

# Programlamayı Nasıl Öğretmeliyiz

—

## Programlamayı Nasıl Öğretmemeliyiz

Chris Stephenson  
İstanbul Bilgi Üniversitesi

March 7, 2007

### Abstract

The results we achieve when teaching programming are not good. Many Computer Science / Computer Engineering students graduate unable to program well. I suggest adoption of some new approaches, emphasising program and data design from the first day of programming teaching. These are contrasted with examples from conventional textbooks in current use. Experience, both in the author's University and elsewhere suggest that this is a fruitful path to follow.

### Özet

Programlamayı öğretirken aldığımız sonuçlar pek de iç açıcı değil. Bilgisayar Bilimleri/Bilgisayar Mühendisliği bölümlerinden mezun olan öğrencilerin birçoğu program yazmakta yetersizler. Benim önerim eğitimin daha ilk gününden itibaren program ve veri tasarımı vurgulamak. Yazar, hem kendi üniversitesinden hem de başka diğer üniversitelerden kazanılan tecrübelerden dolayı bu yolun çok verimli olduğu görüşünde.

## 1 Programlarda gerçekten önemli olan nedir?

Stratejik atılım sıklıkla verinin ya da tabloların değişiminden ortaya çıkar. Burası programın kalbinin attığı yerdir. Bana akış şemalarını göster ve tablolarını sakla, programın gizemini korusun. Tablolarını göster, çoğunlukla akış şemalarını görmem gerekmez, zaten belirgin olacaklardır.[1] –F.P. Brooks

Muhtemelen yazılımlar iki parçası birbirine benzemediği için insan ürünü başka herhangi bir yapıdan daha karmaşıktır (en azından belirtim aşamasında). Eğer benzeşen kısımları olursa alt-rutin haline getirip tekrarı engelleriz. Bu hususta yazılımlar bilgisayar, bina ya da otomobil gibi tekrarın sıkça bulunduğu ürünlerden derin bir çizgi ile ayrılırlar.[2] –F.P. Brooks

Bu 30 yıldan önce, 1975'te söylendi. Brooks çıkarımlarını 1960'larda başında olduğu devasa ve zor programlama projesi olan IBM OS/360 deneyiminden türetti. Çıkarımları günümüzde hala geçerli. Günümüz terimlerine tercüme ettiğimizde Brooks 2 maddeyi yazılım tasarımının merkezine koyuyor: yapının tasarımı ve verinin tanımı ile kodun yeniden kullanımı için gerekli olan soyutlama.

Zamanımızda popüler olan programlama tekniklerine baktığımızda hepsi Brooks'un veri tasarımı ve soyutlamasının doğruluğunu yansıtıyor.

Veri tasarımının merkeziliği pek çok alanda karşımıza çıkıyor. Nesne yönelimli programlama, diğer özelliklerinin yanında veri yapılarına kodu da ekleyerek kodu programın yapısında bağlayıcı bir konuma yerleştiriyor. Sistem tasarımının Model-View-Controller tekniği model tasarıma – yani veri yapısı ve onun anlamsal tanımına – belirleyici rolü yüklüyor. J2EE sistemlerinde kritik bağlayıcı rolünü oldukça yüklü miktarda XML oynuyor. Bu XML işi o kadar büyük ki çoğu Java IDE'sinin ana işi bu XML veri yapılarını düzenlemek ve yaratmak. Servis Odaklı Mimariler yeniden söz dizimi ve anlamı, bu durumda sunucular ve istemciler arasında aktarılan XML verisini yeniden sistem tasarımının merkezine koyun.

Aynı şekilde, soyutlama gereksinimi bilgisayar sistemleri geliştirme tekniklerindeki modern ilerlemelerin merkezindedir. Brooks'un gözlemlediği gibi tekrarlama gördüğümüz yerde kurtulmaya çalışırız. Soyutlama, yeniden kod yazmamızı engellemeye yardım edecek genelleştirmeler için bir öngereksinimdir. Bu genelleştirmeler karşımıza Inheritance, Iteration, Collection, Design Patterns olarak çıkıyor. “Yazılım problemini” çözmek için çırpınırlar.

Biz öğrencilerimize bu yöntemle mi öğretiyoruz? Hayır.

Sadece onları bu yöntemle eğitmemekle kalmıyoruz, genel olarak programlamayı öğretirken, öyle bir yaklaşımla öğrencilerimizi eğitiyoruz ki, bu yöntem onların kavramasını çok *daha zor* kılıyor.

Bunun bir belirtisi “kopyala-yapıştır” programlama. Mezunlarımızın birçoğu son derece kolay programları bile yazmakta yetersiz kalıyorlar. Onların esas programlama teknikleri Google'dan çözümünü bulmak ve sonradan taslağa uyarlamak. Ben bunu Bilgi'de yüzlerce Bilgisayar Bilimleri/Bilgisayar Mühendisliği asistanı adayında gözlemledim. En iyi CV'lere sahip adayları bulduktan sonra, seçtiğimiz adayların %90'ı hala boş kağıt ve bir kalem verildiğinde birkaç satırlık basit bir programı bile yazamıyorlar.

Ben, tasarımı ve veriden başlayan farklı bir yaklaşıma ihtiyacımızın olduğunu dile getirmek, tartışmak istiyorum. Bu makalede ben Bilgi Üniversitesi'nde kullandığımız tasarım temelli yaklaşım ve fonksiyonel bir dil olan Scheme'den kazandığımız tecrübelerden bahsedeceğim.

