Yazılım Mühendisliğinde Performans Yönetimi ve Otomatik Bellek Yönetim Durumunun İncelenmesi

[1]Abdullah Volkan Sevinçok, [2]Murat Osman Ünalır

[1]PERFORM, Performans ve Test Danışmanlık. Hizmetleri A.Ş.
[2]EGE Üniversitesi

[1]volkan.sevincok@perform.com.tr
[2]murat.osman.unalir@ege.edu.tr

Özet. Bu bildiride otomatik bellek yönetimi, bellek çöp toplama ve yazılım mühendisliğinin bilgi teknolojileri metodolojilerinden Bilgi Teknolojisi Altyapı Kütüphanesi (ITIL) ve Yetenek Olgunluk Model Entegrasyonu (CMMI) ile olan bağlantıları ele alınmaktadır. Bu bildiride ayrıca, dünyada yazılım mühendisliğinin bir dalı olarak kabul görmüş fakat Türkiye'de gerek akademik gerekse yazılım sektöründe henüz pek uygulanmayan "Yazılım Performans Mühendisliği" konusu ele alınacaktır.

Java'da JVM, .Net'de CLR tarafından otomatik olarak gerçekleştirilen bellek yönetimi ve çöp toplama işlemlerinin iyi anlaşılamaması yazılımlarda normalden daha uzun süre sistemin askıya alınması (Suspend), yanıt sürelerinin uzaması gibi kritik performans problemlerine yol açmakta, çözüme kavuşturulması ise maliyetli olabilmektedir.

Bununla birlikte yazılım projelerinde bellek çöp toplama işlemlerinde karşılaşılan gerçek bir problem örnek olarak verilmekte ve bu probleme ait çözüm kümeleri ve çözüm uygulanışları da sunulmaktadır.

1 Giriş

Adı her ne kadar çöp toplayıcı da olsa asıl görevi bellek tahsis etme, belleği serbest bırakma ve belleği birleştirmek olan OBY'nin (Otomatik Bellek Yöneticisi) teknik kullanım detaylarına yeterince dikkat edilmediğinde sistemlerde oluşturduğu performans sorunları kaçınılmazdır.

"C" ile yazılım geliştiren yazılımcıların dikkat etmesi gereken en önemli noktalardan bir tanesi bellek yönetimidir. Yazılım geliştirici değişkenin/ nesnenin bellek tahsisini "malloc" komutu ile gerçekleştirir ve "free" komutu ile tahsis ettiği belleği

serbest bırakır. Bir hatadan veya unutkanlıktan dolayı gerçekleşmeyen belleğin serbest bırakılma işlemi ise bir bellek sızıntısı oluşturur ve bu ise bellek ve bellek bağlantılı problemler oluşturur. "C" dili yazılım geliştiren yazılımcılar, bellek yönetimini kendileri yapmak zorunda oldukları için, kod geliştirme hızları düşer.

Java ve .Net teknolojileri başta olmak üzere yönetimi bir sistem tarafından gerçekleştirilen bellek tahsisi ve belleğin geri bırakılma işlemi otomatik olarak gerçekleştirilmekte, bu ise teorik olarak, yazılım geliştiricilerin bellek sızıntılarını ve bellek ile ilgili problemleri düşünmeksizin kodlama yapmalarını sağlamaktadır. Fakat teorik olarak böyle olsa da pratikte tam da bu şekilde değildir. Hatta, otomatik bellek yönetimi, yazılımcıyı içinden çıkılmaz karmaşık bir yapıyla bile karşı karşıya bırakabilir. Bu yüzden OBY'nin çalışma prensipleri iyi anlaşılmalıdır.

2 Yazılım Performans Mühendisliği

Yazılım performans mühendisliği, hizmet verecek bir sistemin tüm bileşenlerini; sunucularını, ağ yapılarını, proje ekibini, yazılım geliştirme süreçlerindeki tüm adımlarını ve araçlarını, rollerini, aktivitelerini ve yeteneklerini göz önüne alarak ve etkileşimde bulunarak, sistem bütününün performans temelli bir mimaride oluşturulmasına yönelik uygulanan bir yaklaşımdır.

Yazılım performans mühendisliği bu bağlamda, fonksiyonel ve fonksiyonel olmayan müşteri gereksinimlerinin belirlenmesiyle başlayıp, analiz, tasarım, kodlama, test, devreye alım ve yazılım bakım safhalarında gerçekleştirilen tüm işlemlerin, performans modeline göre uygulanıp, işlemlerin tanımlanan performans süreci doğrultusunda yapılıp yapılmadığının kontrolünü gerçekleştirir. Bu anlamda yazılım performans mühendisliği, yazılım kalite süreçlerinin önemli evreleriyle de yakından ilişkilidir.

