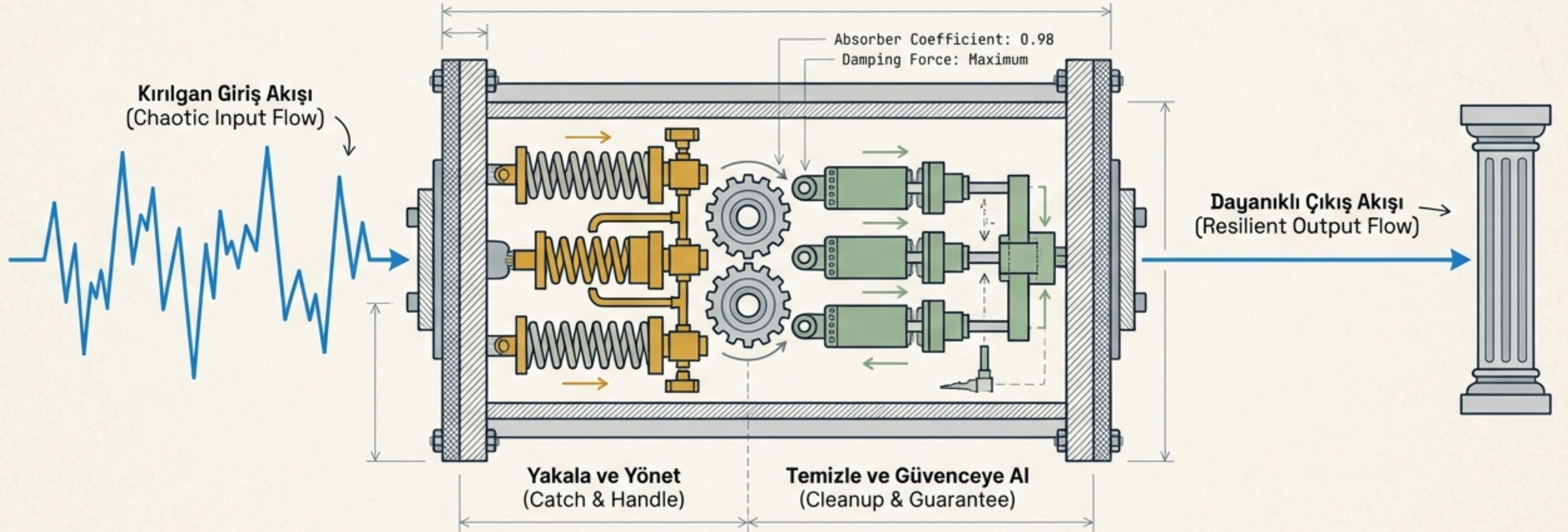


Kırılganlıktan Dayanıklılığa: C#'ta Hata Yönetimi

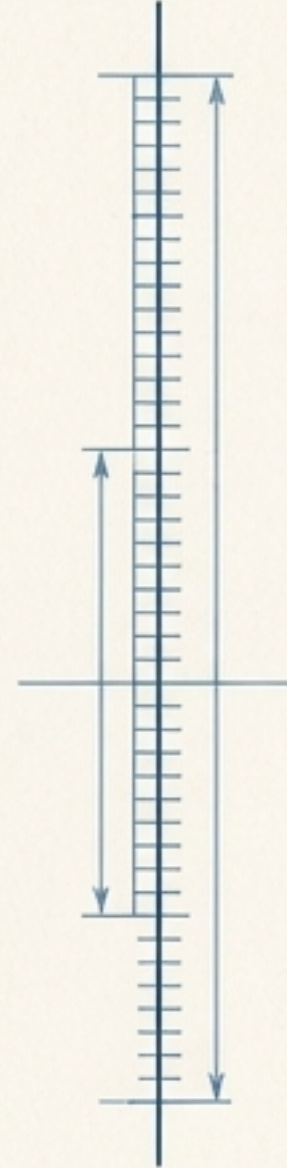
İstisnaları (Exceptions) Anlamak, Yakalamak ve Yönetmek



LEGEND: [Slate Gray] = Structural/Standard | [Blueprint Blue] = Normal Flow |
[Industrial Ocher] = Warning/Catch | [Sage Green] = Safety/Cleanup | [Brick Red] = Alarm/Unhandled

Kod Neden Kırılır?

