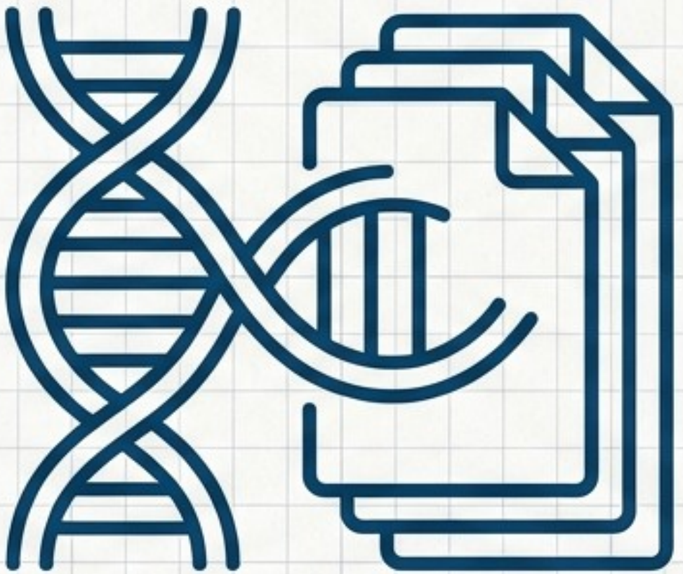
The Prototype Pattern



An Efficiency Blueprint for Object Creation

Mastering the Art of Efficient Cloning

What is the Real Cost of Creation?

We begin by stating that sometimes, creating new objects is not a trivial task. It can be:

- **Expensive**: Consumes significant memory or resources.
- **Slow**: Involves time-consuming operations that impact performance.
- **Complex**: Requires intricate setup logic and multiple steps.

Expensive DB Lookups:
Fetching initial state from a database.

Parsing Configuration:
Loading and interpreting complex configuration files.

Loading Assets:
Initialising objects from large resources like images.

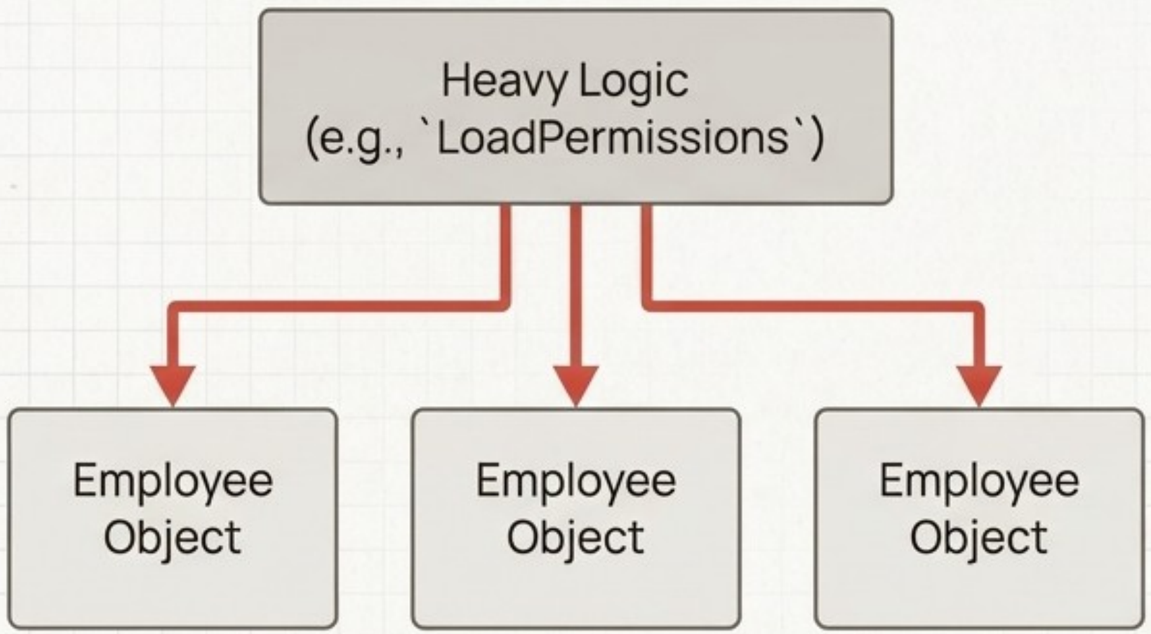
Network Calls:
Constructing objects based on remote data.

