# الگوی Prototype در #C

## تعریف الگوی Prototype

الگوی **Prototype** یک الگوی طراحی **خلاقی** (Creational) است که امکان ایجاد اشیا جدید را با کپی کردن یک نمونه اولیه (Prototype) فراهم می‌کند، به جای ایجاد نمونه‌های جدید از طریق سازنده. این الگو به ویژه زمانی مفید است که ایجاد یک شیء جدید هزینه‌بر باشد یا سیستم نیاز به استقلال از نحوه ایجاد اشیا داشته باشد.

## مشکل اصلی که Prototype حل می‌کند

1. **اجتناب از زیرکلاس‌سازی**: وقتی ایجاد زیرکلاس‌ها برای پیکربندی اشیا پیچیده می‌شود
2. **کاهش هزینه ایجاد اشیا**: وقتی ایجاد یک شیء جدید از طریق سازنده هزینه‌بر است
3. **پیکربندی پویا**: وقتی نیاز به ایجاد اشیا با پیکربندی‌های مختلف در زمان اجرا داریم
4. **حفظ حالت داخلی**: وقتی نیاز به کپی کردن اشیا با حالت داخلی پیچیده داریم

## ساختار پایه Prototype

// 1. رابط Prototype

public interface IPrototype {

IPrototype Clone();

}

// 2. کلاس Concrete Prototype

public class ConcretePrototype : IPrototype {

public int Property1 { get; set; }

public string Property2 { get; set; }

public List<string> Items { get; set; } = new List<string>();

public IPrototype Clone() {

// ایجاد یک کپی سطحی (Shallow Copy)

var copy = (ConcretePrototype)this.MemberwiseClone();

// برای کپی عمیق (Deep Copy) از اشیا مرجع

copy.Items = new List<string>(this.Items);

return copy;

}

public void Display() {

Console.WriteLine($"Property1: {Property1}, Property2: {Property2}");

Console.WriteLine($"Items: {string.Join(", ", Items)}");

}

}

## نحوه استفاده از الگو

class Program

{

static void Main(string[] args) {

// ایجاد نمونه اولیه

ConcretePrototype original = new ConcretePrototype {

Property1 = 10,

Property2 = "Original",

Items = new List<string> { "Item1", "Item2" }

};

Console.WriteLine("Original Object:");

original.Display();

// کپی کردن نمونه اولیه

ConcretePrototype copy = (ConcretePrototype)original.Clone();

copy.Property1 = 20;

copy.Property2 = "Copy";

copy.Items.Add("Item3");

Console.WriteLine("\nOriginal Object after modification:");

original.Display();

Console.WriteLine("\nCopied Object:");

copy.Display();

}

}

## مزایای Prototype

1. **کاهش نیاز به زیرکلاس‌سازی**: نیازی به ایجاد زیرکلاس‌های متعدد برای پیکربندی اشیا نیست
2. **افزایش کارایی**: کپی کردن اشیا موجود اغلب از ایجاد جدید آنها کارآمدتر است
3. **انعطاف‌پذیری در زمان اجرا**: می‌توان اشیا جدید را با تغییر پیکربندی نمونه‌های موجود ایجاد کرد
4. **ساده‌سازی ایجاد اشیا پیچیده**: کپی کردن اشیا پیچیده ساده‌تر از ساخت آنها از ابتدا است
5. **پیروی از اصل Open/Closed**: می‌توان اشیا جدید را بدون تغییر کد موجود اضافه کرد

## معایب Prototype

1. **پیچیدگی کپی عمیق**: پیاده‌سازی کپی عمیق برای اشیا با گراف ارجاعی پیچیده دشوار است
2. **حالت داخلی**: ممکن است دسترسی به حالت داخلی شیء برای کپی کردن لازم باشد که اصل encapsulation را نقض کند
3. **مدیریت چرخه حیات**: مدیریت اشیا کپی شده ممکن است پیچیده باشد

