Kullanılabilir Uygulama Programlama Arayüzleri

Burcu Ardıç[1] Mehmet Göktürk[2]

[1]TÜBİTAK, Ulusal Elektronik ve Kriptoloji Araştırma Enstitüsü, Kocaeli

[2]Gebze Yüksek Teknoloji Enstitüsü, Bilgisayar Mühendisliği, Kocaeli

[1]e-posta: burcu.ardic@uekae.tubitak.gov.tr [2] e-posta: gokturk@gyte.edu.tr

Özetçe

Günümüzde yazılım geliştirme, uygulama programlama arayüzlerinin (UPA) kullanımından bağımsız düşünülemez. Yazılım geliştiricilerin UPA'ları kullanırken yaşadıkları problemler, geliştirdikleri yazılım ürünlerinin kalitesini doğrudan ve olumsuz etkilemektedir. Bu sebeple, kullanılabilirlik göz önünde tutularak geliştirilmiş uygulama programla arayüzlerine ihtiyaç vardır. Mevcut insan-bilgisayar etkileşimi tekniklerinin UPA'lara uyarlanarak, bu arayüzlerin kullanıcı merkezli bir şekilde tasarlanmasını sağlayacak hedeflerin belirlenmesi gerekmektedir. Bununla birlikte, "kullanıcı" olarak "yazılım geliştirici"nin merkeze koyularak ihtiyaçlarının anlaşılması gerekmektedir. Bu çalışmada, geniş bir literatür taraması yapılarak UPA'ların kullanılabilirliğini etkileyen faktörler ve UPA tasarımcılarına yönelik pratikler bir araya getirilmiş ve açıklanmıştır. Ayrıca kullanıcı grafik arayüzlerinin kullanılabilirlik değerlendirilmesinde faydalanılan geleneksel yöntemler, uygulama programlama arayüzlerine uyarlanarak bir değerlendirme çerçevesi ortaya konmuştur.

1. Giriş

Gerekli işlevsellik ve güvenilirliğin sağlanması, zaman ve bütçe planlaması dışında, yazılım tasarımcılarının dikkat etmeleri gereken bir diğer nokta da insan-bilgisayar etkileşimi (İBE) hedeflerinin yerine getirilmesidir. Kullanılabilirlik, kalitenin vazgeçilmez bir parçasıdır. Gündelik yaşamda kullanılan her üründe olduğu gibi, insanın bilgisayarla etkileşimi sırasında faydalandığı tüm programların, sistemlerin ve arayüzlerin de kullanılabilirliği tercih nedenlerinin başında gelmektedir. İşlevsellik, kullanılabilir olmadığı sürece amacına hizmet etmemektedir.

Son yıllarda, yazılım mühendisliği, insan bilgisayar etkileşimi çalışmalarından faydalanarak kullanılabilirlik açısından kayda değer bir yol kat etmiştir. Bu çalışmalar özellikle kullanıcı grafik arayüzleri üzerinde yoğunlaşmıştır. Ancak, insan-bilgisayar etkileşiminin sorumluluk alanı bu kadarla sınırlı değildir. Yazılım projelerinde maliyet kaygısı ile yeniden kullanım talebinin artması, yazılım geliştiricilerini üçüncü parti bileşenleri daha fazla kullanmaya zorlamaktadır. Uygulamalarına entegre ettikleri kütüphanelerin kendilerine sunduğu arayüzler, yazılım geliştiricilerin bilgisayarla etkileşim noktaları haline gelmiştir.

Uygulama Programlama Arayüzü (UPA) [*İngilizce: Application Programming Interface – API*], işletim sisteminin, bir kütüphanenin veya bir servisin diğer programlara sağladığı fonksiyon ve sınıf kümesidir. UPA'ların tasarımı ve geliştirilmesi konusunda literatürde çok sayıda yazılım mühendisliği çalışması olmasına rağmen, bunların çok azı bu arayüzlerin kullanılabilirliğine işaret etmektedir. Diğer taraftan, kullanılabilirlik üzerine çok sayıda insan-bilgisayar etkileşimi çalışması olmasına rağmen, bunların yalnızca küçük bir kısmı UPA'ları kullanarak yazılım geliştirmeye çalışan geliştiricilerin de aynı zamanda birer "kullanıcı" olduğunu hesaba katmıştır. UPA'larda bulunan kullanılabilirlik problemleri, çok sayıda yazılım projesini olumsuz olarak etkilemeye devam etmektedir.

Programlama psikolojisi, 1970 yılından beri üzerinde çalışılan bir alandır. İnsanın yazılım geliştirme becerileri bilişsel psikoloji tarafından araştırılmaktadır. 1987 yılında The Psychology of Programming Interest Group (PPIG) [1], bu araştırmaları organize etmek için kurulmuştur. Buna rağmen, ancak 1996 yılında Steven Pemberton'ın "Programmers are Humans, Too" başlıklı çarpıcı konuşmasıyla "yazılım geliştirici"ye yönelik kullanılabilirlik fikrine dikkat çekilebilmiştir [2]. Makalede de belirtildiği gibi, programcılar da bilgisayarla etkileşim içinde olan insanlardır. Bu etkileşimin araçları ise, başta programlama dilleri ve kütüphaneler olmak üzere, yazılım geliştirme ortamları, test araçları ve editörlerdir. Yazılım geliştirme faaliyetleri de İBE konusuna dahil olmaktadır. Bu nedenle programlama dilleri ve kütüphaneler üzerinde de görev analizi, gereksinim analizi, kullanıcı testleri ve kullanıcı merkezli yinelemeli tasarım yapılmalıdır. Aslında programlama dilleri ve kütüphaneler programcılar için var olmasına rağmen, çoğumuzun aklında bunların bilgisayarlar için olduğu algısı bulunmaktadır. Belki de bu sebeple, insanlar için tasarlanmamakta, bilgisayarlar için tasarlanmaktadır. İnsan faktörü yeteri kadar göz önünde bulundurulmamaktadır [2].

