9. Hafta
2. GNP
Başlangıç

11.4. Etkinlik

Mühendisliğin en önemli özelliklerinden biri de özkaynakların etkin ve verimli bir şekilde kullanılarak amaçlara ulaşılmasıdır. Bilgisayar sistemlerinin etkinliği de tanımlanan işi amaca uygun hız ve güvenlikle en ekonomik şekilde karşılamasıyla değerlendirilir. Bunun için de özkaynakların iyi bir şekilde kullanılması gerekir. Bilgisayarlı sistemlerin özkaynakları, bilgi işleme kapasitesine sahip makineler, bunların işlemcilerinin zamanları, bellekleri, saklama alanlarıdır. Bu özkaynakların iyi kullanımıyla elde edilen etkinlik, genellikle başarımla ilgili bir ister olarak, sistem çözümlenmesi sırasında tanımlanır, iyi bir tasarımla ilk adım atılır ve kodlama sırasında gerçekleştirilir. Ancak, salt etkinlik uğruna, kodun anlaşılabilirliğinden, okunabilirliğinden ve diğer niteliksel özelliklerinden ödün verilmemelidir. Öte yandan, mükemmel özelliklere sahip ancak başarımı düşük yazılımın da bir işe yaramayacağı akıldan çıkartılmamalıdır. Şimdi etkinlikle ilgili dikkat edilmesi gereken ve [5]'te de değinilen birkaç özelliği ön plana çıkartalım:

11.4.1. Kod Etkinliği

Bir yazılımın kaynak kodunun etkinliği ayrıntılı tasarım sırasında kullanılan algoritmaların etkinliği ile doğru orantılıdır. Buna ek olarak, kodlama biçimi yürütme hızına ve bellek yönetimine etki eden çok önemli bir unsurdur. Kod etkinliğini artırmak üzere aşağıdaki kuralları uygulamaya çalışmak gereklidir:

- Tasarım sırasında belirlenen algoritmaları gerçekleştirmek için mantıksal ve aritmetik deyimler kodlamaya geçmeden önce en verimli hale getirilmeye çalışılmalıdır.
- Etkinlik için uzun yordamlar kullanılmamalı, ancak gereksiz yordam çağrılarından da kaçınılmalıdır.
- Etkinliği düşürecek veri yapıları kullanımından kaçınılmalıdır. Örneğin, arama sıklığı çok olan durumlarda ağaç yapısı tercih edilirken, baştan sona taramanın daha sık olduğu durumlarda sıralı liste tercih edilmelidir.
- Yordamlara parametre geçirirken değer olarak değil, adres ya da işaretçi olarak geçirmek daha yararlıdır.
- Aritmetik işlemlerinden makine mimarisi için en hızlı olanları kullanmak daha yararlıdır.
- Programlama dilleri izin verse dahi veri tipleri birbirlerine karıştırılmamalı, uygunsuz atamalar yapılmamalıdır.

- Veri tipleri tanımlanırken gereksinimi en iyi karşılayacak şekilde bellekte en az miktarda yer tutan tipler kullanılmalıdır. Örneğin, fazla hassasiyet gerektirmeyen kayan nokta tipinde, 8 sekizli yerine 4 sekizli kullanımı tercih edilmelidir. Ancak, işlemci mimarisinin kullanmak zorunda olduğu en düşük bit miktarı da göz önüne alınmalıdır.

11.4.2. Bellek Etkinliği

Eskiden bilgisayar sistemlerinin ana ve yardımcı bellekleri maliyetleri nedeniyle oldukça kısıtlıydı. Zaman içindeki teknolojik gelişmelerle artık çok ucuza, yüksek kapasitede bellek kullanımı mümkün olmaktadır. Bu şekilde kesintisiz fiziksel ana belleğin ve yüksek kapasiteli sabit disklerin kullanımı yaygınlaşmıştır. Ancak yine de sınırsız değillerdir, onun için de tasarrufla kullanımında yarar vardır. Aşırı miktarda bellek kullanımı sonuçta genel başarımı olumsuz olarak etkiler. Özellikle boyut sorunu olan mikroişlemcili gömülü sistemlerde bellek yönetimi oldukça önemlidir. Bir de çok uzun süre kesintisiz çalışması gereken sistemlerde, dinamik bellek yönetimi ve atık toplama çok daha fazla önem taşımaktadır. Bellek etkinliğini artırmanın birkaç yolu aşağıda sıralanmaktadır:

- Sık sık ekleme ve çıkarma yapılan, toplam eleman sayısı belirsiz olan veri yapılarında dinamik bellek kullanılmalı, ancak bu yapıların yapıcı ve yokedici yordamları mutlaka uygun şekilde kullanılmalıdır. Bazı programlama dillerinde atık toplama işlemi otomatik yapılırken bazı dillerde açıkça denetimi gereklidir.
- Veri tipleri tanımlarken arzu edilen hassasiyet ve uçdeğerlere en uygun temel veri tipi seçilmelidir. Örneğin, 1 ile 100 arasında tamsayı kullanılacaksa, bu veri tipi bir sekizli uzunluğunda kısa bir tamsayı (`short int`) olabilir.
- Bileşik veri tiplerinin tanımlarına hiç kullanımı olmayan alanlar (field) dahil edilmemelidir. Hele bu tiplerden dizi yaratılması gereken durumlarda daha da dikkatli olunmalıdır.
- Bilgisayarların bellek erişimlerinde kullanılan veri yolu genişlikleri bugün için 2 ile 8 sekizli arasında değişmektedir. Bu da bir defada bellekten işlemciye aktarılacak veri miktarını belirlediği için bir bitlik veri ile 64 bitlik veri aynı hızda yazılıp okunmuş olur. Onun için, veri tipleri tanımlanırken hedef sisteme uygun sözcük uzunluğuna göre değerler verilmeli, gereksiz kısaltmalar yapılmamalıdır.

