Yazılım Yapılandırma Teknikleri: Temizer Sistemi

Selim TEMİZER

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Artificial Intelligence Laboratory
Cambridge, Massachusetts, USA

e-posta: temizer@alum.mit.edu

Özet

Bu dökümanda, yazılım geliştirme, test etme ve bakım süreçlerine sistematiklik, kolaylık ve hız kazandırmak üzere tasarlanmış olan ve *Temizer Sistemi* olarak adlandırılan bir grup yazılım tekniği anlatılmaktadır. Sistem temel olarak iki kısımdan oluşmaktadır. İlk kısımda, bir yazılımı oluşturan kod parçalarının mantıksal olarak nasıl gruplanabileceği ve bu grupların fiziksel olarak nasıl korunabileceği ile ilgili çalışmaların sonuçları yer almaktadır. İkinci kısımda ise yazılımı oluşturan temel programlama ünitelerinin yapısı ile ilgili tavsiyeler bulunmaktadır. Temizer Sistemi ilk olarak C programlama dili kullanılarak geliştirilen projeler için tasarlanmış olup, daha sonra sistemdeki temel prensipler genelleştirilerek, kullanılan dilden bağımsız olarak her türlü yazılım projesine uygulanabilir hale getirilmiştir.

Abstract

This document describes a set of techniques collectively named as *Temizer System*, which are designed to provide systematic, easy and fast ways of software development, testing and maintenance. The system consists of two parts. The first part deals with logical decomposition of software and elaborates on how to reflect that logical structure physically. The second part contains recommendations on how to structure the basic programming units. Temizer System was originally designed for projects in which the C programming language is used, and the basic principles are then generalized to be applicable to any project independent of the programming language used.

1. Giriş

Yazılım projelerinin hacmi büyüdükçe, yazılım süreci boyunca sistematik yazılım geliştirmeyi ve test süreçlerini mümkün olduğunca destekleyen, kolaylaştıran ve yazılımın büyüklüğünden etkilenmeyen mimariler ve dizayn ilkeleri kullanmanın önemi daha da fazla belli olmaktadır. Belli bir yapı ve düzenin takip edilmediği projelerde küçük yazılım geliştirme takımları bile kısa süreler içinde hacim olarak çok büyük olmasına rağmen anlaşılması çok zor olan, son derece karışık ve bakımı ve hatalardan arındırılması da hemen hemen imkansız olan yazılımlar üretebilirler. Bu tip yazılımların bakım maliyetleri çoğu zaman benzer bir yazılımın yeniden oluşturulmasının maliyetinden daha fazladır. Ayrıca zayıf mimarilerin kullanıldığı projelerde, üretilen parçaların ana sistem ile entegrasyonunun sağlanması sırasında da sorunlar yaşanması neredeyse kaçınılmazdır.

Bu tip sorunlar sadece yazılım geliştirme takımlarını değil, daha önceden başkaları tarafından geliştirilmiş yazılımların hata ayıklamalarını ve bakımlarını yapan takımları da ilgilendirmektedir

ve onların da bu sorunlarla karşılaşmamak için önceden belirlemiş olmaları gereken belli mimariler ve metodlar kullanarak çalışmaları gerekmektedir.

Bu dökümanda, yukarıda bahsedilen sorunları çözmek üzere tasarlanmış olan ve *Temizer Sistemi* olarak adlandırılan bir sistem anlatılmaktadır. Temizer Sistemi, C programlama dilinin kullanıldığı projelere çözüm sunmak amacı ile dizayn edilmiş olup, daha sonra sistemdeki temel prensipler genelleştirilerek herhangi bir programlama dilinin kullanıldığı yazılım projelerinde kullanılabilecek hale getirilmiştir. Bu dökümanda sistemin özelliklerinin çoğunu kapsayabilmesi açısından önişlemci özelliği bulunan C programlama dili kullanılarak hazırlanmış örnekler sunulacaktır.

Temizer Sistemi temel olarak iki tip mimariden oluşmaktadır. Bunlardan ilki, projemize tepeden bir bakış yapmamızı sağlayarak bir yazılımı oluşturan bütün elemanları mantıksal olarak nasıl gruplayabileceğimizden ve fiziksel olarak nasıl saklayabileceğimizden (dosyalara ve klasörlere bölerken kullanabileceğimiz ana fikirlerden) bahsederken, ikinci tip mimari ise bu dosyaların içlerinde bulunan temel programlama ünitelerini (yani kullanılan programlama diline göre fonksiyonlar, prosedürler, sınıflar, vs.) dizayn ederken kullanabileceğimiz teknikleri kapsar.