Temel kullanılabilirlik prensiplerinin UPA'lara uygulanması, yazılım geliştiricilerin bilgisayarla etkileşimini daha etkili, verimli ve tatmin edici hale getirecektir. Bu amaç doğrultusunda, çalışma kapsamında yazılım geliştiricinin ihtiyaçları anlaşılmaya çalışılmıştır. Bildirinin 2. bölümünde UPA'ların kullanılabilirliğine etki eden faktörler açıklanmaktadır. 3. bölümde kullanılabilirlik değerlendirmesi çalışması aşamaları anlatılmakta ve 4. bölümde bu konuya ilgi duyan araştırmacılar tarafından yapılabilecek çalışmalar sunulmaktadır.

2. Kullanılabilir Uygulama Programlama Arayüzleri

2.1 Kullanılabilirlik

Kullanılabilirlik, kullanıcıların bir sistemle etkileşimlerinin kolaylığına odaklanmıştır. ISO 9241'de kullanılabilirlik, *"Hedef kullanıcıların gerekli görevleri etkili, verimli ve tatmin edici bir şekilde yerine getirebilmeleri"* olarak tanımlanmaktadır [3].

Kullanılabilirlik tanımında da belirtilen insan-bilgisayar etkileşimi hedefleri şunlardır:

a. Etkili Olma ('Effectiveness'): Kullanıcının yerine getirmesi gereken görevleri eksiksiz ve doğru bir şekilde tamamlayabilmesi için gerekli işlevselliğin sağlanmasıdır.

b. Verimlilik ('Efficiency'): Kullanıcının eksiksiz ve doğru bir şekilde yerine getirebildiği görevlerin, harcadığı fiziksel ve zihinsel kaynaklara (zaman veya fiziksel uğraş gibi) oranıdır.

c. Kullanıcı Tatmini ('User Satisfaction'): Kullanımın rahat ve kabul edilebilir oluşudur. Kullanıcının bilgisayarla etkileşimi süresince yaşadığı sinir bozukluğu ile ters orantılıdır.

2.2 UPA Kullanılabilirlik Hedefleri

Uygulama programlama arayüzlerinin kullanılabilirliğini sağlamak için ulaşılması gereken hedefler, bölüm 2.1'de sıralanan İBE hedeflerinin UPA'lara uyarlanmasıyla elde edilir. Kullanıcının yazılım geliştirici olduğu, ve kullanılan arayüzün bir UPA olduğu durumda İBE hedefleri özelleştirilmiş ve UPA'ların kullanılabilirliği için aşağıdaki hedefler elde edilmiştir:

a. Kullanıcının ihtiyaç duyduğu fonksiyonları sağlaması: Her UPA'nın bir kapsamı vardır. Yazılım geliştiricinin ihtiyaç duyacağı fonksiyonlar, bu kapsama girmelidir. Bu nedenle UPA, konusu dahilinde kullanıcının ihtiyaç duyacağı tüm fonksiyonları sağlamalıdır. Kullanıcının kendisinin gerçeklemek zorunda kalacağı eksiklikler bırakılmamalıdır.

Diğer taraftan, kullanıcının ihtiyaç duyacağı tüm fonksiyonların sağlanması tek başına yeterli değildir. Kullanıcının da sağlanan fonksiyonları algılayabilmesi gereklidir. Farkına varılamayan işlevsellik, pek çok üründe rastlanan bir kullanılabilirlik hatasıdır [4].

b. Kolay öğrenilmesi: Bir UPA ile yazılım geliştirmenin verimli olabilmesi için, o UPA'nın kolay öğrenilebilmesi gerekmektedir. UPA'ların kullanıcılar tarafından keşfedilmesinin kolaylığı ve yardımcı kaynakların varlığı, kolay öğrenilme özelliğini destekler.

c. Kolay hatırlanması: Kullanıcının yazılım geliştirirken ihtiyaç duyacağı sınıflar ya da metotların akılda kalıcı olması gerekmektedir. Uzun süreli hafızadan ('long term memory') getirilen bilgiler, sınıfların isimleri, metotların ait oldukları sınıflar gibi önceden öğrenilmiş, detaylı bilgilerdir. Kısa süreli hafızadan ('short term memory') getirilen bilgiler ise dokümantasyona bakıldıktan hemen sonra hatırlanması gereken, parametrelerin türleri ve sırası gibi, bilgilerdir. Kolay hatırlanma, uzun ve kısa süreli hafızada depolanacak bilgilerin kalıcı olması özelliğidir.

d. Kodun kolay yazılabilmesi: Kullanıcının kod yazarken karşılaşacağı zorlukların engellenmesi özelliğidir. Kullanıcının ek kontroller yapmak zorunda bırakılmaması örnek olarak verilebilir. Bu özellik diğer tüm maddelerle ilişkilidir.

e. Kullanıcı kodunun anlaşılırlığı: Kullanıcının yazdığı kodun okunabilir ve anlaşılabilir olması özelliğidir. Bu özellik sayesinde, bir UPA kullanılarak yazılmış bir kod parçasının değiştirilebilirliği ve yeniden kullanılabilirliği de artacaktır.

f. Yanlış kullanımın zorluğu: Kullanıcının hata yapmasını engelleyici önlemlerin alınması özelliğidir. Hedeflerin en önemlisi belki de bu maddedir. Bir UPA'nın kullanımı kolay, yanlış kullanımı zor olması gereklidir. Aksi halde, kullanıcının yapacağı hatalar UPA'yı kullanmasına karşı direnç göstermesine sebep olacaktır.