11.4.3. Giriş/Çıkış Etkinliği

Bir bilgisayar sisteminin genellikle iki tür giriş/çıkışı vardır. Bunlardan birincisi, insanla olan etkileşim, yani kullanıcı arayüzü, diğeri de başka aygıtlarla ve çevre birimleriyle olan etkileşim, yani veri arayüzüdür.

Kullanıcı arayüzü, insan mühendisliğinin bir uygulama alanı ortaya çıkar. Etkinliği, kullanım kolaylığı, kullanıcı dostluğu, öğrenme çabasının azlığı ve anlaşılabilirlik açısından değerlendirilir.

Başka donanımlarla olan giriş/çıkış işlemlerinin etkinliği alt düzey programlamada önemli bir deneyim ve bilgi birikimi gerektirir. Bu işlemler uygulama alanı ve bilgisayar türüne göre farklılık göstermesine rağmen bazı genel kuralları şu şekilde sıralayabiliriz:

- Giriş/çıkış istekleri yazılımın genel mimarisi içinde olabildiğince düşük düzeyde tutulmalıdır. Çünkü, yürütme sırasındaki program akışı, işlemciden çok giriş/çıkışın sonuçlanması için bekler.
- Her türlü giriş/çıkış işlemi iletişim yükünü azaltmak için tamponlanmalıdır.
- Giriş/çıkış sırasında kullanılan veri tipi ve büyüklüğü istek sıklığına göre en iyi hale getirilmelidir.
- Veri aktarım hızı ilgili aygıtın algılayabileceği en uygun düzeyde olmalıdır.
- Dışarıdan alınan her türlü verinin geçerlilik testi yapılmalı, ondan sonra kullanılmalıdır.
- Özellikle kontrol sistemleri için, dışarıdan alınan her türlü verinin geçerlilik testi yapılmalı, kullanımdan önce kontrol edilmeli, hata yakalayıcı ve düzeltici yöntemler kullanılmalıdır (Hamming kodları, hata yakalama biti, akış denetimi gibi).
- Dışarıya aktarılan verilerin doğruluğu ve tutarlılığı sona ana kadar kontrol edilmeli ve teknik anlaşmaya uygun veriler gönderilmelidir.
- Giriş/çıkış yordamları genellikle işletim sisteminin türüne göre değişen sürücüler kullanırlar. Taşınabilirliği artırmak için, bu tür sürücüler daha standart arayüzlerle kuşatılmalı (wrapping), asıl program koduna karıştırılmamalıdır.

11.4.4. Atık Toplama

Tüm yazılımlar bilgisayar donanımları üzerinde sınırlı miktarlardaki özkaynakları kullanarak çalışırlar. En önemli özkaynaklardan biri ana ve yardımcı bellektir. Programlama dilinin özelliğine göre, kodlayıcı, fiziksel ana ve yardımcı belleği istediği gibi kullanabilir. Özellikle dinamik sistemlerde bellek kullanım gereksinimi zaman içinde farklılık gösterdiğinden, uzun süreli çalışmalarda bellek sıkıntısına düşmemek için kullanımı sonra eren nesne ve veri yapılarının silinmesi gereklidir. Silme işlemi geliştiricinin kendi algoritmalarıyla yapılabileceği gibi, bir ara katman, bir yürütme sistemi veya işletim sistemi, hatta özel bir donanım tarafından da yapılabilir. Bu işleme *atık toplama* (garbage collection) denir.

Otomatik olarak yapılan atık toplama işlemiyle geliştiriciye yalnızca fiziksel olarak kısıtlanabilen sonsuz büyüklükte bir bellekle çalışma olanağı verilerek kullanımı bitmiş, artık gereksinim duyulmayan nesne ve yapılarla uğraşmamaları sağlanır. Sistem, bir nesnenin başka bir nesne ya da değişken tarafından erişiminin mümkün olmadığını sezebilirse bu nesnenin bulunduğu alanı başka bellek isteklerinde kullanılmak üzere serbest bırakabilir; bu işlem kullanıcı tarafından fark edilmez.

Yazılım biriminin işlemci üzerinde yürütülmesi sırasında havuz bellekten (heap) ayrılan bellek öbekleri kullanımı bitince tekrar işletim sistemine geri verilmelidir. Yine sabit diske yazılan geçici dosyalar, işleri bitince silinmeli, diskte yer kaplamaları engellenmelidir. Bellek kullanımı özellikle sürekli çalışması gereken dinamik sistemlerde son derece önemli olduğundan kullanımı tamamlanmış veri yapılarının ve nesnelerin uygun bir şekilde yok edilebilmesi her geliştirme yönteminde ele alınması gereken önemli bir konudur.

Ana ve yardımcı bellek yanında, bir yazılım birimi, dosya tutamacı (file handle), giriş/çıkış aygıtı, paylaşılır bellek, semafor, soket gibi özkaynakları da kullanabilir. İşletim sistemine göre değişiklik göstermesine rağmen, bu tür özkaynakların kullanımı sınırlıdır. O nedenle, kullanımları sona erdiğinde derhal serbest bırakılmalıdırlar. Normal olmayan bir şekilde çalışması duran bir yazılım birimi kullanmakta olduğu özkaynakları serbest bırakabilmelidir. Bazı işletim sistemleri bunu güvenli bir şekilde sağlarken, bazıları çöken yazılımla birlikte tüm sistemin de çökmesine neden olmaktadır.

Simgesel dillerle yapılan bellek yönetimi, geliştiriciler tarafından yazılan kaynak kod düzeyinin altında, derleyici desteği, işletim sistemi kolaylıkları veya özel donanımlar yardımıyla sağlanır. Bazı dillerde atık toplama düzeneği tamamen saydam olarak ve makine düzeyine en yakın biçimde gerçekleştirilebilir; onun altında bir başka düzey daha yoktur.