Yazılım Mühendisliği kitabında [1] (Sayfa 6, Bölüm 1 , Giriş) yanıtı verilmek üzere 3. sırada bir soru sunulmuşur. "What are the attributes of good software?" Bu sorunun türkçe karşılığına ve yanıtına bakmadan önce soru içindeki "attributes" kelimesinin de incelenmesinde fayda vardır. Bu kelime için sözlükte şu şekilde bir açıklama yapılmaktadır. "a quality or feature, especially one that is considered to be good or useful" [2] ("Bir kalite veya özelliğin iyi ve yararlı/ faydalı/ kullanışlı olma durumu") Bu durumda "What are the attributes of good software?" cümlesinin Türkçe karşılığını "İyi bir yazılımın kalite bileşenleri nedir?" olarak çevirebiliriz.

Sorunun yanıtı ise : "Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable." olarak belirtilmiştir. Bu ifadeyi de Türkçeye şu şekilde çevirebiliriz : "İyi bir yazılım, kullanıcıya gerekli olan fonksiyonaliteyi ve performansı sunarken sürdürülebilir, güvenilir ve kullanılabilir olmalıdır. "

Bu tanıma göre fonksiyonalite ve performans bir yazılım sisteminin öncelikli bileşenleridir. Yazılım mühendisliğinde gereksinimler, fonksiyonel ve fonksiyonel olmayan gereksinimler üzere ikiye ayrılır. Fonksiyonel olmayan gereksinimler; "güvenlik, kullanılabilirlik, güvenilirlik ve performans" olarak ifade edilebilir. Carnegie Mellon Yazılım Mühendisliği Enstitüsü (SEI), yazılım mühendisliğini, bilimsel

bilginin sistematik uygulanarak insanlığın hizmeti için, düşük maliyetli çözümlerin sunulması olarak ifade etmiştir. [3]

Hem bu tanıma hem de yazılım mühendisliği tanımına göre; aslında, fonksiyonel ve fonksiyonel olmayan gereksinimler, iki farklı sınıfta isimlendirilmesi yerine sadece "Yazılım Gereksinimleri" başlığı altında ele alınabilir. Günümüz yazılım mühendisliğinde, fonksiyonel olmayan gereksinimler fonksiyonel olan gereksinimler kadar pekte önemsenmemekte, özellikle performans ve güvenlik sorunları çok büyük maddi ve telafi edilemeyecek prestij kayıplarına yol açmaktadır. Gereksinimlerin, fonksiyonel olan ve olmayan şeklinde ayrıştırılmaması, yazılım kalitesini arttırmak için atılacak yeni ve önemli bir adım olabilir.

Yazılım performans mühendisliği ayrıca;

1. Gereksinimleri test etmek için gerekli yazılım süreçlerini,
2. Hizmet kalite seviyelendirilmesinde, insan ve teknoloji ilişkilerini,
3. Devreye alım öncesi uygulama performans eniyilenmesini,
4. Devreye alım sonrasında ise sistemin belirlenen performans modeline uygunluğunun onaylanması

olarak da ifade edilebilir.

Günümüzde sitelerin işlevselliği ve hızlı olması artık çok önemlidir. Aynı zamanda sistemler çok dağıtık, birbiriyle karmaşık şekilde entegre olmuş büyük ölçekli mimarilere sahiptir. Bu karmaşık sistemleri ölçmek ve eniyilemek için yazılım performans mühendisliği anlayışına gerek duyulmaktadır. Bu yaklaşım, bir uygulama performansının yönetilmesinde yüksek verim sağlayacak ve böylece performans yönetimi daha az karmaşık olacaktır. Performans verisinin toplanması, analiz edilmesi ve raporlanması için bir performans bilgi tabanına sahip olunması gerekmektedir.

Bir site performansının düşük olduğu durumu tanımlamak için genel olarak "uygulama yavaş çalışıyor" ifadesi kullanılır. Fakat bu yavaşlığın yanıt süresinin çok yüksek olduğundan mı yoksa uygulamanın aynı anda beklenenden daha fazla kullanıcıya yanıt veremediğinden mi olup olmadığı bir metodoloji kapsamında genellikle yanıtlanamaz. Sorun, birinde veya diğerinde veya her ikisinin birleşiminden de kaynaklanıyor olabilir. Bununla birlikte burada ifade edilen bu basit örnekte olduğu gibi; performans ve ölçeklenebilirlik kavramlarının birbirlerine karıştırılmaması gereklidir. Gerçekleşebilecek kısa veya daha karmaşık performans sorunlarının çözümü için yazılım performans mühendisliği ve süreçleri gereklidir.

2.1 Performans Ölçüm

İşlem gören bir isteğin durumunu belirleyen bir çok bileşen vardır. O anki yük, diğer uygulamaların istekleri, sistem faktörleri (Bellek, CPU ve ağ kullanımları) ve isteğin karmaşıklığı yanıt süresini etkileyen nedenlerden bazılarıdır.