Malesef bu öğretmek için programlama dillerinin seçimi bakımından teknik bir soru değil. Bütün dillerin arasında en uygunuyla programlamayı oldukça kötü öğretmek mümkün, uygunsuz olanları kullanıp daha kötü de öğretmek mümkün. Bu, yaklaşımla ilgili bir sorun.

## 2 Programlamayı kötü öğretiyoruz

Neden bizim öğrencilerimiz programlamayı öğrenemiyorlar? Sizlere bunun sebebinin, onlara kötü bir şekilde öğretmeye çalışmamız olduğunu kanıtlayacağım. Daha da kötüsü onlara kötü programlamayı öğretiyoruz. Hipokrat yeminindeki

ilk bend der ki, doktor hastasına zarar vermemelidir. Aynı kural programlama öğretmenler için de geçerli olmalıdır.

Bir örnek ele alalım. Öğrencilerimize eğitimlerinin başlarında öğrettiğimiz programlardan birtanesi bir doğal sayı listesini artan şekilde sıralamak. Bizim açımızdan pek bir şey ifade etmese de, onlardan iyi açıklamamış olduğumuz yöntemleri kullanarak gereksiz, kullanışsız ve aynı zamanda hesaplama zamanı açısından da yetersiz olan programlar yazmalarını istiyoruz<sup>1</sup>.

Bununla beraber, sıralama orta zorlukta bir iş ve nasıl öğrettiğimiz hakkında kısa bir araştırma aydınlatıcı olabilir. Programlama öğretme konusunda örnekler aramak için bakılması gereken en iyi yer, doğal olarak programlama ders kitaplarıdır.

Ben örnekler bakmak için bir adet C programlama kitabı ve 6 tane de Java programlama kitabı seçtim. Örnek metinler basitçe bana incelemem için bedava gönderilen programlamaya giriş kitaplarından seçildi. Ben bunların arasından sıralamadan bahsedenleri seçtim. Basit bir sıralama algoritmasının bile kitaplar tarafından “çok zor” addedilmesi, kalitesi düşük programlama eğitimimizin bir göstergesidir. Tabii ki böyle bir algoritma öğrencilere öğretilmesi gereken ilk algoritma olmamalı, ancak bu orta zorlukta bir algoritmadır ve öğrencilere yaptırılabilir makul bir programlama görevidir.

*Java for Students*[3] kitabı numaralardan oluşan bir dizinin sıralanmasını egzersiz olarak bırakmış ve okuyucuyu “yazılabilecek en kolay program değildir” diyerek basitçe uyarılmış. Bu programcıları eğitmek için en iyi yöntem değildir. Öğrencilerimize doğru programları nasıl kolayca yazabileceklerini göstermek bizim işimizdir, onlara “kolay değildir” demek değil.

*Java: How to Program*[4] kitabı ilginç bir şekilde kendinden kopya çekme yoluna gitmiş. Bu kitaptaki bubble sort programı *C: How to Program*[5]’daki kodun bir kopyası. Aslına bakarsanız, yazılan program bir Java programı bile değil, sadece Java’ya çevrilmiş bir C programı. Ve test verisi de programın içine “elle yazılarak” girilmiş. Bu durum bir testin yeterli olduğunu söyleyen ve aslında çok yanlış olan düşünceyi teşvik eder.

Deitel’in şu kod parçasına bakalım:

```
// swap two elements of an array
public void swap( int array3[], int first, int second )
{
int hold; // temporary holding area for swap
    hold = array3[ first ];
    array3[ first ] = array3[ second ];
    array3[ second ] = hold;
}
```

swap metodunun içinde değerleri değiş-tokuş ettirmek için gereken tuhaf hile birçok tehlike barındırır ve öğrencilerin sorması gereken 4 soruyu doğurur:

---

<sup>1</sup>Bir anahtardan sıralanmış ikililer için sıralamaya ihtiyacımız vardır. Eğer gerçekten sayılardan oluşan bir listeyi sıralamak istiyorsanız, ve bu sayılar sabit uzunlukta ise, o zaman metod  $O(n)$ ’dir. Eğer  $n$ ’tane ikili içeren bir listeyi sıralamak istiyorsanız o zaman  $O(n \log n)$  olan iyi bilinen ve göreceli olarak basit metodlar vardır. Insertion sortu (ve bubble ve selection sort) genellikle anlaması kolay olmayan, verimli olmayan ve hesaplama süresi  $O(n^2)$  olan yöntemlerle öğretiriz. Böyle bir yöntemin kullanımı da gerçek boyutlarda olan bir problem için kabul edilemez.

Neden deđiş-tokuş ettirmek için bir array ve iki indeks gönderiyoruz da, sadece deđiş-tokuş edeceğimiz iki deđeri yollamıyoruz?

Neden hold isminde üçüncü bir deđişkene ihtiyac duydunuz?

Swap metodu neden public olarak tanımlanmış?

Dizi parametresine neden array3 ismini verdiniz? Büyük ihtimalle bubble-Sort da kullanılan array parametresinin ismi array2'ydi.