Kod içinde açıkça yapılan atık toplama yanında (işaretçilerin gösterdikleri yerde yaratılan bellek parçasının işi bitince açıkça silinmesi) otomatik atık toplama için çeşitli algoritmalar geliştirilmiştir. Şimdi bunların bazılarına kısaca değinelim.

- **İşaretle ve temizle yöntemi**

  Bu yöntemde, dinamik bellekte yer tutan tüm nesne ve veri yapıları incelenerek hangi nesne ve yapılarla ilişki halinde oldukları belirlenir; aralarında ilişki bulunan nesneler işaretlenir. Tüm bellek içeriği tarandıktan sonra, ikinci bir geçiş yapılarak hiç işaretlenmemiş nesneler ve yapılar temizlenir. Bu işlem belirli aralıklarla tekrarlanır. Tarama sırasında genellikle tüm yürütülen işlemler kısa bir süre için durdurulur. Sistem genelinin bu kısa duraklamadan etkilenmemesi için önlem alınmalı, mümkünse sistem boşta iken veya kritik anlar dışında yapılmalıdır.

- **Referans sayma yöntemi**

  İşletim sistemi veya yazılım birimi tarafından kullanılabilen bu yöntemde isteğe göre ayrılan bellek öbeklerinin bir kaydı tutulur. Öbekleri kullananların sayısı belirli bir yerde bulunan sayaçlara kaydedilir. Kullanıcı sayısı azaldıkça sayaçların değerleri de azaltılır. Sayacın sıfır değerini taşıması o öbeğin kullanılmadığını gösterir. Uygun aralıklarla ana denetleyici tarafından yapılan taramalarda referans değeri sıfır olan bu öbekler belirlenerek gerekli temizlik yapılır. Aralıklı tarama yerine belirli bir algoritma kullanmak da mümkündür. Bu yöntem kod içinde açıkça yapılan bellek denetimlerinde de kullanılabilir.

- **Yarılalan kopyalama tekniği**
  Genellikle işletim sistemi düzeyinde kullanılan bu teknik, işaretle ve temizle yönteminde taramadan dolayı oluşan ara vermeyi kaldırarak gerekli temizliği sağlar, ancak daha fazla bellek gereklidir. Bu yöntemde bilgisayar belleği **A** ve **B** olarak ikiye ayrılır. Yeni nesneler ve yapılar **A** alanında yaratılır. Belirli bir zaman sonra, ya da sistem boşta (idle) iken **A** yarılalanındaki tüm erişilebilir nesne ve yapılar **B** yarılalanının bir ucundan başlayarak yan yana kopyalanır. Bir yapı içinde olup diğer bir yapıyı gösteren işaretçiler de buna göre ayarlanır. O anda, **A** yarılalanında kalan yapıların erişilemez durumda, yani atık oldukları bilinmektedir. Buna göre de **A** alanında gerekli temizlik yapılır. Bundan sonra **A** ve **B** yarılalanlarının rolleri değiştirilerek çevrime devam edilir.

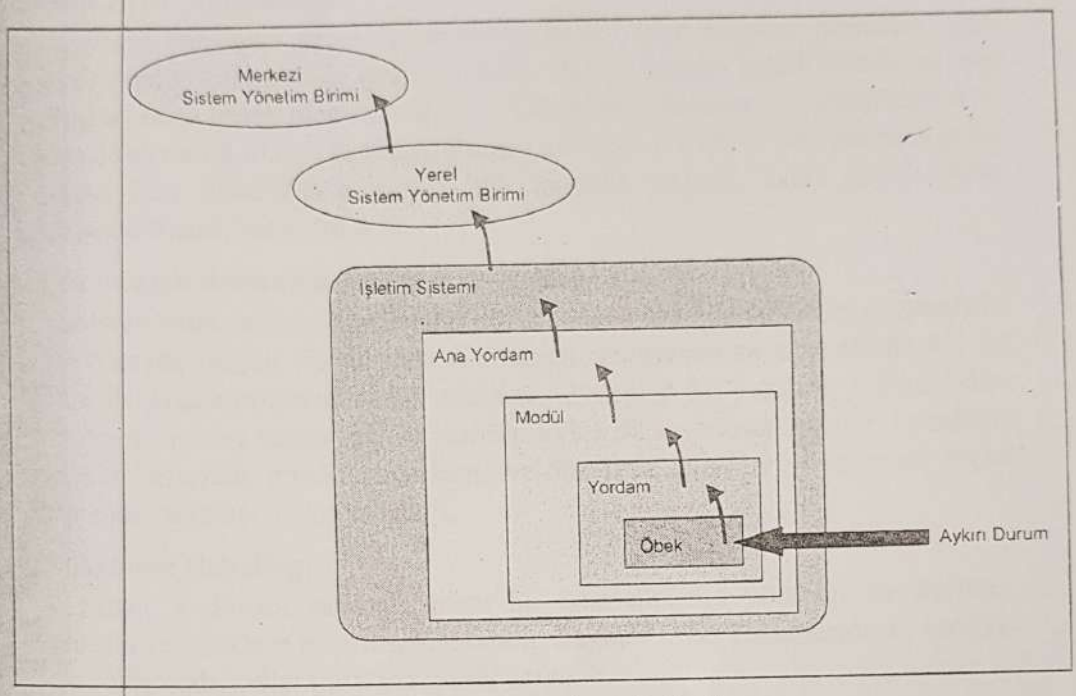
Günümüzde bazı programlama dilleri dinamik bellek yönetimini kendi içinde yapmakta, hatta bazıları dinamik yapılanmaya izin vermemektedir. JAVA dili işaretçi kullanmayan ve kendi atık toplama düzeneğine sahip bu dillerden biridir.

