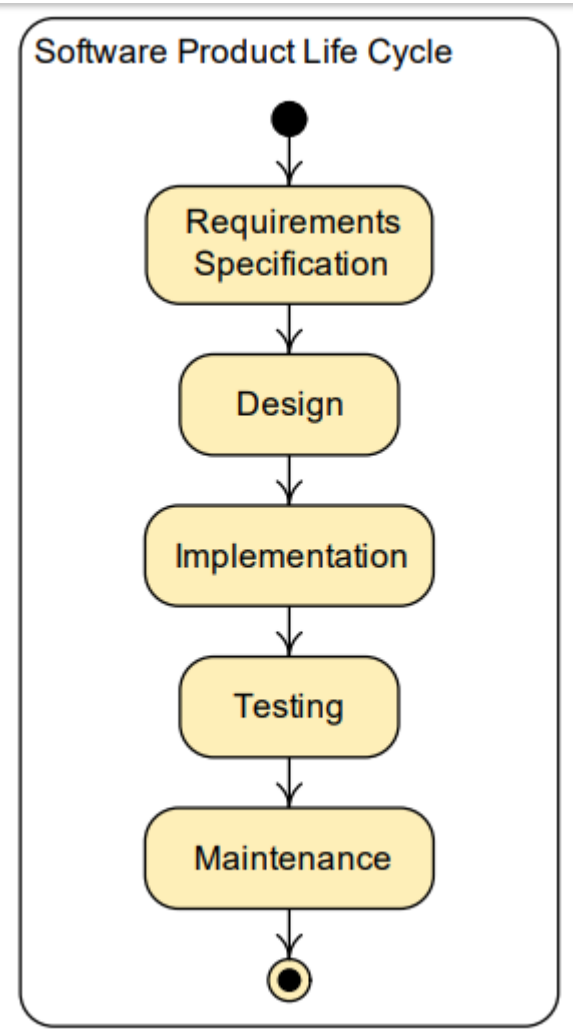

C#, VB.net, VC++.net - Cool Blue Modern Dashboard - Bunifu UI 現代設計用戶界面 - YouTube

# Yazılım Mühendislik Tasarımı

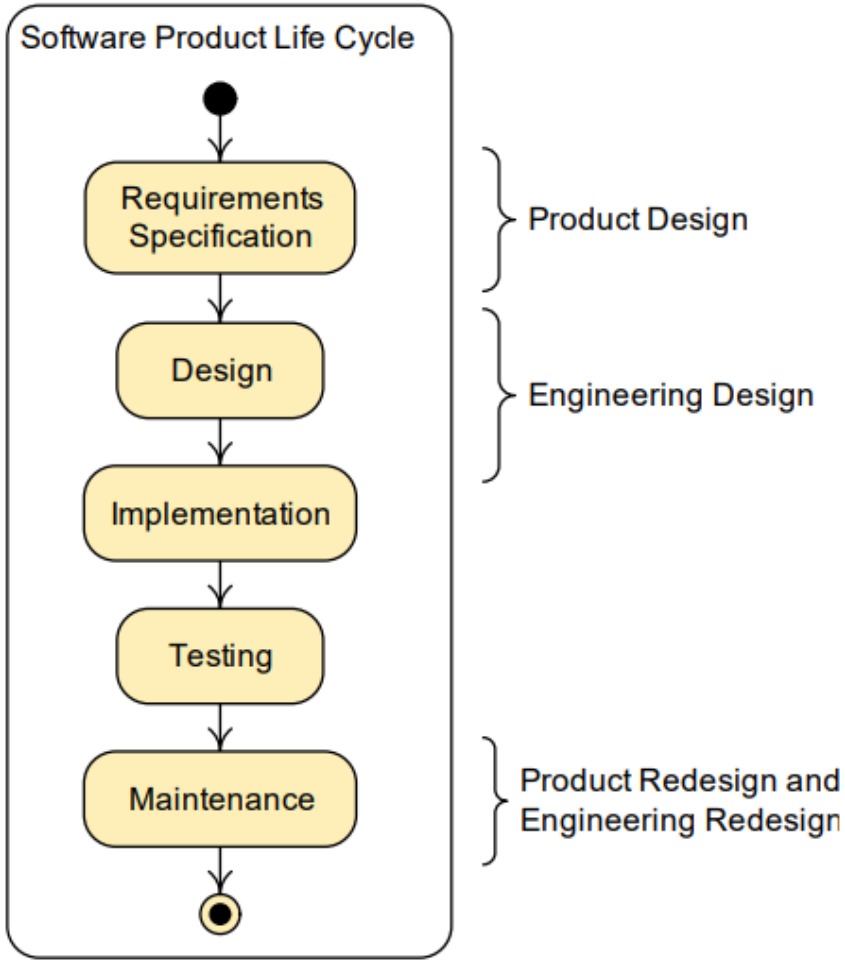
- Yazılım mühendislik tasarımı, yazılım ürün şartnamesini karşılamak üzere programların, altsistemlerin, ve bunları oluşturan parçaların ve çalışma biçimlerinin belirlenmesi etkinliğidir.
- Programlama, algoritmalar, veri yapıları, yazılım tasarım prensipleri, uygulamaları, süreçleri, teknikleri, mimarileri ve kalıpları konusularında bilgi ve beceri gerektirir