Dökümanın devamında, önce Temizer Sistemi'nin ilk dizayn hedefi olan ve örnek olarak seçtiğimiz C programlama dilinin sistem açısından önemli olan özelliklerini kısaca gözden geçireceğiz. Daha sonra Temizer Sistemi'ni detaylı olarak inceleyeceğiz. Ve son olarak da sistemin diğer dilleri kapsayacak şekilde nasıl genişletilebileceği, avantajları ve dezavantajları konusunda yorumlarda bulunacağız.

2. C Programlama Dilinin Özellikleri

Yazılım ürünlerini iki kategoriye ayırabiliriz: birincisi, bağımsız olarak çalışmak üzere tasarlanmış uygulamalar, diğeri ise başka projelerde kullanılmak üzere hazırlanmış kütüphaneler. Bağımsız uygulamalarda, aşağıda bahsedeceğimiz yapılardan birer küme bulunmasına rağmen, kütüphane yazılımlarında bazı yapılardan ikişer küme bulunur. Bu kümelerden bir tanesi kütüphanenin, kendisini kullanacak olan sisteme sunduğu servisler için gerekli olan yazılımları içerir. Diğer kümede ise kütüphane yazılımının kendi iç kullanımına özel parçalar bulunur. Ayrıca bizim kütüphanemiz de başka kütüphane yazılımlarına ihtiyaç duyuyor olabilir. Temizer Sistemi'ni anlatırken vereceğimiz örneklerde, daha geniş bir kapsamı olması açısından bir kütüphane yazılımı yaptığımızı varsayacağız. C programlama dilinin kullanıldığı bir projede şu yapı grupları bulunur:

- **Derleyici parametreleri** – Önişlemci makroları ile aynı teknik özelliklere sahiptirler, fakat kullanım amaçları farklıdır. Kodun bir kısmını derleme aşamasında aktive etmek veya deaktive etmek amacı ile kullandığımız yapılardır. En yaygın örnekleri arasında DEBUG, NDEBUG vs. gibi, hataları bulabilmek amacı ile yazdığımız kısımları (testler sırasında) kodda bırakmak için veya (testler bitince) koddan çıkarmak için kullandığımız tanımlamalar bulunur.
- **Önişlemci makroları** – Bu makrolar “`#define PI 3.14`“ şeklinde parametresiz olabileceği gibi “`#define MAX(a,b) ((a)>(b)?(a):(b))`“ şeklinde parametreli de olabilirler. Önişlemci tarafından, kod derleyiciye girmeden önce, bu makroların gerekli şekilde açılımları yapılır.
- **Veri Tipleri** – C dilinin sunduğu (char, int, float, vs. gibi) temel veri tiplerine ek olarak, *typedef*, *struct*, *union*, işaretçiler ve dizileri kullanarak kendi veri tiplerimizi de oluşturabiliriz.
- **Değişkenler** – Bu kategoride bahsettiğimiz değişkenler global olan değişkenlerdir. Çeşitli bilgileri depolamak için global değişkenlerden faydalanabiliriz.
- **Fonksiyonlar** – C programlama dilindeki temel programlama üniteleridir.

Yukarıda bahsettiğimiz yapılardan global değişkenler ve fonksiyonlar, C dilindeki *objeler* olarak da adlandırılırlar [1]. Buradaki obje terimini, C++ ve Java gibi nesneye yönelik programlama dillerindeki objeler ile karıştırmamalıyız. Bu dökümanda örnek olarak C programlama dilini seçtiğimiz için, aksi belirtilmedikçe obje terimi global değişkenleri ve fonksiyonları ifade etmektedir.