11.4.5. Aykırı Durumların Kotarılması

Aykırı durumlar (exception), bir programın çalışmasını geçersiz bir komut yürütme, yanlış veri işleme ya da başka nedenlerle istenmeyen bir şekilde ve denetim dışında sonlanmasına neden olan durumlardır. Normalde bir yazılım birimi hiçbir zaman ana denetim dışında sonlanmamalıdır. Bunun için de, öncelikle yazılımın kodlaması yapılırken hata üretebilecek kısımların çok dikkatle yazılması gereklidir. Buna rağmen olağan dışı durumların oluşması durumunda programın nasıl davranacağına ilişkin standartlar ve yöntemler önceden belirlenmeli, kodlayıcıların bunları uyguladıkları denetlenmelidir.

Günümüzdeki programlama dillerinin bir kısmı aykırı durumları yakalayabilen düzeneklere sahiptir (C++ için `try-catch`, ADA için `exception` öbekleri gibi). Bu düzeneklerin kullanımı için de kodlayıcı tarafından aykırı durumlar tanımlanmalıdır. Çok büyük yazılımlarda, standardı sağlamak için bu tanımlar daha üst düzey tasarımlar sırasında yapılmalıdır. Yürütme sırasında herhangi bir olağan dışı durum oluştuğunda Şekil-7.6 da gösterilen sıraya göre çeşitli geri kazanma işlemleri yapılabilir. Aşağıda belirttiğimiz herbir basamakta durum düzeltilemezse hatanın cinsi ve yeri tanımlarak üste doğru taşınmalıdır.

- **Kod öbeği içinde kotarma**
  Önceden potansiyel bir hata kaynağı olarak belirlenen kod parçası bir öbek şeklinde hata yakalama düzeneği ile koruma altına alınarak hata oluştuğu anda neler yapılacağı kotarma kısmında yer alır. Bu öbek bir döngü şeklinde olabilir. Hataya neden olan işlem birkaç kez veya sonsuza dek tekrar edilir (bir aygıttan okuma yapma gibi). Hata kotarımı döngü içinde kalır, döngüden çıkıldığında program akışı devam eder. Eğer hatayı düzeltmek mümkün değilse hata bir üst düzeye iletilir.



Şekil-7.6. Aykırı durumların kotarılması.

- **Yordam içinde kotarma**
  Her yordamın sonuna bir hata kotarıcı yerleştirmek alışkanlık halinde olmalıdır. Yordam bedeninde bulunan deyimlerin herhangi birinde hata oluştuğunda veya bir kotarma öbeğinden hata iletildiğinde sonraki deyimler yürütülmeden program akışı kesilir ve denetim yordam sonundaki hata yakalama düzeneğine geçer. Buradaki kotarma durumuna göre, hatayı giderecek bir önlem alınabilir, hatanın hangi modül ve hangi yordam içinde oluştuğu bilgisi raporlanabilir ya da hata bir üst düzeye iletilir. Bu arada yordam dışında yapılmış olan işler geri alınır (evrensel değişkenlerin değerlerinin eski haline alınması gibi). Bir üst düzeyde genellikle bir başka yordam vardır. Bu sıra ana yordama kadar gider. Ana yordamın kotaramaması kontrollü sonlanmayı gerektirir.

- **Modül içinde kotarma**
  Bazı programlama dilleri modül olarak paket yapıları kullanırlar. ADA dili bunlardan biridir. Paketlerin son kısımlarında hata yakalama düzenekleri bulunur. Genellikle ilk yaratmadan (elaboration) kaynaklanan veya yordamlardan iletilen hataları paket düzeyinde yakalamak ve raporlamak için kullanılır.

- **Ana yordamda kotarma**
  Her programın bir tek ana yordamı bulunur. Program belleğe yüklendikten sonra ana yordamın deyimleri sırayla yürütülür. Ana yordamın son kısmında en son hata yakalama düzeneği bulunur. Bundan sonra hata program dışına, yani işletim sistemine iletilir, kontrollü sonlama meydana gelir.

- **Durdurma**
  İşletim sistemine hata kodu ile dönen bir programın yürütülmesi sona ermiş demektir. Bundan sonra sistemin genel durumu hakkında ya işletmen ya da sistemi denetleyen bir sistem yönetim birimi karar verir. Bazı sistemlerde, hataya düşen programın kendi durumunu sistem yönetim birimine bildirmesi durumunda o program yönetim birimi tarafından sonlandırılır ve yeniden başlatılır. Programın çökerek devre dışı kaldığının fark edilmesi durumunda ise program yönetim birimi tarafından yeniden başlatılır.

11.5. Temel İlkeler

İyi bir yazılım geliştirmek, eski deyimiyle, program yazmak, kişinin aklını kullanma yeteneğine, yeterli bir beğeni duygusuna ve sabıra sahip olmasını gerektirir. Bunların hepsini bir anda uygulayıp bir kerede mükemmel bir yazılım geliştirilemez. Denemeler yaparak, başka kodları inceleyerek ve karşılaştırarak bu konudaki deneyim arttırılabilir. Bu kısımda iyi bir yazılım gerçekleştirimi için gerekli kurallara değineceğiz.

11.5.1. Kodlamada Niteliksel Özellikler

Programlama dillerinin tasarımı ve karşılaştırılması için çeşitli niteliksel özellikler kullanılır. Yazılımlarda aranan nitelikler arasında da yer alan bu ilkeler, mutlaka uyulması gereken katılıkta değillerdir; ancak ideal duruma ulaşmada temel adım niteliğini taşırlar. Şimdi, [11]'de belirtildiği gibi, özel bir dil seçmeksizin bu ilkeleri görelim:

- **Soyutlama (abstraction)**
  Aynı şeyi birçok defa ifade etmekten kaçınmak gereklidir. Bunun için tekrarlama yapan döngü ortadan kaldırılmalıdır. Örnek olarak, bir döngü içinde yer alan bir atama deyimi döngü dışına çıkarıldığında işlem bozulmuyorsa bu atamanın sürekli tekrarlanmasına gerek yoktur. Küçük bir ayrıntı olarak görülebilen bu noktanın aslında programın genel başarımına büyük etkisi vardır. Bir başka örnek de program içinde tekrarlanan kısımların bir modülde toplanması ve daha sonra bu modülün dilin kurallarına göre ilgili yerlere dahil edilmesidir. Bu şekilde ayrı ayrı derleme avantajı da doğar. Bunun bir ileri düzeyi de hatalardan ayıklanmış ve test edilmiş kütüphanelerin kullanımıdır. C++ ve JAVA dillerinde sınıfların ve nesnelerin kullanımı, ADA dilindeki paketler bu ilkenin en güzel uygulanmış halidir.
- **Bilgi Gizleme (information hiding)**
  Modüllerin yalnızca gerekli bilgileri saklaması sağlanmalıdır. Modül kullanıcısı, bir modülü doğru bir şekilde kullanabilmek için gerekli ve yeterli bilgilere sahip olmalıdır. Kaynak kod dosyasının modül olarak kullanımı, görünürlük kurallarının uygulandığı kod öbekleri, sınıfların özel ve korunmuş (private, public) kısımlarındaki veri denetimi bu ilkenin uygulanışına birer örnektir. Bu ilkenin bir başka söyleniş şekli de verilerin yalnızca "bilmesi gerekenler" tarafından bilinmesidir.

- **Otomasyon (automation)**
  Ne zaman çalışması gerektiği önceden belirli olan işlevler otomatik, yani kendiliğinden çalışır hale getirilmelidir. Belirli zaman aralıklarında çalışan döngüler buna örnek olarak verilebilir. Döngünün denetimi ve yürütülmesi bu yapı ile otomatik olarak sağlanır. Bunun için ayrı bir işlem yapılmasına gerek yoktur. Bazı dillerdeki otomatik atık toplama sistemi, saate dayalı işlem başlatma önemli birer örnektir.

- **Çok Düzeyli Korunma (defense in depth)**
  Yazılımın hatalara karşı korunması için düzeyler halinde önlemler alınmalıdır. Bir düzeyde oluşan bir hatanın o düzeyde yakalanamaması halinde bir üst düzeyde yakalanmasına olanak sağlanır. Güvenilir bir yazılım, kullanıcıdan, donanımdan veya başka yazılım birimlerinden dolayı oluşan hataları yakalayıp kendini korumalı, gerekli raporlama ve düzeltme işlemlerini yapmalı, hiçbir durumda tamamen çökmemelidir.

- **Etiketleme (labeling)**
  Kullanıcı, kendisine anlamlı gelen bir isimlendirme yöntemi ile verilere, yapılara ve öbeklere erişim sağlamalıdır. Saymal tipler (enumeration) bu ilkenin kodlamada kullanılmasına iyi bir örnektir.

- **Belirgin Arayüz (apparent interface)**
  Tüm arayüzler açık ve belirgin olmalıdır. Kullanıcı neyi nasıl kullanacağını rahatlıkla anlayabilmelidir. Gerek yordam bildirimleri gerekse paket veya sınıf arayüzleri dilin elverdiği ölçüde açıkça tanımlanmalı, gerekirse ek açıklamalarla anlatılmalıdır.

- **Taşınabilirlik (portability)**
  Kodlama yapılırken belirli bir donanıma bağımlı kalınmamalıdır. Kod yazımı sırasında yalnızca o an kullanılan ortamı düşünmek hatalı olur. Yazılımın başka bir donanım üzerine taşınması halinde aynen çalıştırılması veya yeri bilinen, çok küçük değişikliklerle aynı sonucun alınması gereklidir. Kullanılan bilgisayarın sözcük uzunluğu, tamsayı ya da kayan nokta için ayrılan bit adedi, bellek kullanım şekli, grafik donanım özelliklerinin kullanımı önemli örnekler olarak verilebilir. Taşınabilirliği artırmak için olası seçeneklere göre kodlama yapılmalı, temel tipler, veri yapıları dilin tanımladığından başka isimlerle kullanılmalıdır (örneğin C dilinde bulunan ve yazılım boyunca değerinin 16 bit olduğu bilinen `short int` temel tipi kullanılmışsa, donanımın 32 bite yükseltildiği durumda uyumsuzluk yaşanacak ve `short int` ile yapılan tüm bildirimlerin teker teker değiştirilmesi gerekecektir. Bu nedenle `typedef` ile yeni bir tip tanımlanmalıdır).

- **Güvenlik (security)**
  Yazılımda kullanılan verilerin güvenliğinin sağlanması ve bunlara erişimin uygun şekilde kısıtlanması gereklidir. Bir paket ya da sınıfın iç verilerinin yetkisiz kullanıcılar tarafından değiştirilebilmesi önlenmelidir. Bu amaçla,

sınıfların veya modüllerin verileri dilin özellikleri (`private` gibi) kullanılarak olabildiğince korunmalı, evrensel veri kullanımından kaçınılmalıdır.

- **Basitlik (simplicity)**
  Yazılım olabildiğince basit olmalıdır. Dilin yapısında çok az sayıda kavram olmalı ve bunları birleştiren basit kurallar bulunmalıdır. Yazılan programlarda da basitlik hedef alınmalı, anlaşılabilirliği azaltan karmaşık yapılardan kaçınılmalıdır.

- **Genel Yapı (general structure)**
  Bir yazılımın durağan yapısı yürütme sırasında kullanılan dinamik yapısı ile açık bir şekilde bağdaşmalıdır. Yazılımın kaynak kodunun metinsel görüntüsü onun dinamik olarak çalışması hakkında fikir verebilmelidir. Kodun kısa yordam ve fonksiyonlardan oluşturulması, bir yordamın çok sayıda ardışık satırlar yerine bunların ayrı ayrı bulunduğu başka yordamları çağırarak işlem yaptırılması, görünürlük kuralları ve kod öbeklerinin kullanılması, kod yazılırken satırbaşı ve öbek işaretleri gibi metinsel özelliklere dikkat edilmesi bu ilkenin uygulandığı örneklerdir.