2.3 UPA Kullanılabilirliğine Etki Eden Faktörler

Bölüm 2.2'de verilen kullanılabilirlik hedefleri göz önünde bulundurularak yapılan literatür çalışması neticesinde aşağıdaki 11 faktör elde edilmiştir. Bu faktörler, bir ya da birden fazla hedefle ilişkili olduğu için herhangi bir üst gruplandırma ya da sıralama yapılmamıştır.

1. Karmaşıklık: Bir UPA'nın çok fazla fonksiyon içermesi, kullanıcıya esneklik sağlaması açısından iyidir, fakat aralarından seçim yapmak zor olacağından kullanılabilirliği olumsuz etkileyebilir. UPA'nın sunduğu fonksiyon listesi, kullanıcının ihtiyaç duyacağı tüm fonksiyonları içermeli, fakat hatırlama ve öğrenme zorluğu yaratacak kadar da karmaşık olmamalıdır. UPA'nın çok karmaşık olması hata yapma ihtimalini de arttırır. Soyutlama-karmaşıklık dengesi iyi ayarlanmalıdır.

2. İsimlendirme: UPA geliştiricileri, sınıfların, metotların ve parametrelerin isimlendirmesini özenli yapmalıdır. Tutarsız ve ilgisiz isimlendirmeler öğrenim ve kullanım zorluğu yaratır. Başarılı isimlendirme, UPA'nın kolay öğrenilmesi ve hatırlanması için anahtar roldedir [5].

Kullanılan isimler kendini açıklayıcı nitelikte olmalı ve mümkünse gerçek hayatla analoji kurabilmelidir. Çünkü bir UPA tıpkı bir dil gibidir. Kullanıcıların o UPA ile okumayı ve yazmayı öğrenmesi gereklidir. Doğru isimlendirme sağlandığı taktirde, yazılan kod düzyazı kadar rahat okunabilecektir [6]. Kullanıcıyı mümkün olduğunca dokümantasyona sevk etmeyecek şekilde isimlendirme yapılmalıdır.

İsimlendirme sırasında yaygın alışkanlıklara uyulmalı ve UPA kendi içinde tutarlı olmalıdır. Her noktada aynı şeylerin her zaman aynı isimlendirilmesi ve aynı olmayan şeylerin de asla aynı isimlendirilmemesi önem taşımaktadır.

3. Dokümantasyon: Dokümantasyon, kolay ve hızlı öğrenmeyi destekleyen her türlü kaynağı içerir. UPA'lar, kullanıcıların mümkün olduğunca dokümantasyona başvurmayacakları şekilde tasarlanmalıdır [5]. Yine de ayrıntılı bir dokümantasyon sağlanmalı, sık gerçeklenen görevler için kod örnekleri hazırlanmalıdır. Açık kaynak kodlu UPA'larda, kullanıcı kodun içine girebileceği için, kaynak kodun yorumlarla ('comment') desteklenmesine dikkat edilmelidir.

UPA'lar yayınlanmalarının ardından birçok yazılım geliştirici tarafından kullanılır ve bu sayede UPA'ya dair belirli bir tecrübe birikir. Dokümantasyonun, yeni UPA kullanıcılarına yol gösterecek şekilde bu tecrübeden beslenmesi, UPA'nın kullanılabilirliğini olumlu yönde etkileyecektir. "Jadeite" adlı çalışmada [7], UPA'ların mevcut dokümantasyonlarının, internette yayınlanan kullanım bilgilerinden elde edilen çeşitli tavsiyelerle zenginleştirilebileceği gösterilmiştir.

Dokümantasyon konusunda sıkça yapılan bir yanlışlık, bir hatayı ya da eksikliği dokümantasyona ekleyerek sorumluluktan kurtulmaya çalışmaktır [4]. Dokümantasyona güvenerek aksaklıkların düzeltilmemesi kullanılabilirliği olumsuz etkilemektedir.

4. Nesne Yaratılması: Nesne yönelimli programlama dillerinde, nesnelerin nasıl yaratılacağı UPA tasarımındaki en önemli noktalardan birisidir. Fabrika veya inşacı gibi tasarım kalıplarının kullanımı ya da yaratıcı metot kullanılarak nesne yaratılması, tasarım sırasında karar verilmesi gereken konulardandır. Yapılan seçimin kullanılabilirliğe etkisi üzerinde de çalışmalar yapılmaktadır.

Fabrika tasarım kalıbının kullanılabilirliğe etkisi üzerinde yapılan bir deneysel kullanılabilirlik çalışmasında [8], yazılım geliştiricilerin nesneleri fabrikalardan elde ederken zorlandıkları gözlenmiştir. Aynı çalışmada, fabrika tasarım kalıbının çoğu durumda kullanılmaktan kaçınılmasının mümkün ve gerekli olduğu belirtilmektedir.

