İçindekiler

[**YAZILIM MÜHENDİSLİĞİ** 3](#_Toc377922142)

[**SORU-1: SDLC (yazılım geliştirme yaşam dairesi) hakkında bilgi veriniz.** 4](#_Toc377922143)

[**SORU-2: Planlama Oyunu (Planning Game) hakkında bilgi veriniz.** 6](#_Toc377922144)

[**SORU-3: İkili Programlama (Pair Programming) hakkında bilgi veriniz.** 9](#_Toc377922145)

[**SORU-4: Observer Design Pattern (Gözlemci Tasarım Kalıbı) hakkında bilgi veriniz.** 10](#_Toc377922146)

[**SORU-5: Document Management Systems (Doküman Yönetim Sistemleri) hakkında bilgi veriniz.** 16](#_Toc377922147)

[**SORU-6: Spagetti Kod (Spaghetti Code) hakkında bilgi veriniz.** 17](#_Toc377922148)

[**SORU-7: Örnek Kodlama Kuralları** 18](#_Toc377922149)

[**SORU-8: Zeki Vekiller (Akıllı Ajanlar, Intelligent Agents, Zeki Etmenler) hakkında bilgi veriniz.** 20](#_Toc377922150)

[**SORU-9: C2 Üslûbu (C2 Style) hakkında bilgi veriniz.** 21](#_Toc377922151)

[**SORU-10: JML ( Java Modelleme Dili) hakkında bilgi veriniz.** 23](#_Toc377922152)

[**SORU-11: Program doğruluğu (Program correctness) hakkında bilgi veriniz.** 24](#_Toc377922153)

[**SORU-12: Kara Kutu Yaklaşımı (Black Box) hakkında bilgi veriniz.** 25](#_Toc377922154)

[**SORU-13: Nokta Kesiciler (PointCuts) hakkında bilgi veriniz.** 28](#_Toc377922155)

[**SORU-14: Birleşim Noktaları (JoinPoints) hakkında bilgi veriniz.** 29](#_Toc377922156)

[**SORU-15: Bağlam Örücüler (Apect Weavers) hakkında bilgi veriniz.** 29](#_Toc377922157)

[**SORU-16: Veri Bütünlüğü (Data Integrity) hakkında bilgi veriniz.** 30](#_Toc377922158)

[**SORU-17: Bağlam Yönelimli Programlama (Aspect Oriented Programming) hakkında bilgi veriniz.** 31](#_Toc377922159)

[**SORU-18: MVC (Model View Controller, Model Bakış Kontrolcü) hakkında bilgi veriniz.** 33](#_Toc377922160)

[**SORU-19: Atomluluk (Atomicity) hakkında bilgi veriniz.** 34](#_Toc377922161)

[**SORU-20: Tekrarlı ve Arttırımlı Geliştirme (Iterative and Incremental Development) hakkında bilgi veriniz.** 35](#_Toc377922162)

[**SORU-21: Bağlama (Coupling) hakkında bilgi veriniz.** 36](#_Toc377922163)

[**SORU-22: SDK hakkında bilgi veriniz.** 36](#_Toc377922164)

[**SORU-23: UML (Unified Modeling Language, Ortak Modelleme Dili) hakkında bilgi veriniz.** 36](#_Toc377922165)

[**SORU-24: CASE Araçları (Case tools) hakkında bilgi veriniz.** 37](#_Toc377922166)

[**SORU-25: En kötü durum analizi (Worst Case Analysis) hakkında bilgi veriniz.** 37](#_Toc377922167)

[**SORU-26: Birliktelik, Münasebet ve Oluşum (Association, Aggregation and Composition) hakkında bilgi veriniz.** 40](#_Toc377922168)

[**SORU-27: Şelale Modeli ( Waterfall Model ) hakkında bilgi veriniz.** 40](#_Toc377922169)

[**SORU-28: Tasarım Kalıpları (Tasmim Kalıpları, Design Patterns) hakkında bilgi veriniz.** 41](#_Toc377922170)

[**SORU-29: Vücubiyet (Modality) hakkında bilgi veriniz.** 42](#_Toc377922171)

[**SORU-30: Sayısallık (Cardinality) hakkında bilgi veriniz.** 43](#_Toc377922172)

[**SORU-31: ERD (Unsur İlişki Çizimi, Entity Relationship Diagram) hakkında bilgi veriniz.** 45](#_Toc377922173)

[**SORU-32: İlişkinin Anlık Görüntüsü (Instance of a Relation) hakkında bilgi veriniz.** 46](#_Toc377922174)

[**SORU-33: İlişki (Relationship) hakkında bilgi veriniz.** 47](#_Toc377922175)

[**SORU-34: Unsur (Entity) hakkında bilgi veriniz.** 48](#_Toc377922176)

# **YAZILIM MÜHENDİSLİĞİ**

## **SORU-1: SDLC (yazılım geliştirme yaşam dairesi) hakkında bilgi veriniz.**

Bu yazının amacı, İngilizcedeki Software Development Life Cycle (SDLC) yani Türkçe karşılığı ile yazılım geliştirme hayat dairesini (yaşam döngüsünü) açıklamaktır.

Her geliştirme işlemi özellikle de ürün geliştirme işlemi belirli aşamalar içerir. Yazılımların da birer ürün olduğu unutulmamalıdır. Bu bağlamda nasıl bir otomobil, bir uçak veya bir televizyonun geliştirilmesi için çeşitli aşamalardan bahsetmek mümkünse, yazılım için de benzer aşamalar mevcuttur. Genel olarak bu ürünlerin tamamını birer sistem olarak görmek ve sistem geliştirme metodolojisi ismi altında incelemek mümkündür.

Her sistem öncelikle tasarlanır, üretilir, piyasaya arz edilir ve neticede piyasadan yok olur, yerine yeni bir sistem arz edilir ve bu daire böylece devam eder. Benzer şekilde yazılım da öncelikle bir ihtiyaçla başlar. Birileri bir bilişim sisteminin neler yapması gerektiği, ihtiyaçlara nasıl cevap vermesi gerektiği konusunda fikirle işe başlar. Bu bilişim sistemini kullanacak olan yapının altında ayrılabilecek kaynaklar belirlenir. Ardından bu yazılımın yerine ikame edileceği mevcut sistem tahlil edilir. Örneğin bir Doküman yönetim sistemi için mevcut işleyiş incelenir. Mevcut işleyişe yeni beklentiler sisteme eklenerek, sistemin tanıtımı yapılır, gerekli eğitimler verilir ve sistemin kurulumu gerçekleştirilir.

Her kurumda benzer aşamalardan geçilen yazılım geliştirme süreçleri ve bilişim sistemleri kavramlarını farklı kelimeler ve aşamalarla anlatmak mümkün olsa da genel olarak aşağıdaki maddeler altında toparlamak mümkündür.

1. Planlama ve seçim aşaması
2. Tahlil (analiz) aşaması
3. Tasarım (Design) aşaması
4. Fiil aşaması (implementation)

Yukarıdaki aşamalar aşağıdaki şekilde temsili olarak resmedilebilir:



Şekil 1: Yazılım geliştirme yaşam dairesi

Yukarıdaki şekilde de temsili olarak resmedildiği üzere bu dört aşama aslında bir dairenin merhaleleridir. Yani bütün aşamalar tamamlandıktan sonra tekrar başa dönülerek herşey yeniden devam eder. Bunu örneğin piyasadaki ürünlerin yaşamlarının son bulması ve yeniden yeni ürünlerin piyasaya girmesi olarak görmek mümkündür. Aslında her yazılımın da bir ömrü vardır ve bu ömür tamama erdikten sonra yazılımda gerekli yenilemeler ve değişikliklerle yeni yazılım müşterinin / kurumun hizmetine sunulur ve bu süreç bir süreklilik arz eder.

Yukarıdaki bu aşamaları aşağıdaki şekilde detaylandırmak mümkündür.

1. Sistem planlama ve seçim aşaması:

Bu aşamada, iki önemli adım atılmalıdır. Birincisi birisinin yeni ve daha gelişmiş bir bilişim sistemine olan ihtiyacı belirtmesi ikincisi ise bilişim sisteminin kullanılacağı ortamdaki ihtiyaçların belirlenmesidir.

Bu iki adım atılırken aşağıdaki yan amaç ve sorular gündeme getirilmelidir:

1. Mevcut sistemdeki problemlerin çözülmesi için kullanılan yöntemler ve bu yöntemlerin iyileştirilmesi
2. Sistemin iyileştirilmesi için ortamdaki beklenti düzeyi ve buna olan inanç
3. Bilişim sisteminin gerekliliği ve sisteme yapacağı katkı düzeyi.

Bu aşama, planlamanın bütün gereklerini tamamlar. Zaman kaynak ayrımı, personel, kurumun mevcut bilişim seviyesi ve yeni sistemin kullanılabilirliği, uzun vadede sistemin bakımı ve sürdürülebilirliği gibi çok sayıda parametre bu aşamada incelenir. Bu aşamanın tamamlanmasının ardından bir uygunluk raporu (feasibility report) hazırlanarak kurumdaki irade sahiplerine (yönetim) sunulur ve bilişim sisteminin yapacağı olumlu katkıların maliyet analizine göre katma değeri olduğu kesinleştirilir.

2. Sistem Tahlili (Analiz): Bu aşamada sistem analizi yapılır ve geliştirilmekte olan bilgi sisteminin etkilediği diğer birimler ve mevcut işleyiş tahlil edilir. Örneğin geliştirilmekte olan sistem ödeme emirleri ile ilgili olsun, bu durumda girilen emirlerin satış, defteri kebir, stok kayıtları gibi çok sayıdaki etkilediği diğer kayıt tahlil edilerek bu sistemlerle olan etkileşimi çalışılır.

3. Sistem Tasarımı (Design): Bu aşamada sistemin bütün bileşenleri ve genel yapısı tasarlanır. Çok çeşitli tasarım yöntemleri ve aşamaları olmakla beraber genelde mantıksal bir tasarım aşamasından geçilerek sistemin hedefleri ve etkileşimde olduğu alt bileşenleri arasındaki uyum gözetilir. Ardından bu mantıksal tasarımın fiziksel tasarıma çevrilmesi aşaması gelir. Fiziksel tasarıma çevrilim aşamasında, programlama dilinden kullanılacak olan teknolojilere ve sistemin bütün parçalarına kadar olan tasarımı tamamlanır. (veri tabanı tasarımı, kullanıcı ekranları, veri akış yolları gibi).

4. Uyarlama ve işletme aşaması (implementation and operation): Bu aşama, şimdiye kadar yapılan bütün kağıt üzerindeki işlemlerin gerçeğe dönüştürüldüğü ve yazılım halini aldığı aşamadır. Kodlama, kod kontrolleri, testleri, kurulumu, yönetimi gibi işlemlerin tamamı bu aşamada yapılır.

Yukarıdaki 4 aşama kapsamında SDLC adımlarının çıktıları aşağıdaki şekilde özetlenebilir:

|  |  |
| --- | --- |
| Safha | Neticeler |
| Planlama ve seçim safhası | Sistem öncelikleri  Veri, ağ, donanım ve bilgi sisteminin mimarisi  Seçilmiş olan proje için detaylı çalışma planı  Sistemin geçerlilik alanı için özellikler  Sistemin iş akışı açısından önemi |
| Sistem Tahlili | Mevcut sistemin tanımı  Mevcut sistemin düzeltilmesi, iyileştirilmesi, değiştirilmesi için öneriler  Muhtelif sistemler için açıklamalar ve alternatiflerin değerlendirilmesi  Yeni teknoloji gereksinimlerinin belirlenmesi ve geçiş için öneriler |
| Sistem Tasarımı | Sistemin bütün alt bileşenleri için detaylı birer tasarım |
| Uyarlama ve işletme | Kod  Dokümantasyon  Eğitim ve destek kaynakları  Yeni sürüm için kod, eğitim ve destek aşamalarındaki değişiklikler. |

Tablo 1: SDLC adımları

Ayrıca yukarıdaki 4 adımdan oluşan SDLC yaklaşımı için alternatif olarak sunulan yöntemler veya araçlardan da bahsetmek mümkündür.

Örneğin bilgisayar destekli yazılım mühendisliği araçları (computer allied software engineering, CASE tools), ortak uygulama tasarımı (joint analysis design, JAD), hızlı uygulama geliştirme (rapid application development, RAD), katılımcı tasarım (participatory design, PD), atik yöntemler (agile methodologies) gibi farklı birer yazı konusu olan yöntemler kullanılabilir.

## **SORU-2: Planlama Oyunu (Planning Game) hakkında bilgi veriniz.**

Bu yazının amacı, özellikle uç programlama (extreme programming) konusunda sıkça kullanılan aşamalardan birisi olan planlama oyunu (planning game) kavramını açıklamaktır.

Planlama oyunu, aslında belirli zaman kutularında (time box, zaman aralıkları) yapılan toplantılara verilen isimdir. Örneğin haftada bir yapılan toplantıladır ve bu toplantılarda iki önemli kısım bulunur:

* Sürüm Planlaması (Release Planning): Bu kısımda, yakın dönem sürümlere odaklanılır, bu planlama aşamasında müşteri ile ortak olarak çalışılmaktadır. Sürüm planlamasında üç temel aşama vardır:
  + Keşif Aşaması (Exploration Phase): Bu aşamada, müşteri kendisi için yüksek öneme haiz işlerin kısa bir listesini sunar. Bu bilgi ayrıca ileride kullanılmak üzere kullanıcı hikayesi kartına (user story card) yazılarak kaydedilir.
  + Taahüt Aşaması (Commitment Phase) : Bu aşamada iş tarafı ve yazılımcı tarafı, bir sonraki sürüm tarihi ve bu sürümdeki fonksiyonel özellikler konusunda taahütleşirler.
  + Yönlendirme Aşaması (Steering Phase) : Bu aşamada, planda değişiklikler yapılarak ilerlenir, yeni ihtiyaçlar eklenebilir veya mevcut ihtiyaçlarda değişiklik yapılarak çıkarılması tartışılabilir.
* Dönüş Planlaması (Iteration Planning): Bu aşamada program geliştiricilerinin yapacakları işlemler ve görevleri planlanır. Bu aşama geliştirme ekibi içinde yapılan planlamadır ve müşteri tarafı katılmaz. Dönüş planlamasının da kendi içinde üç ayrı alt aşaması bulunmaktadır.
  + Keşif Aşaması (Exploration Phase) : Bu aşamada ihtiyaçlar farklı görevlere çevrilir. Bu görevler daha sonra kullanılmak üzere görev kartlarına (task card) kayıt edilir.
  + Taahüt Aşaması (Commitment Phase): Görevler programcılara atanır ve bu görevlerin zamanı ve son tarihinin tahmini yapılır.
  + Yönlendirme Aşaması (Steering Phase) : Görevlerin işletilmesi sağlanır ve sonuçların daha önceden hazırlanan kullanıcı hikayeleri (user story) ile uyumu kontrol edilir.

Planlama oyununun genel amacı, ara sürümlerle üretilen ürünün teslime hazır hale getirilmesidir. Burada klasik programlama yaklaşımına göre en büyük farkı, ürünün tamamlanması ve tam olarak bitmiş ürüne ihtiyaç duyulan tarih ve bitmiş ürünün kullanılma geçirildikten sonra yaşanacak problemlerin tahmini gibi zor işleri bir seferde ve büyük risk alarak yapmak yerine projeyi yönlendirerek teslime hazır hale getirmektir.

Sürüm Planlaması (Release Planning)

* Keşif Aşaması(Exploration Phase) : Bu aşama, dönüş işlemi olarak görülebilir ve bu aşamada ihtiyaçlar toplanır ve aşağıda listelenen ihtiyaçların işe olan etkileri araştırılır.
* Hikaye Yazma : Bu aşamada, kullanıcı hikayesi (user story) ismi verilen bir hikaye hazırlanır. Genelde bir toplantı şeklinde geçen ve kullanıcının problemlerine ve ihtiyaçlarına odaklanılan bu aşamada, kullanıcının istediği sistemi ve bu sistemin parçalarını tanımlamasına ve nasıl çalışması istediğini anlatmasına yardımcı olunur. Bu aşamada, olumsuz etkisi olmaması için, programcıların hikayeye etkisinin olmaması beklenir ve oluşturulan hikaye kullanıcı hikaye kartına (user story card) yazılır.
* Hikaye Tahmini: Bu aşamada programı geliştirecek olan yazılımcı ekibinin, istenen özellikleri içeren kullanıcı hikaye kartına göre bir süre tahmini yapması beklenir. Bu aşamadaki geliştirme çözümlerinin iş mantığına müdahalesine izin verilmez, sadece tanımlı iş mantığının geliştirilmesi için tahmin yapılması beklenir.
* Hikayenin Bölünmesi: Bu aşamada hikaye alt parçalara bölünür. Döngü başlamadan önce kritik karmaşıklıktaki parçalar tespit edilir. Şayet programı geliştirecek olan programcılar, hikayeyi tahmin etmekte güçlük yaşıyorsa, hikayenin yeniden bölünerek yazılması gerekir.