# Çağlayan Yaşam Döngüsü Modeli

- Çağlayan (waterfall) modeli yazılım geliştirme etkinlikleri arasındaki mantıksal ilişkiyi ortaya koyar.



# Yaşam Döngüsünde Tasarımın Yeri



# Tasarım Kalıpları

“Tasarım kalıpları, uzmanların yeni sorunları çözmek için geçmişte çalıştıkları çözümlerin uygulamalarının iyi belgelenmiş halidir.”

Tasarım kalıplarının arkasındaki düşünce, yazılım geliştirilirken sıklıkla karşılaşılan problemler için sunulan genel çözümler için standartlaşmış bir yol geliştirmektir.

# Tasarım Kalıplarının Avantajları

- Kalıpların standartlaştırılması, tüm geliştiricilerin (profesyoneller, yeni başlayanlar veya uzmanların) kararlarını daha kolay vermesini sağlamaktadır.
- Tasarım kalıpları ortak bir kelime haznesi sağlar. Bu geliştiriciler arasındaki iletişimi daha da kolay hale getirir. Bir tasarımı detaylıca açıklamaktansa, planlarımızı açıklamak için kalıp adını kullanabiliriz.
- Kalıplar birbirleri ile ilişkilendirilebilir, böylece geliştiriciler projelerinde hangi kalıpların birlikte bulunması gerektiğini kolayca anlayabilir.

# Tasarım Kalıplarının Avantajları

- Tasarım Kalıpları nesneye yönelik programlama topluluğu aracılıyla tecrübe paylaşımı için etkili bir yöntem sunmaktadır. Örneğin; C++, Smalltalk, C# ya da Java programlama dillerinde kazanılan bilgiler, Web projelerinde ortaya çıkan uzmanlık gibi öğrenilen bilgiler biriktirebilir ve bunlar diğer geliştiricilerle paylaşılabilir.

# Tasarım Kalıpları Tarihçesi

- Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides 1995’te "Design Patterns: Elements of Reusable Object-Oriented Software" kitabını çıkardılar. Bu dörtlü ayrıca "Gang of Four" olarak da bilinir.

# Tasarım Kalıpları Tarihçesi

- Bu dörtlü kitaplarında 3 farklı kategoride toplam 23 tane kalıba yer vermişlerdir:
- Creational: Nesneleri yaratmakla ilgili olan tasarımlardan 5 adet
- Structural: Nesneler arasındaki yapısal ilişkileri ifade eden tasarımlardan 7 adet
- Behavioral: Nesnelerin çalışma zamanı davranışlarını değiştirmek için oluşturulan tasarımlardan 11 adet kalıp bulunmaktadır.