Nesne yaratılmasında bir diğer önemli karar, sınıfların yaratıcı metotlarının ne kadar iş yapacağıdır. Yaratıcı metotlar için iki temel seçim söz konusudur. İlki, metotların bazı parametreler alarak yarattığı nesne üzerinde ilklendirmeler yapmasıdır. Aşağıda bir örneği görülmektedir [9]:

```
var foo = new FooClass(barValue);
foo.Use();
```

Diğer seçim ise, yaratıcı metodun herhangi bir parametre almaması ve herhangi bir ilklendirme yapmamasıdır.

```
var foo = new FooClass();
foo.Bar = barValue;
foo.Use();
```

Gerekli ilklendirmeler, nesne yaratıldıktan sonra kullanıcı tarafından yapılır.

Yaratıcı metodun aldığı parametrelerin kullanılabilirliğe etkisi yapılan deneylerle karşılaştırılmıştır [9]. Parametreli yaratıcı metodun daha kullanılabilir olduğu hipoteziyle yola çıkılmıştır, çünkü parametreli yaratıcı metodun arkasından atama yapmaya gerek yoktur. Ayrıca, yaratıcı metot aldığı parametreler ile nesnenin nasıl kullanılması gerektiği konusunda kılavuzluk etmektedir. Fakat deneylerde parametresiz yaratıcı sınıflarla hem geliştirme yapmanın, hem de yazılmış kodu okumanın daha kolay olduğu ortaya çıkmıştır. Bu nedenle parametreli metot kullanımından kaçınılmalı, kaçınılamıyorsa da en azından parametresiz bir yaratıcı metot mutlaka kullanıcıya sunulmalıdır.

5. Metot Parametreleri ve Dönüş Tipi: Özellikle arka arkaya aynı türde çok sayıda parametre kullanımı kullanılabilirliği olumsuz etkiler. Kısa süreli hafızada çok miktarda veri tutulamayacağından 3 ve daha az parametre kullanılmaya gayret gösterilmelidir. Ayrıca parametrelerin sırası ve türü tutarlı olmalıdır [6].

Dönüş değerinin başarıyı belirtmesi kullanımı kolaylaştırır. Metodun işini başarıyla yerine getirip getirmediği bazı durumlarda dönüş değerinde kullanıcıya sunulabilir. Bu noktada dönüş değerinin ek kontrollere yol açmaması gerektiği belirtilebilir. (Örneğin, kullanıcın hata almaması için, `null` yerine boş bir dizi dönmek.)

6. Metot Yerleşimi: Nesne yönelimli programlama dillerinde, bir metodun hangi sınıf üzerinde yer aldığı kullanılabilirliği büyük ölçüde etkilemektedir [10]. Jeffrey Stylos ve Brad Myers tarafından yapılan çalışmada, yazılım geliştiricilerinin hiç bilmedikleri bir UPA'yı keşfetmeye büyük çoğunlukla aynı sınıf üzerinden başladıkları gözlenmiştir. Bu durum, insanların UPA'ları öğrenirken doğal bir strateji izlediklerini göstermektedir. Yazılım geliştiriciler başlangıç sınıfında işaret edilmeyen sınıfları daha geç keşfetmektedirler.

Aşağıda iki farklı UPA kullanılarak e-posta göndermek için yazılmış kod parçaları bulunmaktadır. Benzer görünümlere sahip bu iki satır kod öğrenilebilirlik açısından farklı sonuçlar doğurmaktadır.

```
1. mailServer.send(mailMessage);
2. mailMessage.send(mailServer);
```

İlk satırda verilen kod açık ve anlaşılır görünmektedir. Fakat yazılım geliştiricilerin bir e-posta UPA'sını keşfetmeye `MailMessage` sınıfından başladıkları kabul edildiğinde, `MailServer` sınıfına ulaşmaları daha çok zaman alacaktır. Bu sebeple ikinci satırdaki alternatif kodun üretilmesi, daha kolay ve yazılım geliştiricinin zihinsel modeline daha uygundur.

UPA tasarımcıları, gerektiğinde kullanıcı deneyleri yaparak UPA'nın başlangıç sınıfına doğru karar vermeli ve ilgili olabilecek metotları bu sınıf üzerine yerleştirmelidir. Sık gerçeklenen görevlerde kullanılacak diğer sınıfların başlangıç sınıfı tarafından işaret edilmesine özen gösterilmelidir.

7. Hata ve Kural Dışı Durum İşleme: Program akışı içinde hatalar ve kural dışı durumlar ('exception') kaçınılmaz olarak oluşmaktadır. Dikkat edilmesi gereken nokta, hata mesajlarının yeterli bilgi içermesi ve 'exception'ların sadece kural dışı durumlarda fırlatılmasıdır [6]. Diğer taraftan, kural dışı durumlar oluştuğu halde 'exception' fırlatmamak ve bunun yerine hata kodu dönmek metotların kullanılabilirliğini düşürmektedir.

Her aşamada kullanıcının hata yapması zorlaştırılmalıdır. Bu sebeple metot ve alan erişimlerinin sınırlandırılması için gerekli erişim niteliklerinin ('private', 'public' gibi) kullanımlarına dikkat etmek doğru bir pratiktir. Kullanıcı erişiminin şart olmadığı metotlara dışarıdan erişim engellendiğinde hem UPA sadeleşmiş olur, hem de sınıflar arası bağlaşım ('coupling') azalır.