Şayet iş sürecinde yeni bir işlem yapılması gerekmiyorsa, taahüt adımına geçilir.

Taahhüt Aşaması

Bu aşamda, yazılımın geliştirme sürecine dahil olan bütün taraflar, belirli taahütler altına girmiş sayılır. Bu aşamanın amacı, maliyet, çıkarlar ve zamanlama konusunda kesin sonuçlar elde etmektir ve dört bileşeni bulunur:

* Değere göre sıralama: Kullanıcı hikayelerini iş sürecindeki kişiler değerine göre sıralar
* Riske göre sıralama: Geliştirme ekibi, hikayeleri taşıdıkları risklere göre sıralar. Bu aşamda üç tip risk tanımı yapılabilir, düşük orta ve yüksek. Ayrıca risk araştırması sırasında farklı açılardan konu ele alınabilir. Tamamlılık (completeness), yani bir hikayenin bütün detaylarını biliyor muyuz? Değişkenlik (volatility) yani bir hikayenin ileride değişme ihtimali bulunuyor mu? Karmaşıklık (complexity) yani bir hikayenin tamamlanması için ne kadar emek harcanması gerekiyor? Bu sorulara düşük orta veya yüksek cevapları verilerek daha sonra ortalaması alınıp hikaye için bir değerlendirme yapılabilir.
* Hıza göre sıralama: Geliştirme ekibi, hikayeleri, geliştirme sürelerine ve projenin en hızlı bitmesine yardımcı olacak şekilde sıralarlar.

Konuya göre seçim: Kullanıcı hikayeleri arasından, çıkacak ilk sürümde bitmesi gerekenler seçilir.

**Yönlendirme aşaması (Steering Phase):** Bu aşamda, iş sürecinden veya programlama sürecinden kişiler projeyi yönlendirebilirler. Buna göre bu kişiler projede değişiklik yapabilir. Bireysel olarak kullanıcı hikayeleri ele alınıp, değişik kullanıcı hikayeleri ile olan bağlantıları incelenip bu hikayelerin öncelikleri belirlenebilir.

Dönüş Planlaması (Iteration Planning): Bir dönüş 3 temel unsurdan oluşur, bunlar keşif aşaması (exploration phase), taahhüt aşaması (commitment phase) ve yönlendirme aşaması (steering phase) olarak sıralanabilir ve detayları aşağıda sunulmuştur.

* Keşif aşamasında, projedeki görevlerin ve bu görevlerin sürelerinin tahmini yapılır. Bu aşamada aşağıdaki adımlar izlenebilir:
* İhtiyaçların göreve dönüşmesi: Görev kartlarına işlenir
* Görevlerin birleştirilmesi veya bölünmesi: Şayet programcı bir görevin çok küçük veya çok büyük olmasından dolayı tahminde bulunamıyorsa, görevi alt parçalara bölmek veya birleştirmek gibi yollar izleyebilir.
* Tahmin: Bu aşamda programcı görevin süresini tahmin eder.
* Taahhüt aşaması (commitment phase): Bu aşamda, kullanıcı hikayelerine dayanılarak programcılara planlama kapsamındaki görevleri atanır. Bu atama işlemi ve programcıların taahhüde girmesi aşağıdaki adımlardan oluşur:
* Programcı görevi kabul eder: Her programcı, kendi sorumluluğunu alabileceği bir görevi kabul eder.
* Programcı görevi tahmin eder: Programcı sorumluluğuna aldığı görev ile ilgili tahminde bulunur.
* Yük çarpanının belirlenmesi (load factor): Yük çarpanı, programcının kendisine atanan görev ile ilgili her döngüde ne kadar vakit harcayacağını belirler. Örneğin haftada 40 saat harcanacak bir iş atandıysa, döngünün her dönüşünde programcı bu görev ile ilgili 40 saat harcamakla yükümlüdür. Elbette bir programcının uygunluğuna göre birden fazla görev alması söz konusu olabilir ve döngü süresi örneğin 1 hafta olan bir çalışma ortamında programcı 5 saatlik 3 görev 10 saatlik 1 görev gibi farklı boyutlarda ve dolayısıyla farklı yük çarpanına sahip görevlere atanabilir.
* Dengeleme (Balancing): Bir ekipte bulunan bütün programcılara görevler atandıktan sonra bu görevlerin süre tahminleri ve yük çarpanları üzerinden son bir kere gçeilerek karşılaştırma yapılır. Örneğin bir proje ekibinde haftalık toplam 5 saat yük çarpanı olan bir programcı ve haftalık toplam 40 saat yük çarpanı olan başka bir programcının bulunması sağlıklı değildir. Şayet 40 saatlik tek bir görev bulunuyorsa ve tek bir kişiye atanması gerekiyorsa, geri dönülerek görevin alt görevlere bölünmesi yolu izlenebilir.
* Yönlendirme Aşaması (Steering Phase) : Görevlerin yönlendirilmesi aslında dönüşün son adımıdır ve aşağıdaki adımlardan oluşur:
* Bir görev kartının alınması: Programcı daha önce taahüdü verdiği görev kartılarından birisini üzerine alır.
* İkili arkadaşını bul: Programcı bu işte beraber çalışacağı ekip arkadaşı ile eşleşir. Bu işlem aslında ikili programlama (pair programming) olarak bilinen yaklaşımdaki iki kişinin bir görev kartı üzerinde eşleşmesidir.
* Görevi tasarla: Şayet gerekiyorsa, görevin fonksiyonel tasarımını programcılar yaparlar.
* Birim testi yaz (unit test): Programcılar, programlamaya başlamadan önce bu programdan bekledikleri görevlere uygun olarak yerine getireceği testleri yazarlar.
* Kodu yaz: Bu aşamada kod yazılır.
* Testi çalıştır: Kod bittikten sonra, programcılar daha önceden belirledikleri testleri bitmiş kod üzerinde çalıştırırlar.
* Temizleme (refactor) : Bu aşamada, kodda bulunan kod kokuları (code smells) temizlenir.
* Fonksiyonel Test: Bu aşamada kullanıcının hazırladığı kullanıcı hikayesine uygun olarak fonksiyonel gereksinimlerin sağlanıp sağlanamadığı test edilir.

Planlama oyunu, sadece planlama aşamasında yapılan işlemlerden oluşur ve yukarıdaki yazıdan da anlaşılacağı üzere her aşamada bir planlama yapılması söz konusudur. Ancak burada dikkat edilecek husus, aslında yukarıdaki adımların tamamının bir döngünün alt adımları olduğudur. Yani birinci dönüşten sonra tekrar en başa dönülerek yeni bir sürüm için yeniden aynı adımlar takip edilir.

## **SORU-3: İkili Programlama (Pair Programming) hakkında bilgi veriniz.**

Bu yazının amacı, özellikle bilgisayar bilimlerinin bir alt konusu olan yazılım mühendisliğinde (software engineering) kullanılan bir teknik olan atik programlama (agile programming) kavramının getirdiği bir yöntem olan ikili programlama (pair programming) kavramını açıklamaktır.

Bu yöntem basitçe, tek bir bilgisayarın önünde iki kişinin oturarak çalışması mantığı üzerine kuruludur. Yani yazılım projelerinde sıkça ihtiyaç duyulan ve risk azaltmak için kullanılan kişilerin yedeklenmesi (ve dolayısıyla kişilerin bilgilerinin ve becerilerinin de, işten çıkma, sağlık problemleri vs. risklere karşı farklı bir kişiye edindirilerek yedeklenmesi) yönteminin anlık olarak yapılmasıdır.

Yapılan işin karmaşıklığı ve programcıların uyumlu çalışma tecrübelerine bağlı olarak, aynı işi iki programcının aynı anda yapmasının aslında ayır ayrı işi paylaşarak yapmalarına göre %15 ile %50 arasında bir iş gücü kaybına sebep olduğu anlaşılmıştır.

Ancak ikili programlama yöntemi, aynı programa iki farklı kişinin aynı anda bakması sonucunda, programcıların hatalarının daha önce tespit edilmesi ve çalışma motivasyonunun arttırılması gibi olumlu sonuçlar doğurur.

Örneğin, programcıların tek başınayken kendilerini disipline etmesi (self disipline) gerekirke, iki programcının aynı anda işten kaytarması, internet üzerinde vakit kaybetmesi gibi etkilerin azaldığı tespit edilmiştir.

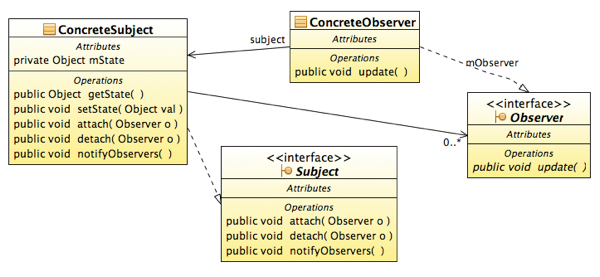
İkili programlamanın da diğer yöntemler gibi bazı problemleri ortaya çıkardığı kesindir. Bu problemlerin bazı emareleri aşğıda listelenmiştir:

1. Uzaklaşma (Disengagement) : Çalışanlardan birsinin, diğerinden fiziksel olarak uzaklaşmaya çalışması, örneğin sandalyesini uzaklaştırması veya klavyesini uzaklaştırması gibi belirtilerdir. Bazı durumlarda çalışanın uyuklaması veya konudan uzaklaşması gibi de görülebilir.
2. Efendiyi İzleme : Bazı durumlarda çalışanlardan birisi çok daha tecrübeli olduğu için efendiliğini ilan eder ve diğer çalışanın çalışmaya olan motivasyonu zaman içerisinde düşer, bu durum genelde uzaklaşma ile sonuçlanır.
3. Sessizlik : Şayet ikili arasında sessizlik oluyorsa, bu beraber çalışmadıklarının en bariz emarelerindendir.
4. Fikir Ayrılıkları : İkili programlamanın en büyük problemlerinden birisidir. Genelde bir programın yazılmasının birden fazla yolu vardır ve programcılar şayet bir fikir ayrılığı yaşarlarsa, bu durum “Efendiyi İzleme” emaresi ile sonuçlanabilir. Genelde bir çözüm olarak 3ncü bir kişiden istifade edilmesi veya takım lideri veya internette daha önceden anlaşılmış bir başvuru kaynağından faydalanmak doğru çözüm olarak görülmektedir.
5. Emek Çeşitliliği: Genelde performans değerlendirmelerinde, bir iş bir ikiliye atandığında, genelde işin yükü ikiye bölünerek iki tarafın da performansı değerlendirilir. Ancak çalışma sırasında iki kişiden birisinin daha fazla emek harcaması durumunda adaletsizlikler ortaya çıkmaktadır ki genelde tam eşit emek ortaya konulması neredeyse imkansızdır. Bu durum, bazı çalışanlarda haksızlık hissi doğurmakta ve yönteme karşı tepki gösterilmesi ile sonuçlanmaktadır. Genel bir çözümü olmamakla birlikte, farklı ekip arkadaşları ile çalıştırarak çeşitliliği arttırmak bir yöntem olarak görülmektedir.

## **SORU-4: Observer Design Pattern (Gözlemci Tasarım Kalıbı) hakkında bilgi veriniz.**

Bu yazının amacı, nesne yönelimli programlama ortamlarında kullanılan bir tasarım kalıbı (design pattern) olan gözlemci tasarım kalıbını (observer design pattern) açıklamak ve kullanımına dair bir örnek vermektir. Nesne yönelimli olmayan programlama dillerinde (örneğin C) aynı yapı, geri çağırım (callback) ismi verilen yaklaşım ile yapılabilir.

Konuya, klasik bir gözlemci tasarım kalıbını açıklayarak başlayalım. Geliştirme ortamı olarak sıklıkla kullandığım Netbeans içerisinde bir observer pattern eklenmesi halinde aşağıdakine benzer bir sınıf diyagramı (class diagram) görülmektedir:



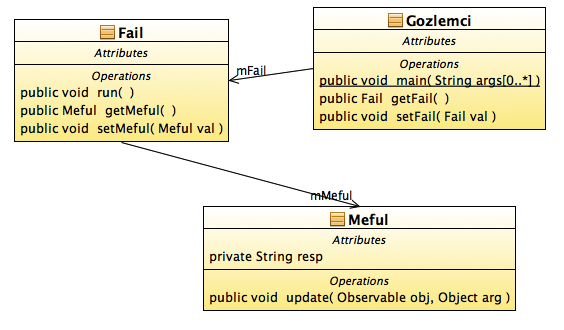
Şekil 1: Sınıf diyagramı

Yukarıdaki kalıpta görüldüğü üzere 4 temel unsur bu kalıbı oluşturmaktadır:

* ConcreteObserver : Somut bir gözlemci
* ConcreteSubject: Somut bir fail (özne)
* Subject : Somut failden (özneden) türemiş olan diğer failler
* Observer: Somut gözlemciden türemiş olan diğer gözlemciler./

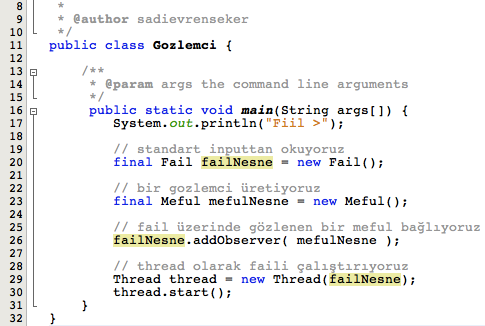
Gözlemci tasarım kalıbı, yukarıdaki şekilde görülebileceği üzere, aslında bir Fail / Meful ilişkisi (subject / object, özne /nesne ) üzerine kuruludur. Somut fail ile kastedilen, gözlemci yapısındaki herhangi bir fiile sebep olan (eylemi yapan) faildir (özne). Buna karşılık, nesne yönelimli programlama dillerinde, üzerine bir fiil uygulanan şey genelde nesne olarak ifade edilir. Yani zaten nesne yönelimli dillerde herhangi bir fiil, sadece nesneler üzerine uygulanmaktadır. Bu durumda fail ile kastedilen, yukarıdaki kalıpta concreteSubject ve bu yapıdan türeyen bütün subject sınıfı nesneler, meful ise bu nesnelerin metotlarına parametre olarak geçirilen ve üzerinde işlem yapılan nesnelerdir.

Bu yaklaşımı bir kod örneği üzerinden açıklamaya çalışalım. Öncelikle tasarımımızın sınıf diyagramını (class diagram) vererek konuya başlayalım:



Şekil 2: Sınıf diyagramı

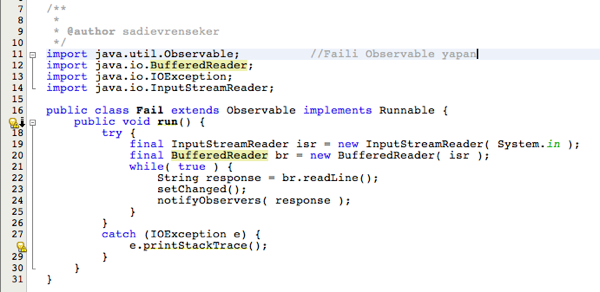
Yukarıdaki şekilde, modelimizde bulunacak olan üç unsur içinde, yani gözlemci, fail ve meful için birer sınıf tanımı yaptık. Bu sınıfların kodu aşağıda verilmiştir:



Şekil 3: Sınıf kodları

İlk olarak gözlemci sınıfı (class) ile konuya başlayalım. Sınıfın tanımında, bir main fonksiyonu bulunuyor ve bu main fonksiyonu aslında fail , meful ve gözlemci kavramlarının bir araya geldiği nokta oluyor. Zaten sınıf diyagramında da görüleceği üzere gözlemci bu iki kavramı bir araya getirir. Buna göre fail ve meful sınıflarından üretilen failNesne ve mefulNesne nesneleri (objects) (kodun 20. ve 23. satırlarında), kodun 26. satırında, addObserver fonksiyonu ile birbirine bağlanmıştır. Bu basit işlem, meful üzerinde bir failin gözlemini tanımlar. Ayrıca kodun 29. satırında klasik thread üretmek için kullanılan ve Runnable arayüzü üzerinden Thread çağırımı ile failNesne’mizi bir thread şeklinde başlatıyoruz.