Beklenen Akış

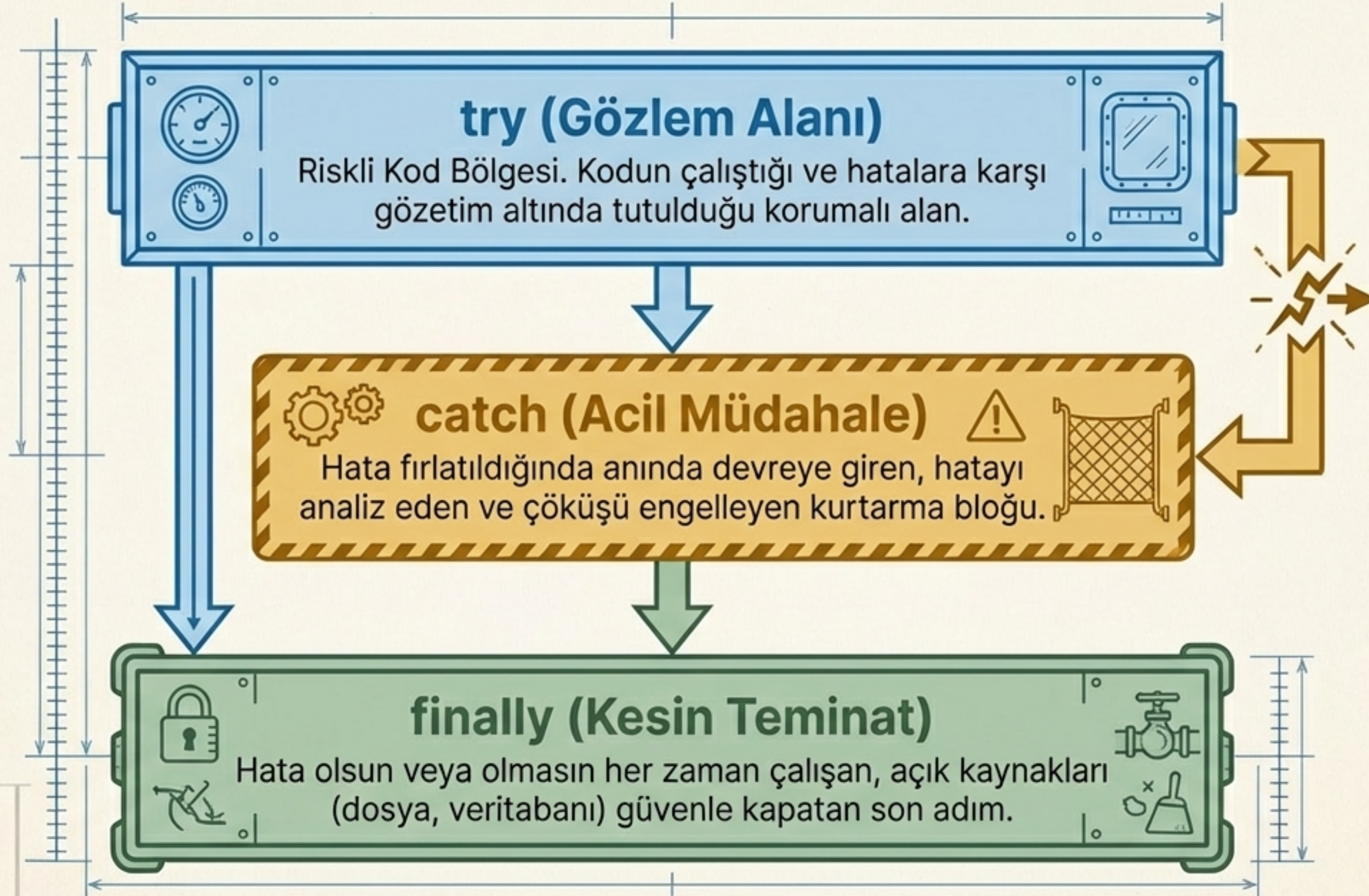


Gerçek Dünya



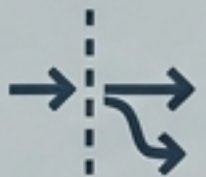
En iyi yazılmış kod bile kontrolümüz dışındaki etkenlerle karşılaşır. Yanlış kullanıcı girdisi, tükenen bellek veya kopan ağ bağlantıları gerçek dünyanın kaçınılmaz bir parçasıdır. C#, bu **hataları sessizce yutmak yerine**, onları **İstisnalar (Exceptions)** aracılığıyla görünür kılar. Bu, uygulamanızı korumak için tasarlanmış yapısal bir **güvenlik mekanizması**dır.