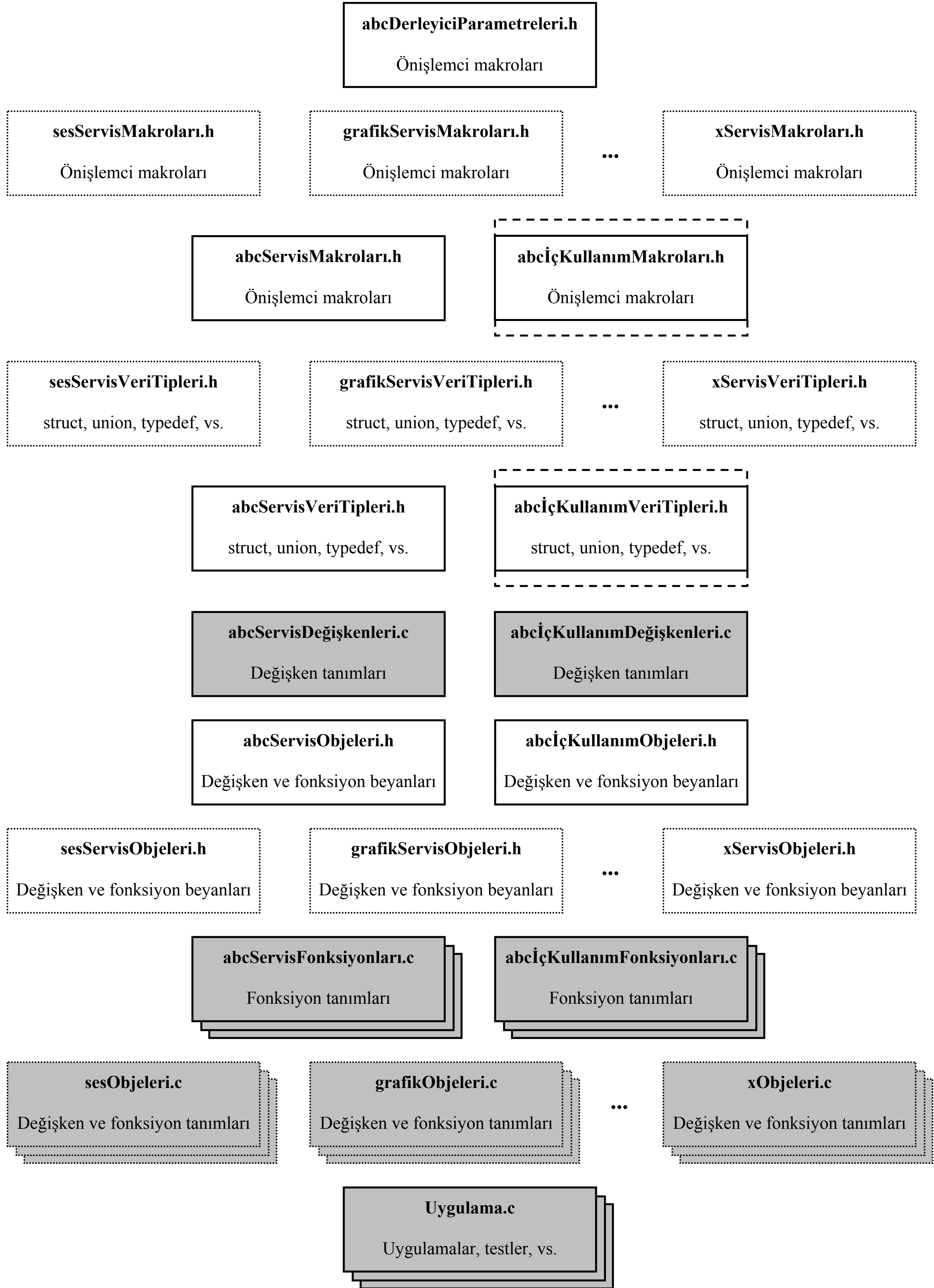
3. Dosya Mimarisi

C programlama dili kullanılarak oluşturulacak bir yazılım için Temizer Sistemi'nde önerilen dosya mimarisi, temel olarak, bir önceki bölümde bahsettiğimiz yapıların mantıksal gruplamasının fiziksel olarak da korunmasını öngörür. Bu bölümde örnek olarak bir *bilgisayar oyunu motoru* yazılımı yaptığımızı varsayalım. Oyun motorumuzun amacı, basit ses ve grafikler için (ve başka herhangi bir *x* yeteneği için) bazı hazır kütüphaneleri kullanarak daha komplike fonksiyonlar oluşturmak ve diğer uygulamalara servis olarak bu fonksiyonları sunmak olsun. Oyun motorumuza da *abc oyun motoru* adını verdiğimizi düşünelim. *Namespace* gibi özelliklerin bulunmadığı dillerde olası isim karışıklıklarını önlemek için genelde bir projeye ait olan değişken, fonksiyon ve hatta dosyaların isimlerinin başına kısa bir ön ek getirmek, kullanılan çözüm tekniklerinden birisidir. Biz de bu örneğimizde *abc* ön ekini kullandığımızı varsayalım. Böyle bir yazılım örneği için Temizer Sistemi'ni kullanırsak, yazılımımızı oluşturacak olan dosyalar ve bu dosyaların içereceği yazılım parçalarını Şekil 1'deki gibi ayırabiliriz. Şekil 1'deki hiyerarşik yapının özellikleri şunlardır:

- Her kutu bir dosyayı temsil eder. Dosyalara verilebilecek olan birer isim tavsiyesi, ve dosyaların içermesi gereken yazılımlar kutuların içlerinde belirtilmiştir.
- Objelerin tanımlarını içeren dosyaları temsil eden kutular gri renk ile doldurulmuştur.
- Kenarları düz çizgilerden oluşan kutuların temsil ettiği dosyalar, proje kapsamında bizim hazırlamamız gereken yazılımlardır. Kenarları noktalardan oluşan kutuların temsil ettiği dosyalar ise bizim ihtiyaç duyduğumuz ama dışarıdan hazır olarak alacağımız yazılımlardır.
- Bizim hazırlamamız gereken bir dosya, sadece kendisinden hiza olarak yukarıda bulunan dosyalardaki bilgilere ihtiyaç duyabilir (ve ihtiyaç duyarsa `include` edebilir). Kendisinden daha aşağıdaki dosyalardaki bilgilerin hiçbirisine ihtiyaç duymaz. Bu noktada, dikkat edilirse *iç kullanım makroları* için çizilmiş olan kutunun büyüklüğü pek kesin değildir ve sınırlarının genişleyebileceği kesikli çizgilerle ifade edilmeye çalışılmıştır. Bunun sebebi ise, bazı durumlarda bazı servis makrolarını tanımlarken, iç kullanıma özel makrolardan faydalanmak, veya bunun tersi gerekebilir. Bu yüzden bu iki kutunun tam olarak hizası, projeden projeye değişebilir. Benzer bir durum da servis veri tipleri ve iç kullanıma özel veri tipleri için söz konusudur ve orada da kesikli çizgiler, yine sınırların genişleyebileceğini göstermektedir.
- Mümkünse her temel programlama ünitesi kendine ait bir dosyada tek başına bulunmalıdır.
- En alt sıradaki kutu(lar), hazırlamış olduğumuz kütüphaneyi test etmek için oluşturabileceğimiz yazılımları veya kütüphanemizi kullanan uygulamaları temsil etmektedir.

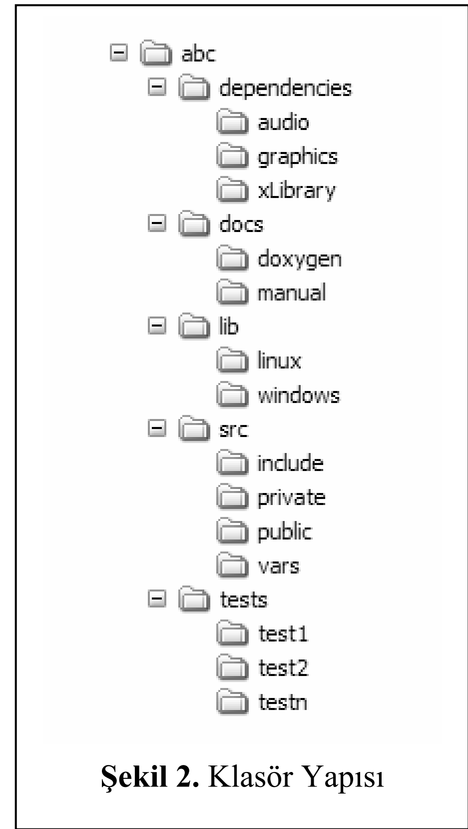
Yazılımımızı içeriklerine göre bu şekilde dosyalara ayırdıktan sonra, Temizer Sistemi bu dosyaları da yine mantıksal olarak tutarlı bir klasör yapısında saklamayı öngörür. Bu proje için önerilen bir klasör yapısı Şekil 2'de gösterilmektedir.