- **Sözdizimsel Tutarlılık (syntactic consistency)**
  Sözdizimsel ve anlamsal benzerlikler anlaşılabilirliği artırmaktadır. Kod yazarken de anlaşılabilirliğin korunması için modül, değişken ve yordam isimlendirmesinde, programlama dilinin sözdizim kurallarına uygun yapılar kullanılmasına dikkat edilmelidir. Kodlama dilinin Türkçe ya da İngilizce olması seçeneğinde bu özellik göz önünde bulundurulmalıdır. Örneğin, `if-then-else` yapısında kullanılan koşul içinde Türkçe ve farklı anlam taşıyan bir sözcük olması bu yapının düz bir şekilde okunurluğunu azaltabilecektir.

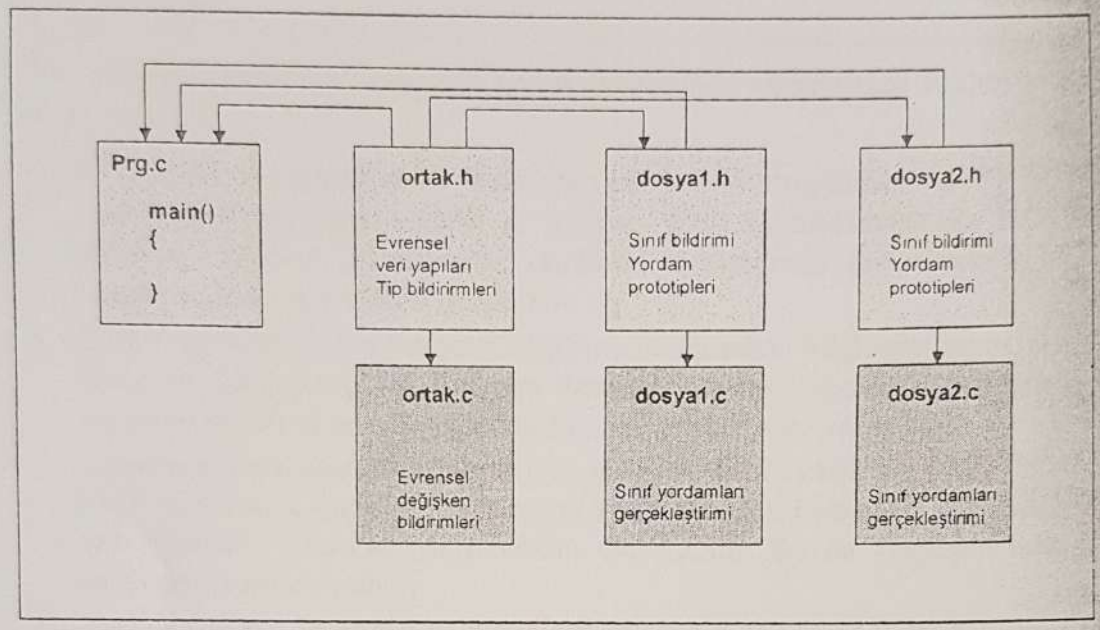
- **Sıfır-Bir-Sonsuz**
  Yazılımda yalnızca, sıfır, bir ve sonsuz değerleri dikkate alınabilir özellik taşımalıdır. İyi bir yazılım kaynak kodu içinde sayılara bağlı herhangi bir kısıtlama bulunmamalıdır. Yazılımın tasarımı sırasında hiçbir özellik, işlev ya da yapı belirli ve sabit bir sayıya göre yapılmamalı, gelecekte doğabilecek artmalara karşı tedbirli olmak gereklidir. Örneğin, bir personel bilgi ve takip yazılımında en fazla 100 kişi için hazırlanmış bir dizinin daha sonra artan personel sayısı nedeniyle genişletilmesi gerektiğinde, tüm kodun taranıp dizi boyutuyla ilgili değişkenlerin değiştirilmesi gerekecektir. Eğer bazı kısımlarda bu tür sabit uzunluklara göre işlem yapılmaktaysa sorunlar ortaya çıkacaktır. Bir başka örnek olarak, başlangıçta iki makine üzerinde çalışacak bir yazılımın kaynak kodunda makinelerden biri ve diğeri olmak üzere iki seçeneğe göre geliştirilen algoritmalar, makine sayısının üç veya daha fazla olması durumunda çalışmayacaktır. O nedenle, "bir" ve sistem içinde bir sabit olarak tanımlı "en fazla makine sayısı" arasında döngü kurulmalıdır.

11.5.2. Modül Oluşturma

Yazılımın büyüklüğü arttıkça yordamları ve verileri birbirlerine olan bağlılıkları düşünülerek birimlere ayırmak gereklidir. Çoğu geliştirme ortamında dosya en küçük yazılım birimidir. Birbirleriyle ilişkili yordamlarla bunlara ait verileri bir dosya içine koyarak bir *modül* oluşturulabilir. Nesneye yönelik programlamada modül olarak sınıflar da kullanılabilir. Ancak her sınıf için bir dosya yaratmak pratikte kullanışlı olmayacağı için benzer sınıfların tanımları bir dosyaya (`dosya_adı.h` gibi), onlara ait işlemlerin bedenleri ayrı bir dosyaya (`dosya_adı.c`) konur. Bu şekilde modüler programlama tekniği uygulanarak Bilgi Gizleme ilkesine bağlı kalınmış olur. Dilin özelliğine göre, dosya şeklinde olan modüller ayrı derleme olanağı sağlayarak büyük yazılımların paralel olarak geliştirilebilmesini kolaylaştırır.