## مثال کاربردی: سیستم اسپریت‌های بازی

public abstract class GameUnit : ICloneable {

public Vector3 Position { get; set; }

public string Name { get; set; }

public int Health { get; set; }

public abstract object Clone();

public virtual void Display() {

Console.WriteLine($"Unit: {Name} at {Position} with {Health} health");

}

}

public class Soldier : GameUnit {

public Weapon Weapon { get; set; }

public Soldier() {

// بارگذاری مدل 3D سنگین و پیچیده

Thread.Sleep(2000); // شبیه‌سازی بارگذاری زمان‌بر

Name = "Soldier";

Position = Vector3.Zero;

Health = 100;

Weapon = new Weapon("Rifle");

}

public override object Clone() {

// کپی سطحی

var copy = (Soldier)this.MemberwiseClone();

// کپی عمیق برای اعضای مرجع

copy.Weapon = new Weapon(this.Weapon.Type);

// تنظیم نام جدید

copy.Name = "Soldier Clone";

return copy;

}

public override void Display() {

base.Display();

Console.WriteLine($"Weapon: {Weapon.Type}");

}

}

public class Weapon {

public string Type { get; set; }

public Weapon(string type) => Type = type;

}

// استفاده در بازی

var originalSoldier = new Soldier(); // بارگذاری سنگین

originalSoldier.Position = new Vector3(10, 0, 10);

// ایجاد سربازان جدید با کپی کردن

var soldier1 = (Soldier)originalSoldier.Clone();

soldier1.Position = new Vector3(20, 0, 20);

var soldier2 = (Soldier)originalSoldier.Clone();

soldier2.Position = new Vector3(30, 0, 30);

soldier2.Weapon.Type = "Shotgun";

originalSoldier.Display();

soldier1.Display();

soldier2.Display();

## تفاوت با الگوهای مشابه

* **Factory Method**: ایجاد اشیا از طریق زیرکلاس‌ها
* **Abstract Factory**: ایجاد خانواده‌ای از اشیا مرتبط
* **Prototype**: ایجاد اشیا جدید با کپی کردن نمونه‌های موجود

## استفاده در #C و .NET

1. **ICloneable**: رابط استاندارد در .NET برای پیاده‌سازی الگوی Prototype
2. **MemberwiseClone()**: متد protected در کلاس Object برای ایجاد کپی سطحی
3. **Copy Constructors**: روش جایگزین برای پیاده‌سازی Prototype در #C

## بهترین روش‌های پیاده‌سازی

1. **کپی سطحی در مقابل عمیق**:
   * از MemberwiseClone() برای کپی سطحی استفاده کنید
   * برای کپی عمیق، باید اعضای مرجع را به صورت دستی کپی کنید
2. **راهکارهای کپی عمیق**:
   * Serialization/Deserialization
   * Reflection
   * Expression Trees
   * استفاده از کتابخانه‌هایی مانند AutoMapper
3. **ثبت نمونه‌های اولیه**:

public class PrototypeRegistry

{

private Dictionary<string, IPrototype> \_prototypes = new Dictionary<string, IPrototype>();

public void AddPrototype(string key, IPrototype prototype)

=> \_prototypes[key] = prototype;

public IPrototype GetPrototype(string key)

=> \_prototypes[key].Clone();

}

**پیاده‌سازی با Serialization**:

public static T DeepCopy<T>(T obj)

{

using (var ms = new MemoryStream())

{

var formatter = new BinaryFormatter();

formatter.Serialize(ms, obj);

ms.Position = 0;

return (T)formatter.Deserialize(ms);

}

}

این الگو زمانی مناسب است که:

* سیستم باید مستقل از نحوه ایجاد اشیا باشد
* ایجاد اشیا جدید از طریق سازنده هزینه‌بر باشد
* نیاز به پیکربندی پویای اشیا در زمان اجرا داشته باشید
* بخواهید از پیچیدگی سلسله مراتب کلاس‌ها بکاهید

-------------------------------------------------------------------------------------------------------