Güvenlik Ağının Üç Sütunu

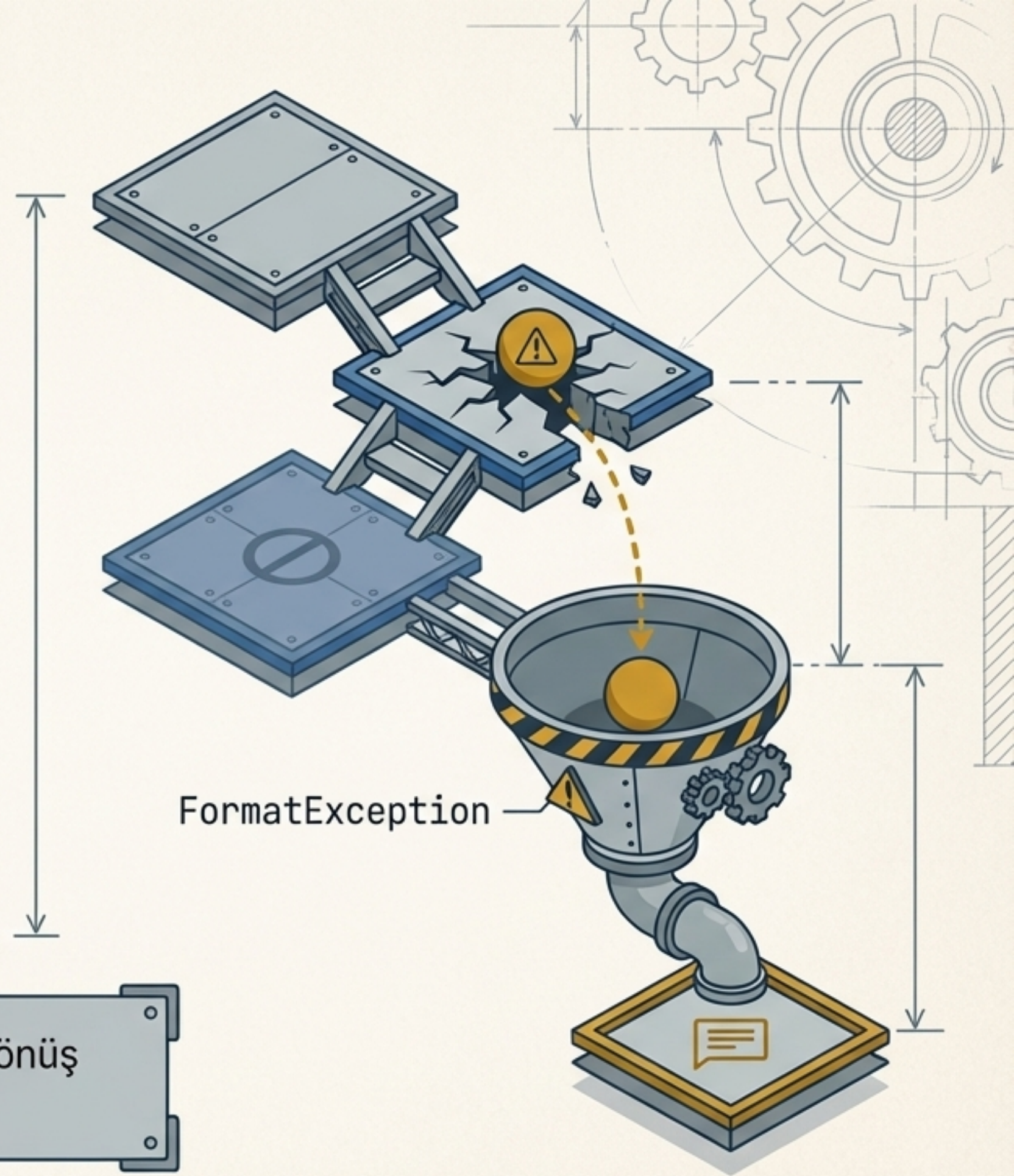


İstisnayı Anında Yakalamak

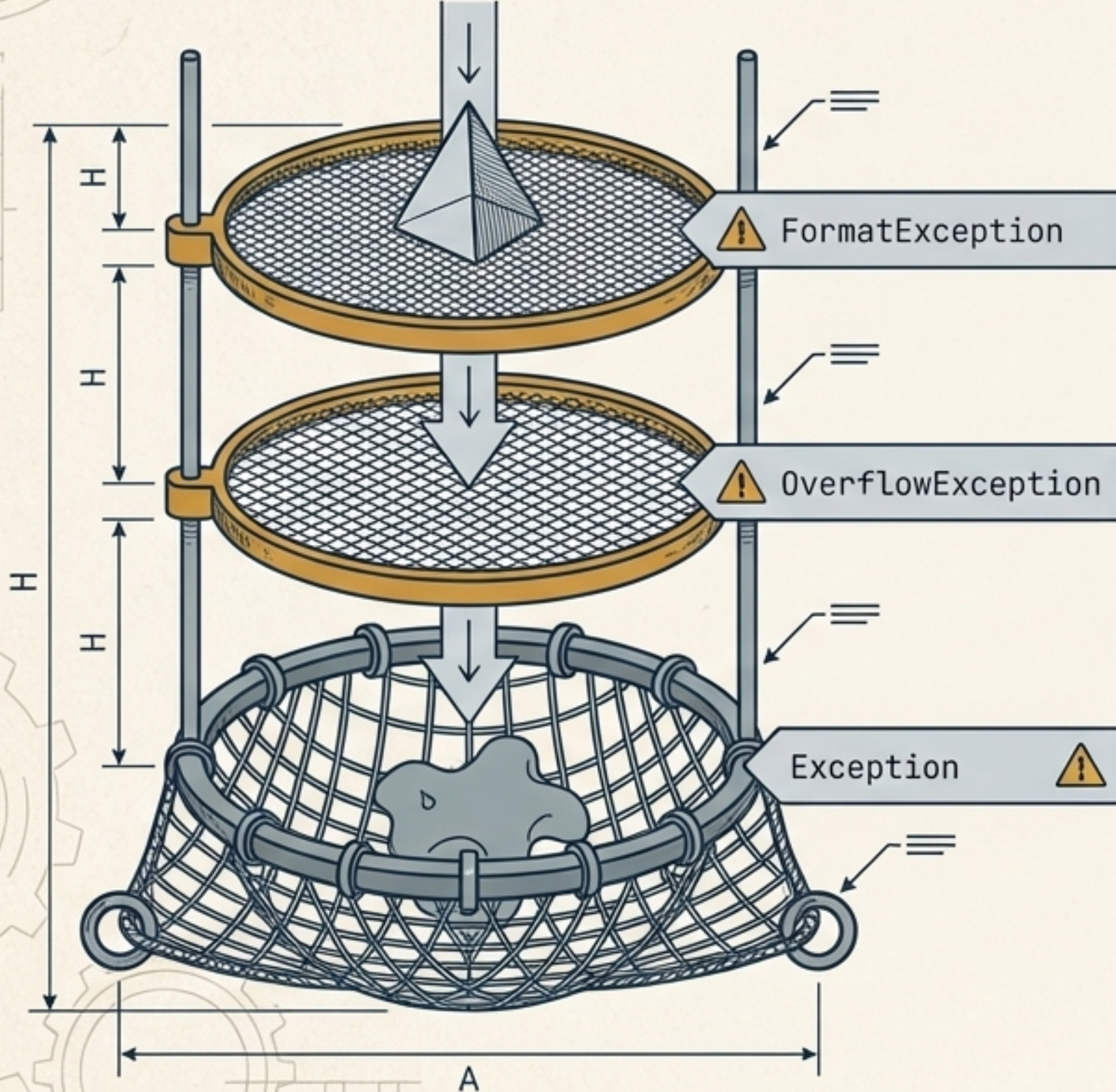
```
try {  
    // "John" metni sayıya çevrilemez!  
    int sonuc = int.Parse(girdi);  
  
    // Bu satır ASLA çalışmaz:  
    Kaydet(sonuc);  
} catch (FormatException fEx) {  
    // Kontrol doğrudan buraya geçer  
    MesajGoster(fEx.Message);  
}
```



Hata oluştuğu milisaniyede normal akış durur. Geri dönüş yoktur. Sistem doğrudan uygun catch bloğuna atlar.



Filtreleme Mantığı: Özelden Genele



Kural

C#, catch bloklarını yukarıdan aşağıya doğru sırayla okur ve ilk uyan bloğu çalıştırır.

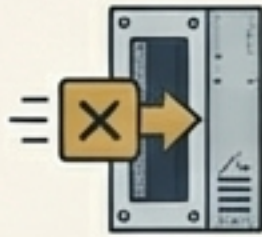
Mimari Yaklaşım

Her zaman en spesifik hata türlerini (örn. `FormatException`) en üste yazın.

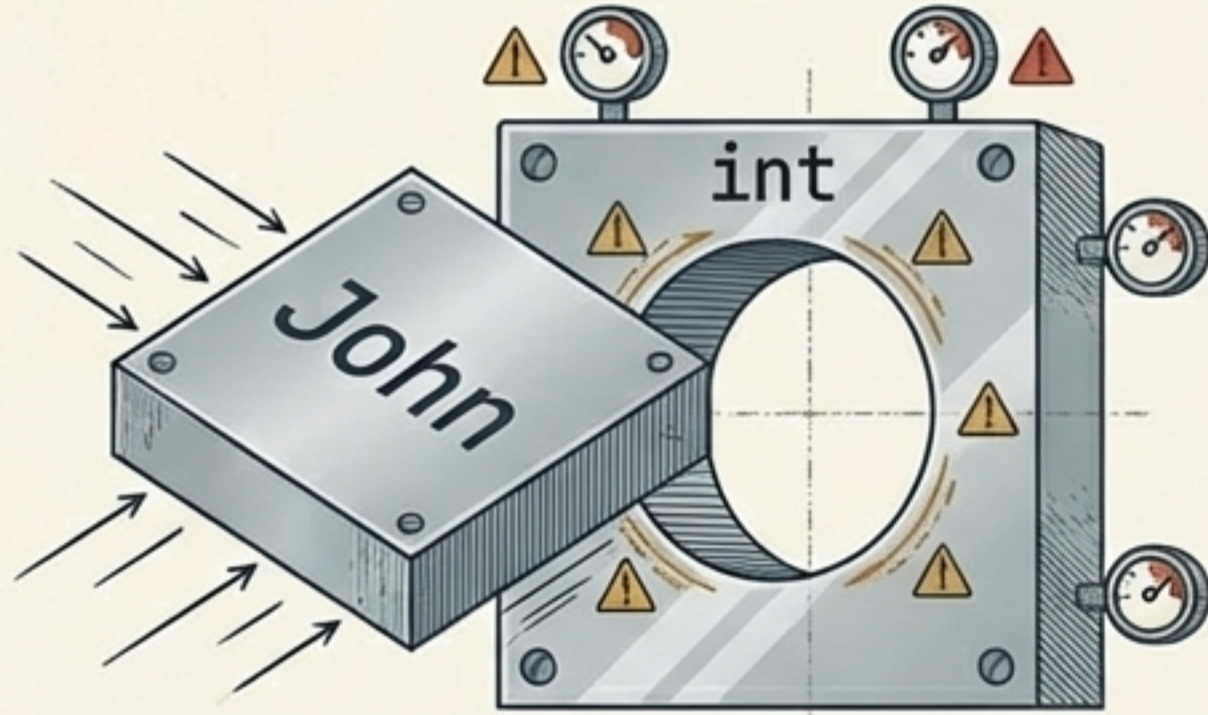
Tüm hataların atası olan genel `Exception` sınıfını ise "son çare" olarak en alta yerleştirin.