8. Kademeli Sınama: Yazılım geliştiriciler, program henüz tamamlanmasa da, yazdıkları kodu belirli noktalarda deneyerek geri besleme elde etme ihtiyacı duyarlar. Özellikle bir UPA'yı ilk kez kullanan kullanıcı, geliştirme süreci sırasında sıklıkla geriye dönerek kendi geliştirme yöntemlerini sınamaya ve aşama aşama elde ettiği sonuçları değerlendirmeye eğilimlidir. Kademeli olarak gerçekleştirilen bu geriye dönüşler ve değerlendirmeler, UPA'nın öğrenilmesine büyük katkıda bulunur. UPA'lar, henüz tamamlanmamış kod parçalarının denenebilmesine olanak verecek şekilde tasarlanmalıdır [11]. Bu pratik bir örnekle daha rahat anlaşılabilir [4]:

```
IPAddress hostIPAddress =
      IPAddress.Parse("130.23.43.234");
IPAddress theAddress =
      new IPAddress(hostIPAddress);
CEmailServer theServer =
      new CEmailServer (theAddress);
if (theServer.Connect ())
{
      CMailbox theMailbox =
        theServer.GetMailbox ("stevencl");
      theMailbox.Open("stevencl", true);
      CEmailMessage theMsg =
        theMailbox.NewMessage();
      theMsg.Build("message", " subject",
        "to@you.com");
      theMailbox.Send (theMsg, true);
      theMailbox.Close();
}
else
      Console.WriteLine ("Could not connect to
        the mail server");
```

Yukarıda, bir UPA kullanılarak yeni bir e-posta oluşturulması ve gönderilmesini sağlayan bir kod parçası verilmiştir. Bu UPA'nın kullanıcıları için, senaryo tamamlanmadan ilerlemelerini kontrol etmeleri zordur. Çünkü sunucuya bağlanıp posta kutusunu açmadan bir `CEmailMessage` nesnesi yaratmak mümkün değildir. Bu sebeple bu UPA'nın kademeli sınamayı desteklediği söylenemez. Oysa UPA aşağıdaki kullanıma uygun olarak tasarlanmış olsa, kademeli olarak sınanabilecektir:

```
CEmailMessage theMsg = new CEmailMessage();
theMsg.Build("message", " subject",
       "to@you.com");
/* ** CEmailMessage nesnesi bu aşamadan
itibaren sınanabilir ** */
IPAddress hostIPAddress =
      IPAddress.Parse("130.23.43.234");
IPAddress theAddress =
      new IPAddress(hostIPAddress);
CEmailServer theServer =
new CEmailServer (theAddress);
if (theServer.Connect ())
{
      CMailbox theMailbox =
        theServer.GetMailbox ("stevencl");
      theMailbox.Open("stevencl", true);
      theMailbox.Send (theMsg, true);
      theMailbox.Close();
}
else
      Console.WriteLine ("Could not connect to
        the mail server");
```

Kademeli sınama sayesinde, nihai ürüne varana kadar, gerçeklenen tüm program bileşenlerinin gözden geçirilmesi ve sınanması sağlanmış olur.

9. Çağrı Sonrası Anlaşılabilirlik: UPA metotları çağrıldıktan sonra anlaşılabilirliklerini koruyacak şekilde tasarlanmalıdır. Çözülmek istenen probleme UPA kullanıcısının perspektifi ile yaklaşılmalıdır. Bu özellik basit bir örnekle açıklanabilir. Aşağıda, özenle isimlendirilmiş bir metot tanımlaması yer almaktadır [5]:

```
void makeTV(
     bool isBlackAndWhite,
     bool isFlatScreen)
{/* ... */ }
```

Fakat, bu metot çağrı sonrası anlaşılabilirliğini yitirmektedir. Renkli ve düz ekran bir TV yaratmak isteyen bir kullanıcı aşağıdaki çağrıyı yapacaktır:

```
makeTV(false, true);
```

Bu satırda bir TV yaratıldığı isimlendirmeden anlaşılmaktadır. Fakat parametrelerin ne anlama geldiğini kestirmek olanaksızdır. Aşağıda çok daha anlaşılabilir bir çağrı bulunmaktadır:

```
makeTV(Color, FlatScreen);
```

İkinci alternatifte renkli ve düz ekran bir TV yaratıldığı anlaşılabilmektedir. Hem çağrı sonrası anlaşılabilirlik artmış, hem de hata yapma ihtimali azaltılmıştır. Farkı yaratan aşağıdaki `enum` tanımlamalarını kullanan yeni metot imzasıdır:

```
enum ColorType {
     Color,
     BlackAndWhite };

enum ScreenType {
     CRT,
     FlatScreen };

void makeTV(ColorType col, ScreenType st);
```

UPA tasarımcısı açısından, kullanıcı perspektifini göz önünde bulundurmak bir miktar daha fazla çaba gerektirebilir, fakat sonuçta daha kullanılabilir bir UPA ortaya çıkacaktır.

10. Kaynak Kodunun Anlaşılabilirliği: Özellikle açık kaynak kodlu UPA'lar için, kaynak kodunun anlaşılabilirliği önemlidir. Kod okunabilir yazılmış olmalı ve kullanılan programlama dilinin geleneksel yazım standartlarına uyulmalıdır. Girintilemeye ('indentation') ve kod bloklarını dilin ayraçlarını kullanarak ayırmaya özen gösterilmelidir. Tekrar eden kod parçaları, çok büyük sınıflar ve uzun metotlar, uzun parametre listeleri kodun okunabilirliğini azaltan ve yazılım kalitesini düşüren unsurlar sayılır [12]. Bu özelliğin sağlandığını kontrol eden, hatta bazı ufak düzeltmeler yapan yazılımlar mevcuttur, bunlardan faydalanılabilir (Bkz. [13]).