Temel performans ölçütleri;

- Yanıt süresi (Response Time) : Bir isteğin ne kadar sürede tamamlandığı
- Birim zamanda yapılan işlem (Throughput) : Uygulamanın, tanımlanan zaman (saniye, milisaniye v.b.) içinde, işlem yapabileceği toplam istek sayısı

- Sistem Durumu (Availability) : Bir kaynağın gereksinim duyulduğu zamanlarda, kullanıma uygun olma seviyesi

olarak tanımlanabilir.

Performans ayrıca, uygulamaların sistemden talep ettiği kaynak istekleri ve bu kaynak isteklerinin birim zamanda yapılan işlemlerle ilgili ölçümlemelerin yapılması olarak da ifade edilebilir.

Ölçümlerin sağlıklı yapılabilmesi için; ölçütlerin belirlenmesi ve tanımlanmış bir performas modeli çerçevesinde performans analiz ve değerlendirmelerinin gerçekleştiriliyor olması gerekmektedir.

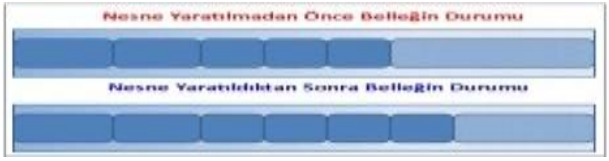
2.2 Performans Modelleme

Performans modelleme; yazılım projelerinde gereksinimlerin alınması ile başlayan, analiz, tasarım ve kodlama ile birlikte devam ederek yazılım geliştirmenin tüm yaşam döngüsü boyunca devam eden bir yöntemdir. Bu modellemeyi yapabilmek için ise yapısal ve tekrarlanabilir bir yaklaşım gereklidir. Performans sorunlarının temeli sıklıkta projenin tasarım kısmında oluşmaya başlar. Müşteri gereksinimlerinin alınmasıyla birlikte, talep edilen özellikler performans kaygısı duyulmadan analiz edilerek tasarımı gerçekleştirilebilir. Tasarımdan sonra kodlaması gerçekleşen proje, belli bir olgunluğa eriştikten sonra, test sunucuları üzerinde çeşitli performans testleriyle sınanmalıdır. Bu şekilde projenin etkileşimde olduğu kaynakların kısıtları, eşik değerleri gibi çok önemli metrikler elde edilerek, mümkün olan ve kabul edilebilen CPU, Disk ve Ağ I/O işlemlerinin sınırları elde edilir. Bunun en önemli getirisi de proje bütçe tahminlemesine veri sağlamasıdır.

3 Otomatik Bellek Yönetimi

3.1 Otomatik Bellek Yönetim Durumun İncelenmesi

Nesneler ve statik değişkenler dinamik olarak “heap” te yaratılır [4]. Her tekil nesne yaratımı için işletim sistemi ile senkronize olunmadığından gerçekleştirilen bu tahsis işlemi çok hızlı gerçekleşir. Talep edilen bellek yaratıldıktan sonra, “C” dilindeki bağlı listelerde kullanılan gösterge işaretçisi gibi, OBY’nin göstergesi otomatik olarak bir sonraki tahsis edilecek noktaya kaydırılır.



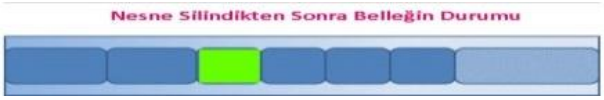
Şekil 1. : Belleğin Durumu-1

OBY, heap’teki kullanılmayan nesnelerin değil, kullanılan nesnelerin peşindedir. Kullanılan nesneler bulunur ve bunun haricindeki herşey “çöp” olarak işaretlenir. Bu

küçük fakat çok önemli nokta yazılımlarda performans sorunlarının başını çekmektedir. Artık kullanılmayacak bir nesne, OBY tarafından heap'ten fiziksel olarak silinmez, sadece daha sonra gerçekleştirilecek bellek tahsisleri için ilgili alanı kullanılabilir olarak işaretler. Bu işlem sonucunda oluşan iki önemli konunun farkında olunması bellek yönetimi için çok önemlidir.

Serbest bırakılan bellek işletim sistemine iade edilmez. [5]

Serbest kalan bellek alanın boyutu, yeni yaratılacak bir nesnenin veya statik değişkenin boyutundan küçük olduğu sürece belleğin bu bölümü kullanılamaz duruma gelir. (Heap Fragmentation) [5] Bu olay, sabit disklerde disk alanlarının parçalanması ile aynı mantıktadır. Parçalanmış bellek OBY'nin belleği toplayarak düzenlemesine (Heap Compacting) [6] veya uygun bir nesnenin tahsisine kadar devam eder.



