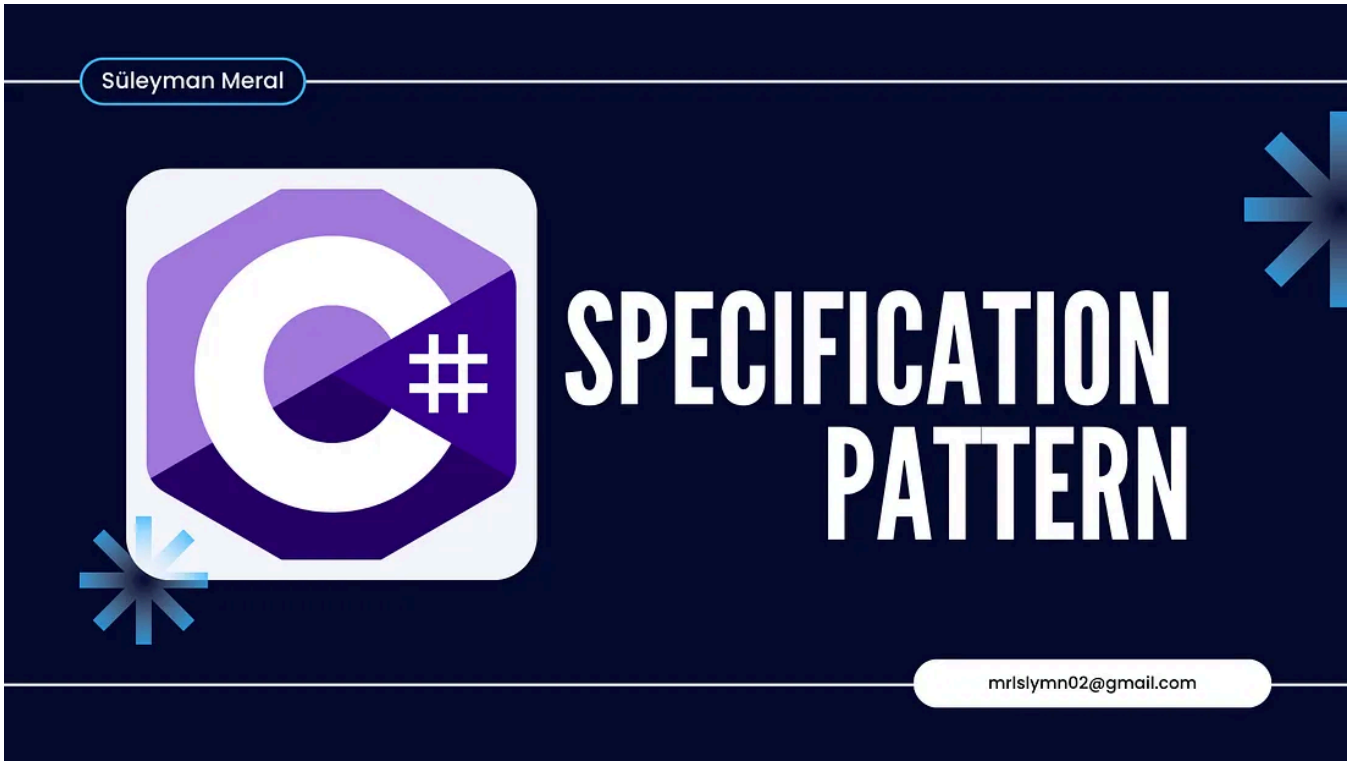
# Specification Pattern Nedir?(.NET)

5 min read · Jun 1, 2025

Süleyman Meral






Herkese Merhaba!

Bu yazımda, API üzerinde filtreleme işlemi yaparken aklıma takılan bir sorunun araştırılması sırasında karşıma çıkan ve yazılım dünyasında oldukça önemli bir yere sahip olan bir yaklaşımdan, **Specification Pattern** tasarım deseninden bahsetmek istiyorum.