Birinci sorunun cevabı Java dizilerinin kararsız olmasından dolayı deđerler, referanslar ve pointerlar hakkında derin bir tartışmayı içerir. Tartışma, program örneğinin dizinin tamamının gönderilmesini önleyen pointer yapısını içeren C kitabından direk olarak kopyalanması nedeniyle daha da belirsiz bir hal alır. İkinci ise imperative programlama stiline kaçınılmaz sonucudur. Üçüncü sorunun cevabı ise gayet basit: fonksiyon public olmamalı. Kod örneđi yanlış. Son soru ise gösteriyor ki kitap öğrencilere son derece yanlış olarak metod parametreleri için tamamen farklı isimler seçmek gerektiđini söylüyor.

*Deitel and Deitel* kitabında verilen bubble sort ise sadece zor anlaşılır olmakla kalmıyor, aynı zamanda devasa miktarda gereksiz iş içeriyor. Halihazırda sıralı olan bir dizi için  $n^2$  kıyaslama yapıyor. Daha da önemlisi, verilen programa nasıl ulaşıldığı hakkında herhangi bir ipucu verilmemiş ve programın çalışacağı hususunda gerçek bir garanti yok. Örnek kod “kopyala-yapıştır” yaklaşımına davet ediyor ve öğrencilere gerçekte bir testin yeterli olacağını öğretiyor.

Son olarak bu bölümün sonunda, yazarlar aşağıdaki “bilgeliğın incisi”ni sunuyorlar, *C:How to Program* kitabında “başarıım ipucu 6.5” olarak geçen metni aynen kopyalıyorum.

#### Başarıım ipucu 7.2

Bazen en basit algoritmalar çok kötü çalışırlar. Onların meziyeti kolay programlanmaları, kolayca test ve debug edilebilmeleridir. Bazen daha kompleks algoritmalar...

Bu “piyasada” bulunan programlamaya giriş kitaplarının genel kuralına bir örnektir. Eğer bir yazar bir noktayı vurgulamak üzere yazıyı kutu içine almış ise, yada yazıyı renklendirmiş ise, dikkat edin. Büyük ihtimalle o kutunun içindeki mor, kahverengi yada kırmızı yazılar *dođru deđildir*. Bu gözleme Mor kutu yasası diyelim. Bu yasaya uygun “ispata” makalenin ilerleyen kısımlarında deđineceğim.

Basit algoritmalar kompleks olanlardan daha iyi yada daha kötü çalışabilirler. Basit ve şık yazılmış bir algoritmanın performansını ölçmek karışık yazılmış bir algoritmanınkini ölçmekten çok daha kolaydır, ve basit algoritmalarda performans tuzaklarına düşme olasılıđınız daha azdır.

*Java Software Solutions Foundations of Program Design*[6]

Diđer kitaplarda olduđu gibi, bunda da açıklamalar programın önemli parçalarının üzerinden şöyle bir geçiyor, öyleki masum deđişiklikler programın çalışmasını durduracakmış gibi duruyor. Bu da tek testin sıralama algoritması için yeterli olacağını varsayıyor.

Örnek:

```
public static void insertionSort (int [] numbers)
{
    for (int index = 1; index < numbers.length; index++)
    {
        int key = numbers[index];
```

```

        int position = index;

        // shift larger values to the right
        while (position > 0 && numbers[position-1] > key)/**
        {
            numbers[position] = numbers[position-1];
            position--;
        }

        numbers[position] = key;
    }
}

```

Yıldız ile işaretlenmiş satıra bakın. Eğer && işareti ile birleştirilmiş değerlerin sırasını değiştirirsek:

```
while (numbers[position-1] > key && position > 0 )/**
```

program, dizinin sınırları aşıldığı gerekçesi ile duracaktır. Bunu anlayabilmek için, && işaretinin simetrik bir operator olmadığını, aksine ilk kısım “false” olduğunda ikinci kısmın çalıştırılmadığı bir testler dizisi olduğunu bilmek gereklidir.

*Java Programming from the Beginning[7]*

Bu kitap sanki sıralama algoritmasını anlamayı kasten zorlaştırmış. Bir doğal sayıyı halihazırda sıralanmış bir listeye eklemek için bir parça kod yazılmış. Daha sonra bu kod terkedilip özgün koda tam ters yönde bir insertion sort yapan yeni bir algoritmaya başlanmış.

Kafa karıştırıcı 5 sayfa yazıdan sonra kitap bize başka bir Mor Kutu örneği veriyor.

Bir algoritma tasarladığımızda, en başta özel durumlar için endişe etmeyin. Algoritmayı tipik durumlarda çalışacak şekilde tasarlayın. Tamamladığımızda, geri dönüp özel durumlar için çalışıp çalışmadığına bakın. Eğer çalışmıyorsa özel durumları da kapsayacak testler ekleyin.

Kaçırılmış olma ihtimalinize karşın, Mor kutu ana metin içerisinde siyah beyaz şekilde tekrar yinelenmiş.

Diğer bir deyişle, genel algoritmayı tasarlamadan önce, özel durumlar için endişelenerek vakit kaybetmeyin ve kodunuzu daha da karmaşık hale getirmeyin.

Umarım programlamayı bu kitaplardan öğrenen öğrenciler uçak veya nükleer santral programlamada çalışmazlar. Yazılım mühendisliği alanında kritik durumlar için test seçimi konusunda çok geniş bir kaynak araştırması yapılabilir. Özel durumlar çoğunlukla programların bittiği yerlerdir. Diğer bir deyişle bir program tasarlanırken ana merkeze bu “özel durumlar” oturtulmalıdır.