11. UPA'larda Değişiklik: UPA'lar kullanılmaya başlandıktan sonra değişiklik yapılması istenen bir durum olmasa da kaçınılmazdır. Bu değişikliklerin mümkün olduğunca seyrek yapılması gerekliliği bir tarafa, en önemli ayrıntı, bir UPA'da yapılan değişikliğin, o UPA'nın eski versiyonunu kullanan programlarda bir uyumsuzluğa sebep olmamasıdır. Bir değişiklik söz konusu ise, bu zorunluluk zamana yayılabilmeli, önceden yazılmış kodlar "geçersiz" duruma düşmemelidir. Java programlama dili, geriye dönük desteğin mümkün olduğunun en güzel ispatıdır. Java'nın tüm eski versiyonlarında yazılmış tüm kodlar, şu anda Java'nın son sürümü olan 1.6'da halen çalışabilmektedir.

Ayrıca, bir UPA'dan tamamen desteğin kesilerek yepyeni bir UPA ile kullanıcı karşısına çıkmak da büyük zorluklara yol açmaktadır. Bu nedenle UPA'lar öldürülmemeli, yavaş yavaş yaşlandırılmalıdır [14].

3. UPA'larda Kullanılabilirlik Değerlendirmesi

UPA'ların teknik olarak değerlendirilmesi, daha sık yapılan ve daha somut sonuçları olan bir çalışmadır. Diğer taraftan, kullanılabilirlik açısından değerlendirilmesi, çok gerekli olmakla birlikte, henüz oturmuş yöntemler bulunmadığı için teknik değerlendirmeye kıyasla daha zordur.

UPA'ların kullanıcı merkezli bir şekilde tasarımı için ilk defa kullanıma sunulmadan önce mutlaka ayrıntılarıyla değerlendirilmesi ve elde edilen bulgulara göre adımlar atılması gerekmektedir. Ayrıca UPA'lar üzerinde yapılacak değişikliklerin yayınlanmasından önce de, değişikliklerin kullanılabilirliğe zarar vermediğinden emin olmak için değerlendirme yapılmalıdır.

3.1 Sezgisel Değerlendirme

Sezgisel değerlendirme ('heuristic evaluation'), kullanıcı arayüzünün tasarımındaki problemleri ortaya çıkarmak için kullanılan geleneksel yöntemlerden biridir. Kullanıcı arayüzü, uzman bir değerlendirici tarafından, belirli bir kriter kümesine göre değerlendirilerek o kriterlere uyum ölçülür. Bu yöntem, kullanıcı grafik arayüzleri için 10 kriter ile birlikte, kullanılabilirlik uzmanı Jacob Nielsen tarafından ortaya konmuştur [15]. Sezgisel değerlendirme, ciddi kullanılabilirlik hatalarının tasarım aşamasında yakalanarak düzeltilmesine olanak tanır. Kullanıcı davranışlarının yorumlanmasını gerektirmediği ve belirli bir kriter kümesine dayandığı için kolay ve hızlıdır. Tasarım aşamasında kullanılabilirlik hatalarını teşhis edebildiği için maliyet açısından avantajlıdır. Diğer taraftan değerlendiricilerin taraflı veya yetersiz olmaları durumunda hata oluşabileceğinden kullanıcı deneyleriyle birlikte kullanılmalıdır.

Kullanıcı grafik arayüzleri değerlendirmesinde kullanılan sezgisel değerlendirme yöntemi, uygulama programlama arayüzlerinin değerlendirilmesinde de kullanılabilir. Bu çalışmada UPA'ların sezgisel değerlendirilmesinde kullanılmak üzere Bölüm 2.3'te verilen faktörlerin kriter kümesi olarak kullanılması önerilmektedir.

3.2 Kullanıcı Deneyleri

İnsan-bilgisayar etkileşimi hedeflerini yerine getirmek için en önemli nokta, kullanıcının zihinsel modelinin doğru anlaşılmasıdır. Bu amaçla, kullanıcı ile insan-bilgisayar etkileşimi laboratuarında ya da kullanıcının kendi çalışma ortamında bazı deneyler yapılır. Deneye katılan kullanıcılara sembolik de olsa bir karşılık verilir (bir kitap ya da lisanslı bir yazılım olabilir.). Denekler, test edilen UPA'nın alanında daha önce çalışmamış, fakat programlama diline hakim programcılar arasından seçilmelidir. Bu çalışmalar 3 aşamalı olarak tasarlanmıştır:

3.2.1. Sesli Düşünme

Sesli düşünme protokolü ('think aloud protocol'), kullanıcı grafik arayüzlerinin kullanılabilirlik açısından değerlendirilmeleri için yapılan deneyler sırasında, kullanıcıdan doğru veriyi toplayabilmek için sıkça başvurulan bir yöntemdir. Kullanıcı kendisine verilen görevleri gerçeklerken aklından geçenleri sesli ifade ederek, zihinsel modelindeki ürün ile değerlendirilen ürün arasındaki farkı ortaya çıkarır.