Şekil 2. : Belleğin Durumu-2

3.2 Belleğin Verimli Kullanılması

OBY'nin işini daha verimli yapması için kod geliştirirken dikkat edilmesi gereken noktalar vardır.

1. Gizli bellek tahsislerini azaltmak :Nesne yaratımı mümkün olduğunca bellek tahsis komutu ile yapılmalıdır.

```
System.out.println("Topla Sayı + sayıTopla(1.0, 2.0 ,
3.0, 4.0));
```

Performans yönelimli kod:

```
double dizi[] = new double[4];
dizi [0] = 1.0;
dizi [2] = 2.0;
dizi [3] = 3.0;
dizi [4] = 4.0;
System.out.println("Topla Sayı" + sayıTopla (dizi[0]+
dizi[2]+ dizi[3]+ dizi[4]));
```

Kullanılacak değişkenlerin "new" komudu ile bellek tahsisinin sağlanması, OBY'nin performansını arttıracaktır.

2. Kısa ömürlü nesnelerin kullanımı : Uzun ömürlü nesnelerden kısa ömürlü nesnelere başvuruda bulunmamalıdır.

```
Class Customer
 {
Order _lastOrder;

void insertOrder (int ID, int adet);
    {
      Order currentOrder = new Order(ID, adet);
      currentOrder.Insert();
      this._lastOrder = currentOrder;
    }
}
```

Performans yönelimli kod:

```
Class Customer
 {
    int _lastOrderID;

     void ProcessOrder (int ID, int adet)
      {
      ...
      this._lastOrderID = ID;
      ...
      }

  }
```

3. Büyük Ölçekli Nesneler İçin Belleğin Önceden Tahsisi :

Belleğin önceden ve büyük miktarda tahsisi "C" dilinde yazılım geliştiren yazılımcıların başvurdukları bir yöntemdir. Bu işlem birden fazla bellek tahsis çağrımından kaçınmak için gerçekleştirilirdi, fakat bu işlem OBY özelliği taşıyan dillerde önerilmemektedir.

Çünkü;

- Bellek tahsis işlemi zaten hızlı gerçekleşmektedir.
- Önceden fazla ve büyük bellek tahsis etmek OBY'nin gereksiz çalışmasına neden olmaktadır.

4. OBY'nin manüel çalıştırılması : OBY'nin kod içinden manüel çağrılması OBY'nin heap'te belli bir metodoloji ile kontrol altında tuttuğu tüm kısımların taranması anlamına gelmektedir. Bu ise OBY'nin otomatik çalışmasından daha uzun süren bir işlemdir.

4 Tecrübe Edilen Problem

“ Çok yüksek CPU (~% 90) kullanımı ve gerçekleşen kullanıcı isteklerinin çok geç tamamlanması.”

Karşılaşılan problem; her zaman olmamakla birlikte bazı zamanlarda Web sunucusunun çok yüksek CPU kullanımıyla eşzamanlı kullanıcı isteklerinin çok geç yanıtlanması şeklinde idi. Problemi tespit etmek için sistem izlenmeye alınarak CPU kullanımının yükseldiği anlar kayıt altına alınmaya başlanmıştır.

Yüksek CPU kullanımına genellikle neden olan konular;

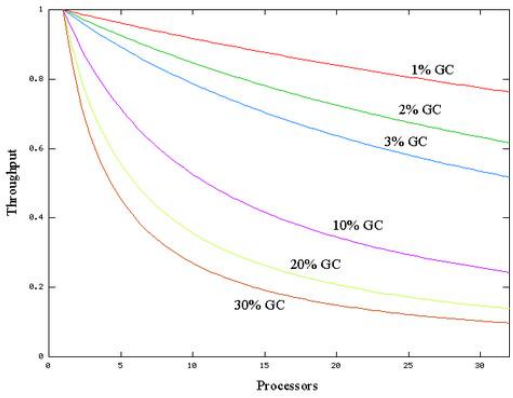
- Virüsler,
- Uyumsuz yazılımlar,
- Yazılım çakışmaları,
- OBY’nin çok fazla çalışması,
- Program içindeki olası sonsuz döngü(ler),
- Çok fazla istek ile aşırı yük.

olarak ifade edilebilir.

CPU kullanımının izlenmesiyle birlikte, OBY işlemlerini gerçekleştirirken, “OBY’nin CPU tüketim yüzdesi”, “bir saniyede tahsis edilen byte” ve “saniyede gelen web istek sayısı” metrikleri izlenmeye başlanmıştır.