Aksi takdirde, genel filtre üstte olursa özel hatalarınız asla kendi bloklarına ulaşamaz.

Sık Karşılaşılan İstisnalar: Teşhis Matrisi

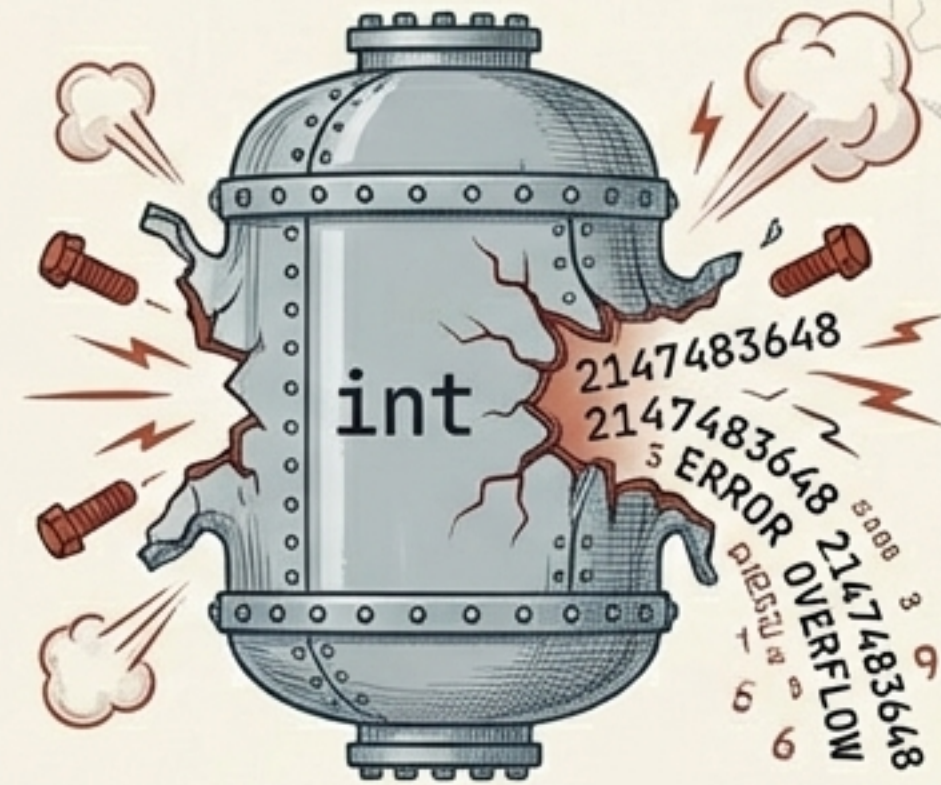
İstisna Türü	Tetikleyici Neden	Örnek Senaryo
FormatException	Veri Biçimi Uyumsuzluğu 	"Ahmet" metnini tam sayıya (int.Parse) çevirmeye çalışmak.
OverflowException	Kapasite (Bellek) Aşımı 	checked blok içinde 32-bit int sınırını aşan bir matematiksel işlem yapmak.
ArgumentException	Geçersiz Parametre 	Bir metoda beklenen kurallara uymayan (örn. null veya boş liste) argüman göndermek.
InvalidOperationException	Geçersiz Sistem Durumu 	Seçili bir operatör yokken "Hesapla" metodunu çağırmak veya başka bir thread'in UI nesnesine erişmesi.

Derinlemesine Bakış: Veri ve Matematik Hataları



FormatException

Veri tipi doğru görünse de içeriği yanlıştır. Kullanıcı girdilerini işlerken en sık karşılaştığımız savunma hattıdır.