Instead of rebuilding the object from scratch every single time, what if there was a more efficient way?

The Inefficient Way: Rebuilding Again and Again

The Problem in Practice

Consider creating multiple 'Developer' employees. Each employee object needs a standard set of permissions loaded, which is a heavy, repetitive operation.

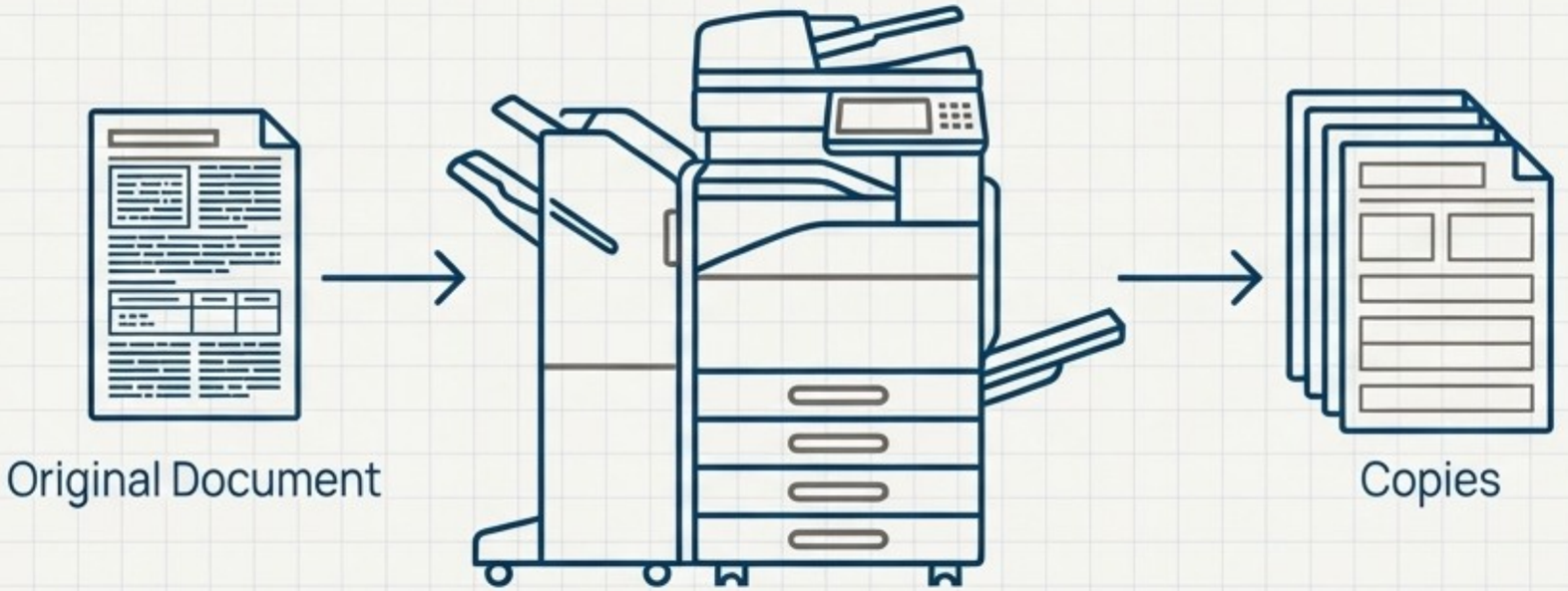


C# Example

```
public class Employee
{
    public string Role;
    public string Department;
    public List<string> Permissions;

    public Employee(string role, string department)
    {
        Role = role;
        Department = department;
        // This is the expensive part!
        Permissions = LoadPermissions(role);
    }
}

// Usage
var dev1 = new Employee("Developer", "IT"); // ❌ Loads permissions
var dev2 = new Employee("Developer", "IT"); // ❌ Loads permissions again
var dev3 = new Employee("Developer", "IT"); // ❌ Wastes CPU & time
```

👉 Heavy logic executed every time

The Core Idea: Think ‘Photocopier’, Not ‘Factory’



1. **Create Once:** You meticulously draft an application form. This is your ‘prototype’ object.
2. **Copy on Demand:** Instead of rewriting the entire form for each new applicant, you place the original in the copier.
3. **Customise the Copy:** You get a perfect copy of the structure and layout. All you need to do is fill in the unique details.