Yazılımımızın ihtiyaç duyduğu kütüphaneler, bizim hazırladığımız kullanma kılavuzları (veya Doxygen [2] gibi otomatik dökümantasyon üreten yazılımların çıktıları), kütüphanemiz için (gerekiyorsa değişik işletim sistemleri için ayrı ayrı) oluşturduğumuz yeniden derlemeksizin kullanılabilecek hazır makina dili kodları, yazılımımızın kaynak kodları, ve test yazılımları için ayrı ayrı oluşturacağımız klasörler, bilgilerin düzenli bir yapıda saklanmasını sağlayacaktır.



Şekil 1. Dosya Mimarisi

Kaynak kodlarımızı, obje tanımı içeren ve içermeyen dosyalar olarak gruplayabiliriz. Obje tanımı içeren dosyaları da değişkenler ve fonksiyonlar olarak gruplamak mümkündür. Ve son olarak fonksiyonlarımızı da servis fonksiyonları ve iç kullanıma özel fonksiyonlar olarak gruplayabiliriz. Şekil 2'deki klasör yapısında kaynak kodlarımız için bu tarz bir gruplama örneği de resmedilmektedir.



Şekil 2. Klasör Yapısı

4. Programlama Ünitelerinin Yapısı

Temizer Sistemi, bir programlama ünitesinin (örnek olarak C dilini kullandığımız için *fonksiyon*ların) iç yapısı ile ilgili olarak şu tavsiyeleri içermektedir:

- Fonksiyonlar iki kısımdan oluşur. Birinci kısım, fonksiyona takılmış bir kanca gibidir ve fonksiyonun ana gövdesi çalışmadan önce (özellikle test aşamasında) çeşitli işlemler yapabilmemiz için kontrolü gerektiğinde bize aktarabilecek bir yapıda olmalıdır. İkinci kısım ise, fonksiyonun kendi gövdesidir. Eğer kullandığımız dil bir önişlemci içeriyorsa, o zaman bu kısımların ikisini de, gerektiğinde çıkarılabilecek şekilde yazmalıyız. (Kancayı testlerimiz bitince çıkarmalıyız, fonksiyonun gövdesini ise, birim testleri yaparken sadece kanca kısmını kullanmak istersek, test kodumuzun makina diline dönüştüğündeki büyüklüğü az olsun diye çıkarmak isteyebiliriz).
- Fonksiyonun gövdesi de iki kısımdan oluşur. Birinci kısımda, fonksiyonun aldığı parametrelerin hatalı olup olmadıklarını kontrol eden parça bulunur. İkinci kısımda ise fonksiyon, parametrelerin geçerli olduklarını varsayıp, gerekli operasyonları uygular. Bir sistemin doğru olarak çalıştığını test edip onayladıktan sonra, parametre kontrolü yapan kısımları, hızı artırmak ve büyüklüğü azaltmak açısından çıkarmak isteyebileceğimiz için, yine eğer kullandığımız dil bir önişlemci içeriyorsa, bu kısmı da gerektiğinde çıkarılabilecek şekilde yazmalıyız.
- Fonksiyonların tek bir çıkış noktası olmalıdır (o da fonksiyonun sonunda olmalıdır). Bu özelliğin uygulanması her zaman gerekli değildir. Hatta bazı durumlarda hız kaybına sebep olabilir. Yine de hem Temizer Sistemi'nde, hem DO-178 [3] gibi bazı dökümanlarda, hem de bazı şirketlerin kendileri için belirleyip kullandığı kodlama standartlarında tavsiye edilmektedir.

Bu bölümde örnek olarak, verilen bir sayının, büyüklük olarak yine verilen iki sayının arasında olup olmadığını kontrol eden *isInRange* adlı bir fonksiyon hazırladığımızı düşünelim. Yukarıdaki özellikler dikkate alınarak yazılmış bu fonksiyonu, anlatılan kanca yapısı için örnek bir implementasyonu ve de bu kanca mekanizması ile yapılabilecekleri göstermek üzere hazırlanmış bir birim test örneğini içeren, C dilinde yazılmış komple bir program ve gerekli açıklamalar Ek'te verilmiştir.

5. Sistemin diğer programlama dillerini kapsayacak şekilde genişletilmesi

Sistemin C++ programlama dilini içine alacak şekilde genişlemesi için ek hiçbir gereksinim yoktur. Veri tiplerine ek olarak sınıfları ve global değişkenlere ek olarak da nesneleri dahil ettiğimiz zaman, sistem C++ programlama dillerini kullanan projelerde de direk uygulanabilir. Bunun dışında, temel gruplama mantığını kullanarak diğer dillere de sistemi uygulamak mümkündür. Ama bir önişlemci yeteneği sunmayan dillerde, programlama ünitelerinin iç yapısı için önerilen özelliklerin basit bir

parametre ile koda eklenebilmesi veya çıkarılabilmesi zor olabilir veya mümkün olmayabilir.