Gelelim Fail sınıfının tanımına:

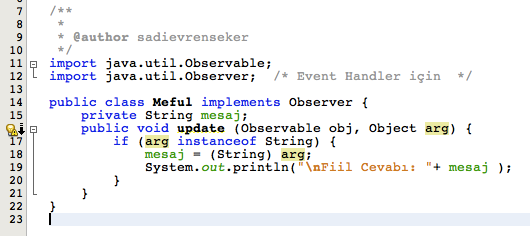


Şekil 4: Fail sınıfı

Yukarıdaki kodda, basit bir şekilde, kulanıcıdan, ekranda yazı okumak için BufferedReader, InputStreamReader’a ve InputStreamReader da System.in’e bağlanmış. Bu sayede, System.in üzerinden girilen bir veri, BufferedReader tarafından okunabilecektir. response ismindeki String değişkeni (variable) bir sonsuz döngü içerisinde (21. satırdaki while) kullanıcıdan veri okumakta ve sırasıyla setChanged() ve notifyObservers() fonksiyonlarını çağırmaktadır. Buradaki notifyObservers fonksiyonu aslında bütün bu yazının kalbini oluşturuyoru. Dikkat edilirse bu fonkisyona klavyeden okuduğumuz response isimli değişkeni parametre olarak veriyoruz.

Kodda dikkat edilecek diğer iki husus ise, Observable sınıfını miras almamız (inheritance) ve bir thread olarak çalışmasını istediğimiz için Runnable arayüzünü (interface) implements etmemizdir.

Son olarak meful sınıfını açıklayıp, kodun çalışmasına bakalım:



Şekil 5: Meful sınıfı

Kodda görüldüğü üzere, Meful isimli sınıfımız, basitçe Observer arayüzünü kendi üzerine uygulamıştır (implements).

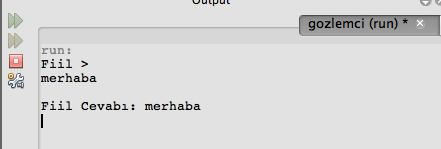
Meful sınıfının yegane fonksiyonu update fonksiyonudur ve parametre olarak bir Observable nesnesi bir de arg isimli Object tipinde (yani JAVA açısından tipsiz) bir parametre almaktadır.

Burada tipsiz tipler ile ilgili aşağıdaki yazıyı okumakta yarar olabilir

Son olarak update fonksiyonu içerisinde alınan mesaj, tip inkılabı (typecasting) ile Object tipinden String tipine inkılap ettirilmiş ve ardından ekrana basılmıştır.

Şimdi yukarıdaki senaryoyu toparlayacak olursak, gözlemci sınıfımızdaki main fonksiyonu çalıştırılarak bütün senaryo başlamaktadır. Bu main fonksiyonu içerisinde bir fail bir de meful nesnesi üretilmekte ve üretilen nesneler birbirine addObserver fonksiyonu ile bağlanmaktadır. Yani artık meful sınıfından türeyen nesneler, fail sınıfının birer observer’ı olmaktadır. Ardından fail sınıfından üretilen nesne bir thread olarak çalıştırılmakta ve kendi içerisinde bulundurduğu, ve threadlerin ilk çalışan fonksiyonu olan run() fonksiyonunda tanımlı olan kullanıcıdan bir dizgi (String) okuma işlemini gerçekletirmektedir. Hemen ardından okunan mesajı, aslında hiçbir fikri olmayan ismini, ne tür bir parametre aldığını dahi bilmediği notifyObservers () fonksiyonuna parametre geçirmektedir. Bu fonksiyon, bir sevki tabii olarak meful sınıfındaki update isimli fonksiyonu tetiklemektedir. Update fonksiyonu ise kendisine parametre olarak gelen nesneyi String sınıfına çevirmekte ve ekrana bamaktadır.

Yukarıdaki kodu çalışması sonucu ekran görüntüsü aşağıdaki şekildedir:



Şekil 6: Ekran görüntüsü

Görüldüğü üzere kodumuz bir fiil sorusu sormakta ve klavyeden benim yazdığım “merhaba” yazısını ekrana geri basmaktadır. Buradaki mesajı okuyan fail isimli sınıftan türeyen nesne iken, ekrana cevabı yazan meful isimli sınıftan türeyen nesnedir

Yukarıdaki bu tasarım kalıbını, daha gerçekçi bir örnek üzerinden açıklamaya çalışalım.

Örneğin bir müzayede (açık arttırma) ortamını, gözlemci tasarım kalıbı ile modellemek isteyelim. Buna göre müzayedeyi yöneten bir kişi ve bu müzayedeye katılan müzayedeciler bulunacaktır. Buna göre müzayedeyi yöneten kişi bir gözlemci olmakta, müzayedecilerin her biri birer fail (fiili yapan kişi) olmakta ve müzeyede sırasında verilen her teklif (fiyat) bir meful (fiilin etkisindeki değer) olmaktadır.

Bu yaklaşımı aşağıdaki şekilde göstermeye çalışalım:



Şekil 7: Yaklaşım

Yukarıdaki yaklaşımda, muzayede sırasında verilen değerler, gözlemcinin update fonksiyonuna birer parametre olarak geçirilmekte ve nihayetinde gözlemciden bağımsız olarak ilgili fonksiyonlar çağrılmaktadır.

Gözlemci tipi tasarım kalıbının bir dezavantajı, bu noktada çıkmaktadır. Gözlemci tasarım kalıbında, meful yapısında olan ve fiilden etkilenen nesnelerin sistemde etkin rol oynaması beklenir. Diğer bir deyişle, açık arttırmaya katılan her bir katılımcı, fail olarak fiyatlarını belirtirken, gözlemci bu durumu sadece gözlemekte ve bir fiilde bulunmamakta ancak fiileri gözlemektedir. Bu anlamda, gözlemci tasarım kalıbı, çekme iletişim modeli (pull interaction model) olarak düşünülebilir. Alternatifi olan itme iletişim modelleri (push interaction model) failin doğrudan eylemde bulunması ile sonuca ulaşmakta ve dışarıdaki nesneleri çalışmaya zorlamaktadır. Yani gözlemci örüntü modelinde, dışarıda bulunan nesnelerin çalışmaları sonuçları toplanmaktayken, itme iletişim modeli olan diğer kalıplarda, bir nesne, diğer nesnelerin sonuç üretmesini zorlayabilir. Burada, tasarıma bağlı olarak bir seçim yapmak gerekir.

## **SORU-5: Document Management Systems (Doküman Yönetim Sistemleri) hakkında bilgi veriniz.**

Bu yazının amacı, bilgisayar dünyasında kullanılan döküman yönetim sistemlerinin genel bir tanımını yapmak ve bir doküman yönetim sisteminin yapısını açıklamaktır.

Genel olarak bir doküman yönetim sistemi (document management system), herhangi bir organizasyondaki dokümanların (yazışmalar, evraklar, formlar veya resimler gibi) takibini yapmayı, tasnif ederek erişimini kolay hale getirmeyi, sorgulamalara ve aramalara cevap vermeyi hedefler.

Daha büyük ölçekte, içerik yönetim sisteminin bir parçası olarak görülebilir.

Bir doküman yönetim sisteminde, şart olmamakla birlikte genelde aşağıdaki öğeler bulunur:

**Üst bilgi (Metadata):** Bir dokümanın, doğrudan içeriğine ait olmayan ancak dokümanın algılanması ve kullanılması sırasında işe yarayan bilgilerdir. Örneğin bir resmin çekildiği tarih, veya sisteme kaydedildiği tarih, dokümanın üst bilgisidir ancak bu bilgi resmin içnde yer almak zorunda değildir.

**Havuz (repository):** Doküman yönetim sistemlerinde (document management systems) kullanılan ve dokümanların saklandığı alandır. Basitçe havuza nasıl erişileceği, kimlerin ne kadarına erişeceği veya dokümanların havuzda nasıl tutulacağı, doküman yönetim sistemi tarafından belirlenir. Örneğin dokümanların doğru bir veri tabanı yönetim sistemi tarafından (database management system , DBMS) nasıl kullanılacağı bir havuz problemidir.

**Bütünleşme (Integration):** Dokümanlar ile uygulamalar arasındaki ilişkidir. Örneğin bir kullanıcı bilgisayarında tuttuğu hesaplama tablolarını bir uygulama üzerinden kullanmaktayken (örneğin MS Excel veya Open Office Calc gibi) bu veriler doküman yönetim sisteminin havuzunda durmaktadır. Kullanıcı şeffaf bir doküman yönetim sisteminde (transparent) sadece uygulama ile muhatap olurken, doküman yönetim sistemi, arka planda kullanıcının verileri ile uygulamayı bütünleştirmektedir.

**Fihristleyici (Indexer):** Dokümanların kolay erişilmesi için belirli bir fihrist yapısına oturtulmasını sağlayan programdır. Örneğin dokümanların ağaç yapısında (tree) veya özetleme tablolarında (hash table) tutulması, arama sırasında, daha hızlı erişilmesini sağlar. İşte fihristleyici bu görevi üstlenerek uygun veri yapısı üzerinde (data structure) dokümanlara erişim sağlamaktadır.

**Getirme (Retrieval):** Dokümanın havuzdan getirilmesi işlemidir. Çeşitli sistemlerde oldukça karmaşık yapılara ulaşabilir. Örneğin internet üzerinde arama yapan bir motorun (örneğin google) desteklediği arama cümlelerini kabul eden bir doküman yönetim sisteminin, fihristleyici üzerinde doğru erişimleri yapması ve getirmesi, normalden daha karmaşık algoritmalar gerektirmektedir. Hatta günümüzde doğal dil işleme (natural language processing) kullanılarak, kullanıcıların konuşur gibi, sistemden doküman aramaları ve doküman yönetim sisteminin (DMS) bu dokümanı bulup getirmesi mümkün hale gelmiştir. Ayrıca getirilen sonuçların kullanıcıya gösterilmesi sırasındaki sıralama da ayrıca önem taşımaktadır. Örneğin 100 adet doküman döndüren bir aramanın sonucunda hangi dokümanın ilk sırada gösterileceği, skorlama algoritmaları ile çözülmektedir ve bütün bu işlemler “getirme” (retrieval) olarak geçmektedir.

**Dağıtım (Distribution):**Bu terim aslında literatürde bilgisayar bilimi temelli kişiler için bir karmaşıklık oluşturmaktadır. Literatürde iki farklı anlamda kullanıldığı görülebilir. Birincisi çok yoğun ve yüksek miktarda verilerin saklandığı ortamlarda, kaynakların verimli kullanılması için dokümanların dağıtılmasıdır. Bu klasik bilgisayar bilimleri tanımı olup tamamen yük kaygısı ile performans arttırma amaçlı yapılan bir işlemdir. Örneğin bir bankanın her şubesinin kendi dokümanını, kendi şubesinde tutması ve verilere erişimde hız kazanması gibi. Öte yandan doküman yönetim sistemlerinde bir dokümanın, taraflar arasında dağıtılması da bir problemdir. Örneğin hukuki bir metni, bütün muhataplarının okumuş olduğunun garanti edilmesi, hatta dijital olarak imzalaması (tebliğ / tebellüğ edilmesi) bir sözleşmenin bütün ilgili ve sadece ilgili kişiler tarafından okunması gibi problemler de dağıtım başlığı altında geçmekte olup bazı doküman yönetim sistemlerinde desteklenen özelliklerdendir.

**Güvenlik (Security):** Doküman yönetim sistemindeki kullanıcıların kendi yetkileri (authentication) çerçevesinde erişime sahip olması ve dokümanların tam olarak korunuyor olmasıdır.

**İş akışı (workflow**): Bir dokümanın izlemesi gereken yolları belirtir. Örneğin bir satış işlemi sırasında, önce satış sözleşmesinin taraflarca onaylanması, satıcı tarafın deposuna sevk emrinin geçirilmesi, depodan çıkan ürünlerin taşıyıcı firma tarafından onayı, aradaki sigorta evraklarının tanzimi, alıcı firma tarafından malların alındığını onaylayan dokümanlar ve nihayetinde para transferini onaylayan banka dokümanlarının tamamı doküman yönetim sistemi tarafından takip edilip doğru sıra ile, doğru taraflar tarafından, evrak atlanmadan yapılmış mı diye sorgulanabilir. Elektronik doküman yönetim sistemleri (electronic document management system), bu kontrol işlemlerinin tamamının otomatik olarak yapılmasını ve ilgili kullanıcılara raporlanmasını hedefler.

**Sürümleme (Versioning) :** Bir doküman üzerinde değişik zamanlarda yapılan çalışmaların takip edilmesini hedefler. Örneğin dokümanda yapılan her işlem yeni bir sürüm numarası ile saklanır ve kullanıcılar gerekli gördükleri durumlarda, eski sürümlere dönerek işlem yapabilir.

**Yayınlama (Publishing):** Dokümanın tam olarak doğru algılanmasını sağlamak amacıyla, doküman üzerinde, yanlış anlamalara mahal verebilecek bütün eksiklik ve hataların düzeltilmes, tahsis edilmesi ve redaksiyonu işlemidir. Yayına hazırlanan bir dokümanın standartlarının belirlenmesinde ayrıca dokümanlar kullanılmaktadır.

## **SORU-6: Spagetti Kod (Spaghetti Code) hakkında bilgi veriniz.**

Bilgisayar kodlamasında, bir kodun okunabilirliğinin düşük olması, yani kod takibinin zor olması durumunda, koda verilen isimdir.

Genellikle yapısal programlama dillerinde (structured programming languages) fonksiyonların bulunması ile birlikte GOTO veya JMP gibi, kodun içerisinde bir yerden başka bir yere atlayan komutların kaldırılması mümkündür.

Bu tip komutların kullanılması durumunda, kodun hem diğer programcılar tarafından okunabilirliği düşer, hem de kodun karmaşıklığı kestirilemez bir hâle gelebilir.

Spagetti kodu engellemek için 3 temel adım kullanılır:

1. Kodda bulunan ve koşula bağlanması istenen satırlar için if bloğu kullanılır
2. Kodda bulunan ve tekrarlanması istenen satırlar için döngü kullanılır
3. Kodda bulunan ve parametrize edilmesi istenen satırlar için (aynı satırların farklı değerlerle çalışması isteniyorsa) fonksiyon kullanılır.

Yukarıdaki bu üç adım tamamlandıktan sonra kodda herhangi bir GOTO satırı kalmamalıdır.

Ayrıca spagetti kodundan esinlenerek katmanlı mimarinin kullanıldığı ve projelerin birden fazla seviyeye bölünerek ele alındığı yaklaşımlara lazanya kod (lasagna code) ; veya özellikle nesne yönelimli programlama dillerinde (object oriented programming languages), program parçalarının birbiri ile olan ilişkisinin asgariye indirildiği ve neredeyse her parçacığın tek başına varlığını sürdürebildiği kodlama yaklaşımlarına da ravioli kod (ravioli code) isimleri verilmektedir.

## **SORU-7: Örnek Kodlama Kuralları**

Bu yazının amacı, geliştirme ortamlarında kullanılan kodlama standartları için örnek bir doküman oluşturmaktır. Kodlama standardı, daha sonradan yapılacak olan bakım, güncelleme ve ekleme işlemlerinin kolaylaşmasını sağlarken, dokümantasyon ve projeye dahil olan yeni personelin adaptasyonunu hızlandıracaktır. Kurallar, aşağıdaki başlıklar halinde sunulmuştur.

**Karakter Kodlaması**

Kod geliştirilmesi sırasında, kullanılan isimlerin hiçbirisinde Türkçe karakter tercih edilmemelidir. Kullanılan isimlerin, fonksiyon veya sınıfı tanımlar nitelikte olması ve ayrıca kodun içerisinde yorum satırı şeklinde içeriğine dair bilgi vermesi beklenir.

**Deve İsimlendirmesi (Camel Notation)**

Bu yazım türünde, birden fazla kelimeden oluşan isimlerin, her kelimesinin ilk harfi büyük diğer harfleri küçük olarak yazılır.

**Örnek**

DenemeBasligi

**Sınıf isimleri**

Bütün sınıf isimleri büyük harflerden oluşan bir ön ek alır. Bu ön ek, BoonEX projesi kapsamında yazılması durumunda BX’tir. Bunun dışında, özel bir proje için eklenen sınıflar, projeyi belirten ön ekler alır. Örneğin öğrenci portalı şeklindeki bir proje için OP ön eki eklenebilir. Ayrıca sınıf isimleri yegane olmalıdır (unique) ve deve isimlendirilmesi kullanılmalıdır.

**Örnek**

OPDenemeSınıfı

**Değişken Tanımlama**

Bütün değişken isimleri, küçük harf ile başlamalıdır. Ayrıca değişkenin tipine dair bir ön ek almalıdır. PHP dilinde mevcut bulunan değişken tipleri ve bu değişken tiplerine göre eklenebilecek ön ekler aşağıda listelenmiştir.

* i Integer
* f Float, Double
* s String
* a Array
* r Resource
* b Boolean
* is Boolean

**Örnek:**

$sIsim = “Variable Value”;

$iSayi = 0; $iSayi++;

$rPrjDosya = fopen (“dosya.txt”, “r”);

var $isYaz;

**Fonksiyon Tanımları**

Fonksiyon isimleri, değişken tanımına uymalıdır. Fonksiyonların küme parantezleri, fonksiyonun tanımlandığı satırın sonunda açılmalıdır.