Bu kitaptaki insertion sort kodu da Lewis ve Loftus’un kitabındaki gibi && tuzağını içeriyor.

*Introduction to Java Programming[8]*

Bu örnekte de diğerlerindeki kusurların çoğu mevcut, Mor Kutu'su dışında da değinilmesi gereken bir özelliği yok.

Çoğu öğrenciler bu algoritmayı ilk seferde çalıştıramayabilirler. Benim size tavsiyem, kodun önce ilk adımını yazarak listenin en büyük elemanını bulun ve son eleman ile yer değiştirin. Daha sonra ikinci adımda nelerin farklı olması gerektiğini bulmaya çalışın, sonra üçüncü adım, daha sonra da diğerleri. Bu incelemeyi sonra dış döngüyü oluşturup algoritmayı çalıştırabilirsiniz.

*The Object of Java Introduction to Programming Using Software Engineering Principles [9]*

Bu kitap daha iyi olmaya çalışıyor, fakat daha fazla kafa karıştırıyor. SimpleList kütüphanesini kullanarak tek bir döngü ile programı yazıyor. Kitapta ön koşullar ve bitiş koşullarına değinilmiş. Ancak && tuzağı bu kitapta da karşımıza çıkıyor.

```
public void insert (int j) {
    start();
    while (!isOff() && j > itemInt())    {
        forth();
    }
    super.insert(new Integer(j));
}
```

İlgili satırdaki && ifadesi ile birleştirilmiş değerlerin yerlerini değiştirdiğimizde programın hata verip çalışmadığını göreceksiniz.

Hoare'dan bir alıntı:

“Yazılım üretmenin iki yolu vardır. Ya çok basit yazarsın, hata içermediği hemen anlaşılır, ya da çok karmaşık yazarsınız ve hataların görünmemesini sağlarsınız.”

### 3 Özet

Bahsi geçen kitaplarda sıralama ile ilgili algoritma hakkında bir kaç sayı:

|             | Sıralama türü | 1 test yeterli              | Sf. sayısı | Kodun satır sayısı <sup>2</sup> | Genel olarak doğru mu? | so-runlu kod <sup>3</sup> | Mor Kutu |
|-------------|---------------|-----------------------------|------------|---------------------------------|------------------------|---------------------------|----------|
| Deitel C    | Bubble        | Evet                        | 3          | 22                              | hayır                  | Evet                      | Hatalı   |
| Deitel Java | bubble        | Evet                        | 4          | 37                              | hayır                  | Evet                      | Hatalı   |
| Lewis       | insert        | Evet                        | 6          | 27                              | hayır                  | Evet                      | Yok      |
| King        | insert        | Teste gerek yok!            | 5          | 17 <sup>4</sup>                 | hayır                  | Evet                      | Hatalı   |
| Liang       | selection     | Evet                        | 3          | 32                              | hayır                  | hayır                     | Hatalı   |
| Riley       | insert        | Evet, tek bir örnek yeterli | 5          | 24 <sup>5</sup>                 | hayır                  | Evet                      | Yok      |

Bu incelemeye göre 6 örneğin 4'ünde Mor kutular hatalı, ikisinde ise Mor kutu yok. Başka bir deyişle “Mor kutu yasamız” ile çelişen bir örneğe rastlanmadı. Bunu Mor kutu yasasının doğruluğu için bir kanıt kabul edebiliriz.

Bu kitapların ortak özelliği yanlış şeyler öğretiyor olmaları. Yetersiz test yapmayı öğretiyorlar. Öğrencilerin kafasını değişkenlerin kapsamları, private ve public gibi terimlerle karıştırıyorlar. Bu kitaplar oldukça kırılğan program örnekleri veriyorlar. Bu yüzden öğrenciler yeniden çalıştıramama korkusu ile çalışan koda müdahale etmekten korkuyorlar. En kötüsü de bu kitapların öğrencilerin kendine olan özgüvenlerini azaltıyor olması. Bu kitapların verdikleri kötü programlama örnekleri ile anlamsız bir araç haline gelişinin incelenmesi, bu makalenin kapsamı dışındadır.

## 4 Programlamayı daha iyi öğretebiliriz

Bilgi Üniversitesi'nde biz tasarım temelli bir yaklaşım uyguluyoruz. Birinci sınıf derslerinde How to Design Programs[10] adlı kitabı kullanıyoruz. Uyguladığımız yöntemin tam olarak anlaşılabilmesi için, okuyucunun HtDP kitabına bakması gerekir. Neyse ki bu kitaba [www.htdp.org](http://www.htdp.org) sitesinden ücretsiz erişilebilir. Ayrıca burada DrScheme tarafından desteklenen Scheme dilinin “Beginning Student” özelliğindeki sözdiziminin okuyucu tarafından kolaylıkla kavranabileceğini kabul ediyoruz ve direk olarak birinci sınıf dersimizin yaklaşık onuncu haftasında gösterdiğimiz bir örnekle devam ediyoruz.

HtDP yaklaşımını nasıl yorumladığımızı anlamak için, gelin insert sort algoritmasının tasarımını yeniden yapalım.