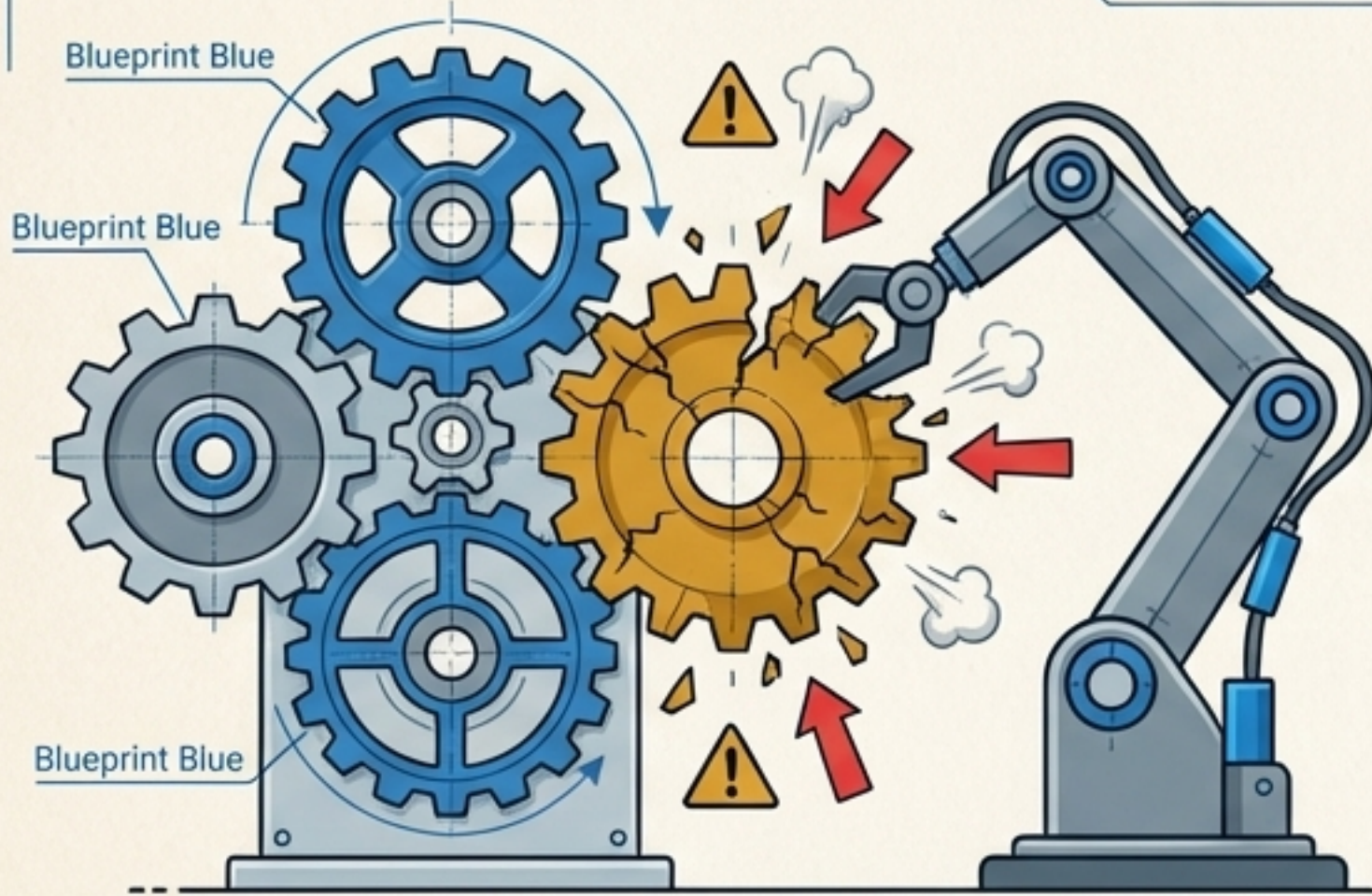
OverflowException ve Sessiz Tehlike

C#, performans sebebiyle tamsayı (integer) taşmalarını varsayılan olarak görmezden gelir ve sessizce yanlış sonuç üretir. Taşmayı anında yakalamak için riskli matematiksel işleminizi checked { ... } anahtar kelimesi ile sarmalamanız şarttır.

Derinlemesine Bakış: Mantık ve Durum Hataları

ArgumentException (Dış Etken)

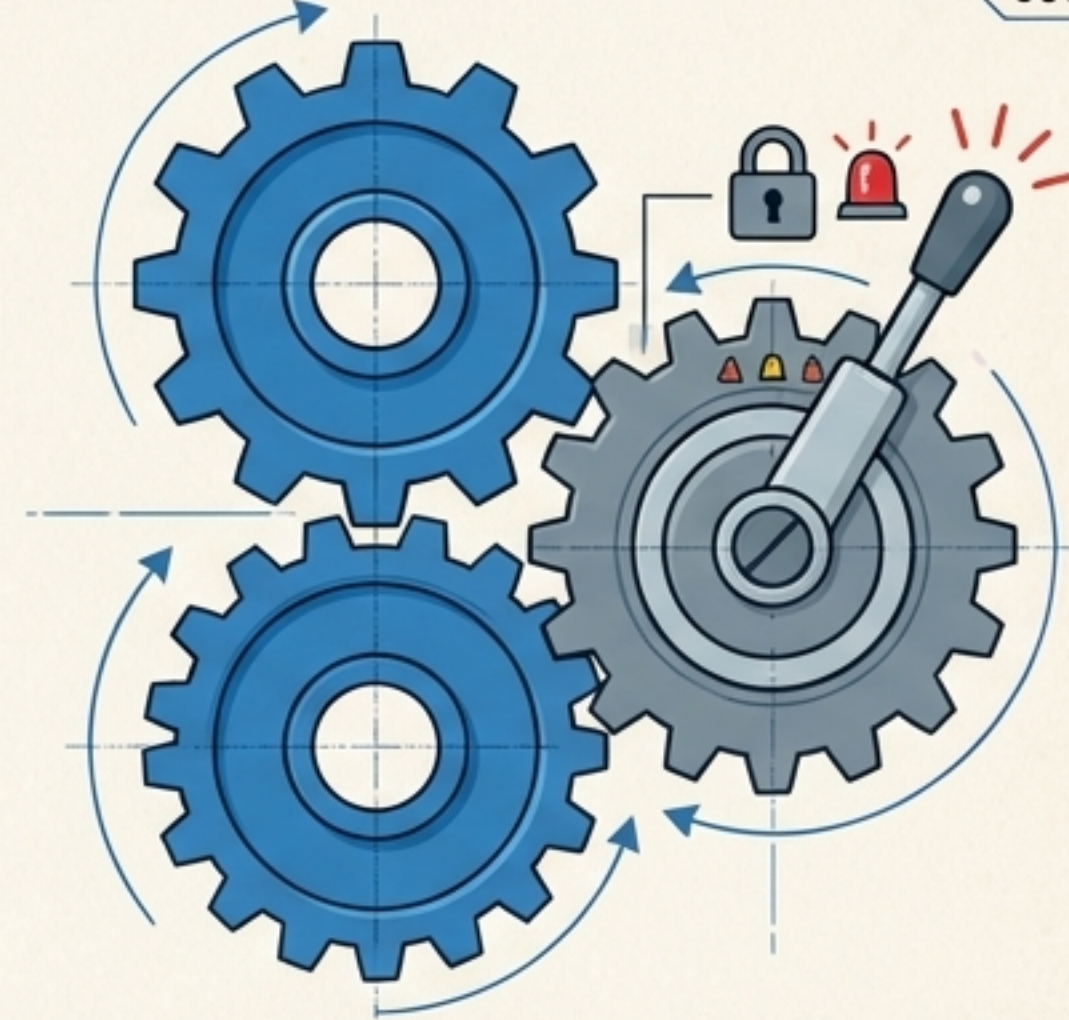
JetBrains Mono



Sorun dışarıdan gelen veridedir. Metodunuz, kendisine verilen parametrelerin çalışma kurallarına aykırı olduğunu reddeder. Makine sağlam, girdi hatalı.

InvalidOperationException (İç Etken)

JetBrains Mono



Parametreler tamamen doğru olabilir, ancak nesnenin mevcut durumu bu işleme izin vermiyordur (Örn: Henüz açılmamış bir veritabanı bağlantısından veri okumaya çalışmak).