**Örnek:**

function denemeFonksiyonu($iSayi, $sIsim){

//kod

}

**Yapısal Dil Formatları**

PHP diline özgü blok kodları (for, while, do.. while, foreach, if…else gibi), aşağıdaki kurallara uymalıdır.

* Bu blokların, parantez ile blok kelimesi arasında bir boşluk olmalıdır.
* Blok parantezleri, satır sonunda başlamalıdır.
* İşlem kullanılması durumunda, işlem ile operantlar arasında boşluk bulunmalıdır

**Örnek:**

foreach ($aIsimler as $sAnahtar => $sDeger){

//Kod

}

**switch** yapısı

Bu yapı, kod okunabilirliği açısından kullanılmayacaktır. Bunun yerine if / else blokları kullanılacaktır.

**Kod Ekleme Standartları**

Kod eklemesi yapılırken, eklemenin ismi, kim tarafından eklendiği ve hangi proje kapsamında yapıldığı yorum satırı olarak kodun başına ve sonuna yazılır.

**Örnek:**

//Ekleme ismi, kodlayan, proje ismi.//

Kod();

//[Bitti] Ekleme ismi//

Ekleme işlemi yapılmadan önce, orijinal dosyanın yedeği alınmalı ve ekleme işlemi, kopya dosya üzerinde yapılarak orijinal kod saklanmalıdır.

**Database Tables**

Veri tabanından dosya oluşturulması gerektiği durumlarda, proje ismi ve eklenen modül ismi ön ek olarak konulmalıdır.

**Örnek**

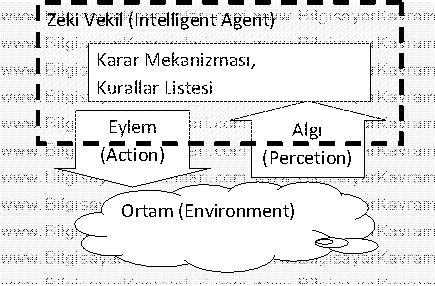
OPModIsmi\_Kullanicilar

## **SORU-8: Zeki Vekiller (Akıllı Ajanlar, Intelligent Agents, Zeki Etmenler) hakkında bilgi veriniz.**

Zeki vekiller (etmenler, ajanlar) kavram olarak, bilgisayar bilimlerine, felsefe, biyoloji ve ekonomi alanındaki çalışmalardan sonra girmiştir. Bu alanlardaki anlamı ve kullanımı, genellikle herhangi bir işin farklı bir vekil tarafından yürütülmesi olarak anlaşılabilir.

Bilgisayar bilimlerin açısından zeki kelimesi, bir vekilin herhangi bir işlemi belirli inisiyatifler kullanarak yerine getirmesidir. Örneğin zeki olmayan bir vekil, her adımda ve her işlemde kullanıcıya bir şeyler sorarken, zeki vekilde daha çok otonom bir yapıdan (autonomous) bahsedilebilir.

Bu anlamda her zeki vekilin (intelligent agent) , çalıştığı ortam ile iletişimini sağladığı ve bu iletişim üzerinde karar verdiği bir mekanizması vardır denilebilir.



Şekil 1: Zeki vekil

Yukarıdaki temsili resimde, bir zeki vekil, çizgili alanda gösterilen üç ana unsurdan ibarettir. Bunlar kısaca vekilin çalıştığı ortamı gözlediği algı (perception), vekilin bu ortamda bir işlem yapmasını sağlayan eylem (action) ve vekilin bu ortamdaki algısına göre nasıl bir eylem yapacağına karar vermesini sağlayan karar mekanizmasıdır. Bu karar mekanizması çoğu zaman bir kurallar listesi (rule base) olabileceği gibi bazı durumlarda basit bir if – else bloğu da olabilmektedir.

Russel ve Norving tarafından 2003 yılında yayınlanan yapay zeka kitabında, zeki vekiller 5 seviyede listelenmiştir. Bu seviyeleri basitten karmaşığa doğru aşağıdaki şekilde sıralayabiliriz:

1. Fiil-i Münakıs Vekiller (reflex agents)
2. Fiil-i Münakıs Kalıp Vekiller (Model-based reflex agents)
3. Hedef vekilleri ( goal-based agents)
4. Fayda vekilleri ( utility-based agents)
5. Öğrenen vekiller ( learning agents)

Reflex Agents

Basit bir koşul ve eylem sıralamasından ibaret olan vekiller. Belirlenen koşul gerçekleşince yine daha önceden belirlenen fiili yerine getirir. Kurulu bir düzenek olarak düşünülebilir. Örneğin fare kapanı, bir insan için fareyi yakalayan bir vekildir ve farenin peyniri yemesiyle birlikte fareyi yakalar. Buradaki peynir yenmesi koşul ve farenin yakalanması fiil olarak düşünülebilir. Bazı refleks ajanlarında durum takibi de yapılabilir. Örneğin fare kapanı misalinde olduğu gibi kapanın kurulu olma durumu, kapanın fareyi yakalamış olma durumu gibi durumlar ayrı ayrı tahlil edilebilir.

Model Bazlı Refleks etkenler

Bu tip ajanlarda (etkenlerde) ise içinde çalışılan ortam modellenir. Yani ajan kendi yapısına göre ortamı anlamaya ve bir modelini kendi hafızasında tutmaya çalışır. Bu ajanlar modeldeki durumlara göre davranış sergilerler. Yani bir önceki tipte olan refleks ajanlarının ortamdan aldıkları doğrudan koşullarından farklı olarak bu ajanların modellerinde bazı refleksler tanımlıdır.

Hedef Güdümlü Vekiller

Bu vekiller ise belirli bir hedefe ulaşmak için bir dizi şart-fiil gerçekleştirirler. Basit bir durum-geçiş diyagramı (state transition diagram) olarak düşünülebilecek yapılarına göre, ortamı algılayarak mevcut yapılarındaki bir duruma benzetir ve bu durumu ulaşmak istedikleri hedefe en uygun şekilde eylemlerle değiştirmeye çalışır.

Çıkar Amaçlı Vekiller

Hedef amaçlı vekillerden farklı olarak, durumlar arasındaki geçişin oransal olması durumudur. Yani hedef güdümlü ajanlarda bir durumdan her zaman diğer duruma geçiş hedeflenir. Çıkar amaçlı vekillerde bundan farklı olarak oransal bir fonksiyon kullanılması söz konusudur. Bu fonksiyona çıkar fonksiyonu ( fayda fonksiyonu, utility function) ismi verilir.

Öğrenen Etkenler

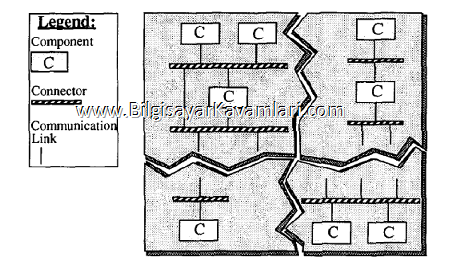
Bu etken tipinde, ortamda yapılan bazı eylemlerin beklenen sonuca nasıl hizmet ettiğine göre yeni kurallar tanımlanır. Ajanın çalıştığı ortamın bilinmemesi halinde kullanılışlıdırlar. Kendi kurallarını ve durum makinelerini oluşturabilir veya değiştirebilirler.

## **SORU-9: C2 Üslûbu (C2 Style) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde özellikle yazılım mühendisliği (Software engineering) konusunda kullanılan ve bileşen (component) ve mesaj (message) temelli yazılım geliştirmeyi amaçlayan bir üsluptur.

C2 tasarım kültüründe yazılım bileşenler şeklinde ele alınır ve yazılımı oluşturan bu bileşenler üzerinden bir ağ (network) çizilir. Bu ağda bileşenler arası haberleşme gösterilir. Bu sayede tasarlanan bir bileşenin daha verimli ve tekrar tekrar kullanılması mümkün olmuş olur.

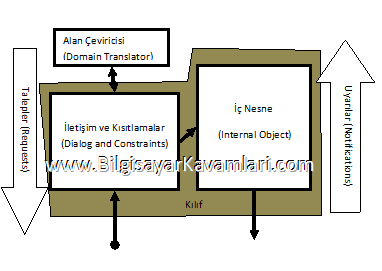
1996 yılında bu yaklaşımı ilk kez ortaya koyan Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, ve Richard N. Taylor, “Using Object-Oriented Typing to Support Architectural Design in the C2 Style (C2 stili ile nesne yönelimli ve destek mimarili tasarım)” isimli makalelerinde bu nesneler arası ilişkiyi aşağıdaki şekille temsil etmişlerdir:



Şekil 1: Nesneler arası ilişki

Yukarıdaki şekilde görüleceği üzere her bileşen C harfi ile temsil edilmiş ve bağlar birer çizgiyle gösterilmiştir. Ayrıca kalın çizgiler ana bağlantı noktalarını göstermektedir.

Yine aynı makalede bulunan bir C2 bileşeninin iç yapısını aşağıdaki şekilde Türkçe terminoloji ile gösterebiliriz:



Şekil 2: C2 bileşeninin iç yapısı

Yukarıdaki şekilde anlatılmak istenen bir nesnenin bir kılıf içerisinde ele alındığıdır. Yani C2 tasarımında bileşen olarak görülen herşey aslında nesne yönelimli programlamada bir nesne(object) ile karşılanır ancak bu nesneye bir kılıf giydirilmiş olarak kabul edilebilir. Nesne ile iletişim ayrıca bir arayüz ile yaıpılırken nesnenin dışarıya doğrudan erişimi bulunmaktadır. Bu durum kılıftan çıkan ve talepler (request) yönünde olan bir ok ile temsil edilmiştir.

Doğrudan kılıf (wrapper) dışına erişim yapabilen nesnenin yanında bütün uyarılar (notifications) iletişim ve kısıtlama nesnesi üzerinden iç nesneye erişebilmektedir. Aslında tasarım olaran nesne yönelimli programlamanın kapsülleme (encapsulation) mantığına oldukça yakın olan bu durumda, ayrıca nesnelerin hepsinin kendi adreslerinde çalıştıklarını düşünebiliriz. Kendi adres alanlarında (own address space) çalışan nesneler değişken paylaşımı (Shared variable) korunmuş olacağı için, çok kullanıcılı (multiuser), çok işlemli (multiprocessing/threading) ve eşzamanlı (concurrent) modellemeleri desteklemektedir.

## **SORU-10: JML ( Java Modelleme Dili) hakkında bilgi veriniz.**

JML ingilizce Java modelling language kelimelerinin baş harflerinden oluşan bir kısaltmadır. Basitçe bir java kaynak koduna eklenen ilave satırlar ile progam doğruluğunu (program correctness) sağlamayı amaçlar (program verification).

İlave olarak eklenecek satırlar java kodunun içerisine yorum satırı gibi ilave edilir. Normal java yorum satırlarından tek farkı ilave olarak konulan @ işaretidir.

Örneğin

//@ örnek jml satırı

veya

/\*@ örnek

çoklu jml

satırı @\*/

şeklinde java koduna ilave edilebilirler.

Aslında bütün jml komutlarını basit üç grupta toplamak mümkündür:

1. Önkoşullar (preconditions)
2. Sonkoşullar (postconditions)
3. Değişmezler (invariants)

Yukarıdaki bu üç grubtan ilk ikisi hoare mantığındaki ön ve son koşullara benzetilebilir. Üçüncü grup ise döngülere özgü kullanılan bir koşuldur. Bu gruplardaki komutları ve kullanımlarını aşağıdaki örneklerden inceleyelim:

static void arraycopy (int[] src, int srcPos,int[] dest, int destPos,int len);

Örnek olarak yukarıdaki [dizi (array)](http://www.bilgisayarkavramlari.com/2007/05/04/array-dizi/) kopyalama fonksiyonunu ele alalım. Burada amaç, src dizisinden dest dizisine derin kopyalama yapmaktır. (deep copy). Ancak bu satırda yaşanabilecek bazı programlar aşağıdaki şekildedir:

* src dizisi null olabilir
* dest dizisi null olabilir
* src dizisinin izin verilen eleman sayısının dışına programcı tarafından çıkılabilir (örneğin 10 elemanlı bir dizinin 10. elemanına erişilebilir (java terminolojisi olarak arrayindexoutofboundsexception olabilir))
* dest dizisinin izin verilen eleman sayısının dışına programcı tarafından çıkılabilir.

Yukarıdaki bu olası problemleri çözmek için fonksiyonu yazarken başına ilave olarak bu requires komutu ile ön koşullarımızı ekliyoruz.

/\*@ requires src != null && dest != null &&

0 <= srcPos && srcPos + len < src.length &&

0 <= destPos && srcPos + len < dest.length;

@\*/

static void arraycopy (int[] src, int srcPos,int[] dest, int destPos,int len);

Yukarıdaki yeni kodda ilave olarak src ve dest’in null olamayacakları ve kopyalama sırasında kullanılan indis değişkeni olan srcPos’un 0 ile src.length ve dest.length arasında olacağı belirtilmiştir.

Yukarıdaki ön koşula ilave olarak diğer bir isteğimiz fonksiyon çalıştıktan sonra src ile dest dizilerinin elemanlarının birebir aynı olmalarıdır. Bu son koşulu (postcondition) kontrol etmek için ilave bir jml satırı daha eklememiz gerekir:

/\*@ requires src != null && dest != null &&  
0 <= srcPos && srcPos + len < src.length &&  
0 <= destPos && srcPos + len < dest.length;  
ensures (forall int i; 0 <= i && i < len;dest[dstPos+i] == src[srcPos+i] )  
@\*/  
static void arraycopy (int[] src, int srcPos,int[] dest, int destPos,int len);

Yukarıdaki yeni jml satırı olan ensures ile son koşul kontrolü yapılıyor.  Burada yeni kullanılan forall döngüsü sayesinde 0′dan len değişkenine kadar olan değerler için (döngü değişkeni i olarak) bütün dizi içeriği karşılaştırılıyor ve ancak aynıysa son koşul sağlanmış oluyor.

## **SORU-11: Program doğruluğu (Program correctness) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde bir programın istenen özellikleri yerine getirip getirememesine verilen isimdir. Buna göre şayet bir program, beklenen özellikleri tam ve eksiksiz yerine getiriyor, istenmeyen sonuçlar ortaya çıkmıyor ve program başladıktan sonra her durumda başarılı bir şekilde bitiyorsa bu programa tam doğru ( total correctness) ismi verilir.

Durma probeleminden (halting problem) bilindiği üzere bir programın bitip bitmemesinin test edilmesi ayrı bir muammadır. Dolayısıyla tam doğru bir programın elde edilmesi veya ispatlanması farklı güçlükleri beraberinde getirir. Şayet bir program, kendinden beklenen özellikleri başarılı bir şekilde yerine getiriyor ancak bitip bitmemeyi garanti edemiyorsa (bir programın biteceği ispatlanamıyorsa) bu durumda programa kısmi doğru (partial correctness) ismi verilir.

Aslında programların doğruluğu (correctness), karar problemlerinin (decision problems) çözülmesidir.

Tam bu noktada bilgisayar bilimleri konusunda veciz sözleri ile konuya aydınlık getiren Dijkstranın bir sözünü hatırlamakta yarar var. “Testler hataların varlığını gösterebilir ama yokluğunu gösteremez.”

Örneğin doğal sayılar kümesi üzerinde sayıların asal olup olmadığını test eden bir program yazdığımızı düşünelim. Bu program basitçe bir sayıyı alacak ve sayının çarpanlarını bulacaktır. Şayet sayının kendisi ve 1 dışında başka çarpanı yoksa bu sayıyı asal sayı ilan edecektir. Bu şekilde bir program yazmamızda herhangi bir sorun yoktur ancak programın biteceğini garanti edemeyiz. Elbette buradaki problem doğal sayıların bir sonu olup olmadığıdır. Yani program bütün asal doğal sayıları bulmaya çalışırsa sonsuz sayıda işlem yapması gerekir (ya da sonsuz bilinen bir değerse sonlu sayıda). Bu noktada bilgisayarın hafıza gibi kısıtları devreye girer. Yani sonsuz sayı işleyecek bir hafıza kapasitesi henüz bilgisayarlarda bulunmamaktadır.

Bu ispat sırasında programın bir lambda cebirine (lambda calculus) çevrilmesi ve bu cebirde doğruluğunun ispatlanması mümkündür. Bu tip ispat yöntemlerine program çıkarımı (program extraction) ismi verilir. Örneğin Curry Howard karşılığı (curry howard correspondence) yaklaşımı buna bir örnek olarak gösterilebilir.

Bir programın doğruluğunun tetkikine, program tetkiki (program verification) ismi verilir.