The Prototype pattern works on the same principle: create one expensive “template” object, then produce cheap copies of it as needed.

The Blueprint: Implementing the Clone Method

The goal is simple: create a base object, then clone it. This is achieved by implementing a common cloning interface.

Step 1 - Define the Prototype Interface

First, we define a contract that guarantees an object can clone itself.

```
// A simple, generic interface for any clonable object
public interface IPrototype<T>
{
    T Clone();
}
```

Step 2 - Implement the Cloning Logic

Next, the Employee class implements the interface. The Clone method uses MemberwiseClone to create a copy.

```
public class Employee : IPrototype<Employee>
{
    public string Role;
    public string Department;
    public List<string> Permissions;

    // ... constructor ...

    public Employee Clone()
    {
        // MemberwiseClone creates a bit-by-bit copy
        return (Employee)this.MemberwiseClone();
    }
}
```

Note: We will discuss the crucial details of MemberwiseClone shortly.

The Transformation: From Rebuilding to Cloning

WITHOUT Prototype

```
// Expensive constructor is called repeatedly
var dev1 = new Employee("Developer", "IT"); ❌
var dev2 = new Employee("Developer", "IT"); ❌
var dev3 = new Employee("Developer", "IT"); ❌
```

❌ Wasted CPU ❌ Slow ❌ Repetitive

WITH Prototype

```
// Create the expensive prototype just once
var baseDeveloper = new Employee("Developer", "IT");

// Clone it instantly
var dev1 = baseDeveloper.Clone(); ✅
var dev2 = baseDeveloper.Clone(); ✅
var dev3 = baseDeveloper.Clone(); ✅
```

✅ Fast ✅ Simple ✅ Efficient

The result: Drastically reduced overhead for creating similar objects.

The Critical Detail: The Danger of a Shallow Copy

`MemberwiseClone` performs a **shallow copy** by default. This is a crucial distinction to master.

Visual Diagram & Explanation

- **Value Types (e.g., string, int):** Are copied directly. Changes in the clone do not affect the original.
- **Reference Types (e.g., Lists, custom objects):** The ***reference*** is copied, not the object itself. Both the original and the clone end up pointing to the **exact same object in memory**.

Code Example

```
// We clone the base developer
var dev1 = baseDeveloper.Clone();

// Now, we modify the permissions list of the
clone
dev1.Permissions.Add("NewAdminAccess");

// The BUG: The original's permissions list is
also modified!
// baseDeveloper.Permissions now also contains
"NewAdminAccess".
```

The Solution: Mastering the Deep Copy

To achieve **true independence** between clones, you must perform a **deep copy** by manually duplicating any **nested reference-type** objects.