Öncelikle bu desenin teorik temellerini ele alacak, ardından konuyu daha iyi pekiştirmek adına C# programlama dili ile basit bir örnek üzerinden ilerleyeceğiz.

Her ne kadar örneğimizi C# ile gerçekleştirecek olsak da, tasarım desenlerinin en güzel yönlerinden biri, **programlama dilinden bağımsız** olarak birçok farklı dilde uygulanabilir olmalarıdır.

## Specification Pattern Nedir?

Yazılım geliştirirken, özellikle karmaşık iş kurallarını uygularken, kodun okunabilirliğini ve sürdürülebilirliğini korumak zor olabilir. Specification Pattern, bu sorunu çözmek için tasarlanmış bir domain-driven design (DDD) pattern'idir. Temel amacı, filtreleme ve sorgulama mantığını nesnelere ayırarak daha esnek ve okunabilir bir kod yapısı oluşturmaktır.

Specification Pattern, bir nesnenin belirli bir kritere uyup uymadığını kontrol etmek için kullanılan bir desendir.

## Specification Pattern Kullanmak Bize Ne Avantaj Sağlayabilir?

- SOLID prensiplerini destekler ve sürdürülebilir kod yazmamıza yardımcı olur.
- Kodun okunabilirliğini arttırır.
- Reusability (Yeniden Kullanılabilirlik). Aynı spesifikasyonu farklı yerlerde kullanabilirsiniz.
- Test etme kolaylığı

Şimdi örnek bir senaryo ile konuyu biraz daha pekiştirelim.

```
public interface IGenericRepository<T> where T : BaseEntity
{
    Task<T?> GetByIdAsync(int id);
    Task<IReadOnlyList<T>> ListAllAsync();
    void Add(T entity);
    void Update(T entity);
    void Remove(T entity);
    Task<bool> SaveAllASync();
    bool Exists(int id);
}
```

Bu yapıda, veritabanı işlemlerini gerçekleştirmek için temel CRUD metotları tanımlanmış durumda. Ancak, özelleştirilmiş filtreleme işlemleri gerektiğinde bu arayüz yetersiz kalmaktadır. Örneğin, "**Türü telefon olan ürünleri listele**" gibi bir

sorgu için yeni metotlar yazmak gerekir. Bu durum, interface'in gereksiz yere büyümesine, kod tekrarına ve zamanla artan bir karmaşaya yol açar.

Yüzlerce varlık (entity) barındıran büyük projelerde bu yapı, yönetilmesi zor hale gelir. İşte bu noktada, devreye **Specification Pattern** girer.

Aslında, tasarım desenlerinin ortaya çıkış amacı da tam olarak budur: geliştiricilerin tekrar eden problemler karşısında **yeniden kullanılabilir ve sürdürülebilir çözümler** üretmesi.

İlk olarak spesifikasyonların uyması gereken kurallar için bir Interface oluşturalım. Bu generic bir yapı olup tüm entityler ile çalışmaya izin verecek.

```
public interface ISpecification<T>
{
    Expression<Func<T, bool>>? Criteria { get; }
    Expression<Func<T, object>>? OrderBy { get; }
    Expression<Func<T, object>>? OrderByDesc { get; }
}
```

Criteria ifadesi koşulumuza , OrderBy ve OrderByDesc ifadesi aran ve azalan sıralama işlemine karşılık gelmektedir.

Şimdi bu interface'den türeyen bir base class oluşturalım.

```
public class BaseSpecification<T>(Expression<Func<T, bool>>? criteria) : ISpeci
{
    protected BaseSpecification() : this(null) { }

    public Expression<Func<T, bool>> Criteria => criteria;

    public Expression<Func<T, object>>? OrderBy { get; private set; }

    public Expression<Func<T, object>>? OrderByDesc { get; private set; }

    protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)
    {
        OrderBy = orderByExpression;
    }
    protected void AddOrderByDesc(Expression<Func<T, object>> orderByDescExpres
    {
```

```
            OrderByDesc = orderByDescExpression;
        }
    }
```

`Expression<Func<T, bool>>` — LINQ ifadelerini temsil eder. Mesela:

`x => x.Brand==brand`

```
    protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)
```

Protected yapısı sadece bu sınıftan türeyen yapıların bu metodu kullanmasına izin verir.