Dosya halinde modül oluşturmada dikkate alınması gereken noktalara değinmekte yarar vardır:

- Aynı tür işlevlere sahip yordamlar belirlenmeli ve bir modülde toplanmalıdır.
- Modülün açık, anlaşılır bir arayüzü olmalıdır. Bu da genellikle bir başlık dosyası (header) veya paket belirtimidir.
- Modül olan dosya içinde evrensel olarak kullanılması gerekli tip, veri yapısı, değişkenler ve sabitler görünürlük kurallarına göre modülün başında bildirilmelidir.
- Bir yazılım birimi, yani program için evrensel olan tipler, veri yapıları, sabitler ve makrolar ayrı bir dosya içinde toplanmalı ve bu dosya diğerleri tarafından erişilmelidir.
- Programı çalışmaya başlatan ana yordamı tek başına ayrı bir dosyaya yerleştirmek program akışını takip edebilmek ve en son hata yakalama düzeyini gerçekleştirebilmek için daha uygun olur.
- Her tür program geliştirmede örnek olacak C++ dili için bir dosya yapısı Şekil-7.7 de görülmektedir. Dosya adedi, o anki gereksinime göre belirlenmeli, modüller bu dosyalara dağıtılmalı, dosya sayısında aşırıya kaçılmamalıdır.
- Dosya sayısının çok artması halinde her modül için gerekli kod dosyaları ayrı dizinlere yerleştirilmelidir.
- Yaratılan herbir dosyanın baş kısmına yeterli bir açıklama konmalıdır. Bu açıklama, dosyanın adını, varsa düzenleşim yönetim sistemi numarasını, sürüm numarasını, yazan kişi veya kişilerin adlarını, ilk yaratma ve değişiklik tarihlerini, o dosyada toplanmış işlevlerin ve veri yapılarının genel bir özetini, varsa başka kaynaklardan alınmış kod parçaları hakkında bilgileri içermelidir. Bu amaçla, standart bir biçim yaratılması ve tüm kaynak dosyalarının başına konması kodlama disiplininin korunması açısından oldukça iyi bir yöntemdir.



Şekil-7.7. Modüler program yapısı (C ve C++).

11.5.3. Kod Yazımı

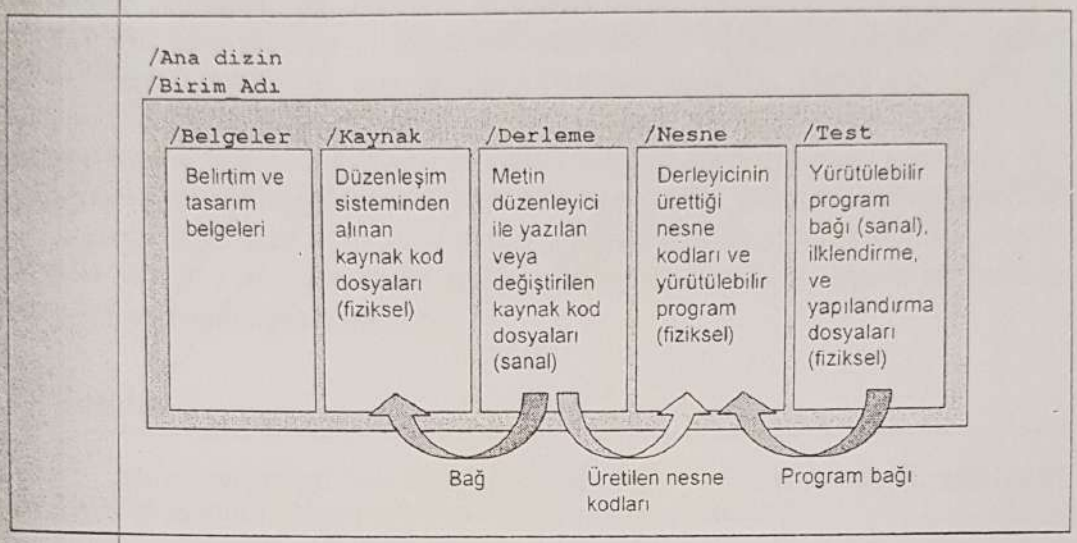
Yazılım geliştirmenin çözümleme ve tasarım aşamaları ne kadar iyi olursa olsun gerçekleştirim iyi olmadığı takdirde sonuca ulaşılamaz. Kod yazmak, her ne kadar bir sanat niteliğinde olsa da, bir takım kurallara uymak ve disiplinli olmak gereklidir.

Bir programlama dili bilen herkes "program" yazabilir, fakat herkes "yazılım" geliştiremez. Tüm yazılım geliştirme aşamalarının gerçekleşmesi ve belirli bir mühendislik bilgisinin denetiminde kod yazılması nitelikli yazılım için temel oluşturur.

Yazılımın kaynak kodu seçilen bir programlama dilinin kurallarına uygun olarak yazılır. Dilin kendi sözdizim kuralları yanında bazı temel ilkelerin de kullanılmasında yarar vardır. Dile yönelik bazı standartlar, yazım biçimleri tanımlanmıştır. Örneğin, ADA dili için Amerika Birleşik Devletleri Savunma Bakanlığının çıkarmış olduğu "Ada Style Guide" bu dil için çok yararlı kodlama önerileri sunmaktadır. C ve C++ için öneriler çok çeşitli kaynaklarda yer almaktadır. Bunların yanında, genel olarak kodlama önerilerimiz Ayrıt 14.3 te verilmektedir:

Kodlama, ana sistem üzerinde, uygun bir dizin yapısı kullanılarak oluşturulan proje ortamında yapılır. Dosya bağlarını destekleyen bir işletim sistemi kullanılıyorsa, asıl kaynak kod dosyaları fiziksel olarak bir dizinde, ona bağlı sanal dosyalar asıl derleme dizininde oluşturulursa, tehlikeli bir işletim sistemi komutunun yanlışlıkla kullanılması sonucu dosyaların silinmesi engellenmiş olur.