Yukarıda da bahsedildiği gibi OBY artık kullanılmayan nesnelerle değil halen aktif olan nesnelerle ilgilenmektedir. OBY çalışma sırasında nesne hiyerarşininin düzenini sağlamak için ilgili uygulamanın çalışmasının geçici olarak durdurur, işlemini tamamlar ve uygulamayı tekrar çalıştırır. Buradaki kritik konu ise, ne kadar çok aktif nesne yaratılmışsa, uygulamanın daha uzun bekleme sürelerine sahip olacağıdır. Bu ise, daha uzun yanıt süreleri ve birim zamanda daha az işlem anlamına gelmektedir. Bununla birlikte CPU adedinin ve çoklu kanal (multi-thread) işlemlerinin artışıyla OBY’nin uygulamaları durdurma süreleri arasında doğrudan bir ilişki söz konusudur.

Gerçekleştirilen bir çalışmaya göre; CPU ve çoklu kanal işlemlerinin artışıyla, OBY işlemlerinin daha geç bittiği ve birim zamandaki gerçekleşen işlemin de düştüğü saptamıştır. Aşağıdaki örnekte; OBY’nin çalışma süresi %1’den %2’ye çıktığında birim zamandaki işlem sayısının %20’lik ek kaybını gösterilmektedir. [7]



Şekil 3. CPU, Birim Zamanda İşlem Sayısı ve OBY Kullanımı

Yüksek CPU kullanımının izlenmesinde, saniyede gelen istek sayılarında bir artış gözlenmemiş fakat OBY'nin uygulamayı bekletme sürelerinde bir artış olduğu belirlenmiştir. Bu sorunun detayını belirleyebilmek için bir uygulama performans yönetimi aracı ile yüksek CPU kullanım zamanlarındaki "Sınıf ve Metod" kullanımlarının raporu alınmış ve bir metodun OBY nedenli uzun süreli beklemeye neden olduğu görülmüştür. Önemli soru ise, neden bir metodun OBY'yi uzun süreli çalıştırdığı ve uygulamayı uzun süreli durdurduğudur? İlgili metod incelendiğinde, metodun sadece bir XML çıktısı oluşturduğu ve ilk bakışta metodun içeriğinde de aykırı bir durum görülmemiştir.

Beklemeye neden olan metod:

Method	Class	Suspension Sum [ms]	Reason
List(System.DateTime, System.DateTime, int, int, int, ...	Package	120254.73	Garbage Collection

Metodun kısaltılmış olarak içeriği :

```
string xListeleme (System.DateTime tarih1, Sys-
tem.DateTime tarih2, int adet,.....)
{
    String tarihIcerik=”<Tarih>”;
    for (int loop=0;  loop<adet; loop++)
    {
     tarihIcerik += "<T1><Ay>" + tarih1.Month +
"</Ay><Gun>".. + Int32.Parse (loop.Text);
    }
   tarihIcerik +=  “</Tarih>”;

   return tarihIcerik;}
```

Fakat XML'in içeriği incelediğinde olması gerekenden çok daha farklı olduğu görülmüştür. XML basit içeriği yerine çok büyük ve tekrarlı verilerden oluşmaktaydı.

Bu tekrarlı veri, ancak metodunun kendisine gelen döngü değişkenin yanlışlıkla büyük bir değere sahip olmasından kaynaklanabilirdi. XML'in sahip olması gerçek tekil veri büyüklüğü ile XML'in sahip olduğu veriye göre hesaplama yapılınca döngü değerinin ~ 72,000 olduğu belirlenmiştir. Böyle büyük bir değer büyük nesneler yaratarak bellekten büyük bir parçayı tahsis etmiştir. Döngü değerinin çok yüksek olmasının yanında, metodun içinde gerçekleştirilen "String +=" işlemi de bellekten giderek artan bir değerin tahsis edilmesine dramatik biçimde etki etmiştir. String nesneleri durağan nesnelerdir. Aslında bir String'e "+=" işlemi yaptığınızda bellekte oluşan durum String'in birleştirilmiş hali değil yeni bir String nesnesinin değeridir. Mevcut metod üzerinden benzeterek örneklendirmek gerekirse;

```
tarihIcerik = "<Tarih> <T1> <9> ........1";
tarihIcerik += "_Tarih 10";
tarihIcerik += "_Tarih 11";
tarihIcerik += "_Tarih 12";
```

Sonuç olarak "<Tarih><T1><9>.....1_Tarih 10_Tarih 12_Tarih 12" çıktısını elde ederiz. Fakat bu işlem için bellekte 4 farklı String oluşturulur.

"<Tarih><T1><9>.....1"
"<Tarih><T1><9>.....1_Tarih 10"
"<Tarih><T1><9>.....1_Tarih 10_Tarih 11"
"<Tarih><T1><9>.....1_Tarih 10_Tarih 11_Tarih 12"

Bu durumda; büyük döngüsel ve String birleştirme işlemlerinde hem Java hem de .Net teknolojierinde "Stringbuilder" kullanılmalıdır. Stringbuilder metinsel ifadeleri birleştirmek için geçici "char" tipinde bellek tahsisini önceden gerçekleştirerek yeni "String" nesne yaratılmasının önüne geçer. Gerçekleştirilen 100,000 adetlik deneysel döngüde Stringbuilder işleminin String'e göre 1/250 oranında daha performanslı olduğu raporlanmıştır. [8].