**Program doğrulamanın kullanıldığı yerler**

Program doğrulama, bir programın geliştirilme sürecinde, son adım olan test aşamasına gelinmeden hataların bulunması ve programın kodlanma hatta tasarım aşamasında hatalarının yakalanmasını sağladığı için oldukça önemlidir. Örneğin bitmiş bir programdaki bir hatanın test ile bulunması bütün tasarım ve kodlama adımlarına geri dönülmesini ve dolayısıyla hatanın telafisi için harcanan maliyetin yüksek olmasını sağlarken, daha henüz kodlama aşamsında hatanın tespiti maliyeti düşürmektedir.

Ayrıca yazılan programın hatasının test aşamasında hiçbir zaman bulunamaması da söz konusu olabilir. Bu ihtimale karşı programın matematiksel olarak doğruluğunun ispatlanması çok önemlidir.

## **SORU-12: Kara Kutu Yaklaşımı (Black Box) hakkında bilgi veriniz.**

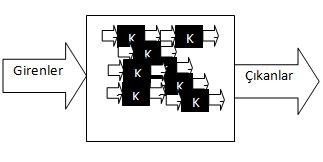
Çok klasik bir mühendislik yaklaşımıdır. Bu yaklaşım sistemi analiz ederken sistemin içeriğin görmezden gelerek sisteme girenler ve çıkanların analiz edilmesini söyler.

[](http://www.bilgisayarkavramlari.com/wp-content/uploads/2009/08/karakutu.jpg)

Şekil 1: Sisteme giren ve çıkanlar

Yukarıdaki temsili resimde gösterildiği üzere, tahlil edilen sistem bir kara kutuya benzetilerek sisteme giren ve çıkanlar incelenir.

Bu anlamda yazılım mühendisliğinde sıklıkla kullanılan kara kutu yaklaşımında sistem önce büyük bir kara kutu olarak düşünülüp ardından alt kutulara bölünebilir:

[](http://www.bilgisayarkavramlari.com/wp-content/uploads/2009/08/karakutu2.jpg)

Şekil 2: Kutuların incelenmesi

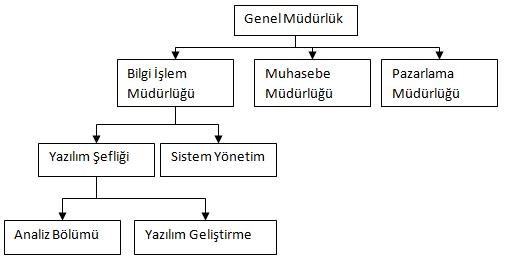
Sistemde bulunan her kutu kendi başına incelenip ardından sistemin tamamının incelenmesi mümkündür. Buradaki her kutuyu ayrı bir sistem gibi düşünmek ve ardından bu sistemlerin birbiri ile olan bağlantısını düşünmek mümkündür.

Örneğin yazılım testlerinde her alt sistem ayrı ayrı test edilip (unity test) ardından bu alt sistmlerin birbiri ile olan uyumu ayrıca test edilebilir (integrity test).

Yazılım mühendisliğinde ayrıca giriş işlem çıkış (input process output) kelimelerinin baş harflerinden oluşan IPO terimi de tam bir kara kutu (black box) yaklaşımıdır.

IPO modellemesinin biraz daha gelişmiş hali olarak kabul edilebilecek olan HIPO (hierarchy input process output veya bazı kaynaklarda hierarchy plus input process output yada hierearchical input process output olarak da geçmektedir) yaklaşımında ise sisteme hiyerarşik bir yapı kazandırılmaktadır. Yani sistemde bulunan kara kutular arasında hiyerarşik bir yapı oluşturulabilir.

Aşağıda bir hiyerarşi grafiği örneği ile konuyu açıklamaya çalışalım:

[](http://www.bilgisayarkavramlari.com/wp-content/uploads/2009/08/hiyerarsi.jpg)

Şekil 3: Varlıklar

Yukarıdaki hiyerarşi grafiğinde çeşitli hiyerarşik seviyedeki varlıklar modellenmiştir. Buna göre örneğin en altta bulunan analiz bölümü genel müdürlük altında bir bölümdür ancak arada iki seviye bulunmaktadır.

Yazılım modellemsinde bu hiyerarşik yapılanmadan faydalanılabilir. Örneğin bir yazılımı alt sistemlere bölerken seviye seviye bölümlemeye gidilebilir.

Bu durumu internette rast gele bulduğum bir yemek tarifi üzerinden anlatmaya çalışayım.  Yemektarifi.com sitesinde Ispanaklı kiş isminde bir yemek aşağıdaki şekilde anlatılmış:

Malzemeler:

1 Kg ıspanak  
500 Gr mantar  
200 Gr dilimlenmiş kaşar peyniri  
500 Gr rendelenmiş havuç  
2 Çay kaşığı tuz  
2 Çorba kaşığı kekik  
5 Yaprak Miflöy hamuru  
2 Çorba kaşığı süt  
1 Yumurta  
Sos İçin:  
2 Su bardağı süt  
4 Çorba kaşığı un  
2 Soğan  
4 Kahve fincanı margarin

Hazırlanışı:

* Ispanakları yıkayıp az haşlayın.
* Kevgirle süzüp bekletin.
* Mantarı ince ince doğrayarak az haşlayın.
* Haşladığınız ıspanakları doğrayıp mantarla karıştırın.
* Tuz ve kekik ilave edip bekletin.
* Sosu hazırlamak için soğanı ince kıyın.
* Tavada margarini eritip soğanları pembeleşinceye kadar kavurun.
* Unu ekleyip tekrar kavurduktan sonra sütü ekleyin ve muhallebi kıvamına gelinceye kadar pişirin.
* Hazırladığınız sosa ıspanak ve mantar karışımını ekleyip karıştırın.
* Miflöy hamurlarını hafif açın ve tepsisine yayarak tamamını kaplayın.
* Üzerine hazırladığınız ıspanak karışımının yarısını ilave edin.
* Üzerine havuç ve kaşar dilimlerini yerleştirin.
* Kalan ıspanak karışımını da ekledikten sonra 200 dereceye ayarlı fırında 30 dakika pişirin.
* Bir kapta 2 çorba kaşığı sütü ve 1 yumurtayı çırpın.
* Ispanaklı kişin pişmesine 5 dakika kala fırından alıp üzerlerine süt ve 5 dakika daha pişirip sıcak olarak servis yapın.

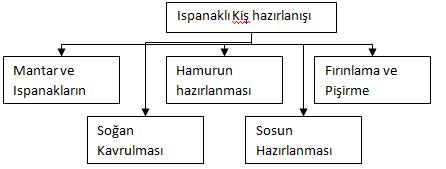
Şimdi yukarıdaki açıklamayı modellemek istersek  aslında bir kara kutu yaklaşımı ile bu işlemi (Ispanaklı kiş hazırlama işlemini) aşağıdaki şekilde modelleyebiliriz:

[](http://www.bilgisayarkavramlari.com/wp-content/uploads/2009/08/ispanak.jpg)

Şekil 4: Yemeği yapılma aşamaları

Yukarıdaki kara kutu yaklaşımında sistemi “Ispanaklı Kiş hazırlanışı” olarak görmek ve sisteme girenler olarak yemek tarifimizdeki malzelemeleri ve çıkan olarak da Ispanaklı Kiş görmek doğru bir yaklaşım olur.

Bu sistemi hiyerarşik olarak bölecek olursak alt kademeeleri bulunmaktadır. Örneğin sos hazırlanması, ıspanakların hazırlanması, soğanların kavrulması gibi alt işlemler tanımlanabilir.

[](http://www.bilgisayarkavramlari.com/wp-content/uploads/2009/08/ispanakhiyerarsi.jpg)

Şekil 5: Aşamalar

Yukardaki hiyerarşik ilişki içerisindeki her kutu aslında bir kara kutu olarak düşünülebilir. Örneğin Mantar ve Ispanağın hazırlanması aşamasında

* Ispanakları yıkayıp az haşlayın.
* Kevgirle süzüp bekletin.
* Mantarı ince ince doğrayarak az haşlayın.
* Haşladığınız ıspanakları doğrayıp mantarla karıştırın.
* Tuz ve kekik ilave edip bekletin.

İşlemleri yapılmaktadır. Dolayısıyla girenler : “ıspanak, mantar, tuz ve kekik” olarak sıralanabilir. Ve bu işlemde örneğin Ispanakların yıkanması, Mantarların doğranması, Ispanak ve mantarların karıştırılması, Tuz ve kekik ilavesi gibi alt kutulara (işlemlere) bölünebilir.

Sonuçta bütün bu işlemleri ve bu işlemlerin seviye seviye gösterildiği analiz yöntemine hipo ismi verilmektedir.

## **SORU-13: Nokta Kesiciler (PointCuts) hakkında bilgi veriniz.**

Bağlam yönelimli programlama (Aspect oriented programming) konusunda kullanılan en önemli yapı taşlarındandır. Basitçe bir programın akışı içerisinde bağalam noktaları (joinpoints) belirlendikten sonra bu noktalara hangi bağlamların (Aspects) bağlanacağını belirleyen kümedir.

Yani nesne yönelimli bir ortamda yazılan programımız üzerine örmek istediğimiz (aspect weaving) bağlamları (aspects) kod üzerindeki birleşim noktaları (joinpoints) ile belirliyoruz. Belirlenen bu noktaları tutan ve hangi noktaya hangi bağlamın bağlanacağını belirleyen kümeye ise noktakesici (pointcut) ismi verilir.

Yapı olarak nokta kesicileri (pointcuts) iş mantığına daha yakın görmek mümkündür. Yani mantıksal bir tasarım sırasında bir bağlamı belirten bir nokta kesici bulunur ve bu nokta kesici birden fazla birleşim noktasından oluşabilir.

Örneğin program geliştirme sırasında sıkça kullanılan işlemlerden birisi kullanıcıdan bir form doldurmasını istemek ve doldurulan bu formu veri tabanına kaydetmektir. Bu eylem başlı başına bir bağlamdır (Aspect) ve bu olayı meydana getiren birleşme noktaları bir nokta kesici (pointcut) belirler. Örneğin formdaki her alan, bu alanların doğrulanması veya veritabanına kayıt yapan metotlar birer birleşim noktası (join point) olabilir.

## **SORU-14: Birleşim Noktaları (JoinPoints) hakkında bilgi veriniz.**

Bağlam yönelimli (aspect oriented) programlamada kullanılan terimlerdendir. Anlam olarak programda bulunan bir kontrol noktasının bir veya daha fazla bağlamın etkisinde olduğu nokta demektir.

Bu anlamda bir kodda bulunan bütün satırlar birleşim noktası (joinpoint) olabilir ancak bu yaklaşım bağlam yönelimli programlama açısından doğru olmaz. Daha doğrusu bütün olası birleşme noktalarından en mantıklı ve optimum noktaları almaktır. Ayrıca bilinmelidir ki aynı sonucu veren iki farklı birleşim noktası kümesi almak mümkündür.

**Birleşim noktalarının çeşitleri**

Bağlam yönelimli programlama yaklaşımı yapı olarak nesne yönelimi programlamanın üzerine kurulu olduğu için birleşme noktaları da nesne yönelimli programlama öğelerinden oluşacaktır. Örneğin nesne yönelimli programlamada kullanılan metotlar, yapıcılar (constructors), istisnalar (exceptions) veya özellikler aday birer birleşim noktasıdır.

Örneğin metotların birer birleşim noktası olarak görülmesi aslında metotların bir programın çalışma akışının parçası olmasından kaynaklanmaktadır. Yani bir programı belirli sayıdaki metotun belirli bir sırayla çağrılması olarak düşünmek mümkündür. Bu çalışma sırasında bir metot başlar çalışır ve biter. Dolayısıyla metotların başlaması, çalışması veya bitmesi birer birleşim noktası (join point) olabileceği gibi metotun tamamı da bir birleşim noktası olabilir.

Benzer şekilde bir istisna (exception), üretilerek fırlatılan (throw) bir yapıdadır. Dolayısyla bir istisnanın fırlatıldığı veya yakalandığı (catch) noktların ikisi de sık kullanılan birleşme noktlarıdır.

## **SORU-15: Bağlam Örücüler (Apect Weavers) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde yazılım geliştirme sürecinde kullanılan yaklaşımlardan birisi de bağlam yönelimli programlamadır (Aspect oriented programming). Bu yaklaşım, yazılım geliştirme yaklaşımları sıralasında nesne yönelimli programlamadan sonra gelmektedir. Yani bir anlamda her bağlam yönelimli programlama yaklaşımı aslında bir nesne yönelimli programlama yaklaşımını kapsar.

Nesne yönelimli programlama yaklaşımından farklı olarak bağlam yönelimli programlama yaklaşımında ilave bağlamlar tutulur. Yani yazılımın modellenmesinde mevcut nesne yönelimli yaklaşım izlenir ancak bunun üzerine yazılımın kod yayılımı (code scattering) ve kestirme fonksiyonellikler (crosscut functions) gibi özelliklerden dolayı ilave olarak bağlamlar (aspects) tutulur.

Bu bağlamların yazılım ile bütünleştirilmesine bağlam örmek (aspect weaving) ismi verilir ve bağlam ile veriler modellemede farklı dururken çalışma sırasında bağlam örülmü (aspect woven) olarak dururlar.

**Derleme zamanı örgü (Compile time weaving)**

Yazılım modellemesinde kullanılan sınıflar (class) ile bağlamların (aspects) birleştirilemsi işleminin derleme sırasında olması durumudur. Örneğin java üzerinde çalışan aspectj bu gruba girmektedir.

Derleme zamanı örgü de kendi içinde iki gruba ayrılır. Genelde C# ve JAVA gibi nesne yönelimli programlamanın öncü dilleri ara bir kod üretirler (byte code) bu kod kullanılan çerçeve (framework) üzerinde çalışır ve platform bağımsızlığını (platform independence) sağlayan da bu ara kodun farklı ortamlarda çalışmasını sağlayacak ara katmanlar bulunmasıdır. Örneğin .net veya jvm (java virtual machine, java sanal makinesi) gibi.

Dolayısıyla derleme zamanı örgü seçenekleri bu ara kodun (byte code) üretilmesinden önce kaynak kod (source code) üzerinden çalışanlar ve byte kod üzerinden çalışanalr olarak ikye ayrılabilirler.

Byte kod üzerinde işlem yapmak genelde daha yaygındır. Bunun birinci sebebi kolay olmasıdır.  Örneğin java dili, üretilen byte koddan çok daha karmaşıktır. Diğer bir sebep ise bağlam örücülerin (aspect weavers) kaynak koda erişmesinin engellenmesidir. Yani örneğin java dilinde kodlanan bir uygulamanın kaynak kodlarına üçüncü parti bir bağlam örücünün erişmesi genelde istenmez. Son olarak byte kod işlemek kaynak kod işlemeye göre daha hızlıdır (basit olması da bir etken olduğundan) dolayısıyla performans olarak byte kod üzerinden bağlam örmek daha uygundur.

Derleme zamanı örgü sonucunda üretilen kod sabit bir şekilde bağlamları içermektedir. Yani herhangi bir bağlamın değişme şansı yoktur veya değişmesi istenirse yeniden derleme süreci gerekmektedir.

**Çalışma zamanı örgü (Run-time weaving)**

Derleme zamanından farklı olarak derlenmiş ve çalıştırılmaya başlamış uygulamalar üzerinde örülen bağlamlardır. Bu tip bağlam örme işlemlerinde uygulamanın üretilmiş olan byte kodu ortamda çalıştırılırken uygulama geliştirme sürecinde üretilen bağlamlar uygulamaya örülür.

Çalışma zamanı örgünün en önemli avantajı, uygulamaya müteharrik (dinamik) bir şekilde bağlam eklenebilmesidir. Örneğin web sunucuları veya çoklu ortam sunucuları (ses, video gibi) gibi sürekli erişim altında olan suncular için bu özellik oldukça önemlidir.

## **SORU-16: Veri Bütünlüğü (Data Integrity) hakkında bilgi veriniz.**

Bilgisayar bilimlerinin birinci derece uğraştığı varlık [veridir](http://www.bilgisayarkavramlari.com/2008/03/26/bilgiverimalumatirfanknowledgedatainformationwisdom/). Yani bilgisayar bilimlerinde yapılan her işi bir bakışa göre veriyi işlemek olarak görmek mümkündür. İşlenen bu verinin bütünlüğü ise ayrı bir problemdir. Veri bütünlüğü ile genelde verinin birden fazla parçaya bölünmesi durumunda bu parçaların bütün olarak tutulması kastedilir.

**Veri’nin bölünmesi ne zaman gerekir**

Günümüzde kullanılan ilişkisel veri tabanlarında veya nesne yönelimli programlama yaklaşımlarında bir [bilgi](http://www.bilgisayarkavramlari.com/2008/03/26/bilgiverimalumatirfanknowledgedatainformationwisdom/) birden fazla parçaya bölünmektedir. Örneğin gerçek hayattaki bir müşteri sipariş formunu (kağıt ile tutulan) ele alalım. Mu formun üzerinde muhtemelen müşterinin iletişim bilgilerini (adres, telefon gibi) istediği ürün bilgilerini (örneğin kitap alacaksa, kitabın ismi fiyatı gibi) ve sipariş bilgilerini (örneğin sipariş tarihi toplam ücreti gibi) görebiliriz.