Sesli düşünme yöntemi, UPA değerlendirmesine uyarlanırken bir miktar genişletilir. Görev gerçekleme aşamasından önce, deneklere bazı programlama görevleri verilir. Uygulama programlama arayüzü ya da herhangi bir dokümantasyona erişim sağlanmaz. Deneklerden verilen görevleri yerine getirmeleri için 'pseudo-code' yazmaları istenir [16]. Böylece, kullanıcı beklentileri ilk olarak *yazılı* olarak ifade edilmiş olur. İsimlendirme, karmaşıklık, parametre türleri ve sırası, nesne yaratımı, metot yerleşimi gibi başlıklarda beklentiler elde edilebilir. Bu bilgilerle UPA'nın kullanıcının zihinsel modeline ne kadar uyduğunun değerlendirmesi yapılır. Yazılı ifadenin ardından, görev gerçekleme aşamasına geçilir (Bkz. Bölüm 3.2.2). Görevler gerçeklenirken sesli düşünme protokolü uygulanır. Kullanıcı, karşılaştığı problemleri ya da düşünmekte olduğu şeyleri kendi cümleleri ile ifade eder.

3.2.2. Görev Gerçekleme

Bu aşamada deneklerden Sesli Düşünme bölümünde bahsedilen görevleri, UPA'yı kullanarak gerçeklemeleri istenir. Görev gerçekleme aşaması boyunca kamera ile kayıt yapılır. Bu yöntemle aşağıdaki veriler toplanabilir [17]:

a. Öğrenme zamanı: Kullanıcıların verilen görevleri yerine getirmek için gerekli sınıflar ve metotlar ile bunlar arasındaki ilişkiyi öğrenmeleri için geçen zaman.

b. İşlem hızı: Kullanıcıların verilen görevleri yerine getirmek için harcadıkları zaman.

c. Hata sayısı: Kullanıcıların verilen görevleri yerine getirirken yaptıkları hataların türleri ve sayısı. (Kullanıcılar hata yaparken işlem hızları da olumsuz etkilenecektir. Fakat, kullanıcıların yaptıkları hatalar, kullanılabilirlik problemlerine doğrudan etki ettiği için hataların da ayrıca değerlendirilmesi gerekmektedir.)

d. Hatırlama oranı: Kullanımın üzerinden geçen belirli bir süre sonunda, öğrenilenlerin ne kadarının akılda kaldığı.

Deneye başlamadan önce her görev için ölçülebilir hedefler koyulur. Deneylerin tamamlanmasıyla elde edilen veriler, hedeflenenlerle karşılaştırılarak iyileştirmeye gerek olup olmadığına karar verilir. Ayrıca, kaydedilen görüntüler daha sonra incelenerek kullanıcıların yaşadıkları zorluklar ve dokümantasyona başvurdukları noktalar belirlenir.

3.2.3. Anket

Kullanıcı tatminini ölçmek için doğrudan kullanıcıdan görüş almak etkili bir yöntemdir. Verilen bir dizi görev, değerlendirilen UPA ile gerçeklendikten sonra, kullanıcıya bir anket sunularak memnuniyet derecesi anlaşılmaya çalışılır. Bir kullanıcı için düşünüldüğünde verilen yanıtın öznel olacağı ve sağlıklı sonuç vermeyeceği doğru olsa da, tüm deneklerden alınan sonuçların ortalaması alındığında kullanıcı tatmini ile ilgili nesnel veri elde edilir [18].

Cognitive Dimensions (CD), sistemler üzerinde hızlı ve etkili kullanılabilirlik değerlendirmesi yapabilmek için Thomas Green tarafından ortaya konmuş bir çatıdır [11]. UPA'ların kullanılabilirlik problemlerini ortaya çıkarmak için kabul görmüş ve uygulanmıştır [19]. CD, her biri kullanılabilirliği etkileyen 13 boyuttan oluşur. Bu 13 boyut, sezgisel değerlendirme sırasında bir uzman tarafından kullanılabileceği gibi, kullanılabilirlik anketlerinin oluşturulmasında da çerçeve görevi görmüştür [20]. Deneyin bu aşamasında kullanıcıdan, CD'deki her bir boyutla ilgili düşüncelerini ortaya çıkarıcı nitelikte bir anketi yanıtlaması istenir. Böylelikle 13 boyut üzerinden görev gerçekleme aşamasında gözden kaçırılan, fakat kullanıcının zorlandığı veya eksik bulduğu bir durum varsa yakalanmış olur.

4. Sonuçlar

Yazılım sektöründe hem iş hacmi hem de yapılan işlerin karmaşıklığı gün geçtikçe artmaktadır. Her geçen gün yeni gereksinimler ortaya çıkmakta ve yeni iş süreçleri elektronik ortama taşınmaktadır. Bu koşullar altında yeniden kullanım kaçınılmaz hale gelmektedir. Günümüzde bir yazılımcı, program geliştirirken her özelliği kendisi gerçeklemeye gerek duymamaktadır. İşin önemli bir kısmı var olan fonksiyonları kullanmak ve birleştirmektir. Üretilen ve ihtiyaç duyulan yazılım miktarının artmasıyla, UPA'lar ile var olanın yeniden kullanılması daha da artacaktır. Fakat, bir UPA kullanılabilir değilse, yeniden kullanılabilir de olamaz.

UPA'ları kullanan insanların sayısı, geliştiren insanların binlerce katına ulaşmaktadır. Bu nedenle, kullanıma girmesinden sonra bir UPA'yı değiştirmek, çok sayıda projeyi etkileyeceğinden, oldukça zordur. Kullanılabilirlik hesaplarını yapmak için en doğru zaman tasarım aşamasıdır. UPA geliştiricilerinin tasarım aşamasında göz önünde bulunduracakları prensiplerin bulunması ve geliştirilen UPA'ların kullanıma sunulmadan önce kullanılabilirlik değerlendirmesi yapılabilmesi büyük önem taşımaktadır.