4.1 Çözüm Kümesi, Uygulanışı ve Yazılım Kalite Süreçleri Açısından Bakış

Metoda gönderilen sayısal değişkenlerin sıralamasında yapılan bir yazılım hatasının bu kadar büyük döngü değerine sahip olduğu yapılan incelemelerde bulunmuştur. Değişkenlerin yerleri olması gerektiği gibi güncellenerek sorun giderilmiştir. Fakat bu küçük hata donanım ve yazılım kaynaklarını, yazılım proje ekibinin prestijini ve kullanıcıların operasyonel işlerini oldukça olumsuz etkilemiştir.

Sorunun kök-neden analizi gerçekleştirildiğinde, metod parametre değerlerinin metod içinde çalışma durumlarının incelenmemesi olarak ifade edilebilir. Kusursuz döngüsel işlemde ve normal koşullar altında yazılımın beklenen çıktıyı üretmesi beklenen bir durumdur. Fakat asıl önemli olan kusurlu bir durumda yazılımın nasıl davranacağıdır. Bu ve benzeri sorunları belirlemenin birincil yolu birim testlerin gerçekleştirimidir.

4.2 Çevik Yöntemler ve Birim Testler

Sürekli kod entegrasyonu ve test güdümlü yazılım geliştirme gibi çevik yöntemler, kod-bileşen seviyesinde ve performans ölçütlerinin belirlenmesinde önemli yer tutmaktadır. Örneğin birim testler, bir bileşenin işlem süresinin, veritabanı çağrı sayısının veya CPU kullanımının kontrolünde kullanılabilir. Birim testlerin sağladığı en büyük faydalardan biri de hataların daha kodlama aşamasında bulunarak düzeltilmesine olanak vermesidir. Paylaşılan bu deneyimdeki sorun, kodlama aşamasında bir birim test ile sınansaydı, üretim ortamında hiç gerçekleşmemiş olacaktı. Birim testler, kodların fonksiyonalitesini doğrulamak için basit ve etkin bir sistem olduğu gibi, performans analizi, ölçeklenebilirlik ve bileşenlerin nasıl çalıştığına dair kontrolleri sağlayan çok güçlü bir yöntemdir.

4.3 Performans Verisi Toplama

Performans yönetimi için gerçekleştirilmesi gereken ilk adımlardan birisi de performans verisinin doğru olarak elde edilmesidir. Performansa ait verilerin toplanması performans ölçme ve modellemeye önemli bir girdi sağlayacaktır.

Performans analizini gerçekleştirmek için ölçütlerin önemi çok kritiktir. Fakat önemli soru nelerin ölçüleceğidir.

Bellek, CPU , hatalar, süreler, disk I/O, ağ I/O gibi değerler ölçülmesi gereken ana bileşenlerdir. Performans iyileştirme tekrarlamalı bir süreçtir. İyileştirme sonuçlarını/faydalarını görmek için testler tekrarlanmalı ve ölçülmelidir. Bu süreç, performans bileşenlerinin sonuçlar ile örtüşünceye, kod optimizasyonu, tasarım değişikliği veya alternatif bir yönteme başvuruncaya kadar sürmelidir.

Öncelikle anahtar senaryolar oluşturulmadır. Performans senaryoları, öncelikleri ve çeşitli değerlere ait çalışma yükünü tanımlar. Anahtar senaryolarda iyi tanımlanmış performans bileşenleri ile nereye odaklanılması gerektiği ve tekrarlı sürecin ne zaman tamamlanacağı bilinir.

Performans iyileştirme disiplininden kopulmamalı, sisteme uzak, az fayda getirecek çalışmalardan vazgeçilmelidir. Performans bileşenleri ile net olarak çalışmayacak teknik özelliklerin proje dışı bırakılması gereklidir. Gerçekleştirilen bu önemli çözümlerin kuruma ve son kullanıcıya/müşterinize olan maliyetleri çok önemlidir.

Veri toplamak için oldukça farklı araçlar vardır. Örnek olarak;

- JMX (Java Management Extensions)
- Sistem kaynaklarını ve uygulamaları izler.
- JVMPI (Java Virtual Machine Profiler Interface) / JVMTI (Java Virtual Machine Tools Interface)
- C/C++ ile geliştirilmiş ve JVM'e erişerek hata ayıklama, profilleme ve izleme imkanı sunar.
- Perfmon (Windows Performance Monitor)

- Ağ, CPU, Disk, Veritabanı, .NET, Hyper-V gibi ana bileşenler başta olmak üzere bir çok metriği dinler ve raporlar.
- SPA (Windows Server Performance Advisor)
- Sunuculardan, tanımlanan metrikleri toplayarak, potansiyel performans sorunlarını belirler, kıyaslar, bilgi verir ve raporlar.