# Tasarım Kalıpları

**Creational Patterns(Yaratımsal Kalıplar)**

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

# Tasarım Kalıpları

**Structural Patterns(Yapısal Kalıplar)**

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

# Tasarım Kalıpları

**Behavioral Patterns(Davranışsal Kalıplar)**

- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- Null Object Pattern
- Strategy Pattern
- State Pattern
- Visitor Pattern

# Yazılım Kalitesi

- Yazılım kalitesi, kullanım amaçlarına göre açıkça tanımlanmış işlev ve gereksinimlere uyum, kullanıcı isterlerine yanıt verebilme, açıkça belgelendirilmiş yazılım geliştirme standartlarına sadık kalma, yüksek güvenilirlik sağlama, üretilen yazılımda çeşitli teknik özelliklere sahip olma ve teslim sonrası destek olarak tanımlanabilir.
- Kötü tasarımlar kalitesiz yazılımların ortaya çıkmasındaki en büyük etmenlerdendir

# Yazılım Kalitesi

## Kötü Tasarım Belirtileri

Kötü Tasarım belirtilerini genel olarak şöyle gruplandırabiliriz:

1. Esnemezlik (Rijidite)
2. Kırılganlık(Fragility)
3. İmmobalite

# Yazılım Kalitesi

## Kötü Tasarım Belirtileri – Esnemezlik

Genel olarak esnemezlik, bir sistemin değişime karşı gösterdiği direnç olarak tanımlanabilir.

Esnemezliğin ölçütü regresyon olarak adlandırılır.

Bir yazılım sistemi zaman içinde değişen ve yeni gündeme gelen gereksinimleri karşılayabilmelidir. Eğer bir değişim ve gelişime karşı sistem, aşırı direnç gösteriyorsa ve *hatta değişime izin vermiyorsa* bu esnemez bir sistemdir.

# Yazılım Kalitesi

## Kötü Tasarım Belirtileri – Esnemezlik

- İyi bir tasarımda regresyonun düşük olması beklenir.
- Düşük regresyonlu sistemlerde olası değişimler büyük sorunlara neden olmadan yapılabilmektedir.
- Bir sistem ne kadar esnemez ise o kadar kırılgandır.

# Yazılım Kalitesi

## Kötü Tasarım Belirtileri – Kırılganlık

- Yazılım sistemleri olabildiğince esnek tasarlanmalı dolayısıyla daha az kırılgan olmalıdır.
- Muhtemel değişikliklere daha az direnç gösteren yani daha az kırılgan tasarlanmalıdır.
- Aksi takdirde yapılacak bir değişiklik, sistemde domino etkisi yaratıp, zincirleme bir şekilde birden çok şeyin değişmesini gerektirebilir.
- Değişimler yönetilemez hale gelir ve sistemi çökmeye (katastrofi) götürebilir.

# Yazılım Kalitesi

## Kötü Tasarım Belirtileri – İmmobilite(Taşınamazlık)

- Yazılım sistemleri ve bu sistemlerle ilişkili sınıflar, kütüphaneler gibi bileşenler modüler şekilde tasarlanmalıdır.
- Bir projede kullanılmış olan herhangi bir bileşenin farklı bir projeye taşınıp orada da kullanılabilmesi "yeniden kullanılabilirlik (reusability)" olarak bilinir.
- Pratikte çok kolay ve sorunsuz olabiliyorken çoğu zaman da ya çok zor ya da imkansız olmaktadır.
- **İmmobilite** diye bilinen bu durumun sebebi; genelde bağımlılıkların sayısı ve derinliğiyle yakından ilgilidir.

# Tasarım Prensipleri

1.Ayrıştırma (Decomposition)
2. Kohezyon (Cohesion)
3. Tek Sorumluluk Prensibi (Single Responsibility)
4. Zayıf Bağlaşım Prensibi (Low Coupling)
5. Yeniden Kullanılabilirlik prensibi (Reusability)
6. Açık – Kapalı Prensibi (OCP)
7. Liskov Yerine Geçme Prensibi (LSP)
8. Bağımlılığı Ters Çevirme Prensibi (DIP)

# Ayrıştırma (Decomposition)

- Nesne yönelimli tasarımın ilk prensibi olan ayrıştırmanın tanımı; problem alanındaki ilgili varlıkların tespit edilip, doğru nesneler biçiminde ifade edilmesidir.
- Bir yöntem olarak da nitelendirilebilecek ayrıştırma, sistemdeki karmaşıklıkla (complexity) başa çıkabilmek için yapılır.
- Buna kısaca böl ve fethet (divide and conquer) denilmektedir.