Şimdi ise sorguyu uygulayan SpecificationEvulator sınıfını yazalım.

```
    public class SpecificationEvaluator<T> where T : BaseEntity
    {
        public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T>
        {
            if (spec.Criteria != null)
            {
                query = query.Where(spec.Criteria); // x=>x.Brand==brand
            }
            if (spec.OrderBy != null)
            {
                query = query.OrderBy(spec.OrderBy);
            }
            if (spec.OrderByDesc != null)
            {
                query = query.OrderByDescending(spec.OrderByDesc);
            }
            return query;
        }

    }
```

SpecificationEvulator sınıfına sadece BaseEntity classından türeyen sınıflar girebilir. BaseEntity sınıfını miras alan sınıflar ise bizim Entity'lerimizi oluşturmaktadır.

Şimdi Repo arayüzümüzü ve sınıfımızı güncelleyelim.

```
public interface IGenericRepository<T> where T : BaseEntity  //base entitiyden
{
    Task<T?> GetByIdAsync(int id);
    Task<IReadOnlyList<T>> ListAllAsync();
    Task<T?> GetEntityWithSpec(ISpecification<T> spec);
    Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec);
    void Add(T entity);
    void Update(T entity);
    void Remove(T entity);
    Task<bool> SaveAllASync();
    bool Exists(int id);

}
```

Görüldüğü gibi 2 yeni metot ekledik. Bunlar filtreleme ve sıralama işlemlerinde bize yardımcı olacak metotlar.

```
public class GenericRepository<T>(StoreContext context) : IGenericRepository<T>
{
    public void Add(T entity)
    {
        context.Set<T>().Add(entity);
    }

    public bool Exists(int id)
    {
        return context.Set<T>().Any(x => x.Id == id);
    }

    public async Task<T?> GetByIdAsync(int id)
    {
        return await context.Set<T>().FindAsync(id);
    }

    public async Task<T?> GetEntityWithSpec(ISpecification<T> spec)
    {
        return await ApplySpecification(spec).FirstOrDefaultAsync();
```

```
    }

    public async Task<IReadOnlyList<T>> ListAllAsync()
    {
        return await context.Set<T>().ToListAsync();
    }

    public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)
    {
        return await ApplySpecification(spec).ToListAsync();
    }

    public void Remove(T entity)
    {
        context.Set<T>().Remove(entity);
    }

    public async Task<bool> SaveAllASync()
    {
        return await context.SaveChangesAsync() > 0;
    }

    public void Update(T entity)
    {
        context.Set<T>().Attach(entity);
        context.Entry(entity).State = EntityState.Modified;
    }

    private IQueryable<T> ApplySpecification(ISpecification<T> spec)
    {
        return SpecificationEvaluator<T>.GetQuery(context.Set<T>().AsQueryable(
    }
}
```

GenericRepository classımıza yeni metotları da implement ettik. Ve ApplySpecification adında bir private metot tanımladık. SpecificationEvulator classından GetQuery metoduna ulaşarak ilgili spesifikasyonlar kullanılıyor. Eğer aklımızda örnek oluşturmadan nasıl metoda ulaştık sorusu varsa bunun cevabı GetQuery metodunun static olarak tanımlanmasıdır.

```
public async Task<T?> GetEntityWithSpec(ISpecification<T> spec)
{
    return await ApplySpecification(spec).FirstOrDefaultAsync();
}
public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)
{
```

```
        return await ApplySpecification(spec).ToListAsync();
    }
```

Şimdi deseni Product entitymize uygulayalım. ProductSpecification adında bir class oluşturalım.