Sürecin ilk aşaması verimizi tanımlamak. Yapmak istediğimiz şey sayılardan oluşan bir listeyi sıraya sokmak. Öncelikle giriş ve çıkış değerlerimizi tam olarak tanımlamalıyız.

Giriş değerimiz bir sayı listesi, çıkış değerimiz ise sıralı bir sayı listesi.

Sayı listesi ne demektir? Sayı listesi iki şey olabilir. Ya hiç bir şey içermeyen boş bir listedir, yada bir sayı ve bu sayıyı takip eden sıralı bir sayı listesidir. Bu, iki koşul içeren basit bir fonksiyon ile kontrol edilebilen kusursuz bir tanımdır.

Şimdi tanımı ve fonksiyonu yazalım:

```
;; data definition list-of-number
;; list-of-number is empty
;; OR
;; (pair number list-of-number)
(define (is-a-list-of-number? l)
  (cond
    ((null? l) true)
    (else (and (number? (first l))
               (is-a-list-of-numbers? (rest l))))))
```

Bu ve buna benzer programlardan, listeler üzerinde çalışan bir çok programda da kullanabileceğimiz, şablon adı verilen, bir kalıp çıkarabiliriz.

```
(define (some-func l)
  (cond
    ((null? l) ...)
    (else ... (first l) ... (some-func (rest l)))))
```

Peki, sıralanmış sayı listesi ne demektir? Nasıl tanımlarsak tanımlayalım, tanımımızda her zaman üç, iki değil, koşulla karşılaşırız. Sıralanmış bir sayı

listesi ya hiç bir şey içermeyen boş bir liste, ya bir tek sayı içeren bir liste, yada bir sayı ve sıralanmış bir listeden oluşan ve listenin ilk elemanı o sayıdan büyük olan bir ikilidir.

```
;; data definition sorted-list-of-number
;; sorted-list-of-number is
;; empty
;; OR
;; (pair number empty)
;; OR
;; (pair number sorted-list-of-number) where
;;     number <= (first sorted-list-of-number)
```

Veri tanımını kullanarak, sıralı listeler için bir test yazabiliriz. Bu da bize, veri yapısına bakarak nasıl program yazabileceğimizi gösteren başka bir örnek teşkil edecektir. Ayrıca sıralama programımızı test edebileceğimiz faydalı bir uygulamadır.

```
(define (is-a-sorted-list-of-number? l)
  (cond
    ((null? l) true)
    ((null? (rest l)) true)
    (else (and ( <(first l) (first (rest l)))
                (is-a-sorted-list-of-numbers? (rest l))))))
```

Bizim sıralama programımızın ne yapması gerekiyor? Argüman olarak bir sayı listesi almalı. Bunun için bir şablonumuz var zaten. Şablonumuzu bir kez oluşturduktan sonra, düşünmemiz gereken bu şablonu nasıl doldurmalıyız ki sıralı bir liste elde edebilelim.

Şablon bizim için programı yazmayacaktır. Bizim programımızın ne yapması gerektiğini de tanımlamamız gerekir. Programımızın iki koşulu sağlaması gerekecek. Sonuç sıralı bir sayı listesi olmalı, ve bu sayı listesi sadece ve sadece argüman olarak verdiğimiz listedeki sayıları içermeli. Bir kaç örnek yazarsak hem bize yardımcı olacaktır, hem de daha sonra test ederken kullanabiliriz. Marjinal durumları kontrol ederek yazılım mühendisliğinin de temel metodlarından birini de tatbik etmiş olacağız.

Burada örneklerimizi sıralama programımızın bitmiş halini de test edebileceğimiz şekilde yazabiliriz.

```
(equal? (sort empty) empty)
(equal? (sort (pair 1 empty)) (pair 1 empty))
(equal? (sort (pair 1 (pair 2 empty))) (pair 1 (pair 2 empty)))
(equal? (sort (pair 2 (pair 1 empty))) (pair 1 (pair 2 empty)))
(equal? (sort (pair 3 (pair 2 (pair 1 empty))))
        (pair 1 (pair 2 (pair 3 empty))))
```

Şablona baktığımızda görürüz ki, sağ tarafta “...” şeklinde boş bırakılan yerleri doldurabilmek için bir şekilde boş ve sadece bir eleman içeren listeden oluşan sıralı bir liste oluşturmalıyız, ki bu durum çok da zor olmayacaktır. Asıl zor olan ise bir sayı ve sıralı bir liste içeren listeyi sıralamaktır. Tasarımımızı tamamladığımızda bu problemlere geri dönebiliriz.

Böylece sıralama programımız için tasarımı tamamlamış ve şablonu çıkarmış olduk. Fakat programımızın koşullarından bir tanesi başka bir dizi üzerinde işlem yapmayı içeriyor. Bu bizim şablonumuza uymayabilir. Böyle bir durumda tasarım kuralımız bize yardımcı fonksiyonlar yazmamızı söyler. Artık yardımcı fonksiyonumuz için kullanacağımız kısıtları biliyoruz. Argüman olarak bir sayı ve sıralı bir liste olacak, sıralı bir liste döndürecek. Diğer bir kısıtımız ise geri dönüş değeri olarak çıkacak listenin sadece ve sadece argüman olarak verdiğimiz listenin elemanlarını içermesi. Aslen, açık olan bir şey var ki, eğer yardımcı fonksiyonumuz bu kısıtları sağlarsa, ana fonksiyonumuz da sağlayacaktır.