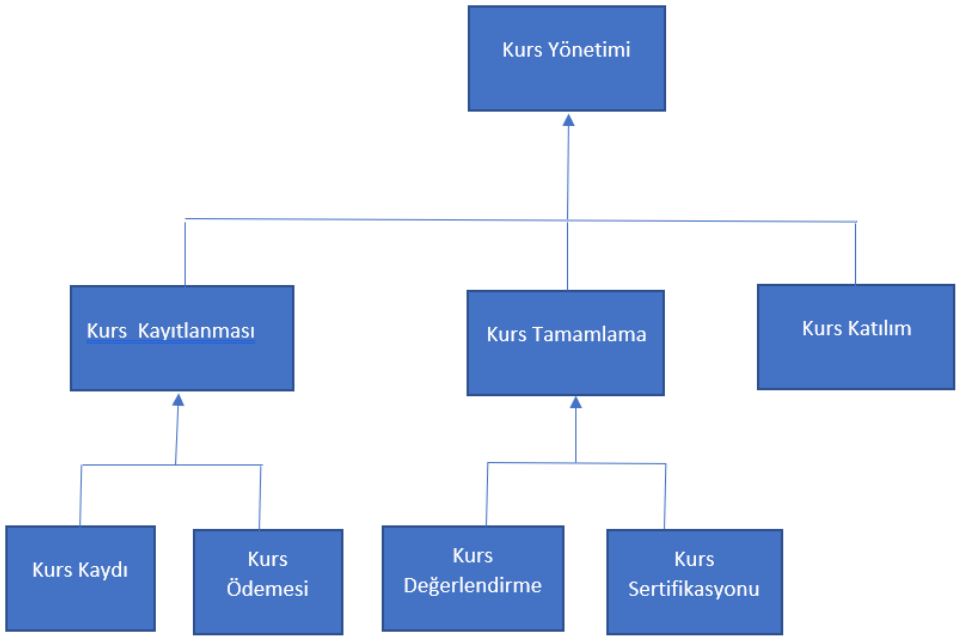
# Ayrıştırma (Decomposition)

- Yazılımcının çözümlemeyi gerçekleştirirken en büyük yardımcısı analiz sırasında ortaya çıkan use case tanımlarıdır.
- Analizle alakalı bir prensip diyebiliriz.
- Use case tanımlarını isim ve fiilleri ortaya çıkaracak şekilde okumak olası nesnelerin tespitine yardımcı olacaktır.
- Bu aşamada yazılımcı, "NASIL?" sorusu yerine "NE(LER)?” sorusunun cevabına odaklanmalıdır.
- Böylece algoritma düşünmek yerine nesneleri düşünmeye yoğunlaşabilir.

# Ayrıştırma (Decomposition)

## Fonksiyonel Ayrıştırma

- Büyük ve karmaşık fonksiyonlar bu yöntemle daha küçük ve anlaşılabilir alt fonksiyonlara ayrılırlar.
- Genellikle analiz aşamasında diagram şeklinde üretilirler.



# Kohezyon(Cohesion/Yapışıklık)

- Cohesion, modülün elemanlarının işlevsel olarak ilişkili olma derecesinin bir ölçüsüdür.
- Yazılım mühendisliğinde kohezyon; bir sınıftaki tanımlı üyelerin birbirlerine mantıksal ilişkilenmesi biçiminde tanımlanabilir.
- Diğer bir deyişle kohezyon, sınıf içerisindeki üyelerin ya da **fonksiyonlar içindeki görevlerin birbirlerine olan mantıksal uzaklığını** belirler.
- Tasarımda kohezyon faktörünün yüksek olması arzu edilir.
- Zira bir sınıfa ilişkin üye fonksiyonlar aralarında mantıksal bir kopukluk olmayacak şekilde yazılmış olmalıdır.