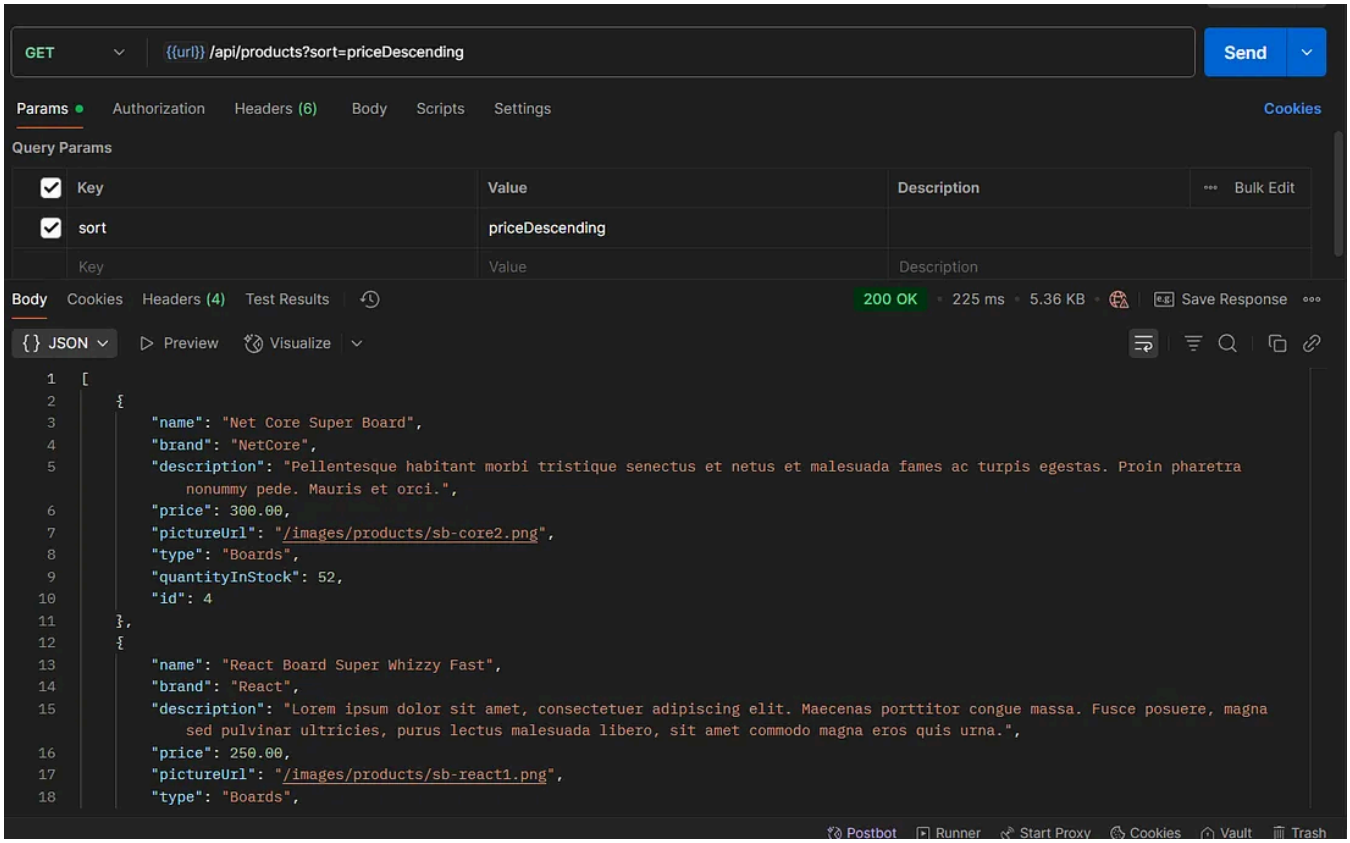
```
public class ProductSpecification : BaseSpecification<Product>
{

    public ProductSpecification(string? brand, string? type,string? sort):base(
    (string.IsNullOrWhiteSpace(brand)|| x.Brand==brand) &&
    (string.IsNullOrWhiteSpace(type)|| x.Type==type)
    )
    {
        switch (sort)
        {
            case "priceAsc":
                AddOrderBy(x => x.Price);
                break;
            case "priceDescending":
                AddOrderByDesc(x => x.Price);
                break;
            default:
                AddOrderBy(x => x.Name);
                break;
        }
    }

}
```