6. Sistemin avantajları ve dezavantajları

Temizer Sistemi sadece yeni başlanan yazılım projelerinde değil, daha önceden başkaları tarafından geliştirilmiş yazılımlarının temizlenmesi, bakımı ve yeniden yapılandırılmasında da kullanılabilir. Karışık yapıdaki herhangi bir yazılımdan teker teker çıkarılacak parçalar, sisteme uygun olarak hazırlanmış ve başta içi boş olan bir dosya mimarisine yerleştirilerek yazılımın yapılandırılması sağlanabilir. Sistem, yazılımın her mantıksal parçasını dikkatlice belirlenmiş ayrı ayrı dosyalarda saklamayı öngördüğü için, yazılımın anlaşılması ve bakımı son derece kolaylaşacaktır ve yazılımın içermiş olabileceği gereksiz parçalar da kolaylıkla saptanılabilecektir (kendiliklerinden ortaya çıkacaklardır). Ayrıca yazılımın dışarıdan ihtiyaç duyduğu parçalar da kendi dosyaları içinde lokalize edilmiş olacağından, yazılımı bir başka platformda çalıştırmak gibi bir ihtiyaç duyulursa, müdahale edilmesi gereken noktalar son derece keskin hatlarla belirlenmiş olacaktır. Yine yazılımın dosyalara ayrılmış olması, Doxygen [2] gibi otomatik dökümantasyon üreten programlara yazılımın çeşitli parçaları için ayrı ayrı ve birbirinden bağımsız dökümanlar ürettirebilmeyi mümkün kılar.

Sistemde tavsiye edilen kanca yapısı, özellikle birim test sürecini son derece kolaylaştıracaktır. (Örnek olarak C programlama dilini ele alırsak ve) her fonksiyon için, Ek'te verilen örneğe uygun olarak birer adet de *stub* fonksiyon hazırladığımızı düşünürsek, tek bir derleme sonucunda, her fonksiyon hem kendi görevini yapabilecek ve hem de başka bir fonksiyonun testi sırasında boş bir fonksiyon (*stub*) gibi davranabilecek özellikler ile donanmış olacaktır ve bütün birim testlerinin hepsi tek bir derleme ile arka arkaya koşturulabilecektir.

Bu faydalarının yanısıra, Temizer Sistemi'nin bazı dezavantajları da vardır. Mesela bütün değişkenlerin tek bir dosyada toplanması ve her fonksiyonun kendine ait bir dosyada tek başına bulunması, C programlama dilindeki *static* global obje özelliğini kullanabilmemizi engeller ve bilgi saklama açısından olumsuz bir etki yapar. Ayrıca, küçük ölçekli projeler için, bu kadar fazla sayıda dosyaya ve anlatılan karmaşıklıktaki bir fonksiyon yapısına gerek olmayabilir.

7. Sonuç

Temizer Sistemi, yazılım geliştirme, test ve bakım süreçlerinde sistematiklik sağlayan, test sürecini yazılımın ilk evrelerinden itibaren göz önüne alan, destekleyen ve son derece kolaylaştıran bir sistem olarak tasarlanmıştır. Yazılımın içerdiği mantıksal grupları aynı zamanda fiziksel hale de getirerek yazılıma yapılabilecek hızlı müdahaleleri, geliştirmeleri ve yazılımın kolayca yeniden yapılandırılabilmesini destekler. Bu özellikleri itibarı ile, yazılım sürecini genel olarak hızlandırdığı gibi, ortaya çıkan ürünün kalitesini de artırır. Şu nicel örnek ile sözlerimizi bağlayabiliriz: Sistemdeki kanca yapısı, (1 Temmuz 2005 tarihinde) Aydın Yazılım ve Elektronik San. A.Ş. (AYESAŞ) tarafından 70,000 satır civarında koddan oluşan bir aviyonik projesinin birim testleri sırasında kullanılmaktadır. Özel bir donanım üzerinde koşması gereken bu testlerin her biri için, derlenmiş kodun donanıma yüklenmesi, testin koşması ve sonuçların donanımdan alınması ortalama 15 dakika sürmektedir. 400+ fonksiyon içeren bu proje için normal yollarla 400 x 15 = 6000 dakika sürmesi beklenen testlerin, kanca yapısı sayesinde 100-120 dakikada tamamlanacağı öngörülmektedir.