Kesin Teminat: finally ile Kaynak Yönetimi

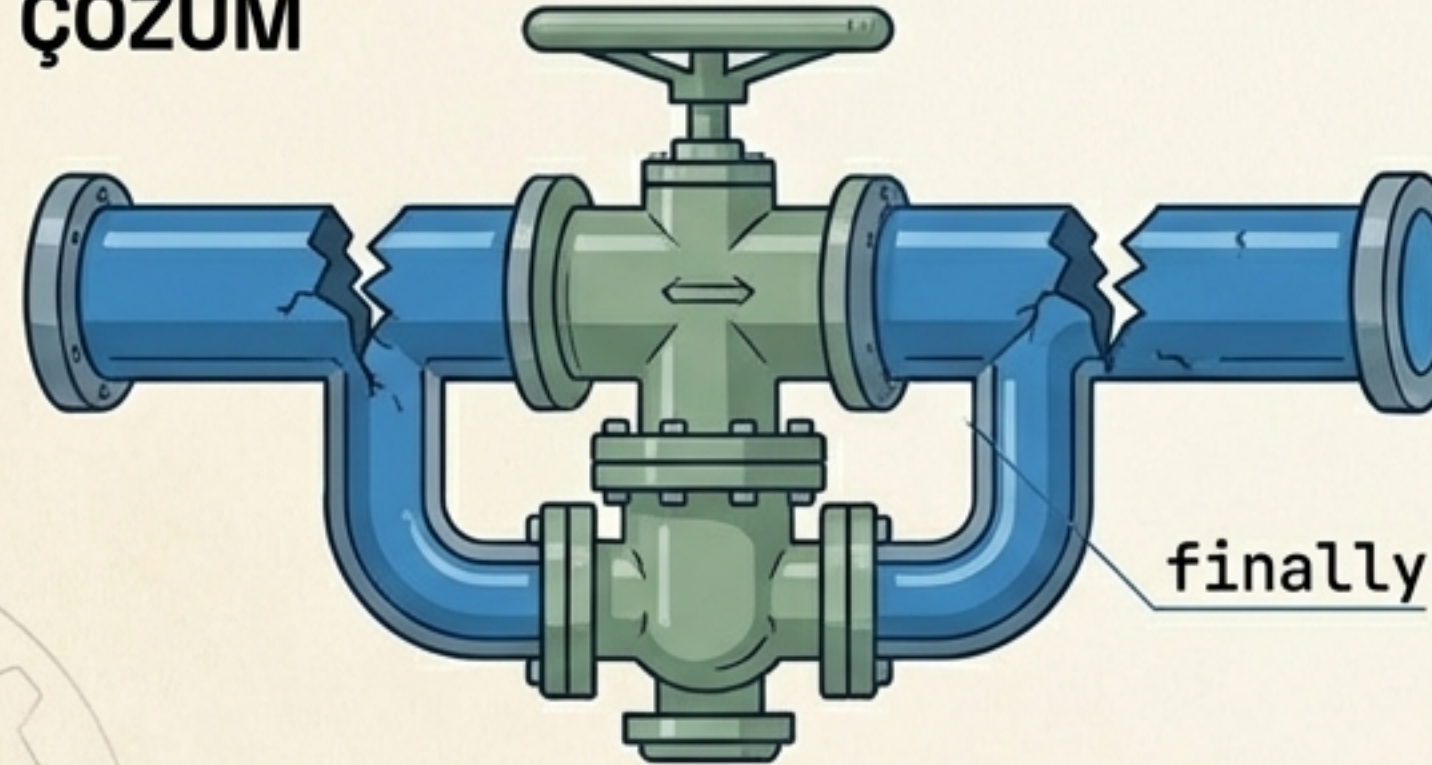
TEHLİKE



Tehlike

Bir dosya okuyucu (TextReader) veya veritabanı bağlantısı açtığınızda, kodunuz kırılırsa kapatma komutu (Close) asla çalışmayabilir. Bu, sistem kaynaklarının tükenmesine yol açar.

ÇÖZÜM



Çözüm

finally bloğu, sistemin sigortasıdır. **try** bloğu başarıyla tamamlansa da, bir hata fırlatılıp **catch** bloğuna girilse de, finally bloğu her zaman çalıştırılır ve kaynakların güvenli iade edilmesini garanti eder.

Zarif Kısayol: using Yapısı

Manuel Yönetim (try-finally):

Kaynakları başlatmak, null kontrolü yapmak ve manuel olarak Dispose() çağırmak kalabalık ve hata yapmaya açık bir koddur.

Modern C# Yaklaşımı: IDisposable

arayüzünü (interface) uygulayan nesneler için using bloğu mükemmel bir soyutlamadır. Arka planda otomatik olarak gizli bir **try-finally** bloğu oluşturur.

```
using (TextReader okuyucu = new StreamReader("dosya.txt")) {  
    // Hata olsa bile blok sonunda okuyucu.Dispose() kesinlikle çağrılır.  
    Console.WriteLine(okuyucu.ReadLine());  
}
```

Derleyici tarafından otomatik try-finally'e çevrilir.

Aktif Savunma: throw ile İnisiyatif Almak



Kavram

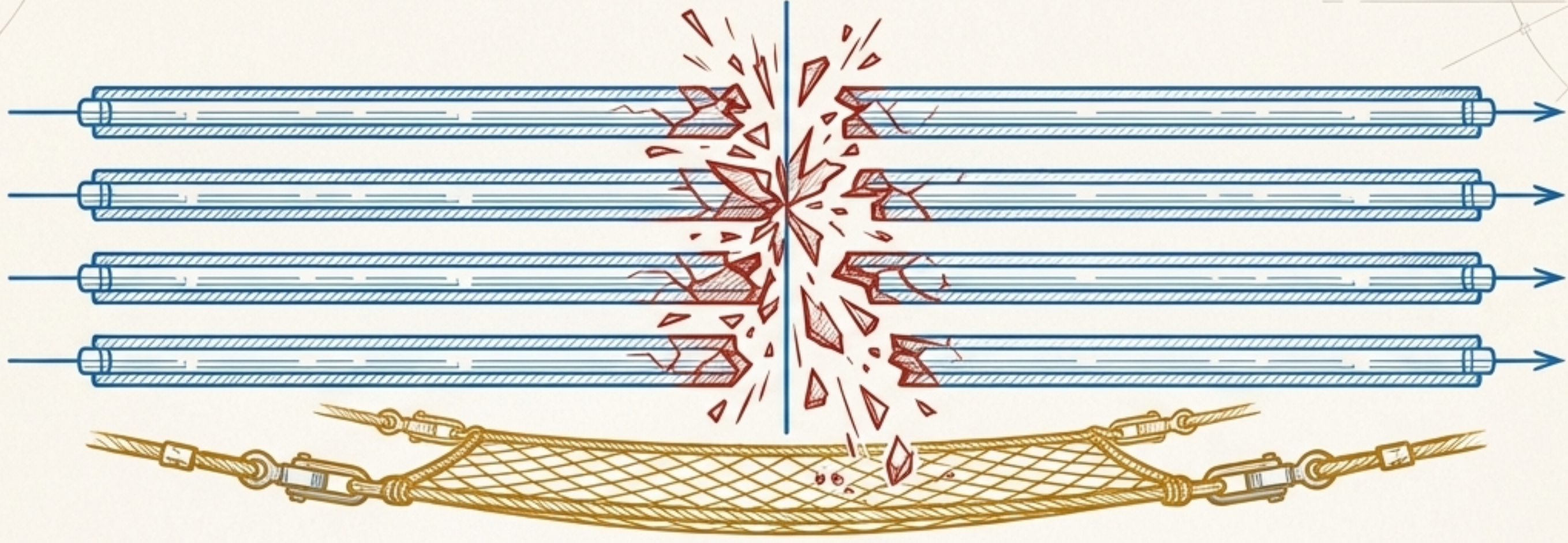
Hataların kendiliğinden (ve sistemin derinliklerinde) oluşmasını beklemek zorunda değilsiniz. "Hızlı Başarısız Ol" (Fail-Fast) prensibi, geçersiz veriyi tespit ettiğiniz an sistemi durdurmanızı emreder.