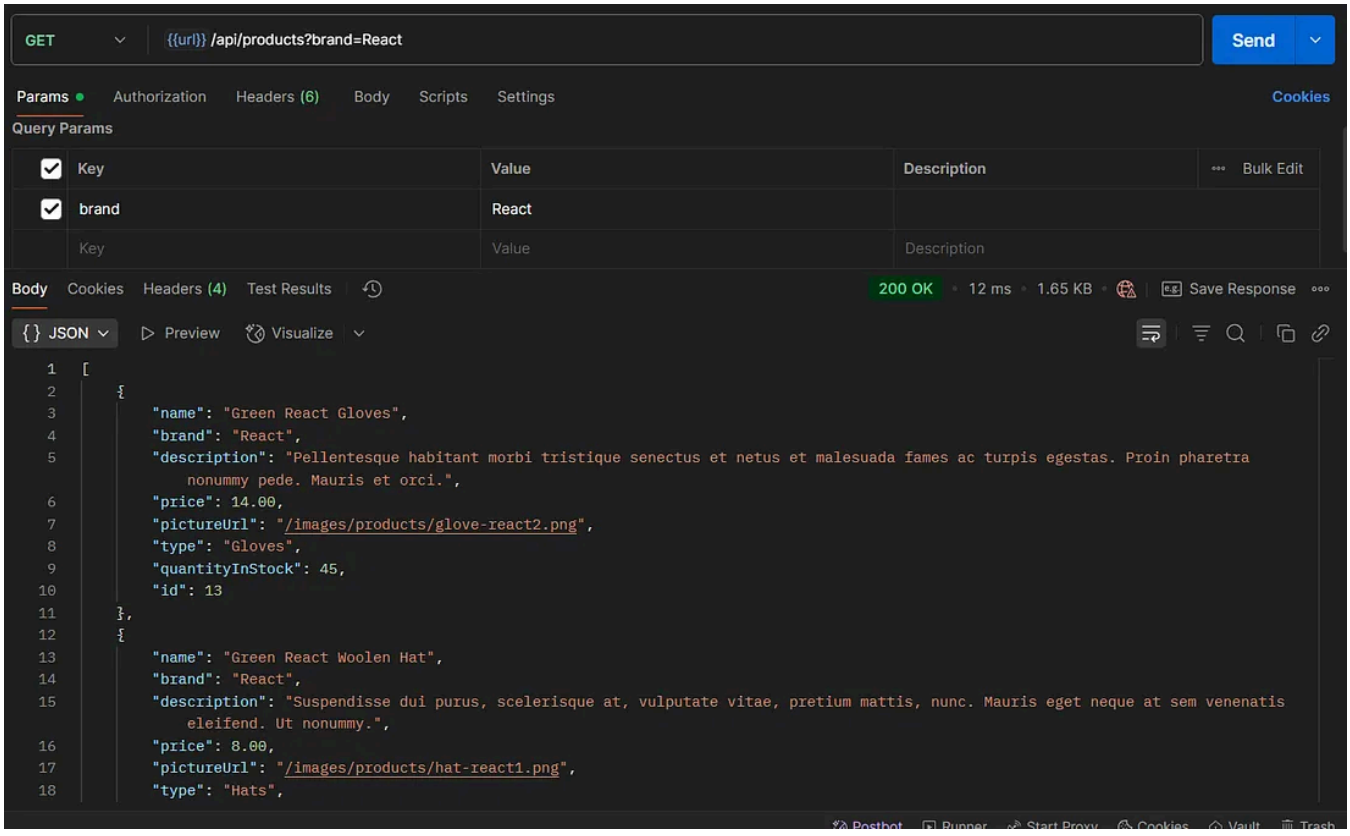
Görüldüğü üzere base classa bir koşul ve sıralama gönderiliyor. Şimdi ProductController sınıfını güncelleyelim.

```
    [HttpGet]
    public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string?
    {
        var spec = new ProductSpecification(brand, type,sort);
        var products = await repo.ListAsync(spec);
```

```
            return Ok(products);
        }
```

Artık product nesnemizin filtreleme işlemlerini yapabiliriz.