4.4 Tecrübe Edilen Deneyim ve CMMI - ITIL

Bu deneyim, genel bir yaklaşım ile, yazılım bakımı kısmı CMMI-SVC kapsamında ele alınabilecek bir konudur (CMMI for Development [9], CMMI for Service [10], and CMMI for Acquisition [11]). Fakat bununla birlikte CMMI ve ITIL açısından konu örtüşebilir. ITIL yönü ile ilgili konu bir "Değişim Yönetimi" (Change Management) kapsamına girer.

ITIL-Değişim Yönetiminin kendi içinde ilişkili iki süreci daha vardır. "Konfigürasyon Yönetimi" (Configuration Management) ve "Sürüm Yönetimi" (Release Management).

"Sürüm Yönetimi" ile "Hizmet Geçiş" (Service Transition) ve buna bağlı olarak "Doğrulama" (Verification) ve "Onaylama" (Validation) süreçleri de ilişkilidir.

İlgili deneyim, ITIL veya CMMI-SVC açısından "Değişim Yönetimi"'ni de ilgilendirir. Bu yüzden gerçekleştirilebilecek etki analizi ile problemi çözmek için, değişmesi gereken başka yazılımların da olup olmadığı, yazılımda "ne değişecek" ve "ne kadar değişecek" kararının verilmesi ve çözümün hayata geçirilmesi için "Değişim Kontrol Kurulu" ndan (Change Control Board) onay alınması gerekmektedir.

Ayrıca, problemin niteliğine göre CMMI-DEV açısından da ilgili olup olmadığı da araştırılabilinir. Eğer problem gereksinimlerdeki bir yanlış anlaşılmadan kaynaklanıyor ise o zaman "Gereksinim Yönetimi" (Requirements Management) de konuyla ilgili olacaktır.

Bununla birlikte, çözüm çalışmalarında "Kök Neden Analizi" (Root Cause Analysis) ve eğer birden fazla alternatif çözüm olasılığı var ise "Karar Analizi ve Çözümü" (Decision Analysis and Resolution) süreçlerine gidilebilinir.

Sonuç olarak; çözüm eğer bir tasarım değişikliği gerektirecek ise "Teknik Çözüm" (Technical Solution), "Ürün Entegrasyonu" (Product Integration) süreçleriyle bütünleşik ve ürünün yeni sürümü için "Sürüm Yönetimi" ile ilişkili çalışmaları gerekecektir. Bu işlemlere bağlantılı olarak fonksiyonel ve/veya performans testlerinin çalıştırılarak "Doğrulama", "Onaylama" işlemlerinin de tamamlanmış olması gerekmektedir.-

5 Yazılım Sorunları ve Yazılım Performans Mühendisliği

Gerçekleştirilen bir araştırmaya göre, yazılım performans mühendisliği uygulanmadan gerçekleştirilen yazılım projelerinde, performans sorunlarının tespiti ve çözümünün tamamı üretim ortamlarında gerçekleşmektedir. [12]

Buna göre yazılım geliştiren kurumların performans sorun çözüm modelleri

Yaklaşım Metodu	Üretim Ortamında Çözülme Oranı
Sorun gerçekleştiği an	%100
Performans Geçerlemeli	%30
Performans Yönelimli	%5

"Sorun gerçekleştiği an" : Bu yaklaşımda ürün, performans kaygıları duyulmadan üretim ortamına sunulmaktadır. Bu en az tercih edilmesi gereken bir yaklaşım olacağı halde, bu model hala çok yaygındır. Bu şekilde kurumlar ticaret yaşamlarını ciddi riske atmış olmaktadırlar.

"Performans Geçerlemeli" : Yazılım geliştiren kurumlar performans kaygısı içindedir, performans modellemesi ve testlerini gerçekleştirmektedirler. Yazılım performans mühendisliği kısmen fakat yeterli verimlilikte uygulanmamaktadır.

"Performans Yönelimli" : Bu yaklaşımda ise, yazılım performans mühendisliği tam olarak uygulanmaktadır.