Çalışma kapsamında, yazılım geliştiricilerin bilgisayarla etkileşimini etkili, verimli ve tatmin edici kılmak, bir başka deyişle insan-bilgisayar etkileşimi hedeflerini yerine getirmek için UPA tasarımında dikkat edilmesi gereken faktörler açıklanmıştır. Bunun yanı sıra, tasarlanan UPA'ların değerlendirilebilmeleri için bir çerçeve sunulmuştur. Sunulan öneriler ve değerlendirme çerçevesi, uygulama programlama arayüzleri için ortaya konulmuş olsa da, yeniden kullanılan her programlama birimine uygulanabilir. Çalışmanın, UPA tasarımcılarına yardımcı olması ve yazılım kalitesi konusunda çalışan araştırmacılara ilham vermesi umulmaktadır.

5. Gelecek Çalışmalar

Çalışma kapsamında, UPA'ların kullanılabilirliğini etkileyen faktörler geniş bir literatür taraması sonucunda elde edilmiş ve listelenmiştir. Çok sayıda ve farklı UPA'lar üzerinde kullanıcı deneyleri yapılarak bu faktörlerin geçerlenmesi önemlidir. Çünkü uygulama programlama arayüzlerinin kullanılabilirliği üzerindeki çalışmalar henüz yeterince olgunlaşmamıştır. Konu üzerinde yapılacak çalışmalarla şüphesiz yeni faktörler ortaya çıkacaktır. Kullanılabilir UPA'lar geliştirmek için, akademik ve endüstriyel çevrelerden kabul gören metodolojilere ihtiyaç vardır.

Bildiride sunulan faktörler açıklanırken, yerine getirilmesi gereken pratiklere değinilmiştir. Bu pratiklerin, tasarım kılavuzları haline getirilmesi, UPA tasarımcılarına doğrudan kullanabilecekleri kaynaklar sunacaktır. Ayrıca, söz edilen pratiklerin bir kısmı, yazılım geliştirmede kullanılan tasarım kalıplarına entegre edilebilir. Böylece UPA'larda kullanılabilirlik pratikleri çok daha geniş çevrelerce uygulanmış olacaktır.

6. Kaynakça

[1] Psychology of Programming Interest Group, http://www.ppig.org.

[2] Pemberton, S., "Programmers are Humans, Too", *ACM SIGCHI Bulletin*, 1996.

[3] "ISO/IEC 9241-11 Ergonomic requirements for office work with visual display terminals (VDT)s - Part 11 Guidance on usability", *ISO/IEC 9241-11*, 1998.

[4] Clarke, S., "Measuring API Usability", *Doctor Dobbs Journal*, 2004.

[5] Henning, M., "API Design Matters", *Communications of the ACM*, Vol 52, Sayfa: 46-56, 2009.

[6] Bloch, J., "How to design a good API and why it matters", OOPSLA'06*:* Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Sayfa: 506-507, 2006.

[7] Stylos, J., Myers, B. A. ve Yang, Z., "Jadeite: improving API documentation using usage information", *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, Sayfa: 4429-4434, Nisan 2009.

[8] Ellis, B., Stylos, J. ve Myers, B., "The Factory Pattern in API Design: A Usability Evaluation", *International Conference on Software Engineering*, 2007.

[9] Stylos, J. ve Clarke, S, "Usability Implications of Requiring Parameters in Objects' Constructors", *Proc. ICSE, ACM Press*, Sayfa: 529-539, 2007.

[10] Stylos, J. ve Myers, B. A., "The Implications of Method Placement on API Learnability", *Symp. on the Foundations of Software Engineering FSE '08*, 2008.

[11] Green, T. R. G. ve Petre, M., "Usability analysis of visual programming environments: A 'cognitive dimensions' framework". *Journal of Visual Languages and Computing*, Sayfa:131–174, 1996.

[12] Fowler, M, Beck, K., "Bad smells in code" in Refactoring: Improving the Design of Existing Code, *Addison-Wesley*, 2000.

[13] Checkstyle, http://checkstyle.sourceforge.net.

[14] Zibran, M. F., "What Makes APIs Difficult to Use?", *IJCSNS International Journal of Computer Science and Network Security*, VOL.8 No.4, Nisan 2008.

[15] Nielsen, J. ve Molich, R., "Heuristic evaluation of user interfaces", *Proc. ACM CHI'90 Conf.*, Sayfa: 249-256, 1990.

[16] Beaton, J., Myers, B. A., "Usability Evaluation for Enterprise SOA APIs", *SDSOA'2008*, 2008.

[17] Shneiderman, B., "Designing the User Interface", *Addison Wesley*, 3. Baskı, 1997.

[18] Nielsen, J., "Usability Engineering", *Academic Press, San Diego*, Sayfa: 34-37, 1993.

[19] Clarke, S., "Describing and Measuring API Usability with the Cognitive Dimensions", *Cognitive Dimensions of Notations 10th Anniversary Workshop*, 2005.

[20] Blackwell, A. ve Green, T., "A cognitive dimensions questionnaire optimised for users", *Proceedings of the 12th Annual Meeting of Psychology Programming Interest Group*, Sayfa: 137-152, 2000.