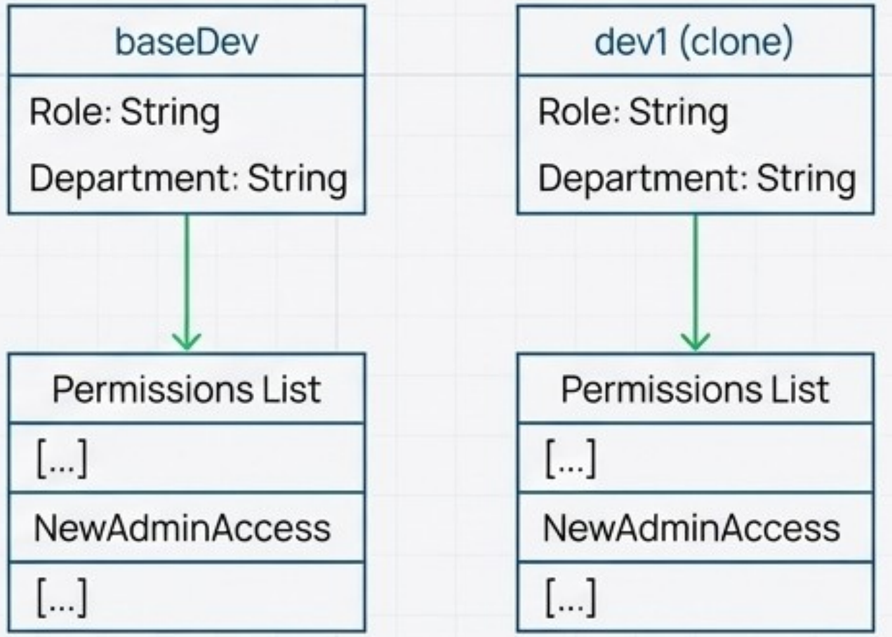
Corrected Code

We override the `Clone` method to first create the shallow copy, then explicitly create a *new* list for the `Permissions` property, copying the elements from the original.

```
public class Employee : IPrototype<Employee>
{
    // ... properties ...

    public Employee Clone()
    {
        // 1. Start with a shallow copy
        var copy = (Employee)this.MemberwiseClone();

        // 2. Manually clone the reference types
        copy.Permissions = new List<string>(this.Permissions);

        // 3. Return the fully independent clone
        return copy;
    }
}
```

👉 The critical fix

Updated Visual Diagram

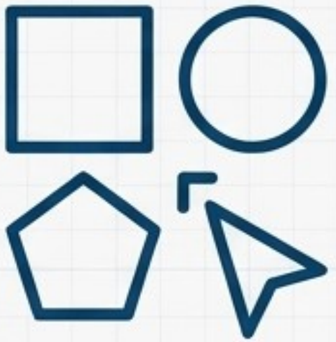


The Prototype Pattern in the Wild

This pattern is not just theoretical; it's a pragmatic solution used heavily in performance-critical and complex systems.

Game Engines

Cloning enemy objects, projectiles, or environmental assets without re-loading them from disk.



GUI & Document Editors

Implementing 'copy/paste' or duplicating shapes. The copied object (e.g., a rectangle) is a clone of the original, ready to be moved or resized.

ORM & Caching Systems

Creating copies of cached entities to avoid giving direct references to the cached object, preventing accidental modification.

Test Data Generation

Quickly prototyping and creating numerous variations of test objects that share a common base state.

A Framework for Deciding: To Clone or Not to Clone?

Use the Prototype Pattern When...

- Object creation is genuinely expensive (computation, I/O).
- You need many instances of similar objects that only differ slightly in their state.
- The system needs to be independent of how its products are created, composed, and represented.
- Copying an existing instance is demonstrably cheaper than creating a new one.

Consider an Alternative When...

- The object is small, simple, and cheap to create (e.g., a simple DTO).
- The cloning logic becomes overly complex due to a deep and tangled graph of nested objects.
- Your design philosophy favours immutability.
- A different creational pattern like **Builder** (for complex, step-by-step construction) or **Factory** (to decouple client from concrete classes) is a better fit for the problem.

Grounded in Solid Design Principles

The Prototype pattern naturally aligns with core object-oriented design principles, leading to more maintainable and extensible code.

Single Responsibility Principle (SRP)

> “An object should have only one reason to change.”
>
> How Prototype Applies: The object itself encapsulates the logic required for its own cloning. The responsibility of duplication is handled by the object, not by an external manager or factory.

Open/Closed Principle (OCP)

> Software entities should be open for extension, but closed for modification.”
>
> How Prototype Applies: You can introduce new clonable classes that implement the `IPrototype` interface without changing the client code that uses them. The client simply works with the `Clone()` method, regardless of the object's concrete type.

The Prototype Pattern: At a Glance

Item	Meaning
The Core Problem	The **high cost** and **complexity** of repeatedly re-creating heavy objects from scratch.
The Solution	**Clone** a pre-configured, existing “prototype” object instead of building a new one.
Key Idea	**Copy instead of rebuild.**
Primary Benefits	Performance gains, code simplification, avoids duplicated setup logic.
Critical Risk	Unintentionally sharing state due to shallow copies. Always be mindful of deep vs. shallow copy requirements.
Guiding Principles	Adheres to Single Responsibility (SRP) and Open/Closed (OCP).