İdeal olarak, her çalışma gününün sonunda geliştirilen kodlar birleştirilerek yeni bir sürüm üretilmeli ve çalışma alanının yedeklenmesi sağlanmalıdır. İdeal bir çalışma ortamı dizin yapısı Şekil-7.8 de görülmektedir:



Şekil-7.8. Çalışma ortamı dizin yapısı.

7.6. Belgelendirme

Yazılım gerçekleştirim aşamasında mutlaka bir belge üretilmesine gerek yoktur. Ancak, kodlama sırasında elde edilen bazı bilgileri (deneyim, test sonuçları, önemli noktalar gibi) gelecekte kullanabilmek üzere düzenleşim yönetim sisteminde saklamak gerekebilir.

Gerçekleştirime yönelik belgelendirme çalışmalarını şu şekilde özetleyebiliriz:

- Herbir kaynak kod dosyasının başında, bir başlık kısmı bulunmalıdır. Bu kısımda, dosyanın tanımlayısı olabilece ad veya numara, dosya içeriği, yazarı, yaratılma ve değiştirilme tarihleri, mülkiyet hakları gibi bilgiler yer almalıdır.
- Kodlamada çalışmış kişilere daha sonradan yararlı olacağı düşünülen önemli noktaları, derleme sırasında karşılaşılan sorunları ve bunların nasıl çözüldüklerini anlatan kayıtlar bir dosyada tutulmalıdır (Bazen bu dosyalara "`okubeni.txt`" şeklinde isim verilir). Bu açıklama dosyaları da mutlaka diğer kaynak kod dosyaları gibi düzenleşim yönetim sistemine konmalıdır.
- Küçük programlar için her türlü derleme bilgisi ana fonksiyonun bulunduğu dosyanın başına, açıklama halinde yazılmalıdır.
- Ana ve test sistemi üzerinde yapılan testlerde kullanılan her türlü yazılım, ayarlama ve yapılandırma gerekli açıklamalarla beraber kaynak kod yanında düzenleşim sistemine girmelidir.
- Kod içinde kullanılan açıklama satırları, anlamlı değişken ve yordam isimleri, etiketler de kodun anlaşılabilirliğini artırarak bir tür belgelendirme sağlarlar.
- Kod yapısını, tasarım belgelerinde yer almayan çeşitli bilgileri içeren, açıklayıcı özellikte her türlü belge dosyası kaynak kod ile beraber düzenleşim sisteminde saklanmalıdır.

7.7. Riskler

Her aşamada olduğu gibi yazılım gerçekleştiriminde oluşabilecek riskleri de şöyle sıralayabiliriz:

- *Dilin etkin kullanılamaması*: Kodlayıcı personelin programlama diline yeterince hakim olmaması sonucu dil yapıları etkin kullanılamayabilir. Kodun deneyimli kişilerce incelenmesi, eksikliklerin bulunarak giderilmesi ile bir öğrenme süreci geçirilmesi gerekebilir.
- *Geliştirme ortamının kısıtları*: Geliştirme ortamındaki bilgisayar sayısı, derleyici ve diğer gereçlerin kullanım lisansları gibi bazı kısıtlamalar personel sayısının yeterli olması durumunda dahi önemli sorunlar yaratabilir.
- *Tasarımın tamamının koda dönüştürülememesi*: Karmaşıklık veya bazı eksiklikler sonucunda tasarımın tamamının koda dönüştürülememesi durumunda bazı isterlerin karşılanması mümkün olmayabilir. Bu da yazılımın hatalı çalışmasına neden olabilir.
- *Yanlış kodlama*: Tasarımın yanlış anlaşılarak yanlış kodlanması ile çözümleme ve tasarım doğru yapılmış olsa dahi hatalı bir yazılım ortaya çıkabilir.
- *Mantık hataları*: Kodlama sırasında bulması çok güç mantık hataları yapılabilir.
- *Eksik hata yakalama düzenekleri*: Hata yakalama düzeneklerinin eksik yerleştirilmesi nedeniyle küçük hatalar yazılımı tamamen çökertebilir.
- *Düşük okunabilirlik*: Kodun okunabilirliğinin az olması geliştiricinin kendisine dahi zorluk çıkarabilir.
- *Gereksiz kod parçaları*: Dikkatsizce yazılmış bazı kod parçaları genel başarımı olumsuz olarak etkileyebilir. Örneğin, bir döngü içinde fazladan çağrılan bir yordam, birçok kez tekrarlanınca işlevsel sorun yaratmamasına rağmen önemli miktarda özkaynak (işlemci, ana ve yardımcı bellek) kaybına neden olabilir.
- *Kod inceleme yapılmaması*: Kodlayıcılar işi en iyi kendilerinin bildiği kanısında olabilirler. Bu nedenle, bir başkası tarafından yapılması gereken kod inceleme işleminin atlanması ileride başka sorunlar çıkarabilir.

Burada belirtilen riskler göz önünde bulundurularak kod gözden geçirmesi yapılmalı ve doğrulama işlemlerinde dikkatle takip edilmelidir.

7.8. Özet

Gerçekleştirim aşamasında, tasarım, bir geliştirme ortamı üzerinde programlama diline ve makine koduna dönüştürülerek yürütülebilir kod oluşturulur. Programlama dillerinin tarihçesinin ve genel özelliklerinin bilinmesi, uygulamada dil seçimi için yarar sağlar. Görev kritik olan ve sıkı gerçek zamanlılık gerektiren sistemler için uygun özellikler taşıyan programlama dillerinin kullanılması daha yararlıdır. Kod çevrim işleminde kullanılan derleyiciler, kaynak kod dosyasında bulunan metni, dilin sözdizimi kurallarına uygun olarak okur, hata yoksa bunları bilgisayarın anlayabileceği komutlardan oluşan makine diline çevirir. Birden fazla dosya ve kütüphaneler de