Yine, bazı test örnekleri bize yardımcı olacak. Tekrar bunları, daha sonra da kullanabilmek için, uygun testler biçiminde ifade edebiliriz.

```
(equal? (insert 3 empty) (pair 3 empty))
(equal? (insert 2 (pair 3 empty)) (pair 2 (pair 3 empty)))
(equal? (insert 3 (pair 2 empty)) (pair 2 (pair 3 empty)))
(equal? (insert 2 (pair 1 (pair 3 empty))) (pair 1
      (pair 2 (pair 3 empty))))
(equal? (insert 1 (pair 2 (pair 3 empty))) (pair 1
      (pair 2 (pair 3 empty))))
(equal? (insert 3 (pair 1 (pair 2 empty))) (pair 1
      (pair 2 (pair 3 empty))))
```

Bir kez daha önceden hazırladığımız liste işleme şablonumuzu kullanabiliriz. Fonksiyonumuz bir liste üretmeli. Yeni bir liste oluşturmak için, bir elemanı listenin ilk elemanı olarak baş tarafa ekleyen cons fonksiyonunu kullanıyoruz. Bu fonksiyon için “null” durumu gayet basit – eğer “null” listeye (hiç bir eleman içermeyen, boş listeye) ekleme yaparsak, basitçe yeni eleman ile boş listeyi ikili oluşturacak şekilde birleştiriyoruz. Diğer durumda ise sıralı listenin veri tanımı bize yol gösterecek. Eğer eklenecek sayı sıralı listenin ilk elemanından küçük ise, basitçe bu elemanı listenin ilk elemanı olacak şekilde listeye birleştiririz, böylece geri dönüş listemiz diğer kısıtları da karşılayan sıralı bir liste olacaktır. Eğer değilse, listenin ilk elemanı ile sayının listenin geri kalanına eklenmiş halini ikili oluşturacak şekilde birleştiririz.

```
;; insert number sorted-list-of-number -> sorted-list-of-number
;; purpose: insert a number into a sorted-list-of-number
;; template
;; (define (insert n slon)
;;   (cond
;;     ((empty? slon) ...)
;;     ((<= n (first slon)) ...)
;;     (else ... (first slon) ... (insert n (rest slon)))))
```

Artık elimizde verimizin tanımı, iki fonksiyonumuzun özellikleri, test durumlarının tanımları ve fonksiyonlarımızın şablonları var.

Dikkatinizi çektiyse şu ana kadar asıl programımızla ilgili tek satır kod bile yazmadık. Yazdıklarımız tanımlardan, testlerden ve şablonlardan ibaret. Geriye kalan tek şey boşlukları doldurarak sıralama programımızı yazmak. Örneklere bakarak bu işi yapmak çok kolay.

Fonksiyonların doldurulmuş hali şu şekilde.

```

(define (insert n slon)
  (cond
    ((empty? slon) (pair n empty))
    ((<= n (first slon)) (pair n slon))
    (else (pair (first slon) (insert n (rest slon))))))

(define (sort lon)
  (cond
    ((empty? lon) empty)
    (else (insert (first lon) (sort (rest lon))))))

```

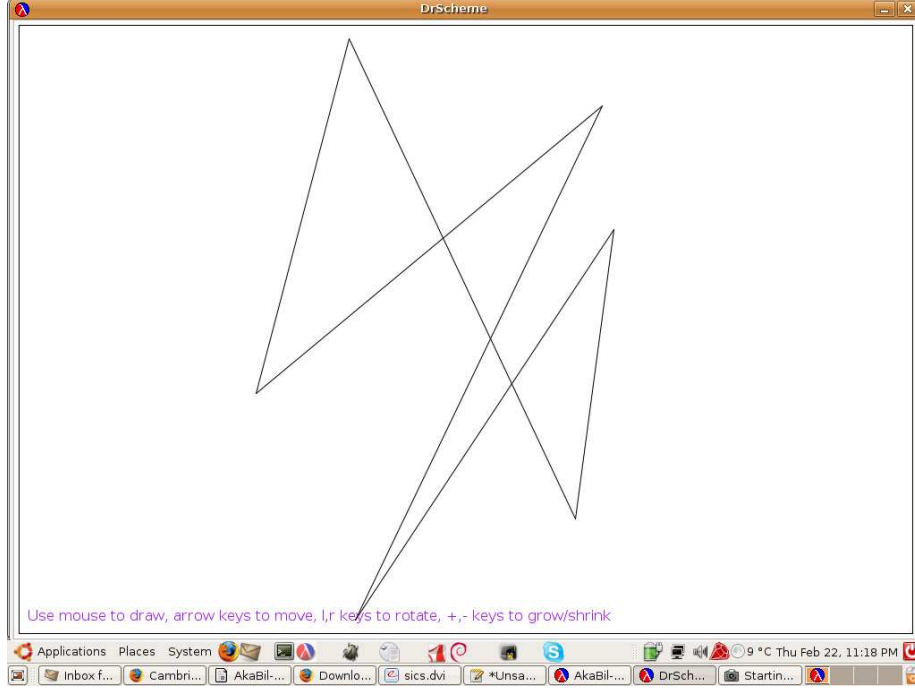
Artık programımızı çalıştırabiliriz.