Eylem

Kendi uygulamanızın mantığı ihlal edildiğinde **throw** anahtar kelimesi ile kendi istisnanızı fırlatın. Bu, verinin bozulmasını engeller ve hatanın kaynağını netleştirir.

C# Validation Check

Modern Savaş Alanı: Paralel Görevler (TPL)



Zorluk

Task Parallel Library (TPL) ile çoklu görevler asenkron olarak aynı anda çalışır. Peki ya birden fazla görev aynı anda çöküp hata fırlatırsa?

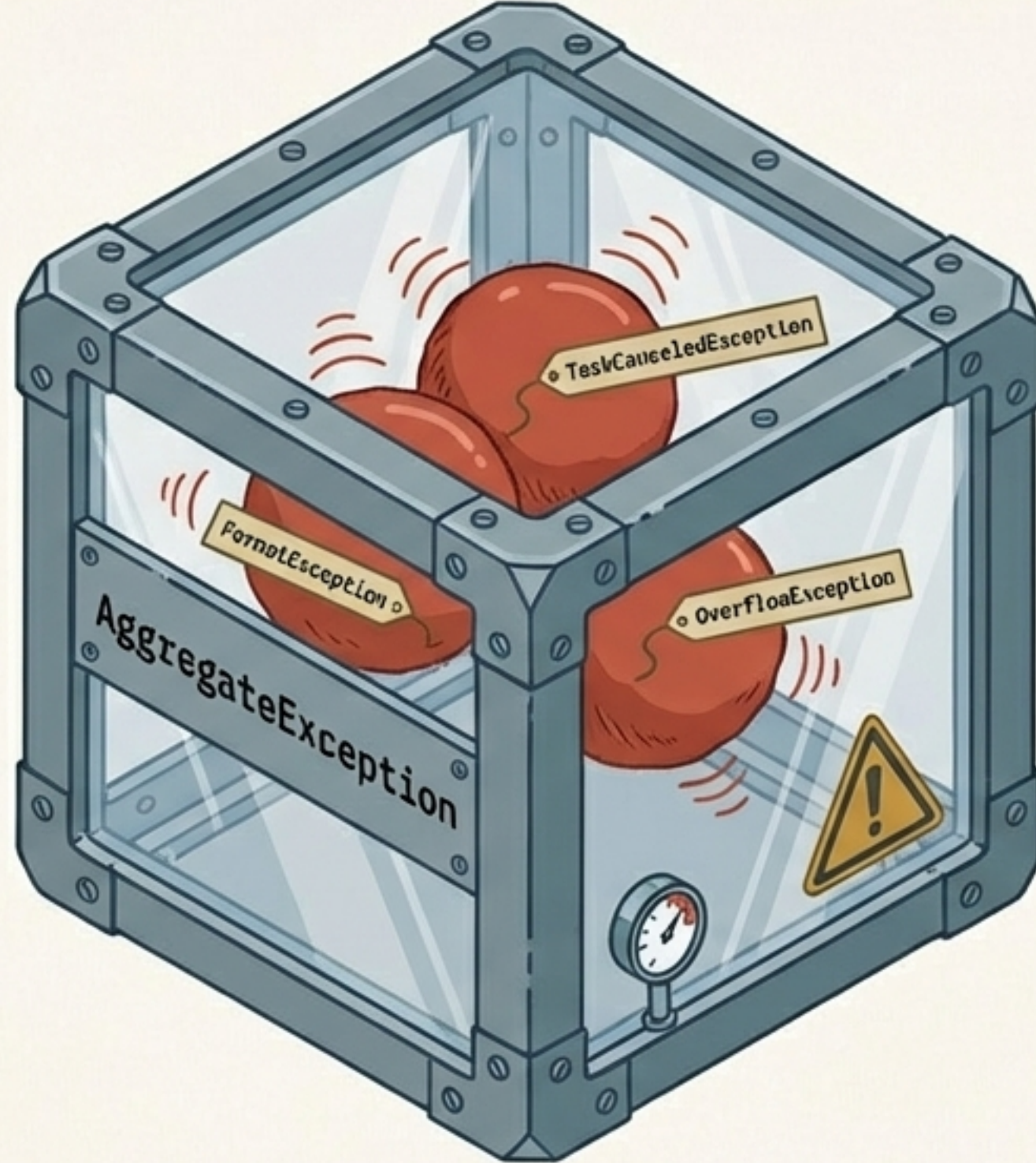
Yetersizlik

Standart bir catch(Exception) bloğu tek boyutludur; aynı milisaniyede farklı thread'lerden gelen çoklu hataları tek seferde yakalayıp işleyemez.

Hataları Paketlemek: AggregateException

Çözüm

C#, paralel görevlerde oluşan tüm hataları kaybetmez. Onları tek bir büyük şemsiye kutunun, yani AggregateException nesnesinin içine hapseder.