Ancak ilişkisel veri tabanında (relational database) bu bilgilerin tek bir yerde tutulması teorik olarak doğru kabul edilmez. Normalleştirme (normalisation) teorilerine gore verinin parçalanması (docomposition) gerekmektedir.

Görüldüğü üzere bir veri tabanı modellemesi sırasında veri parçalara bölünmektedir.

Benzer durum nesne yönelimli yaklaşımda da yaşanabilir. Yani veriyi nesneler ile modellemek isteyelim. Her nesne kendisi ile ilgili bilgiyi tutacaktır. Örneğin yukarıdaki örnek için, müşteri ve kitap gibi iki farklı nesne bulunacak ve veri bu nesneler arasında dağıtılacaktır.

**Veri bütünlüğünü tehdit eden durumlar**

Veri bütünlüğünü tehdit eden birinci durum silme işlemleridir. Yani veriyi modelimiz içinde parçalara böldüğümüzü ve her parçayı başarılı bir şekilde tuttuğumuz düşünelim. Bu bütünün bir parçasını müstakil olarak silersek veri bütünlüğünu tehdit ederiz.

Örneğin yukarıdaki misale devam edelim ve sipariş formundaki bilgileri veri tabanımızdaki üç ayrı tabloya:

* müşteri
* sipariş
* kitap

tablolarına dağıtalım.

Şimdi gidip bir müşteriyi sildiğimizde artık elimizdeki bazı siparişlerin kime ait olduğunu bilmiyoruz. Ya da bir kitabı sildiğimizde bazı müşterilerin hangi kitabı aldığını bilmiyoruz.  İşte bu durum veri bütünlüğünün bozulmasıdır.

Veri bütünlüğü silme işlemi dışında yanlış tasarımdan da bozulabilir. Tasarım hataları verinin tutulması ve işlenmesi sırasında ortaya çıkabilir ancak sonuçta anlatılmak istenen verinin bir kısmına ulaşılamaz olmasıdır.

## **SORU-17: Bağlam Yönelimli Programlama (Aspect Oriented Programming) hakkında bilgi veriniz.**

Yazılım mühendisliğinde kullanılan bir programlama yaklaşımıdır. AOP olarak kısaltılmış halde de geçer. Türkçe literatüründe bağlam / cephe / kesit / görünüm yönelimli programlama kelimelerinin hepsi farklı kaynaklarda kullanılmıştır. 1990′lı yılların ortalarında özellikle JAVA ve nesne yönelimli programlama ihe ivme kazanmış bir yaklaşımdır.

Kısaca aop’yi tanımlamak gerekirse bir program geliştirilmesi sırasında iki farklı yaklaşım görülmektedir. Birinci tip yaklaşım fonksiyonel ve programın yapacağı işlere ait yaklaşımdır. Yani programı yazdığımızda ve program çalıştığında karşılayacağı ihtiyaçlar, yapacağı işlemler şeklindeki ve iş mantığı (business logic) olarak isimlendirilebilecek bakıştır. İkinci bakış ise teknik bakıştır (technicak concerns) yani programın çalıştığı ortam, işletim sistemi, veri tabanı gibi unsurlardır.

AOP kısaca bu iki unsurlar kümesinin birbirinden ayrılmasını ve bağlam adı altında modülerleştirilmesini söyler. Yani fonksiyonel unsurlar ile teknik unsurların birbirinden ayrılmasını daha sonra modüller kapsamında bu iki unsurun birleştirilmesini söyler.

**AOP, OOP’nin yerini alabilir mi?**

AOP yaklaşımı [nesne yönelimli programlama dilleri (Object Oriented Programming, OOP)](http://www.bilgisayarkavramlari.com/category/nesne-yonelimli-programlama/) üzerinde geliştirilmektedir. Bu anlamda AOP ile OOP birbirini tam olarak ikame eden yaklaşımlar değildir. Ancak AOP yaklaşımı doğası gereği ortam olarak OOP destekleyen bir dile ihtiyaç duyar. Yani yaklaşım olarak bağlam yönelimli programlama (AOP) kullanılıyor olsa bile hâlâ [sınıflar (class)](http://www.bilgisayarkavramlari.com/2008/07/15/sinif-class/) yazmaya ve bu sınıfların metotlarını kodlamaya ihtiyaç vardır. AOP, bu [sınıfların](http://www.bilgisayarkavramlari.com/2008/07/15/sinif-class/) metotların veya özelliklerin sadece hangi bakış açısıyla modelleneceğini belirler. Yani bir proje tasarımı yapılırken sınıfların ve nesne yönelimli kodlama yaklaşımının uygulanması sırasında bakış yönelimli tercihler yapılabilir veya yapılmayabilir.

**AOP’ye ne zaman ihtiyaç vardır?**

Bilindiği üzere normalde bir program geliştirilme sürecinde OOP kullanılması pek çok açıdan avantajldır ve OOP ile pek çok farklı alanda ve farklı içerikte proje geliştirilmesi mümkündür. Ancak AOP’nin farkını anlamak ve OOP’nin limitlerini görmek açısından iki noktada OOP’nin yetersiz olduğunu ve AOP’ye ihtiyaç duyduğumuzu söyleyebiliriz.

* Kod yayılması (Code scattering)
* Kestirme Fonksiyonellikler (Crosscutting Functionalities)

Yukarıdaki bu iki durumda klasik OOP yaklaşımının AOP yaklaşımına göre geri olduğu söylenebilir.

**Kod yayılımı (Code Scattering)**

Bilindiği üzere nesne yönelimli programlama dillerinde nesneler arası ilişkiyi iki türlü ele alabiliriz.

* Çağıran nesneler (invoke)
* Çağrılan nesneler (invoked)

Yani bir nesne ile diğer bir nesne arasında bir protokol bulunuyorsa, bir nesne diğerinin bir fonksiyonunu (method) çağırıyor demektir. Bu durumda nesne yönelimli programlama yaklaşımı bize bu çağrılan nesne veya methodu hakkında kaygılanmamızı ve [kapalı bir kutu gibi görüp (blackbox)](http://www.bilgisayarkavramlari.com/2009/08/29/kara-kutu-yaklasimi-black-box/) beklediğimiz hizmeti almamızı söyler.

Ancak herhangi bir sebeple bu aradaki çağırma işlemini yapan ve protokolümüzün üzerine kurulu olduğu fonksiyonu değiştirecek olursak. Örneğin yeni bir parametre ekleyecek olursak. Bu durudma bütün çağıran nesnelerin de çağırma sırasındaki fonksiyonlarının değişmesi gerekir.

Yani bir metot’un paramtre sayısının değişmesi gibi bir durumda bütün bu metotu çağıran [sınıflara](http://www.bilgisayarkavramlari.com/2008/07/15/sinif-class/) müdahale edilmesi aslında geliştirme ve sonrada kodun yönetilmesi açısından güçlükler doğurur ve bu duruma kod yayılması (code scattering) ismi verilir

**Kestirme Fonksiyonellikler (Crosscutting Functionalities)**

Kestirme fonksiyonellik kısaca nesne yönelimli yaklaşımlarda bir nesnenin modellenmesi sırasında nesneye yüklenen fonksiyonelliğin birden fazla nesne tarafından paylaşılması durumudur. Yani A nesnesinin F fonksiyonunu yüklenmesini istiyorsak ancak bu F fonksiyonunu B,C gibi diğer nesneler de yükleniyorsa veya kontrol ediyorsa bu durumda aslında A nesnesinin tam modellendiği söylenemez. Her ne kadar OOP açısından tasarlamak ve kodlamak mümkün olsa da AOP bu modele izin vermez.

Konuyu net bir örnek üzerinden anlamaya çalışalım. Klasik müşteri – sipariş örneğini düşünelim. Örneğin müşterilerimizi ve siparişleri tutan iki ayrı sınıf modellediğimizi kabul edelim. [Veri bütünlüğü (data integrity)](http://www.bilgisayarkavramlari.com/2009/07/15/veri-butunlugu-data-integrity/) problemi olarak bilindiği üzere bir müşterinin silinmesi ancak hiç siparişi olmayınca mümkündür. Yani şayet bir müşterinin bekleyen bir siparişi varsa ve biz müşteriyi silersek, siparişi gerçekleştiremeyiz çünkü müşterinin iletişim bilgilerini kaybederiz.

Bu durumda aslında müşteri silme fonksiyonu, müşteri sınıfının bir üyesiyken sipariş sınıfının kontrol etmesi gereken bir durum oluşur. Yani müşterinin silme fonksiyonu, sipariş olmadığında çalışabilmektedir.

Bu durum üç sonuç doğurmaktadır:

1. Müşteri sınıfı siparişin olup olmadığını kontrol işlemini gerçekleştirememektedir ve tasarım itibariyle kendi fonksiyonunu kendi başına yapamaz. Bu bir tasarım zaafıdır.
2. Müşteri sınıfı benzer şekillerde diğer veri bütünlüğü (data integrity) kurallarını da bilemek zoruna kalır. Bu da programlama açısından güçlükler doğurur.
3. Müşteri sınıfının yeniden kullanılabilirliği (code reusability) kaybedilir çünkü artık müşteri sınıfı tek başına vâr olamaz. Varlığı sipariş sınıfına bağlıdır. Yani aynı sınıf başka bir projede kullanıldığında sipariş sınıfının da kullanılması gerekir.

Yukarıdaki problemlere çözüm olarak bu veri bütünlüğü kontrolünün sipariş sınıfına konması da birşey değiştirmez çünkü bu durumda aynı problemler sipariş sınıfında tezahür eder.

AOP’nin sağladıkları

AOP yazılım geliştirme yaklaşımlarında son nesil olarak kabul edilebilir. Daha önceki fonksiyonel programlama yaklaşımlarında problem fonksiyonel elemanlara bölünerek her eleman için bir fonksiyon yazılması yoluna gidilmekteydi. Nesne yönelimli yaklaşım biraz daha ileri giderek problemi veri anlamında bütünlük arz eden nesnelerle modelledi. Son olarak bağlam yönelimli programlama (aop) ise bir adım daha ileri gidildi.

## **SORU-18: MVC (Model View Controller, Model Bakış Kontrolcü) hakkında bilgi veriniz.**

Yazılım mühendisliğinde kullanılan bir mimari yaklaşımdır. Basitçe yazılımın tasarımı ve geliştirilmesi aşamalarında etkili olan bir bakış açısını yansıtır.

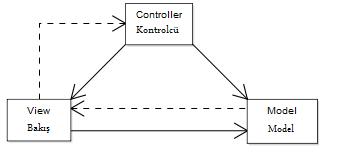
Bu bakış açısına göre kullanıcı ara yüzü (user interface) ile iş mantığı (business logic) birbirinden ayrı olmalıdır. Yani kullanıcıların önündeki ekranların tasarlanması ve geliştirilmesi sırasında kullanıcı gözüyle analiz yapılmalı ve bu analize göre kullanıcıya en kullanışlı (user friendly) ekran tasarımı yapılarak geliştirilmelidir. Arka tarafta ise iş mantığı (business logic) gözetilmeli ve kullanıcı ara yüzlerinden bu mantığa bağlantı kurulmalıdır.

MVC yaklaşımı 3 parçadan oluşur bu parçalar isminde de geçen :

* Model (Model)
* View (Bakış)
* Conroller (Kontrolcü)

parçalarıdır.

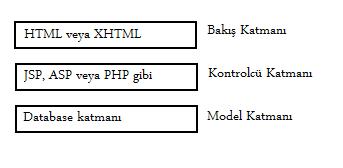
Bu parçalardan model, bilgiyi (information) veya veriyi (data) göstermek için kullanılır. View (Bakış) ise kullanıcı ara yüzünü ve kullanıcının sistem ile olan iletişimini ele alır. Controller (Kontrolcü) ise sistemin veri akışını ve bu verinin model ile olan bağlantısını kontrol etmek amacıyla kullanılır.



Şekil 1: Parçalar arasındaki bağlantı

Yukarıdaki şekilde bu parçalar arasındaki bağlantı görülmektedir. Model parçasında modellenen veriye view(Bakıştan) doğrudan erişim bulunurken kontrolcü parçası iki parçaya da erişerek kontrolü sağlamaktadır.

MVC yaklaşımını gerçek hayattan bir örnek ile anlatmak gerekirse. Örneğin bir web sayfasının geliştirilemsi sırasında Bakış katmanı genelde HTML dilinde üretilir. HTML dilindeki bu sayfaları üreten sunucu tarafında bir kodlama (örneğin PHP, JSP veya ASP gibi) bir katman bulunur ki bu katmana kontrolcü (controller) ismi verilir. Son olarak verinin tutulduğu ve modellendiği bir katman da bulunur ki bu katmanada Model ismi verilir. Dolayısıyla MVC yaklaşımına göre bazı teknolojilerin katmanlandırılması aşağıdaki şekilde olabilir:



Şekil 2: Katmanlar

Yukarıdaki katmanlar birer örnek olarak düşünülüp farklı teknolojilerinde burada kullanılabileceği unutulmamalıdır.

Günümüzde .NET J2EE gibi ortamları MVC mimarisine uygun geliştirme ortamları olarak görmek mümkündür. Ayrıca bu ortamlarda birden fazla MVC alternatifi çerçeve (Framework) de bulunmaktadır. Örneğin J2EE ortamı için [JSF (Java Server Faces)](http://www.bilgisayarkavramlari.com/2009/04/15/jsf-java-server-faces/) , Structs, JSP gibi alternatifler sayılabilir.

## **SORU-19: Atomluluk (Atomicity) hakkında bilgi veriniz.**

Latince bölünemez anlamına gelen atom kökünden üretilen bu kelime, bilgisayar bilimlerinde çeşitli alanlarda bir bilginin veya bir varlığın bölünemediğini ifade eder.

Örneğin programlama dillerinde bir dilin atomic (bölünemez) en küçük üyesi bu anlama gelmektedir. Mesela C dilinde her satır (statement) atomic (bölünemez) bir varlıktır.

Benzer şekilde bir verinin bölünemezliğini ifade etmek için de veri tabanı, veri güvenliği veya veri iletimi konularında kullanılabilir.

Örneğin veri tabanında bir işlemin (transaction) tamamlanmasının bölünemez olması gerekir. Yani basit bir örnekle bir para transferi bir hesabın değerinin artması ve diğer hesabın değerinin azalmasıdır (havale yapılan kaynak hesaptan havale yapılan hedef hesaba doğru paranın yer değiştirmesi) bu sıradaki işlemlerin bölünmeden tamamlanması (atomic olması) gerekir ve bir hesaptan para eksildikten sonra, diğer hesapa para eklenmeden araya başka işlem giremez.

Benzer şekilde işletim sistemi tasarımı, paralel programlama gibi konularda da bir işlemin atomic olması araya başka işlemlerin girmemesi anlamına gelir.

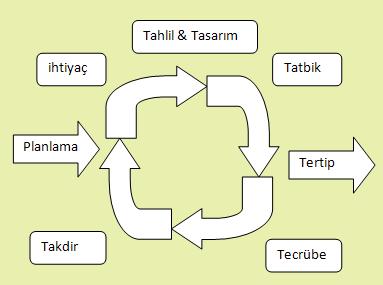
Örneğin sistem tasarımında kullanılan check and set fonksiyonu önce bir değişkeni kontrol edip sonra değerini değiştirmektedir. Bir değişkenin değeri kontrol edildikten sonra içerisine değer atanmadan farklı işlemler araya girerse bu sırada problem yaşanması mümkündür. Pekçok işlemci tasarımında buna benzer fonksiyonlar sunulmaktadır.

Genel olarak bölünemezlik (atomicity) geliştirilen ortamda daha düşük seviyeli kontroller ile sağlanır. Örneğin işletim sistemlerinde kullanılan semafor’lar (semaphores), kilitler (locks), koşullu değişkenler (conditional variables) ve monitörler (monitors) bunlar örnektir ve işletim sisteminde bir işlemin yapılması öncesinde bölünmezlik sağlayabilirler.

Kullanılan ortama göre farklı yöntemlerle benzer bölünmezlikler geliştirilebilir. Örneğin veritabanı programlama sırasında koşul (condition) veya kilit (lock) kullanımı bölünmezliği sağlayabilir.

## **SORU-20: Tekrarlı ve Arttırımlı Geliştirme (Iterative and Incremental Development) hakkında bilgi veriniz.**

Yazılım mühendisliğinde (software engineering) kullanılan bir geliştirme metodudur. Bu metod şelale modelindeki (waterfall model) eksiliklerden yola çıkılarak geliştirilmiştir ve yazılımın geliştirilmesi sırasında bir tekrar ile (döngü) yazılımın daha iyi hale getirilmesi hedeflenir.