Gördüğümüz üzere filtreleme işlemimiz sorunsuz çalışıyor. Şimdi ne yaptığımızı özetleyip yazımızı bitirelim.

```
public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? br
 {
     var spec = new ProductSpecification(brand, type,sort);
     var products = await repo.ListAsync(spec);
     return Ok(products);
 }
```

GetProducts metodumuz içerisine brand,type ve sort değişkenlerini alıyor fakat bunlar zorunlu parametreler olarak işaretlenmemiş. ProductSpecification classımızdan bir örnek alındı ve parametre olarak metodumuzun aldığı parametreler verildi.

```
public ProductSpecification(string? brand, string? type,string? sort):base(
(string.IsNullOrWhiteSpace(brand)|| x.Brand==brand) &&
(string.IsNullOrWhiteSpace(type)|| x.Type==type)
)
{
    switch (sort)
    {
        case "priceAsc":
            AddOrderBy(x => x.Price);
            break;
        case "priceDescending":
            AddOrderByDesc(x => x.Price);
            break;
        default:
            AddOrderBy(x => x.Name);
            break;
    }
}
```

ProductSpecification classımız newlendiği anda bu kodlar çalışıyor. Miras aldığı sınıfa çeşitli parametreler gönderiyor. (BaseSpecification Classı)

```
public class BaseSpecification<T>(Expression<Func<T, bool>>? criteria) : ISpeci
{
    protected BaseSpecification() : this(null) { }

    public Expression<Func<T, bool>> Criteria => criteria;

    public Expression<Func<T, object>>? OrderBy { get; private set; }

    public Expression<Func<T, object>>? OrderByDesc { get; private set; }
    protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)
    {
        OrderBy = orderByExpression;
    }
    protected void AddOrderByDesc(Expression<Func<T, object>> orderByDescExpres
    {
        OrderByDesc = orderByDescExpression;
    }
}
```

Parametreler bu classa veriliyor.

```
var products = await repo.ListAsync(spec);
```

Daha sonra controller'da products adında bir değişken oluşturuluyor. Bu değişken GenericRepository classından ListAsync metoduna ulaşıp spec parametresini veriyor.

```
public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)
    {
        return await ApplySpecification(spec).ToListAsync();
    }
```

```
private IQueryable<T> ApplySpecification(ISpecification<T> spec)
    {
```

```
        return SpecificationEvaluator<T>.GetQuery(context.Set<T>().AsQueryable(
    }
```

ListAsync metodu ApplySpecification metoduna spec parametresini veriyor. ApplySpecification metodu ise bu parametreyi SpecificationEvulator sınıfına veriyor(sorgunun yapıldığı sınıf).

```
    public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T>
    {
        if (spec.Criteria != null)
        {
            query = query.Where(spec.Criteria); // x=>x.Brand==brand
        }
        if (spec.OrderBy != null)
        {
            query = query.OrderBy(spec.OrderBy);
        }
        if (spec.OrderByDesc != null)
        {
            query = query.OrderByDescending(spec.OrderByDesc);
        }
        return query;
    }
```

LINQ kullanılarak sorgular oluşturuluyor ve sorgumuzun listesi bize döndürülüyor.

Specification Pattern'ini kendimce anlatmaya çalıştım. Umarım faydalı olmuştur. Okuduğunuz için teşekkür ederim. Herkese kolay gelsin.

> *Süleyman Meral / .NET Developer*

Specification Pattern | Net Core | Yazılım | Yazılım Tasarım Desenleri

Edit profile

# Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)

## No responses yet

Süleyman Meral

What are your thoughts?

## More from Süleyman Meral