Taslađını hazırladığımız method bu kadar basit bir problem için aşırı derecede uzun görünebilir. Ancak, amaç öğrencilere nasıl program yazmaları gerektiğini öğretmek, bu yüzden yöntem basit problemlerle başlayarak daha sonra aynı yaklaşımla daha büyük problemleri çözmelerini sağlamak olmalı. Her ne kadar bu tasarım kendi içinde usulüne uygun doğrulukta bir kanıt olmasa da, en azından, ikna edici ve oldukça kolay bir şekilde uygun doğrulukta bir kanıtla çevrilebilir.

Aslına bakarsanız, bu makalede anlatılan insertion sort algoritması, incelediğimiz kitapların birçoğundan daha kısadır. Çalıştırılabilir kod toplamı 9 satır. Bunlara 24 satırlık test kodlarını da ilave edebilirsiniz. Toplam hala kitaplardan daha fazla değil, ki her birinde en fazla bir test yapılmış. Fakat bizim programımızda çok daha fazlası mevcut. Giriş ve çıkış değerlerinin doğruluğunu ölçebilecek testleri içeriyor. İçerdiği bağımlılık testleri sayesinde kod üzerinde yapılan değişikliklerin programı bozmasını engelliyor. Her fonsiyonun giriş ve çıkış değerleri için kapsamlı bir dökümantasyon sağlıyor. Diğer bir deyişle öğrencilere profesyonel bir şekilde nasıl program yazmaları gerektiğini öğretiyor.

Bir sonraki bölümde aynı tekniği kullanarak pek de kolay görünmeyen, ama aslında kavramsal olarak insertion sort'dan daha kolay olan, bir başlangıç seviyesi programı inceleyeceğiz.

## 5 Öğrencileri daha iyi motive edebiliriz



Bu program çıktısı, İstanbul Bilgi Üniversitesi birinci sınıf dersi olan “Programlamaya Giriş” dersinin ilk dönemine ait onuncu hafta örneğidir. Bu program tasarım olarak bir önceki bölümde anlatılan insert sort algoritmasına nazaran daha basit olmasına rağmen, oldukça motive edici grafik sonuçlar üretir.

Yapılması istenen, her fare tıklamasını yeni bir nod olarak kaydeden ve en son eklenen nodun pozisyonunu fareyi sürükleyip bırakmak suretiyle değiştirebilen ve bu nodları kullanarak bilgisayar ekranına poligon çizebilecek bir program yazmak. Ekranı çizilen poligonun klavyedeki tuşlar aracılığı ile hareket ettirilebilir, döndürülebilir ve boyutlarının değiştirilebilir olması gerekiyor.

Bir veri yapısının yönlendirmesiyle, soyutlamaya yönelik yaklaşım zor görünen programlama problemlerini acemi bir programcının dahi halledebileceği kadar basit problemlere indirger. İlk izlenim Model-View-Controller yaklaşımına uygun bir model seçmenin çok önemli olduğudur. Model bir kez tasarlandığında, problemin karmaşıklığı muazzam ölçüde azalır. Modeli kullanarak poligonu ekrana çizdirmek – View kısmı – basit yöntemler kullanılarak yapılabilir. Daha sonra poligon üzerinde yapılması istenen değişiklikler (nod ekleme, son nodu hareket ettirme, poligonu döndürme, poligonu hareket ettirme ve boyutunu değiştirme) oldukça basit fonksiyonlar kullanılarak görselleştirmeden (View) ve birbirlerinden bağımsız olacak şekilde halledilebilir.

Program, tasarım tanımlamaları ve diğer tüm yorumlarla beraber, gösteriyor ki bu iş sadece 58 satırlık çalıştırılabilir kodla yapılabilir. Tasarım reçetesinin meziyeti sayesinde, kod düzgün yazılmış, son derece güzel dokümente edilmiş, bağımlılık testleri sayesinde programdaki her bir fonksiyonun yapılacak değişikliklerde anlaşılabilir hatalar üretmeleri engellenmiştir. Zekice yapılmış bir üçkağıtçılık barındırmıyor. Programlama ortamı özel olarak bu tipdeki program-

lar için *ayarlanmamış*. Sunulan programlama ortamı fare hareketlerini ve klavye tuşlarını fonksiyonlara bağlamaktan ve iki nokta arasına çizgi çizmekten daha fazla bir özellik sağlamıyor. Bu gerçekten de ilkel bir grafik ortamı. Avantaj, programlama ortamının sağladığı olanaklardan değil, tasarım metodlarından geliyor.

## 6 Deneyim

İlk bölümde belirtilen fikir yeni değildir. Bilgisayar Bilimleriyle uğraşan bir çok akademisyen yıllardır bir değişim için şikayet edip, kulis yapıyorlar (Dijkstra). MIT'deki birinci yıl mühendislik öğrencilerine verilen efsanevi 6001 dersinin Structure and Interpretation of Computer Programs (SICP)[12] kitabı bu yazar için ilham kaynağı olmuştur. Kitabın kusurları ve kötü yanları olmuştur. Giriş düzeyinde programlama eğitiminde bile bilgisayar programlarının değerlendirilmesi sürecinin işlenmesi hakkında veri tasarımı ve soyutlama büyük bir atılım iken bu kitaptaki tasarım işlemine kıyasla yetersiz kalmıştır. How to Design Programs kitabı, SICP'nin birinci sene için eğitim kaynağı olarak verimsizliğini gören yazarların bunu belirtmek için yazdığı bir kitaptır. Kitap özellikle ilerledikçe daha da karşılaşılan tasarım tarifleri içererek, tasarım aşamasının sırrını çözmeye ve basitleştirmeye çalışıyor. Yazar bu kitabı "Programlamaya Giriş" dersinde ders kitabı olarak bir buçuk senedir kullanmaktadır.