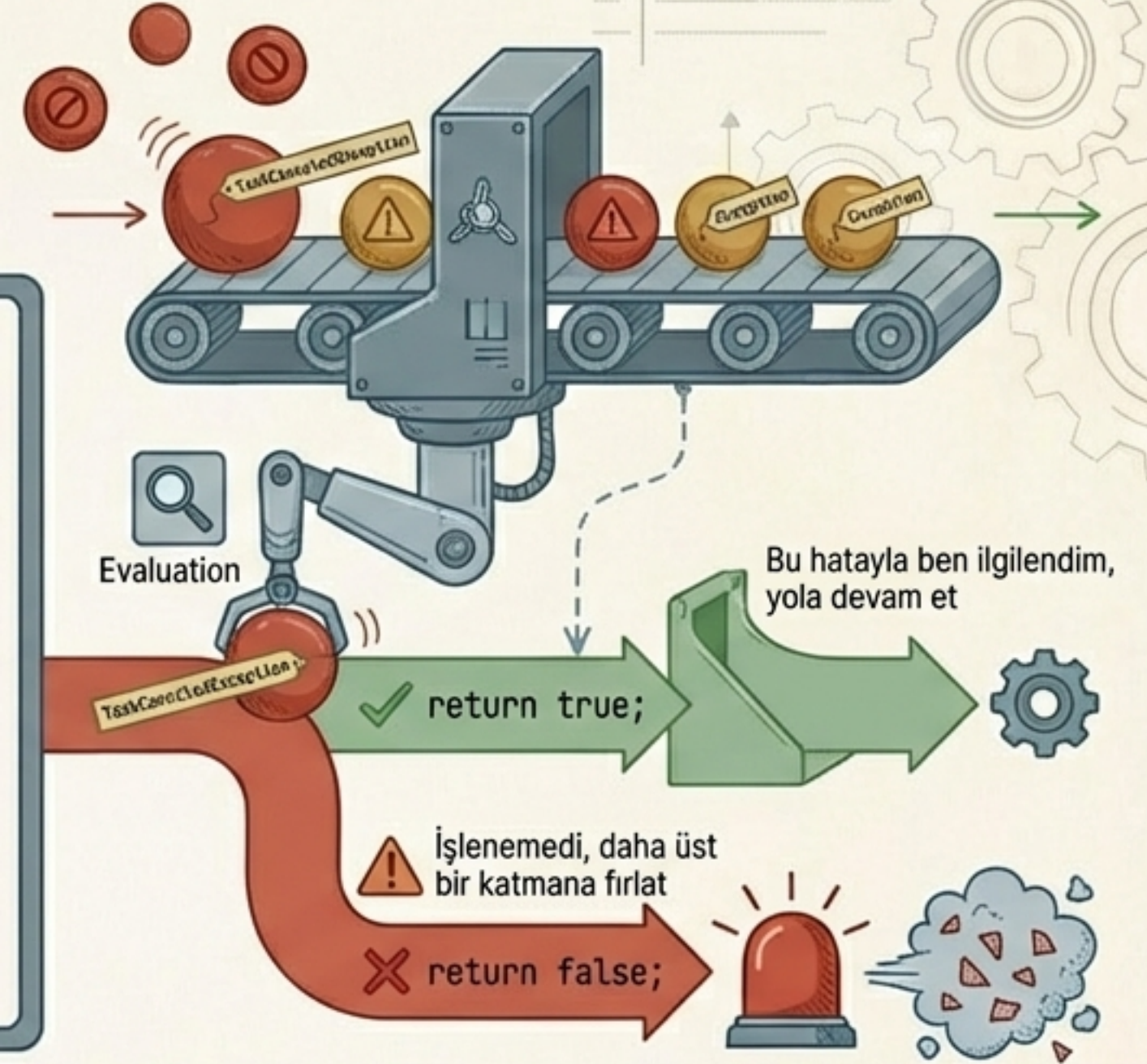


İşleyiş

Bu kutuyu standart bir catch bloğu ile yakalarsınız. Ancak asıl iş, kutuyu açıp içindeki alt hataları (InnerExceptions) tek tek incelemek ve yönetmektir.

Çözümleme Mekanizması: .Handle() Metodu

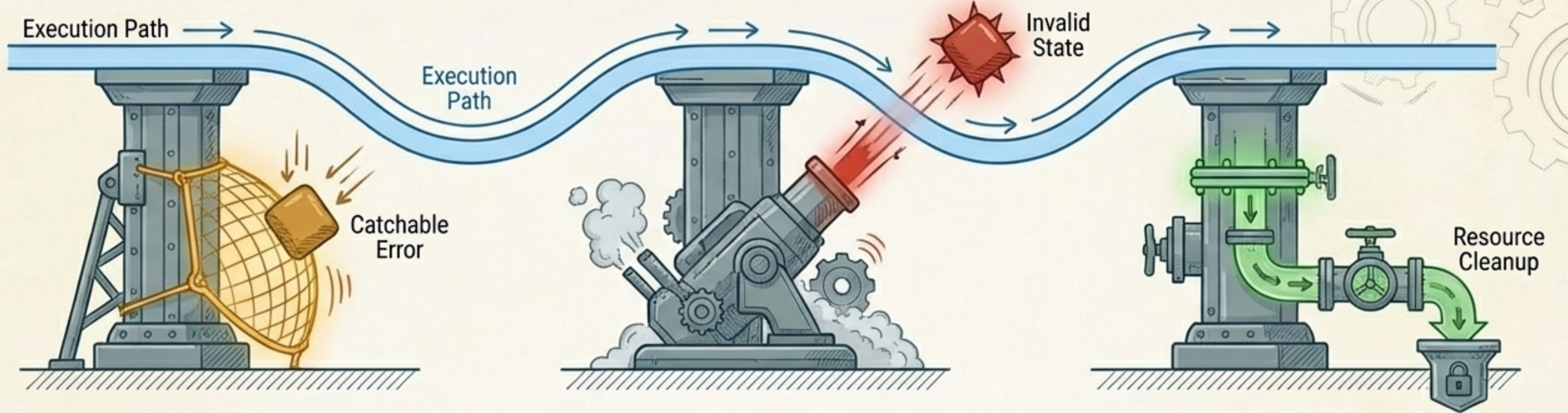
```
catch (AggregateException ae) {  
    ae.Handle(ex => {  
        if (ex is TaskCanceledException) {  
            Console.WriteLine("Görev iptal edildi.");  
            return true; // Hata çözüldü!  
        }  
        return false; // Çözülemedi, sistemi çökert!  
    });  
}
```



Kural

.Handle() metodu her bir alt hatayı döngüye sokar. Kararlarınız sistemin ayakta kalıp kalmayacağını belirler.

Güvenlik Ağı Mimarisi: Altın Kurallar



1. Yakalama (Kalkan)

Industrial Ocher Space Grotesk



Yalnızca nasıl düzelteceğinizi bildiğiniz veya kullanıcıya anlamlı bir mesaj sunabileceğiniz hataları **catch** ile yakalayın. Tüm hataları körü körüne yutmayın.

2. Fırlatma (Alarm)

Brick Red Space Grotesk



Uygulamanız geçersiz bir veriyle veya mantıksız bir durumla karşılaştığında anında kendi özel hatanızı **throw** ile fırlatın (Fail-Fast).

3. Temizleme (Teminat)

Sage Green Space Grotesk



Dış dünyayla kurduğunuz her bağlantıyı (dosya, veritabanı) **finally** veya **using** yapıları ile mutlaka garanti altına alın.

Sonuç: İstisnalar (Exceptions) kodunuzun düşmanı değil, sisteminizin bütünlüğünü koruyan en gelişmiş kontrol mekanizmasıdır.