Şekil 1: Tekrarlı ve Artırımlı Geliştirme

Yukarıdaki şekilde görüldüğü üzere planlama ile başlayan ve tertip (deployment) ile biten süreçte bir döngü (iteration) sürekli olarak tekrarlanmaktadır.

Bu daire içerisinde yapılan işler sırasıyla :

* ihtiyaç (Requirements)
* tahlil (analiz, analysis)
* tasmim (tasarım, design)
* tatbik (uygulama, implementation)

Adımlarından oluşur. Bu adımlar tamamlandıktan sonra şayet yeterli olgunluğa ulaşılmadıysa yazılım geliştirme süreci:

* Tecrübe (Test)
* Takdir (Evaluation)

Aşamaları ile devam eder ve tekrar ihtiyaç aşamasına dönülür. Ancak yazılım belirli bir kabul seviyesine ulaştıktan sonra bu daireden çıkılır.

## **SORU-21: Bağlama (Coupling) hakkında bilgi veriniz.**

Yazılım mühendisliğinde modelleme sırasında sistemde bulunan varlıkların ilişkilerini belirlemeye yarayan bir terimdir. Genellikle yapışma (cohesion) teriminin tersi anlamda kullanılır. Yani yüksek bağlama (high coupling) düşük yapışma (low cohesion) anlamında gelmektedir.

Nesnelerin birbirine bağlanması (coupling) ve yapışması (cohesion) arasındaki en önemli fark bağlanmanın düşük olmasının yani yapışmanın yükesek olmasının iki nesnenin birbirinden mümkün olduğunca izole olmasıdır. Yani bir nesnede yapılan bir değişim diğer nesnede güncelleme gerektirmez. Ayrıca iki nesne iyi tanımlanmış ve kontrolü kolay bir arayüz üzerinden iletişim içerisindedirler.

Sistem modellemede ve nesneler arası ilişki tanımlamada düşük bağlama ve yüksek yapışma (low coupling high cohesion) sloganı kullanılır. Bu sayede nesnelerin birbirinden mümkün olduğunca bağımsız olması ve sistemin modüler bir tasarıma sahip olması hedeflenir. Ancak bazı durumlarda performans ve hız endişesinden dolayı bu kural çiğnenebilir.

## **SORU-22: SDK hakkında bilgi veriniz.**

Yazılım ortamlarında, yazılım geliştirmek için kullanılan araçların bir araya getirildiği paketlere verilen isimdir. Software development kit (yazılım geliştirme araçları) kelimelerinin baş harflerinden oluşan bu kısaltma en az anlamda API (application programming interface , uygulama programlama arayüzü) içeren ve daha gelişmiş paketlerde IDE (integrated development environment), yazılım dökümantasyonu, örnek kaynak kodlar (source codes) içeren paketlerdir.

Geliştirme ortamına göre geliştirildikleri ortam ile ilgili bilgiler de içeren bu araçların en yaygınlarının listesi aşağıda verilmiştir:

* JAVA için geliştirilmiş Eclipse, netbeans, ve JAVA SDK paketleri
* Windows ortamı için Microsoft Windows SDK
* QT grafik ortamı için QTSDK

## **SORU-23: UML (Unified Modeling Language, Ortak Modelleme Dili) hakkında bilgi veriniz.**

Bilgisayar bilimlerinin, yazılım mühendisliği bölümünde yazılım modellemesi sırasında kullanılan gösterim şekilleri standartlaştırılmış ve ortak bir modelleme şekli oluşturulmuştur.  Bu modelleme şekli UML yaygınlaşmadan önce her yazılım modelleme sisteminde farklı şekillerde yapılmaktaydı ve her yazılım firması kendi standartlarını belirleme çalışmasındaydı. UML ile birlikte bu rekabet ortadan kalkmış ve ortak bir standart üzerinden modelleme işlemleri yapılmaya başlanmıştır.

UML’in temel olarak 3 çeşidi bulunur:

* Yapısal modeller (Strucutred Diagrams)
* Davranışsal modeller (Behavioral Diagrams)
* Etkileşimli modeller (Interactive Diagrams)

Yukarıdaki bu üç çeşit modelleme şekillerinin de alt sınıfları bulunur. Örneğin çok sık kullanılan Sınıf diyagramları (Class diagrams) ve Yayılma diyagramları (Deployment Diagrams) birer yapısal model örneğidir. Veya Kullanıcı Durum Diyagramları (Use Case Diagrams) bir davranışsal model örneğidir.

## **SORU-24: CASE Araçları (Case tools) hakkında bilgi veriniz.**

Bilgisayar bilimlerinini yazılım mühendisliği alanında kullanılan araçların genel ismidir. Computer Aided Software Engineering kelimelerinin baş harflerinden oluşan bu araçların amacı yazılım geliştirme süreçlerinin koltrol edilebilmesi, ölçeklenebilmesi ve kolay yönetilebilmesidir.

CASE araçlarını aşağıdaki kriterlere göre birkaç bölümde incelemek mümkündür:

* Kod üreten yazılımlar (Source code generation tools)
* UML (Unified modelling langue) tasarım araçları
* Veri Modelleme araçları (Data Modelling tools)
* Versiyon kontrol ve Yazılım ayarlama araçları (Configuration tools)
* Model Dönüşüm yazılımları (Model Transformation)
* Kod Yeniden üretim yazılımları (Code refactoring)

## **SORU-25: En kötü durum analizi (Worst Case Analysis) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde bir algoritmanın incelenmesi sırasında sıkça kullanılan bu terim çalışmakta olan algoritmanın en kötü ihtimalle ne kadar başarılı olacağını incelemeye yarar.

Bilindiği üzere bilgisayar bilimlerinde yargılamalar kesin ve net olmak zorundadır. Tahmini ve belirsiz karar verilmesi istenmeyen bir durumdur. Bir algoritmanın ne kadar başarılı olacağının belirlenmesi de bu kararların daha kesin olmasını sağlar. Algoritmanın başarısını ise çalıştığı en iyi duruma göre ölçmek yanıltıcı olabilir çünkü her zaman en iyi durumla karşılaşılmaz.

Algoritma analizinde kullanılan en önemli iki ölçü hafıza ve zaman kavramlarıdır. Yani bir algoritmanın ne kadar hızlı çalıştığı ve çalışırken ne kadar hafıza ihtiyacı olduğu, bu algoritmanın performansını belirleyen iki farklı boyuttur.

En iyi algoritma en hızlı ve en az hafıza ihtiyacı ile çalışan algoritmadır. İşte en kötü durum analizi olayın bu iki boyutu için de kullanılabilir. Yani en kötü durumdaki hafıza ihtiyacı ve en kötü durumdaki hızı şeklinde algoritma analiz edilebilir.

Limit teorisindeki master teoremde büyük O ile gösterilen (big-oh) değer de bu en kötü durumu analiz etmektedir. Bu yüzden en kötü durum analizine, büyük O gösterimi (Big-O notation) veya algoritmanın sonsuza giderken nasıl değiştiğini anlatmak amacıyla büyüme oranı (growth rate) isimleri verilmektedir.

**Örnek**

Bir çok terimli fonksiyonun (polynomial function) big-o değerini hesaplamaya çalışalım. Örnek olarak fonksiyonumuz aşağıdaki şekilde olsun:

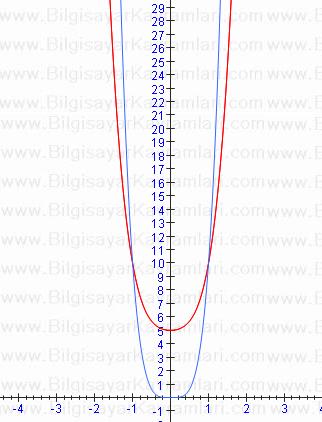
f(x) = 3x4+2x2+5

Fonksiyonun üst asimtotik sınırı (asymptotic upper bound), her zaman için fonksiyona eşit veya daha yüksek değer veren ikinci bir fonksiyondur.

Bu durumda, yukarıdaki f(x) fonksiyonu için O(x4) denilmesinin anlamlı, herhangi bir sayı ile x4 değerinin çarpımının, f(x) fonksiyonuna eşit veya daha yüksek üreteceğidir. Bu durum aşağıdaki şekilde gösterilebilir:

f(x) ≤ cx4

Gerçekten de bu değer denenirse, c=4 için aşağıdaki çizim elde edilebilir:



Şekil 1: Fonksiyon grafiği

Yukarıdaki mavi renkte görülen fonksiyon 4x4 ve kırmızı renkte görülen fonksiyon da f(x) fonksiyonudur. Görüldüğü üzre x>1 için 4x4 fonksiyonu, f(x) fonksiyonundan büyüktür. Dolayısıyla tanımımıza x>1 koşulu eklenebilir.

Kısaca f(x) = O (g(x))

tanımı, f(x) ≤ c.g(x)

şeklinde yorumlanabilir. Buradaki c değeri herhangi sabit bir sayıdır ve sonuçta elde edilen değer eşit veya daha büyüktür.

Büyük-O (Big-O) gösterimi ile kardeş olan bir gösterim de küçük-o (small-o) gösterimidir. Bu iki gösterim arasındaki temel fark küçük-o gösteriminde asimptotik üst sınır (asymptotic upper bound) fonksiyonunun tamamıncan büyük olmasıdır. Yani büyük-O gösterimindeki eşitlik durumu yoktur.

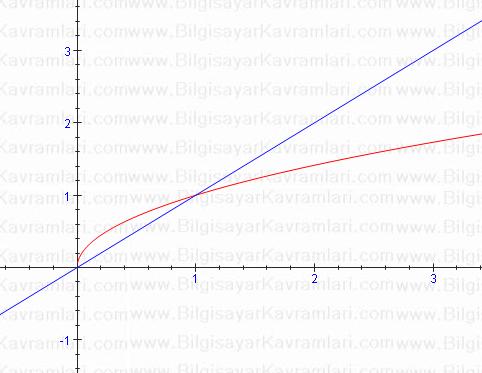
f(x) = og(x) için

f(x) < c g(x)  şartı sağlanmalıdır.

Dolayısıyla f(x) = O (f(x)) tanımı doğru olurken f(x)=o(f(x)) tanımı hatalıdır.

Bazı örnekler aşağıda verilmiştir:

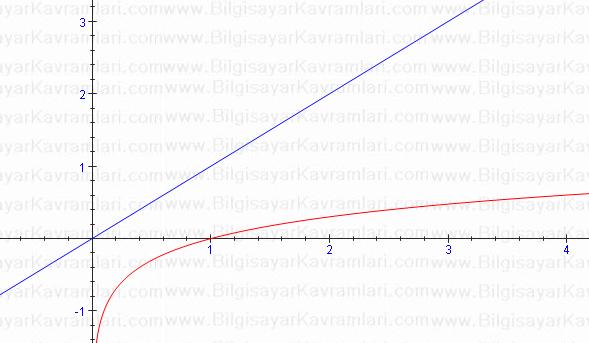
√x=o(x)



Şekil 2: Fonksiyon grafiği

Yukarıda görüldüğü üzere x>1 için √x=o(x) doğrudur. Ancak x≥1 durumunda √x=O(x) yazılmalıdır.

log(x) = o(x)



Şekil 2: Fonksiyon grafiği

Yukarıdaki şekillerde kırmızı ile gösterilen f(x) fonksiyonlarının small-o fonksiyonları mavi renk ile çizilmiştir.

## **SORU-26: Birliktelik, Münasebet ve Oluşum (Association, Aggregation and Composition) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde sistem modelleme ve mimari tasarım sırasında sıkça kullanılan ve genelde birbirine karıştırılması kolay olan konulardır. Özellikle nesne yönelimli programlama (object oriented programming) konusundaki gelişmelerle birlikte kullanılan UML modellerinde nesneler ve sınıflar arası ilişkilerde sıkça rastlanmaktadır.

Temel olarak birliktelik ve münasebet ilişki türleri birer oluşum çeşidi olarak düşünülebilir. Yani oluşum (composition) elde etmek için verilerimiz veya nesnelerimiz arasında kurabileceğimiz ilişki türleri birliktelik (association) veya münasebet (aggregation) olabilir.

Bu iki ilişki türünün arasındaki en temel fark ise birisinin verileri kendi içinde barındırması diğerinin ise atıfta bulunmasıdır.

Örneğin münasebet (aggregation) ilişkisinde bir nesneden diğer nesneye atıfta bulunulur. Bu durum bir ders ile öğrenci arasındaki ilişkiye benzer. Dersin bitmesi ve ders nesnesinin (object) yok olması durumunda bu dersi alan öğrencilerin varlığı devam eder.

Tersi bir örnek bir ev ile odaları arasında kurulabilir. Yani nasıl bir önceki örnekte dersi alan öğrenciler varsa (dolayısıyla dersin öğrencileri varsa) ev’in de odaları vardır. Ev nesnesinden odalara atıfta bulunan bir değişken benzer şekilde tanımlanır. Ancak bu sefer farklı olan ev nesnesinde bu objelerin yaşıyor olmasıdır ve ev yok olursa oda objelerinin de yok olacağıdır.

Bu durum C++ gibi dillerde gösterici (pointer) kullanılarak çözülebilir. Yani ilk örnekteki münasebet ilişkisi için ders nesnesinden öğrenci nesnelerine birer gösterici tanımlanabilir.

Birliktelik ilişkisi içinse ev nesnesinin içerisinde odalar tanımlanmalıdır.

## **SORU-27: Şelale Modeli ( Waterfall Model ) hakkında bilgi veriniz.**

Yazılım mühendisliğinde kullanılan bir yazılım projesi yönetim modelidir. Bu model aşağıdaki 4 temel merhaleden oluşmaktadır:

* tahlil (analiz, analysis)
* tasmim (tasarım, design)
* tatbik (uygulama, implementation
* tecrübe (test,test)

Yazılım mühendisliğindeki diğer modellere temel teşkil eden bu modelde yukarıdaki aşamalar sırasıyla izlenir. Aşamalar arası geçişleri oldukça sıkı olan bu modelde geri dönüşler oldukça maliyetli olmaktadır.

Buna göre ilk aşamada sistemin tahlili yapılır ve uygulanmak istenen yapı tam olarak ortaya konulur. Bu tahlil aşamasından sonra SRS (Software requirements specifications, yazılım ihtiyaç özellikleri) ismi verilen bir döküman ve bu dökümanla birlikte bir tahlil raporu (analiz raporu) çıkarılır.

Sonra bu tahlil ışığında bir tasmim (tasarım) yapılır ve çözüm yolları ortaya konulur.  Tasmim aşaması sonunda da SDD (Software design document, tasmimname, yazılım tasarım dökümanı) adı verilen bir döküman hazırlanır.

İsteklerin ve çözümlerinin dökümantasyonunun ardından uygulamaya geçilir ve bu çözümler tatbik edilir. Bu aşamadan sonra sistem somut (müşahhas) bir hâl almış olur ve yapılan hataları daha net görme imkanı doğar. Bu hataların tam olarak ortaya konulduğu aşama ise tecrübe (test) aşamasıdır. Bu son aşamadan sonra bir test raporu çıkarılarak gerekli adımlara geri dönülür ve hatalar telafi edilerek sistem istenen hale getirilir.

## **SORU-28: Tasarım Kalıpları (Tasmim Kalıpları, Design Patterns) hakkında bilgi veriniz.**

Yazılım mühendisliğinde sıkça kullanılan tasarım kalıplarının bir kütüphane haline getirilmesi ve bu kütüphanenin ileriki projelerde kullanılmasıyla proje gelitşrime sürelerinin kısaltılması hedeflenmektedir. Bu kütüphane (design patterns library) genelde farklı yapılar içerebilmesine karşılık aşağıdaki gruplarda toparlanabilir:

* Web tasarım kalıpları (web design patterns)
* UML tasarım kalıpları (UML Design patterns)
* Grafik arayüzü tasarım kalıpları (GUI design patterns)

Web tasarımı ile ilgili pek çok kalıp kütüphanesi internet üzerinden ulaşılabilir durumdadır. Özellikle AJAX teknolojisinin hızla gelişmesi sonrasında internet üzerinde yayın yapan pek çok gelişmiş kurum (yahoo, google gibi) kendi kütüphanelerini geliştirerek internet kullanıcılarına açmışlardır.