Bilgi'de sahip olduğumuz deneyimden başka sunabileceğimiz bir anekdotumuz yok. Scheme tabanlı bir girişi tamamladıktan sonra Java'ya geçiş bir problem oluşturuyormuş gibi görünmüyor. İkinci yıl Bilgisayar Bilimleri öğretim görevlileri birinci yıldan gelen öğrencilerin programlama yeteneklerinin arttığını söylüyorlar.

Başka bilimsel çalışmalar da yapıldı. Bu çalışmalar gösterdi ki, birinci yıllarında Scheme gibi fonksiyonel bir dil kullanılan tasarım odaklı programlama aldıktan sonra Java gibi daha popüler bir dile geçiş yapan öğrenciler daha yüksek bir performans sergiliyor. Diğer bir entererasan etki ise, bayan öğrencilerin erkek öğrencilere göre başarı oranı; tasarım odaklı bir yöntemle bayan öğrencilerin başarıları büyük bir oranla artmıştır. Bu fark aynı zamanda hem Java hem de program tasarımı odaklı dersler alan daha üst sınıf öğrenciler tarafından da belirtilmiştir. Öğrenciler arasında büyük çoğunluk tasarım tabanlı bir dersi tercih ederken, bayanlarda bu tercih dörde bir oranında olmuştur[14].

Birleşik Devletlerde ve başka yerlerde programlama dersinin genellikle Java olmak üzere belirli bir dil ile yapılması için baskı yapılıyor. Bu bir eğitim sorununu temsil ederken, asıl problemin kendisi değil. Ana problem, dolaylı olarak belirtilen bir dil seçmenin vereceğimiz programlama eğitimi için yeterli olacağı varsayımıdır.

Tabii ki programlamayı Fortran kullanarak bile iyi öğretebilmek mümkün. The Elements of Programming Style[15] içinde Fortran kod örnekleri bulunan harika bir kitaptır. Ancak, eğer sadece kopyala-yapıştır-değiştir tekniklerini kullanarak program yazabilen öğrenciler mezun etmek istemiyorsak, programlama eğitimimizin önemini ilk gününden itibaren tekrar tasarıma kaydırmalıyız.

## 7 Sonuç

Programlamayı düzgün bir şekilde öğretmek gerçekten de zor. İşaret et-ve-tıkla, kopyala-yapıştır kültürü iyi program geliştirmeyi daha da zor hale getiriyor. Özgün, ses getirecek programlar, üzerinde iyice düşünülmeli ve iyi bir tasarıma sahip olmalı, bir kaç fare tıklaması ile şekillendirilmeye çalışılmamalı. İyi programlama yeteneği modern dünyada önemli bir beceri. Eğer problemleri görmezden gelirse, iyi programlama, üniversitelerdeki küçük seçkin gruplar ve sayıları çok az olan teknolojinin dönüm noktasındaki Google gibi şirketler için bir kaynak haline gelecek. Eğer amaç öğrencilerimize iyi hizmet vermekse, onlara programlar hakkında daha derin düşünmelerini öğretmeliyiz.

## Kaynaklar

- [1] Brooks, F.P. The Mythical Man-Month, Addison Wesley 1995 p 102
- [2] ibid p 182
- [3] Bell, P. Parr, M. Java for Students Prentice Hall 2002
- [4] Deitel, H.M. Deitel, P.J. Java: How to Program Prentice Hall 2003
- [5] Deitel, H.M. Deitel, P.J. C: How to Program Prentice Hall 1992
- [6] Lewis, J. Loftus W. Java Software Solutions Foundations of Program Design Addison Wesley 2001
- [7] King K.N. Java Programming from the Beginning Nortto
- [8] Liang, D.Y. Introduction to Java Programming Prentice Hall 2003
- [9] Riley, D.D. The Object of Java Introduction to Programming Using Software Engineering Principles Addison Wesley 2002
- [10] Felleisen, M. Findler, R. Flatt, M. Krishnamurthi, S. How to Design Programs An Introduction to Computing and Programming The MIT Press 2001
- [11] Kod bu adresten ulaşılabilir: <http://cs.bilgi.edu.tr/chris/akab>
- [12] Abelson, H. Sussman, J. and Sussman, J. Structure and Interpretation of Computer Programs MIT Press, 1984
- [13] Felleisen, M. Findler, R. Flatt, M. Krishnamurthi, S. How to Design Programs An Introduction to Computing and Programming The MIT Press 2001
- [14] Felleisen, M Findler, R. Flatt, M. Krishnamurthi, S. Structure and Interpretation of the Computer Science Curriculum Journal of Functional Programming 14, pp 113-123, 2004
- [15] Kernighan, B.W. and Plauger, P.J. The Elements of Programming Style Yourdon 1974