# Kohezyon(Cohesion/Yapışıklık)

- Üyeler arasındaki mantıksal kopukluklar kohezyonu azaltarak sınıfın yazılma amacını çarpıtacağı gibi bağımlılıkların da artmasına neden olur.
- Bir sınıf kendinden başka işler yapmaya başlarsa kohezyon düşük olur.
- Yapışkanlık derecesi düşük sınıflar modülerliği ve yazılım bakımını tehdit eden bir unsurdur.
- Sınıflar arası ilişkinin gevşek bağlı olması istenirken, sınıflar içersindeki yordamların ve veri alanlarının yapışkan/bağlantılı olması istenir.
- Bir sınıf *Single responsibility principle(tek sorumluluk prensibi)* çerçevesinde yalnızca tek bir görevi gerçekleştirmelidir.

# Kohezyon(Cohesion/Yapışıklık)

- Örneğin; string işlemleri için yazılmış bir sınıf, string'in ekrana basılmasını sağlayan fonksiyonlara sahip olmamalıdır.
- Her şeyden önce böylesi yanlış bir tercih, o sınıfı sadece belirli GUI sistemleriyle çalışmaya bağımlı kılacaktır.
- Öte yandan o string sınıfı içinde yazılan ve string işlemleriyle doğrudan ilişkili fonksiyonlar ise olması arzu edilen yüksek kohezyonu yaratacak bir tercihtir.

# Kohezyon(Cohesion/Yapışıklık)

```
public class Uye {

    private String uyeAdi;
    public String getUyeAdi(){ return uyeAdi;}
    public void setUyeAdi(String uyeAdi){}
    private void Ekle(){}
    private void Guncelle(){}

    private void Yazdir(){
    System.out.println("test");
    }
}
```

# Kohezyon(Cohesion/Yapışıklık)

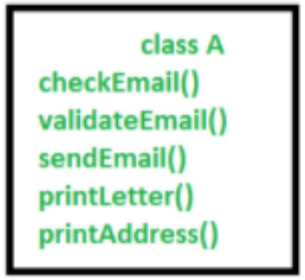



Fig: Low cohesion

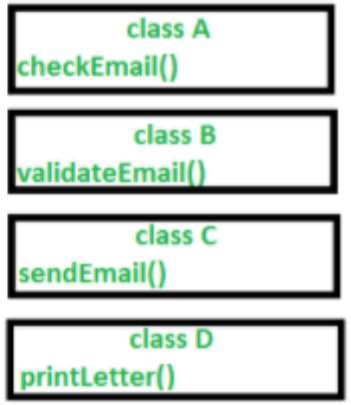



Fig: High cohesion

!! Yüksek kohezyon yazılım sisteminin esnekliğini arttırır, bakım ve yeniden kullanılabilirliği kolaylaştırır.

https://www.geeksforgeeks.org/cohesion-in-java/

# Kaynaklar


- Java ve Java Teknolojileri, *Tevfik KIZILÖREN* – Kodlab Yayınları
- Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014 • Eric Freeman, Head First Design Patterns, O'Reilly Media,2004 • Stephen Stelting & Olav Maassen, Applied Java™ Patterns, Prentice Hall PTR ,2001 • http://www.AlgoritmaveProgramlama.comDr Öğr. Üyesi Zehra Aysun ALTIKARDEŞ Nesne Yönelimli Programlama 2 Ders notları
- https://docs.oracle.com/cd/E17802_01/j2se/j2se/1.5.0/jcp/beta1/apidiffs/java/awt/FlowLayout.html
- Yazılım Mühendisliği CBU-Dr. Öğr. Üyesi Deniz Kılınç Yazılım Mimarisi ve Tasarımı Ders Notları
- Yazılım Kalitesi ve Kötü Tasarım Belirtileri | by Ramazan Ümit Bülbül | Medium
- Coupling ve Cohesion Kavramları Nedir? – KodEdu

# Kaynaklar


- Coupling ve Cohesion Kavramları | by Mehmet Serkan Ekinci | Medium
- Yazılım Çorbası: Cohesion Nedir? - Odaklılık Diyebiliriz (yazilimcorbasi.blogspot.com)
- https://www.geeksforgeeks.org/cohesion-in-java/