Kaynakça

[1]. Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2nd Edition.
[2]. Dimitri van Heesch, *Doxygen*, http://www.doxygen.org
[3]. RTCA, Inc., *Software Considerations in Airborne Systems and Equipment Certification*, 1992.

Ek. Örnek Fonksiyon

```
#include  <stdio.h>

/* --------------------------------------------------------------------------------------------------------------------- */
/* Derleyici parametreleri                                                                                                */

#define  ABC_ACTIVATE_HOOK
#define  ABC_ACTIVATE_FUNCTION_BODY
#define  ABC_ACTIVATE_PARAMETER_CHECKS

/* --------------------------------------------------------------------------------------------------------------------- */
/* Onislemci makrolari                                                                                                    */

#define  ABC_NONVOID_HOOK( returnType, functionName, parameterTypesInParentheses, parametersInParentheses )          \
{                                                                                                                     \
  extern  returnType     (* stubPtr_ ## functionName )  parameterTypesInParentheses ;                                \
  extern  unsigned char    callStub_ ## functionName                                ;                                \
  extern  unsigned char    skipBody_ ## functionName                                ;                                \
                                                                                                                      \
  returnType  hookReturnValue = returnType ## Initializer ;                                                          \
                                                                                                                      \
  if ( callStub_ ## functionName )  { hookReturnValue = (* stubPtr_ ## functionName ) parametersInParentheses ; }    \
  if ( skipBody_ ## functionName )  { return  hookReturnValue ;                                                }     \
}

#define  CheckResultInitializer  TRUE

/* --------------------------------------------------------------------------------------------------------------------- */
/* Veri tipleri                                                                                                           */

typedef  enum { TRUE, FALSE, ERROR }  CheckResult ;

/* --------------------------------------------------------------------------------------------------------------------- */
/* Birim testler sirasinda ihtiyac duyulacak degiskenler                                                                  */

CheckResult    (* stubPtr_isInRange) (int, int, int) ;
CheckResult    stubReturnValue_isInRange            ;
unsigned char  callStub_isInRange                   ;
unsigned char  skipBody_isInRange                   ;

/* --------------------------------------------------------------------------------------------------------------------- */
/* Temizer Sistemi'ne uygun olarak hazirlanmis "isInRange" fonksiyonu                                                     */

CheckResult  isInRange ( int number, int rangeMin, int rangeMax )
```

```
{
  #ifdef  ABC_ACTIVATE_HOOK
  ABC_NONVOID_HOOK( CheckResult, isInRange, (int, int, int), (number, rangeMin, rangeMax) )
  #endif

  #ifdef  ABC_ACTIVATE_FUNCTION_BODY
  {
    CheckResult  returnValue = FALSE ;

    #ifdef  ABC_ACTIVATE_PARAMETER_CHECKS
    if ( rangeMin > rangeMax )  { returnValue = ERROR ;  goto FunctionExitPoint ; }
    #endif

    if ( (number >= rangeMin) && (number <= rangeMax) )  { returnValue = TRUE ; }

    FunctionExitPoint:  return  returnValue ;
  }
  #endif
}

/* ------------------------------------------------------------------------------------------------------------------------ */
/* "isInRange" fonksiyonunu cagiran birimlerin testleri sirasinda "isInRange" fonksiyonunun yerini alacak "stub" fonksiyon */

CheckResult  stub_isInRange ( int parameter1, int parameter2, int parameter3 )
{
  extern  CheckResult  stubReturnValue_isInRange ;

  printf( "isInRange fonksiyonu su argumanlar ile cagirildi: ( %d, %d, %d )\n", parameter1, parameter2, parameter3 ) ;

  return  stubReturnValue_isInRange ;
}

/* ------------------------------------------------------------------------------------------------------------------------ */
/* "isInRange" fonksiyonunun davranisini testler sirasinda istegimize gore nasil degistirebilecegimizi gosteren bir ornek  */

void  main ( void )
{
  CheckResult  result ;

  stubPtr_isInRange         = stub_isInRange ;
  stubReturnValue_isInRange = TRUE           ;
  callStub_isInRange        = 1              ;
  skipBody_isInRange        = 1              ;

  result = isInRange( 4, 1, 10 ) ;
}
```