UML tasarım kalıplarının amacı ise özellikle nesne yönelimli programlama yapılan ortamlarda belirli şablonların saklanarak ileriki projelerin içerisinde kullanılmasıdır. Bu amaçla çeşitli sınıflandırmalara gidilmiştir. Örneğin yapısal tasarımların yer aldığı ve daha çok bileşenlerin konumlandırılmalarını belirleyen mimari tasarım kütüphanesi (architectural design library) veya yazılım tasarımında kullanılan sınıf diyagramlarının (class diagrams) içerildiği GOF bunlardandır.  
Burada GOF’tan bahsetmekte yarar var. GOF, gango of four (dört gangster diye çevrilebilir)’un kısaltmasıdır. Kısaca yazılım mühendisliği konusunda meşhur kitaplardan “Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2)” kitabının yazarı olan 4 kişiye (Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides) ithafen bu isim verilmiştir. Bu kitapta da anlatılan ve bu kişilerin geliştirdikleri tasarım yöntemine göre 3 farklı grup tasarım sınıfından bahsedilebilir, bu sınıfları ve geliştirme aşamalarını içeren tablo aşağıda verilmiştir:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Creational (Oluşturucu) | Structural (Yapısal) | Behavioral (Davranışsal) |
| Interface (arayüz) |  | AdapterBridge  Composite  Facade |  |
| Responsibility (Sorumluluk) | Singleton | FlyweightProxy | Chain of ResponsibilityMediator  Observer |
| Construction (oluşum) | Abstract FactoryBuilder  Factory Model  Prototype |  | Memento |
| Operation (İşlem) |  |  | CommandInterpreter  State  Strategy  Template Method |
| Extension (Uzatma) |  | Decorator | IteratorVisitor |

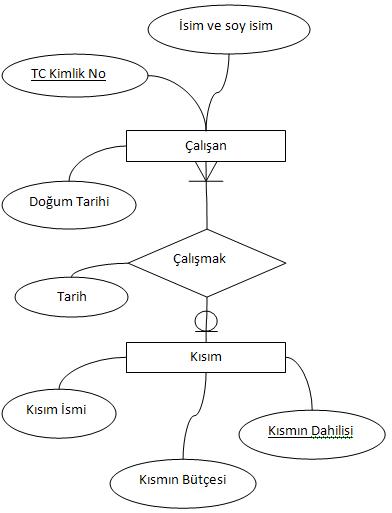
Tablo 1: GOF

Yukarıdaki tabloda GOF içerisinde yer alan 3 önemli geliştirme tasarımına göre sınıflandırma yapılmıştır.

Ayrıca yukarıda bulunmayan ve muvazi fiiller (eşzamanlı işler, concurrent processes) konusunu içeren tasarım kalıpları da bulumaktadır.

## **SORU-29: Vücubiyet (Modality) hakkında bilgi veriniz.**

Yazılım mühendisliği (software engineering) ve veritabanı tasarımı (database design) konularında sistem modellenmesi aşamasında sıkça karşılaşılan bir problem de sistemde modellenen unsurlar (entity) arasındaki ilişkinin (relationship) vücubiyetidir ( modality ) . Bu terim bir unsurun diğerini gerektirmesi anlamında kullanılmaktadır. Mesela sistemimizde bir çalışan bir de kısım (department)  unsuru bulunsun. Her çalışanın bir kısmı bulunur ve kısım unsurunun vâr olması çalışana bağlıdır. Çalışanı olmazsa kısmın bir anlamı kalmaz. Buna karşılık bir kısım bulunmasa da çalışan vâr olabilir. O halde çalışan için vâcib unsur ( gerekir unsur, mandatory) , kısım için ise ihtiyarî unsur (seçimli, optional) denilebilir. Bu durum ERD çiziminde aşağıdaki şekilde gösterilir:



Şekil 1: Modality

Yukarıda da tasvir edildiği üzere kısım unsuru ile çalışan unsuru arasında vücubiyet açısından vacib-ihtiyari (mandatory-optional) ilişkisi bulunmaktadır. Yani ihtiyari olan (seçimlik olan, optional) unsur kısımdır ve bu bir O harfi ile ilişkiyi gösteren dal üzerine yerleştirilmiştir. Buna mukabil çalışanın mevcudiyeti vacib olup (gerekli olup, mandatory) bir çizgi ile bu durum gösterilmiştir.

Şekilde bulunan diğer kaz ayağı ve çizgi ise ilişkinin sayısallığını (cardinality) göstermektedir. Şekilde de görüldüğü gibi bir ERD çiziminde bu iki özellik aynı anda gösterilebilmektedir.

## **SORU-30: Sayısallık (Cardinality) hakkında bilgi veriniz.**

Unsurlar (Entities) arasındaki sayısal bağlantıyı ifade etmek için kullanılan bir terimdir. Literatürde bazı kaynaklarda sayılabilirlik olarak da geçmektedir.  Buna göre bir unsur ile diğer unsur arasında aşağıdaki üç ilişki şeklinden birisi olmalıdır:

* Birebir one-to-one
* Bire çok one-to-many
* Çoka çok many-to-many

Bu durumlara birer misal verecek olursak:

* Bir çalışanın cep telefonu numarası ile TC kimlik no arasında birebir ilişki vardır. Yani her farklı TC kimlik no için farklı bir cep telefonu numarası söz konusudur. Bu ilişki tipi bire bir ilişkidir.
* Bir çalışan ile, çalıştığı kısım arasında bire çok ilişki vardır. Yani bir kısımda (department) birden çok çalışan çalışırken, her çalışan sadece bir kısma bağlıdır. Bu ilişki tipi de bire çok ilişkiye örnektir.
* Bir öğrenci unsuru (entity) ile hoca unsuru (entity) arasında ise çoka çok ilişki tipi vardır. Çünkü bir öğrencinin birden fazla hocası olabilirken bir hocanın da birden fazla öğrencisi bulunur.

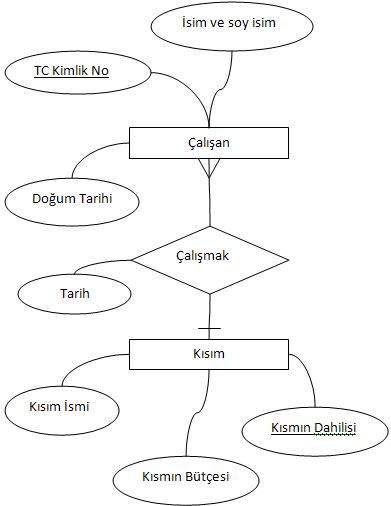
Yukarıdaki bu ikişki tiplerinin hepsinin veritabanı teorisi için anlamları çok büyüktür. İlişki türlerine göre kabaca aşağıdaki yorulmar yapılabilir.  (Bunlar genel yorumlar olup istisnaları bulunmaktadır, buradaki amaç okuyucuya fikir vermektir)

Şayet iki unsur (entity) arasında birebir ilişki varsa bu iki unsurun aslında ayrılmalarına gerek yoktur. Çok büyük ihtimalle bir unsurun iki farklı parçalarıdır ve tek bir çatı altında birleştirilmesi veritabanı teorisi açısından daha doğrudur.

Şayet iki unsur arasında çoka çok ilişki tipi varsa o halde bu ilişki tipi bire çok tipinden iki ilişkiye indirgenmelidir.

Sonuç olarak veritabanında sadece teke çok ilişki tipi elde etmek isteriz bunun sebebi yukarıda da anlatıldığı üzere birebir ilişki tipinin gereksiz oluşu ve çoka çok ilişki tipinin hem performans hemde hafıza olarak sistemde sorun çıkartmasıdır. Bu indirgeme konularını Normal Forms ve Composition konuları altında okuyabilirsiniz.

ERD çizimleri açısından olaya bakıldığında sayısallık (cardinality) kaz ayağı veya birim olarak ifade edilir. Aşağıda bir örnek üzerinde bu durum gösterilmiştir:



Şekil 1: İlişki sayısallığı

Yukarıda iki unsur arasındaki ilişkinin (relation) sayısallığı gösterilmiştir. Buna göre bir çalışanın bir tane kısmı olabilirken bir kısımda birden çok çalışan bulunabilmektedir. Dolayısıyla ilişkinin çalışana bakan tarafı çok, kısma bakan tarafı ise tek olarak gösterilmiştir. Bu gösterimde çok olan taraf kaz ayağı, tek olan taraf ise bir çizgi ile ifade edilmektedir.

Yukarıdaki bu ilişki tipi bire çok ilişki tipine bir örnektir. Çoka çok olması durumunda iki tarafta da kaz ayağı olurken, teke tek olması durumunda iki tarafta da çizgi ile gösterilmektedir.

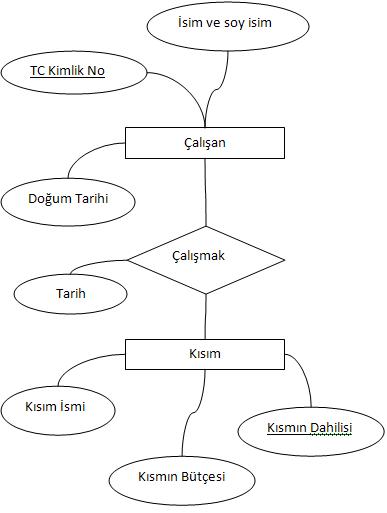
## **SORU-31: ERD (Unsur İlişki Çizimi, Entity Relationship Diagram) hakkında bilgi veriniz.**

Yazılım mühendisliği (Software engineering) ve veritabanı tasarımında (database design) sıkça kullanılan bu çizim yöntemine göre, modellenmek istenen sistemdeki unsurlar (Entities) çıkarılarak bu unsurlar arasındaki ilişkiler (relationships) tanımlanır.

Unsurların özellikleri (attributes), anahtarları (keys) belirlenerek sistemin tamamını kapsayan bir model çizilir ve bu model üzerinde tasarım yapılır.

Tasarımın bitmesinin ardından, tasarımdaki bütün unsurlar ve ilişkiler birer tabloya dönüştürülerek veritabanı uygulaması tamamlanmış olur.

Aşağıda basit bir ERD çizimi bulunmaktadır:



Şekil 1: Çalışma ilişkisi

Yukarıdaki bu ERD çizimine göre iki unsur olan kısım ve çalışan arasında bir çalışma ilişkisi bulunmaktadır. Yukarıda her unsurun ve unsurlar arasındaki ilişkinin özellikleri de verilmiştir.

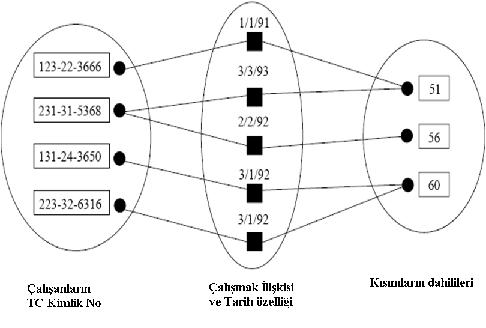
Bu modelleme yönteminde dikkat edilecek önemli bir husus ise sistemimizde bulunması gerçekten gerekli olan özelliklerin ve unsurların modellenmesidir.

ERD çiziminin üzerinde ayrıca ilişki tiplerini tutmak da mümkündür. Bu ilişki tiplerini iki çatıda incelemek mümkündür. İlişkilerin sayısallığını tutan cardinality ilişki şekli ve ilişkilerin gerekirliliğini tutan modality (tazannum eden, vacib-i vucud, birşeyin varlığının diğer şeyin varlığını gerektirip gerektirmemesi) ilişki tipi bu iki çatıdır.

## **SORU-32: İlişkinin Anlık Görüntüsü (Instance of a Relation) hakkında bilgi veriniz.**

Veritabanı teorisinde iki tablonun; veya ERD çiziminde iki unsurun (entity) arasındaki ilişkinin anlık görüntüsüne (Instance) verilen isimdir.

Bir ilişkiyi kümeler arası ilişkiye benzetmek mümkündür. Buna göre her unsurun üyesi olan kayıtlar (records) ile diğer unsurların üyeleri arasındaki ilişki aslında aşağıdaki örnekte olduğu gibi gösterilebilir:



Şekil 1: İlişki ve anlık görüntü

Yukarıdaki örnekte bir çalışan unsurunun, kısım unsuru ile olan ilişkisini (relationship) ve bu ilişkinin anlık görüntüsü gösterilmiştir.

Bu ilişki yapısına bakıldığında anlık olarak bütün çalışanları, bütün kısımları ve hangi çalışanın hangi kısımlarda çalıştığını (aradaki ilişkiyi) görmek mümkündür.

## **SORU-33: İlişki (Relationship) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde ilişki kavramı çok çeşitli anlamlarda kullanılabilir. Örneğin nesnelerin birbiri ile olan ilişkisi veya veritabanındaki tabloların ilişkisi gibi.

**Veritabanı teorisindeki tablo ilişkisi**

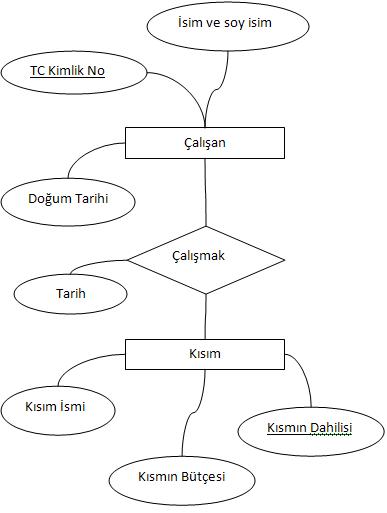
Temel olarak bir ilişkisel veritabanını modellemekte kullanılan ERD (Unsur ilişki çizimi, entity relationship diagram) üzerinden bir ilişkiyi anlamak mümkündür. Bilindiği üzere bir ERD, unsurlardan (entities) oluşmaktadır ve ERD üzerinde bu unsurların ilişkisi modellenmektedir. Aşağıdaki cümleyi ele alalım:

“Ali muhasebe kısmında çalışıyor”

Bu cümlede “Ali” isminde bir çalışandan ve muhasebe departmanında bahsediliyor. Dolayısıyla bu cümleden çıkan anlama göre bir “Çalışan” unsuru bir de “Kısım” unsuru bulunmakta ve Ali, çalışan unsurunun, muhasebe ise departman unsurunun birer örneği olmaktadır.

Dolayısıyla bu iki unsur ( çalışan ve kısım) arasında çalışmak ilişkisi olduğunu yukarıdaki cümleden anlıyoruz.

Yukarıdaki bu cümleyi aşağıdaki model ile tasvir etmek mümkündür:



Şekil 1: Unsurlar arası ilişki

Yukarıda iki unsur arasındaki ilişki görüntülenmiştir. Buna göre ilişkinin de özellikleri olabilir (descriptive attributes). Örneğin yukarıdaki tarih özelliği çalışma ilişkisinin başladığı tarihi tutmak için tasarlanmıştır. Yani bir çalışanın bir kısımda ne kadar zamandır çalıştığı bilgisine ulaşılabilir.

## **SORU-34: Unsur (Entity) hakkında bilgi veriniz.**

Veritabanı tasarımında (database design) ve yazılım mühendisliğinde (software engineering) sıkça kullanılan bir tasarım yöntemi, modellenmek istenen sistedemki unsurları çıkararak bu unsurların özelliklerini ve bu unsurlar arasındaki ilişkileri tutmaktır.

Temel olarak bir unsur nesne yönelimli programlama mantığında olan her nesneye benzetilebilir. Ancak bir unsurun bir nesneden temel farkı, ihtiyaç duyulduktan sonra hemen herşeyin birer unsur olabileceğidir. Örneğin insan sınfının bir özelliği olarak isimi ele alalım. Bu özelli yeterli olgunlukta ise (ünvan, ilk, orta, soy isim gibi karmaşık bir yapıya sahipse) bir unsur olarak değerlendirilmek zorundadır. Ancak nesne yönelimli programlamada  bir sınıf olması mantıksızdır.

* Bir unsur’un tanımını aşağıdaki şekilde yapmak mümkündür:
* Unsur (Entity) gerçek hayattandır.
* Her unsurun özellikleri (attributes) vardır
* Her özelliğin alabileceği değerler (domain) tanımlıdır
* Bir unsuru oluşturan özellikler içerisinden bu usuru tek başına tanımlayan bir anahtar (key) seçilebilir
* Unsurlar kümelenebilir. (Çalışanlar kümesi hem sekreterleri hem de müdürleri kapsar)

Yukarıdaki şekilde tanımlanan bir unsurun ERD (Entitiy relationship diagram) üzerindeki gösterimi aşağıdaki şekildedir:



Şekil 1: Unsurun özellikleri

yukarıda bir çalışan unsuru ve bu unsurun özellikleri gösterilmiştir. Burada bulunan özellikler bir çalışanın sahip olduğu bütün özellikler değildir. Bir unsuru sistem modellemesinde kullanırken dikkat edilecek husus bu unsurun sistemin modellenmesi için gerekli olan özelliklerinin (attributes) bulundurulmasıdır. Bunun dışındaki özellikler sistemin tasarımını etkilemeyeceği için modelemeye de alınmaz.

Yukarıdaki tasvirde dikkat edilecek bir husus da “TC Kimlik No” özelliğinin altının çizili olmasıdır. Bunun sebebi bu özelliğin tek başına bir çalışanı diğer çalışanlardan ayırt etmeye yarayan bir özellikl oluşudur. Bu tip özelliklere anahtar (key) ismi verilmektedir.