5.1 Yazılım Performans Süreci

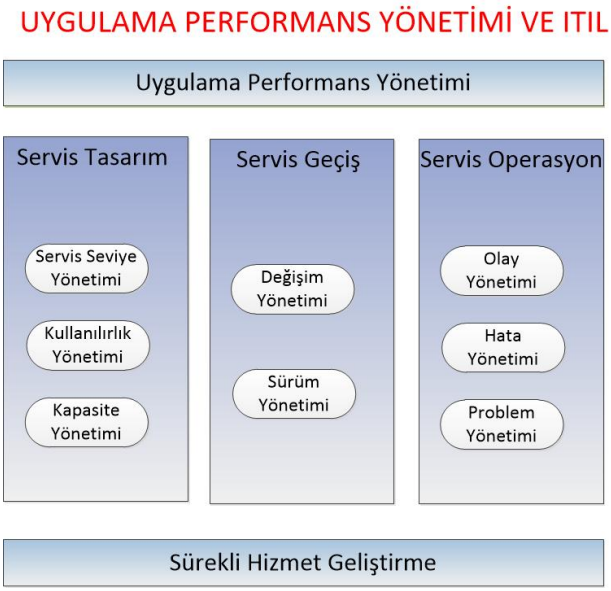
Performansa yönelik bir yapı oluşturmak için, yazılım geliştirme yaşam döngüsüyle bütünleşik bir performans kültürü ve izlenecek bir sürece ihtiyaç vardır. Tasarlanmış bir performans sürecine sahip olunduğu zaman, uygulanacak sistemin, tam olarak nerede başlayacağı, nasıl ilerleneceği ve ne zaman bitirileceği bilinir.

Her kurumun kendine özgü bir performans süreci olabilir fakat genel hatlarıyla değerlendirmek gerekirse;

1. Performans riskleri belirlenmelidir.
2. Kritik kullanım durumları tanımlanmalıdır.
3. Performans senaryoları belirlenmelidir.
4. Performans gereksinimleri belirlenmelidir.
5. Performans modeli oluşturulmalıdır.
6. Performans doğrulanmalı ve geçerlenilmelidir.

olarak maddelenebilir.

Performans yönetim stratejileri birden fazla ITIL süreciyle bağlantılıdır. Uygulama Performans Yönetimi ve Sürekli Hizmet Geliştirme modellerine baktığımızda (Servis Tasarım/Geçiş/Operasyon) bu kavramların birbirlerini tamamladıklarını görürüz. Bu şekilde birlikte oluşturulan hizmet tasarımları ve hizmet sunuşları bilgi teknolojilerinin operasyonel hizmet anlayışında artış ile birlikte kalite çıtasının da yükseltilmesine yardımcı olur.



Şekil 4. Uygulama Performans Yönetimi ve ITIL

6 Sonuç

Java ve .Net gibi yazılım dillerinde otomatik olarak gerçekleştirilen otomatik bellek yönetiminin iyi anlaşılamaması sonucunda sistemlerde oldukça kritik yazılım/donanım performans sorunlarıyla karşılaşılmaktadır. Bunun sonucunda kullanıcı / müşteri mağduriyeti oluşmakta, yazılım proje ekibi prestij kaybı yaşamakta, maddi ve/veya manevi bir çok zarar gerçekleşmektedir. Global anlamda rekabetin kuvvetlenmesi amacıyla, Türkiyede gerek yazılım sektörünün gerekse Üniversitelerin yazılım performans mühendisliğine başvurmaları, gereksinim, analiz/tasarım, kodlama ve test yazılım kalite süreçleriyle bütünleşik, yazılım performans mühendisliği uygulamalarının ihtiyaç olduğu görülmektedir.

Kaynaklar

1. Sommerville, Ian Software engineering / Ian Sommerville. — 9th ed / ISBN-13: 978-0-13-703515-1
2. Sayfa 95 / Longman Dictionary of Contemporary ENGLISH / ISBN : 978-1-4082-1533-3
3. http://www.sei.cmu.edu/reports/90tr003.pdf / 1990 SEI Report on Undergraduate Software Engineering Education

4. http://msdn.microsoft.com/en-us/library/ee787088.aspx / Fundamentals of Garbage Collection
5. http://en.wikibooks.org/wiki/Java_Programming/Object_Lifecycle / Object Lifecycle
6. https://en.wikipedia.org/wiki/Memory_management / Memory management
7. http://www.artima.com/underthehood/gcP.html / Java's Garbage-Collected Heap
8. http://msdn.microsoft.com/en-us/library/ee787088.aspx / What Happens During a Garbage Collection (.NET)
9. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html / Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning
10. http://www.yazgelistir.com/makale/stringbuilder-sinifi-ve-etkinligi / StringBuilder Sınıfı ve Etkinliği
11. http://www.sei.cmu.edu/reports/10tr033.pdf / CMMI® for Development, Version 1.3
12. http://www.sei.cmu.edu/reports/10tr034.pdf / CMMI® for Services, Version 1.3
13. http://www.sei.cmu.edu/reports/10tr032.pdf / CMMI® for Acquisition, Version 1.3
14. http://it-ebooks.info/book/323/The_art_of_application_performance_testing / Forrester Research